Shuvra S. Bhattacharyya
Ed F. Deprettere · Rainer Leupers
Jarmo Takala *Editors*

Handbook of Signal Processing Systems

Third Edition

Springer

Handbook of Signal Processing Systems

Shuvra S. Bhattacharyya • Ed F. Deprettere
Rainer Leupers • Jarmo Takala
Editors

Handbook of Signal Processing Systems

Third Edition

Foreword by S.Y. Kung

Springer

Editors
Shuvra S. Bhattacharyya
Department of ECE and UMIACS
University of Maryland
College Park, MD, USA

Laboratory for Pervasive Computing
Tampere University of Technology
Tampere, Finland

Rainer Leupers
RWTH Aachen University Software
for Systems on Silicon
Aachen, Germany

Ed F. Deprettere
Leiden Embedded Research Center
Leiden University Leiden Institute Advanced
Computer Science
Leiden, The Netherlands

Jarmo Takala
Department of Pervasive Computing
Tampere University of Technology
Tampere, Finland

ISBN 978-3-319-91733-7 ISBN 978-3-319-91734-4 (eBook)
https://doi.org/10.1007/978-3-319-91734-4

Library of Congress Control Number: 2018953763

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To Milu
Shuvra Bhattacharyya

To Deirdre
Ed Deprettere

To Bettina
Rainer Leupers

To Auli
Jarmo Takala

Foreword

It gives me immense pleasure to reintroduce this handbook to the research/development communities in the field of signal processing systems (SPS). The handbook represents the first of its kind to provide a comprehensive coverage on state of the arts of this field. The fact that it is already now the third edition is a clear attestation of the high demand from all the related professional communities. It is truly an influential and timely contribution to the field of SPS.

The driving force behind information technologies (IT) hinges critically upon the major advances in both component integration and system integration. The major breakthrough for the former is undoubtedly the invention of IC in the 1950s by Jack S. Kilby, the Nobel Prize Laureate in Physics in 2000. In an integrated circuit, all components were made of the same semiconductor material. Beginning with the pocket calculator in 1964, there have been many increasingly complex applications followed. In fact, processing gates and memory storage on a chip have since then grown at an exponential rate, following Moore's Law. (Moore himself admitted that Moore's Law had turned out to be more accurate, longer lasting, and deeper in impact than he ever imagined.) With greater device integration, various signal processing systems have been realized for many killer IT applications. Further breakthroughs in computer sciences and Internet technologies have also catalyzed large-scale system integration. All these have led to today's IT revolution which has profound impacts on our lifestyle and overall prospect of humanity. (It is hard to imagine life today without mobiles or the Internet!)

The success of SPS requires a well-concerted integrated approach from multiple disciplines, such as device, design, and application. It is important to recognize that system integration means much more than simply squeezing components onto a chip and, more specifically, there is a symbiotic relationship between applications and technologies. Emerging applications, e.g., 5G communication, big data analysis, machine learning, and the trendy AI, will prompt modern system requirements on performance and power consumption, thus inspiring new intellectual challenges. Therefore, the new paradigm of SPS architectures must be amenable to various design facets such as overall system performance, flexibility, and scalability, power/thermal management, hardware-software partition, and algorithm developments.

With greater integration, system designs become more complex and there exists a huge gap between what can be theoretically designed and what can be practically implemented. It is critical to consider, for instance, how to deploy in concert an ever increasing number of transistors with acceptable power consumption and how to make hardware effective for applications and yet friendly to the users (easy to program). Therefore, major advances in SPS must arise from close collaboration between application, hardware/architecture, algorithm, CAD, and system design.

It is only fitting for Springer/Nature to produce this timely handbook. Springer/-Nature has long played a major role in academic publication on SPS, many of them have been in close cooperation with IEEE's signal processing, circuits and systems, and computer societies. For nearly 30 years, I have been the editor-in-chief of Springer's *Journal of Signal Processing Systems*, considered by many as a major forum for the SPS researchers. Nevertheless, the idea has been around for years that a single-volume reference book would very effectively complement the journal in serving this technical community. Then, during the 2008 IEEE Workshop on Signal Processing Systems, Washington D.C., Jennifer Evans from Springer and the editorial team led by Prof. Shuvra Bhattacharyya met to brainstorm implementation of such idea. The result was this series of right-on-time handbooks. Especially, this edition has collected a vast pool of leaders/pioneers to cover architectures; compilers, programming and simulation tools; and design tools and methodologies.

Indeed, the handbook offers a comprehensive and up-to-date treatment of the driving forces behind SPS, current architectures, and new design trends. It provides a solid foundation for several imminent technical areas, for instance, scalable, reusable, and reliable system architectures, energy-efficient high-performance architectures, IP deployment and integration, system-on-chip, memory hierarchies, and future cloud computing. Moreover, it covers a wide spectrum of applications, including wireless/radio signal processing, image/video/multimedia processing, control and communication, video coding, stereo vision, computer vision, data mining, and machine learning.

Looking into the (near) future, we note that modern AI tools have become heavily data-driven and data-intensive. As of now, on the daily basis, as many as 1 billion photos and 10 billion messages are being handled by a single Internet company and, moreover, such dazzling numbers are rapidly growing on par with Moore's law. In order to unravel useful information hidden in big data, it will require novel (and possibly parallel processing) algorithmic designs which in turn will call for special hardware/software technologies advocated here. In this sense, the handbook is actually well positioned to support the increasingly data-driven AI technologies.

With the utmost enthusiasm, my sincere congratulations go to the authors and editors for putting together such an outstanding contribution.

Department of Electrical Engineering
Princeton University
Princeton, NJ, USA

S. Y. Kung

Preface

In this new edition of the *Handbook of Signal Processing Systems*, many of the chapters from the previous editions have been updated, and several new chapters have been added. The new contributions include chapters on signal processing methods for light field displays, throughput analysis of dataflow graphs, modeling for reconfigurable signal processing systems, fast Fourier transform architectures, deep neural networks, programmable architectures for histogram of oriented gradients processing, high dynamic range video coding, system-on-chip architectures for data analytics, analysis of finite word-length effects in fixed-point systems, and models of architecture.

We hope that this updated edition of the handbook will continue to serve as a useful reference to engineering practitioners, graduate students, and researchers working in the broad area of signal processing systems. Selected chapters from the book can be used as core readings for seminar- or project-oriented graduate courses in signal processing systems. Given the wide range of topics covered in the book, instructors have significant flexibility to orient such a course towards particular themes or levels of abstraction that they would like to emphasize.

This new edition of the handbook is organized in three parts. Part I motivates representative applications that drive and apply state-of-the-art methods for design and implementation of signal processing systems; Part II discusses architectures for implementing these applications; and Part III focuses on compilers, as well as models of computation and their associated design tools and methodologies. The chapters are ordered alphabetically by the first author's last name in Parts I and III, while they are ordered in Part II starting with chapters that cover more general topics, and followed by chapters that are more application-specific.

We are very grateful to all of the authors for their valuable contributions, and for the time and effort they have devoted to preparing the chapters. We would also like

to thank Courtney Clark, Caroline Flanagan, and Jennifer Evans for their support and patience throughout the entire development process of the handbook.

College Park, MD, USA — Shuvra S. Bhattacharyya
Leiden, The Netherlands — Ed F. Deprettere
Aachen, Germany — Rainer Leupers
Tampere, Finland — Jarmo Takala
13 January 2018

Contents

Volume I

Part I Applications

Signal Processing Methods for Light Field Displays 3
Robert Bregovic, Erdem Sahin, Suren Vagharshakyan, and Atanas Gotchev

Inertial Sensors and Their Applications 51
Jussi Collin, Pavel Davidson, Martti Kirkko-Jaakkola, and Helena Leppäkoski

Finding It Now: Networked Classifiers in Real-Time Stream Mining Systems 87
Raphael Ducasse, Cem Tekin, and Mihaela van der Schaar

Deep Neural Networks: A Signal Processing Perspective 133
Heikki Huttunen

High Dynamic Range Video Coding 165
Konstantinos Konstantinides, Guan-Ming Su, and Neeraj Gadgil

Signal Processing for Control 193
William S. Levine

MPEG Reconfigurable Video Coding 213
Marco Mattavelli, Jorn W. Janneck, and Mickaël Raulet

Signal Processing for Wireless Transceivers 251
Markku Renfors, Markku Juntti, and Mikko Valkama

Signal Processing for Radio Astronomy 311
Alle-Jan van der Veen, Stefan J. Wijnholds, and Ahmad Mouri Sardarabadi

Distributed Smart Cameras and Distributed Computer Vision 361
Marilyn Wolf and Jason Schlessman

Volume II

Part II Architectures

Arithmetic 381
Oscar Gustafsson and Lars Wanhammar

Coarse-Grained Reconfigurable Array Architectures 427
Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts

High Performance Stream Processing on FPGA 473
John McAllister

Application-Specific Accelerators for Communications 503
Chance Tarver, Yang Sun, Kiarash Amiri, Michael Brogioli, and Joseph R. Cavallaro

System-on-Chip Architectures for Data Analytics 543
Gwo Giun (Chris) Lee, Chun-Fu Chen, and Tai-Ping Wang

Architectures for Stereo Vision 577
Christian Banz, Nicolai Behmann, Holger Blume, and Peter Pirsch

Hardware Architectures for the Fast Fourier Transform 613
Mario Garrido, Fahad Qureshi, Jarmo Takala, and Oscar Gustafsson

Programmable Architectures for Histogram of Oriented Gradients Processing 649
Colm Kelly, Roger Woods, Moslem Amiri, Fahad Siddiqui, and Karen Rafferty

Part III Design Methods and Tools

Methods and Tools for Mapping Process Networks onto Multi-Processor Systems-On-Chip 685
Iuliana Bacivarov, Wolfgang Haid, Kai Huang, and Lothar Thiele

Intermediate Representations for Simulation and Implementation 721
Jerker Bengtsson

Throughput Analysis of Dataflow Graphs 751
Robert de Groote

Dataflow Modeling for Reconfigurable Signal Processing Systems 787
Karol Desnos and Francesca Palumbo

Integrated Modeling Using Finite State Machines and Dataflow Graphs 825
Joachim Falk, Kai Neubauer, Christian Haubelt, Christian Zebelein, and Jürgen Teich

Kahn Process Networks and a Reactive Extension 865
Marc Geilen and Twan Basten

Decidable Signal Processing Dataflow Graphs 907
Soonhoi Ha and Hyunok Oh

Systolic Arrays 939
Yu Hen Hu and Sun-Yuan Kung

Compiling for VLIW DSPs 979
Christoph W. Kessler

Software Compilation Techniques for Heterogeneous Embedded Multi-Core Systems 1021
Rainer Leupers, Miguel Angel Aguilar, Jeronimo Castrillon, and Weihua Sheng

Analysis of Finite Word-Length Effects in Fixed-Point Systems 1063
D. Menard, G. Caffarena, J. A. Lopez, D. Novo, and O. Sentieys

Models of Architecture for DSP Systems 1103
Maxime Pelcat

Optimization of Number Representations 1141
Wonyong Sung

Dynamic Dataflow Graphs 1173
Bart D. Theelen, Ed F. Deprettere, and Shuvra S. Bhattacharyya

Part I
Applications

Signal Processing Methods for Light Field Displays

Robert Bregovic, Erdem Sahin, Suren Vagharshakyan, and Atanas Gotchev

Abstract This chapter discusses the topic of emerging light field displays from a signal processing perspective. Light field displays are defined as devices which deliver continuous parallax along with the focus and binocular visual cues acting together in rivalry-free manner. In order to ensure such functionality, one has to deal with the light field, conceptualized by the plenoptic function and its adequate parametrization, sampling and reconstruction. The light field basics and the corresponding display technologies are overviewed in order to address the fundamental problems of analyzing light field displays as signal processing channels, and of capturing and representing light field visual content for driving such displays. Spectral analysis of multidimensional sampling operators is utilized to profile the displays in question, and modern sparsification approaches are employed to develop methods for high-quality light field reconstruction and rendering.

1 Introduction

The unequivocal aim of visual media is to provide high realism of the scene being visualized and to provide tools for interacting with visual content. Visual information about real-world objects is carried by the light field, i.e., light of any wavelength travelling in every direction through every point in space. Subsequently, the light field data is *rich* in providing high spatial, angular, and spectral resolution of the visual content. In order to utilize this richness and to convert it into highly realistic and interactive visual experience, extensive research efforts have been made to study the principles of light field formation, propagation, sensing and perception along with the computational methods for extracting, processing and rendering the visual information. In this list of methods, the light field display has a special place

R. Bregovic · E. Sahin · S. Vagharshakyan · A. Gotchev (✉)
Laboratory of Signal Processing, Tampere University of Technology, Tampere, Finland
e-mail: robert.bregovic@tut.fi; erdem.sahin@tut.fi; suren.vagharshakyan@tut.fi; atanas.gotchev@tut.fi

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_1

as the ultimate light field reconstruction stage and device, where optics and signal processing meet.

Attempting high-quality light field reconstruction from a large, yet limited, collection of sensors, has demanded research on new sensing concepts and novel sparse light field representations.

In this chapter, we review the light field basics in Sect. 2 and overview the light field display technologies in Sect. 3, in order to prepare the ground for discussing two fundamental signal processing challenges related with such displays. Departing from the light field representation and propagation formalism, in Sect. 4 we present our approach in profiling light field displays in terms of their throughput, which is analyzed in spectral domain and quantified through the notion of display bandwidth. In Sect. 5, we address the fundamental issue of preparing light field content for any type of display. Our main representation is the so-called densely sampled light field and our main tool is its sparse representation in directional transform domain.

2 Light Field Basics

2.1 Plenoptic Function

The light field (LF) was first conceptualized by Gershun as the amount of light traveling in every direction through every point in space using light vectors [1]. That is, considering rays as the fundamental light carrier, any region of space is interpreted as a collection of light rays. The plenoptic function [2] describes the intensity distribution of these rays. In the most general case, it is a 7-dimensional function parametrizing the crossing points (x, y, z), propagation directions (θ, ϕ), and wavelengths (colors) (λ) of the light rays at a given time (t). The measurement of the plenoptic function can be characterized by considering a space filled with idealized pinhole apertures at every location recording the intensity of the light at every angle passing through it for each possible value of wavelength and time. When three-dimensional (3D) objects (scenes) are viewed by an observer, the human visual system samples the pattern of light rays filling the space around the objects. As such, even though the plenoptic function is often considered as an idealized concept due to difficulties specifying it completely for natural scenes, it can be regarded as a communication link between (the objects in) the scene and the perceived retinal images [2].

2.2 Light Field Parametrization

The plenoptic function can be reduced into a five-dimensional function of spatial (3D) and angular (2D) coordinates for a static scene under monochromatic

illumination. In case there is a transparent medium for the light to propagate in and the analysis is limited to the subset of rays leaving a bounded object, i.e., including only the regions outside the convex hull of the object, the plenoptic function contains redundant information [3, 4]. That is, the radiance along a ray from one point to another remains constant (assuming no participating media). Thus, the dimensionality of the function can be further reduced to 4D. Examples of such LF representations include the 4D LF presented in [3], the Lumigraph [4] and the photic field [5]. The 4D LF information can be parametrized in various ways, e.g., by considering points on a surface and directions for each point, pairs of points on the surface of a 3D shape (cube, sphere) or pairs of points on two planes. The two-plane parametrization of LF is often preferred, since it is well suited for modelling widely studied LF capture and display techniques, such as integral imaging and multiview capture/display, with one plane corresponding to the viewpoints and the other one corresponding to the image/display plane of the camera/display device. Let us consider the notation $L(s,t,u,v)$ for the two-plane parametrization. That is, each ray captured by the LF crosses the two planes at positions (s,t) and (u,v), respectively. Alternatively, the LF can be parametrized as the rays on a single plane (s,t) and two angles (θ,φ) representing the direction of each ray, resulting in notation $L(s,t,\theta,\varphi)$ for the 4D LF. These parametrizations are visualized in Fig. 1.

2.3 Light Ray Propagation

The analysis of LFs usually includes different parametrizations at different depths. In order to link the LF representations at such different depths, it is necessary to formulate the light propagation based on the LF paradigm. For simplicity, let us assume 2D light fields, $L(t,v)$ for fixed s and u, and $L(t,\theta)$ for fixed s and ϕ. Let us also assume that v represents the relative position with respect to crossing point of the ray on t-coordinate (cf. Fig. 2). The relation between these two LF

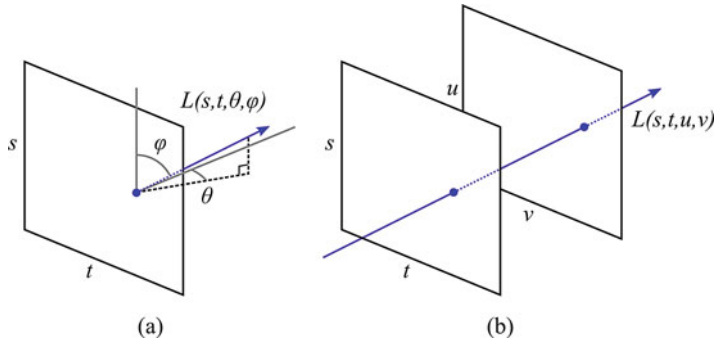


Fig. 1 Two different 4D LF parametrizations. (**a**) Space-angle parametrization. (**b**) Two-plane parametrization

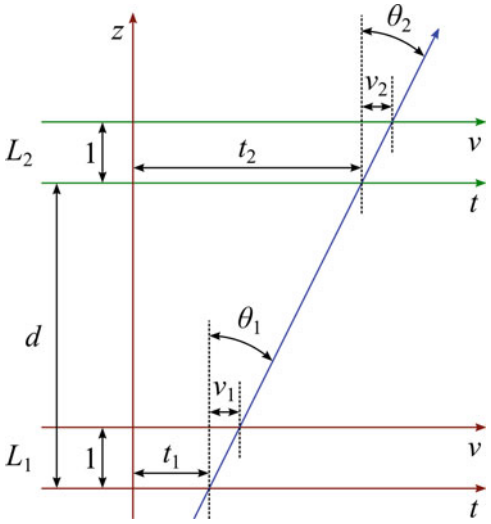


Fig. 2 Light ray propagation in space, represented by two LF parametrizations

representations is given by $v = d_l \tan\theta$, where d_l is the distance between the two planes in the former parametrization and s is same in both representations. The propagation of a light ray in space is illustrated in Fig. 2, where the two t-planes are separated by d and the separation between t and v planes is assumed to be unit distance, i.e., $d_l = 1$. Considering such LF parametrizations on both planes, the light ray propagation can be expressed as [6, 7]

$$L_2\left(\begin{bmatrix} t_2 \\ v_2 \end{bmatrix}\right) = L_1\left(\begin{bmatrix} t_1 \\ v_1 \end{bmatrix}\right) = L_1\left(\begin{bmatrix} 1 & -d \\ 0 & 1 \end{bmatrix}\begin{bmatrix} t_2 \\ v_2 \end{bmatrix}\right) \tag{1}$$

$$L_2\left(\begin{bmatrix} t_2 \\ \theta_2 \end{bmatrix}\right) = L_1\left(\begin{bmatrix} t_1 \\ \theta_1 \end{bmatrix}\right) = L_1\left(\begin{bmatrix} t_2 - d\tan\theta_2 \\ \theta_2 \end{bmatrix}\right), \tag{2}$$

where L_1 and L_2 refer to the two LFs at the first and second plane positions, respectively. Thus, (1) and (2) actually link the two LFs defined at different depths. In the case of two-plane parametrization, the position on the v-axis changes according to a linear transformation of ray direction and distance. That is, a shifting operation (shearing) is performed along the v-axis [8]. However, as can be seen from (2), the relation between the LF parameters is not strictly linear in the plane and angle parametrization.

2.4 Epipolar Plane Images

As discussed in the previous sections, in its simplified form, LF can be described by a 4D function, with the two-plane parameterization being the most common representation (see Fig. 1b) of that function. However, even in this simplified form, due to the complex nature of light, it is still difficult to analyze LF data in a systematic way. Therefore one needs another simplification step—the Epipolar plane images (EPIs) being one of the choices.

EPIs, originally introduced by Bolles et al. [9], are based on the concept that a point in space at a distance z from the camera plane will follow certain geometrical relation, to be described later, when mapped to different camera images at different positions. Forming of an EPI in the case of a horizontal parallax only (HPO) LF is illustrated in Fig. 3. A camera moving along t axis, also referred to as the camera plane, Fig. 3a, captures images at equidistant intervals, Fig. 3b. Those images are then put into a 3D structure, Fig. 3c, referred to as the epipolar cube. An EPI, Fig. 3d, is then obtained by slicing the epipolar cube along the u axis. In other words, an EPI consists of the same row from each image stacked together and as such it can be considered to be a 2D image of size n_{col} (horizontal camera resolution) by n_{im} (number of images or cameras). This maps a complex scene into regular structures that have a higher predictability and are easier to analyze.

Denoting the camera-to-camera distance $\Delta t = t_2 - t_1$ and the change of the position of the point in the images $\Delta v = v_2 - v_1$, it follows that the relation between points in space in terms of camera pixels and camera position are given as

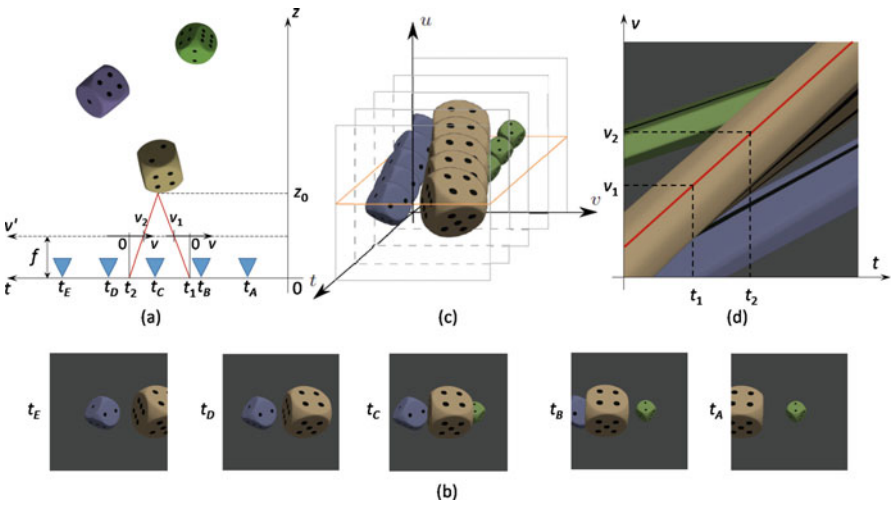


Fig. 3 Forming of EPIs from a 3D scene. (**a**) 3D scene with denoted five camera positions $t_A \ldots t_E$. (**b**) Captured images on camera positions $t_A \ldots t_E$. (**c**) Epipolar cube constructed from captured images. (**d**) One EPI for a large number of captured images with an EPI line marked in red

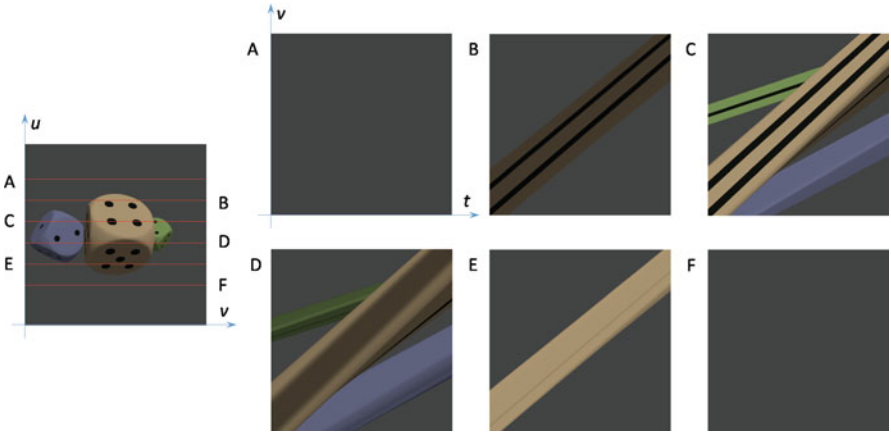


Fig. 4 EPIs formed over different parts (image lines) of the same scene

$$v = \frac{v_2 - v_1}{t_2 - t_1}(t - t_1) + v_1 = \frac{f}{z}(t - t_1) + v_1. \tag{3}$$

Here f is the distance between the camera and image plane and $\Delta v = \frac{f}{z}\Delta t$.

The main benefits of EPI representation for processing LFs lies in the fact that scene points appear as lines in an EPI, see Fig. 3d. The slope of a line is determined by distance of the point from the camera, camera resolution and distance between adjacent cameras. The points closer to the camera make a steeper slope (more vertical) and also occlude points further away that have a more horizontal slope. This will be utilized later on when discussing efficient LF interpolation techniques.

Four comments related to EPIs. First, although EPIs are structured, for the same scene, the structure can differ considerably from EPI to EPI (different lines in the images relate to different parts of the scene) as illustrated in Fig. 4. Second, in comparison with the two-plane parameterization, see Fig. 1b, when forming/denoting the EPIs the t-axis goes in the opposite direction and v axis is sheared such to become relative with the point on the t-axis under consideration (there is no common zero for the v-axis), see Fig. 3a for illustration. Third, there is a one-to-one correspondence between EPI and ray-space notation—only v-axis has to be replaced by ray angles. For small field of view (FoV) it can be assumed that $\Delta v = z \tan \Delta\theta$ with z being the distance to the object and $\Delta\alpha$ angular sampling density [6]. Fourth, in the case of full parallax, in addition to EPIs in horizontal direction, one can also form EPIs in vertical direction by slicing the 4D EPI cube along the us-plane.

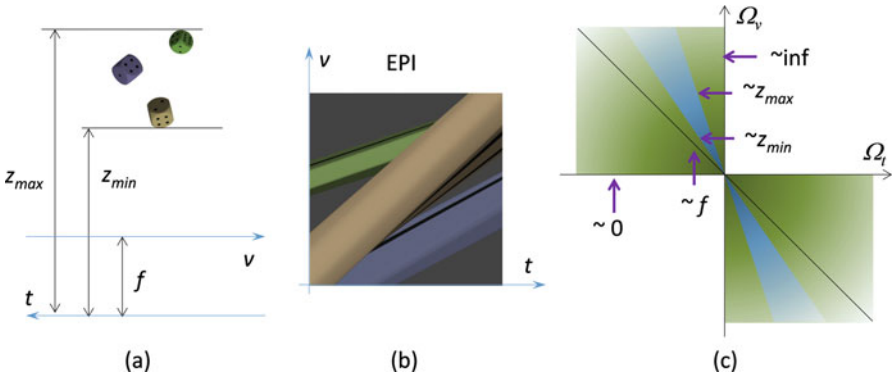


Fig. 5 EPI—from scene to representation in continuous Fourier domain. (**a**) Scene setup. (**b**) EPI in spatial domain. (**c**) EPI in continuous Fourier domain

2.5 Fourier Domain Representation

The regularities in the structure of an EPI can be further exploited by analyzing the EPI in the Fourier domain [10, 11]. Assuming a scene with limited depth, as illustrated in Fig. 5a, captured by a dense set of cameras, the spectrum of an EPI, Fig. 5b, will be limited to a bow-tie shaped area with size (edges) depending on $z_{\min}$ and $z_{\max}$, blue area in Fig. 5c. As one can see the spectrum of an EPI is very well localized, particularly for scenes with shallow depth. Moreover, all points at the same distance (layer) map into a line in spectral domain with the slope proportional to the distance of the layer. Points in infinity map to the vertical axis (in spectral domain) and objects at the camera plane, map to the horizontal axis.

As pointed out earlier, the frequency support of a depth layer is limited to a line in the Fourier domain. In most general case a scene consists of objects at all depths and therefore the spectrum is as the one indicated by the blue bow-tie in Fig. 5c. However, in practice, the objects are typically grouped—each object can be associated (approximated) with one or more depth layers. This is illustrated in Fig. 6. For objects with a single (shallow) layer/depth, as in Fig. 6a, the whole spectrum is localized in the vicinity of one line. For a scene with objects at considerably different depths, we have several layers that are reflected in the spectrum as several lines. As seen for an example in Fig. 6b, there are three dominant layers and those are clearly visible in the spectra. The fact that many scenes can be split into layers has been used in several algorithms that work with LFs, e.g., [12].

In theory, the above discussion applies only to scenes without occlusion. In the case of occlusions the spectra will be more spread out, as illustrated in Fig. 7 for various levels of occlusions [11] with the most right image illustrating the case of so called ‘dominant’ occlusions, that is, when a number of very close objects occludes one or more far objects. However, as we can see in Fig. 6b, in practical scenarios, even if there are occlusions, the aforementioned analysis can still be

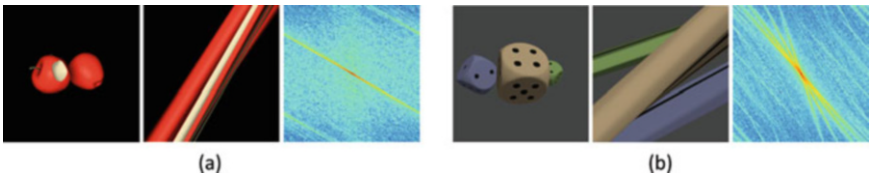


Fig. 6 Scenes of different complexity—Image of the scene, one EPI, and its Fourier domain representation. (**a**) Scene with a 'single' (shallow) layer. (**b**) Scene with several dominant layers

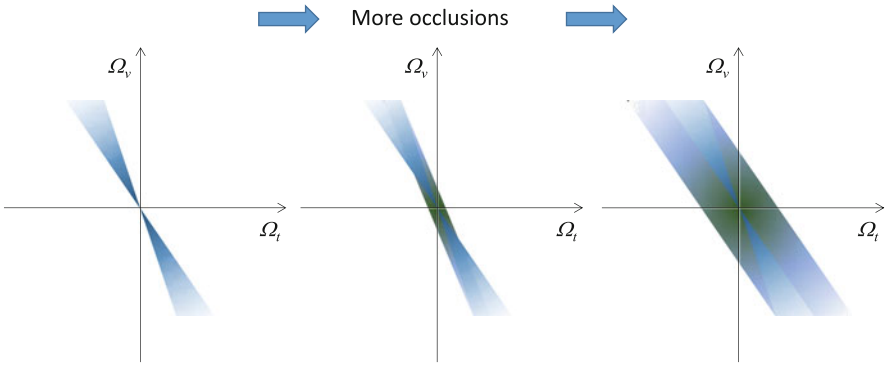


Fig. 7 Fourier domain representation for scenes with different level of occlusions present in the scene

applied. Therefore, in the rest of this chapter we will assume that the spectrum of a scene behaves as in the case of non-occluded scenes.

2.6 Plenoptic Sampling

The continuous Fourier domain representation of an EPI, considered in the previous section, is the natural domain for analyzing EPIs since LF of a scene is a band-unlimited signal [13]. However, in practice one works with discrete systems which means discretization. There are two types of discretization of LF that occur. First one is due to capturing the scene with a limited (finite) number of cameras (angular sampling) and second is due to the fact that each captured image has a finite resolution (spatial sampling). Both of those either require bandlimited signals as input, or proper antialiasing filters before sampling. Otherwise, the sampled LF will be contaminated with spatial and/or interperspective aliasing.

Discretization in the spatial domain is illustrated in Fig. 8a. For a dense-enough sampling (using high resolution cameras), v_1, with respect to the scene there is no overlap between the baseband (blue) and replicas (green). In such case the continuous signal can be easily reconstructed, e.g., by a separable reconstruction

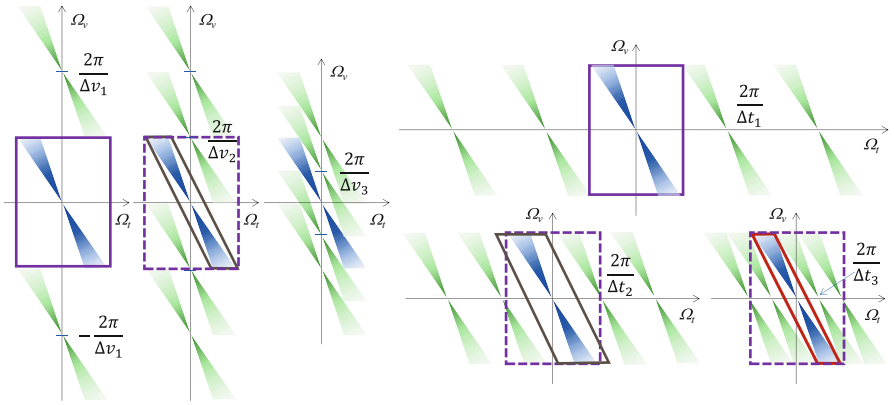


Fig. 8 Discretization of the plenoptic function with different sampling rates. (**a**) Different camera resolution Δv with $\Delta v_1 < \Delta v_2 < \Delta v_3$. (**b**) Different camera to camera distance Δt with $\Delta t_1 < \Delta t_2 < \Delta t_3$

filter marked in purple. For lower sampling densities, v_2, v_3, the spectrum replicas close in on the baseband requiring a tighter 2D reconstruction filter, marked in brown. If the camera resolution is further reduced, $\Delta v > \Delta v_3$, then the baseband and replicas start to overlap and reconstruction using standard multidimensional sampling theory is no more possible. Similarly, the discretization in the angular domain is illustrated in Fig. 8b. In this case, when the cameras are too far apart, $\Delta t > \Delta t_3$, replicas start to overlap and direct reconstruction is not anymore possible.

In practice, the discretization happens simultaneously in spatial and angular domain. It is obvious that smaller scene features need a higher camera resolution and closer objects need a denser set of cameras. However, neither of those can be infinitely increased. The questions that arise here are: How to sample a scene properly (to avoid or minimize aliasing)? What is the optimal sampling that would enable the reconstruction of the scene's continuous plenoptic function?

The answers to these questions, particularly to the second one, depends heavily on the way how one wants to perform the reconstruction of the continuous plenoptic function, that is, in addition to sampled visual data (images) is there any other knowledge about the scene available (in the form of another modality) or are there assumptions that can be made regarding the scene that could assist in the reconstruction.

As discussed earlier, if only images of a scene are available, then, theoretically for every scene one would need proper antialiasing filters in spatial and angular domain in order to sample the scene without aliasing. Spatial antialiasing is typically handled by the camera itself, however, it is not straightforward to implement an antialiasing filter in the angular domain. Therefore, one needs to sample the scene with a dense set of cameras (small$^{\Delta v}$). An attempt to define what means dense enough is proposed in [13] that introduces the concept of essential bandwidth that is defined as: "A compact region in the frequency domain that is symmetrical around

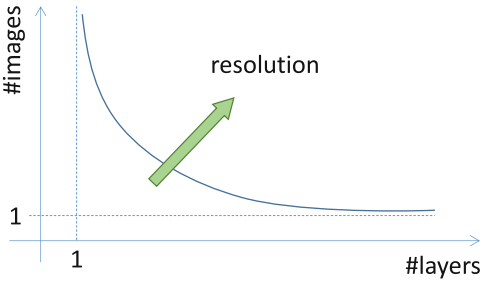


Fig. 9 Optimal sampling for alias-free reconstruction—compromise between the number of layers and number of images

the origin and that contains at least 81% of the plenoptic spectrum's energy." The required sampling is estimated based on the knowledge of the highest frequency in the scene and the depth range—for more about that estimation please see [13].

When in addition to images also depth information is available as an additional modality, it has been shown in [10] that there is a compromise between the required number of images and the amount of depth information expressed in terms of layers (number of depths the scene is quantized) that will result in a similar reconstruction quality. This compromise is illustrated in Fig. 9 and can be summarized as follows: The more one knows about the geometry of the scene, the fewer images are needed and vice versa.

This is as far as one can go using standard sampling theory. Going beyond that, we will show in Sect. 5 that by making few assumptions about the scene (e.g., scene has no reflective surfaces) one can utilize the properties of EPIs, in spatial and Fourier domain and go far beyond the classical sampling theory and reconstruct the continuous function from a sparse (under-sampled) set of images without any knowledge about the depth of objects in the scene.

2.7 Densely Sampled Light Field

As discussed in the previous section, when reconstructing the plenoptic function out of an under-sampled (captured) LF, one needs to use advance reconstruction techniques in order to avoid aliasing. However, in many cases (e.g., when speed is essential) it is beneficial to achieve a good (satisfactory) reconstruction results by using a simple interpolation technique, e.g., bilinear (or quadrilinear) interpolation over the available LF samples. This is possible only if the LF is sampled densely enough. A sampled LF from which the continuous plenoptic function can be reconstructed by simple quadrilinear interpolation is referred to as densely sampled light field (DSLF). Its characteristic is that the maximum disparity of any point in the scene between adjacent views is less than or equal to one pixel. Such sampling ensures that lines in EPI are unambiguous. This is illustrated in Fig. 10 for cases where the disparity is three pixels, Fig. 10a, and one pixel, Fig. 10b, between adjacent views. When interpolating views in between, in the first case one cannot use

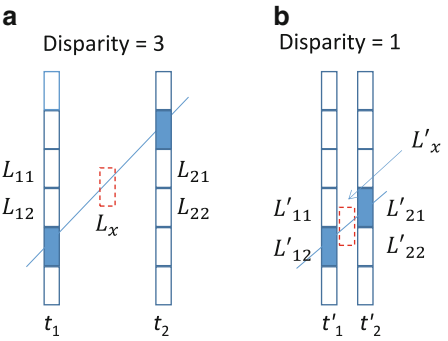


Fig. 10 Difference between differently sampled LFs emphasizing the known samples with the corresponding epipolar line (blue) and the interpolated sample (red). (**a**) Under sampled LF, $L_x = f(L_{11}, L_{12}, L_{21}, L_{22})$. (**b**) Densely sampled LF, $'_x = f\left(L'_{11}, L'_{12}, L'_{21}, L'_{22}\right)$

a simple bilinear interpolation since it will use (adjacent) pixels that are not part of the EPI line, whereas in the second case the correct pixels will be utilized. Although the reconstruction still might not be perfect, the effect of aliasing will be almost negligible—no major errors due to aliasing will be introduced in the reconstructed plenoptic function. One can claim that such sampling allows treating the disparity space as a continuous space.

The required sampling density on the t and v plane to achieve DSLF depends on the (minimal) depth and (smallest) details in the scene. Scenes with objects closer to the camera and more details will require a denser set of cameras and images of higher resolution. As a side benefit, once the DSLF is available, one can properly apply multidimensional filtering (removing/blurring objects in the scene) and then downsample the LF to resolution that can be used, for example on a display, without introducing aliasing at lower sampling rates.

3 Light Field Displays

3.1 Visual Cues

The human visual system (HVS) creates 3D perception based on 3D information acquired via a number of depth cues. These visual cues can be coarsely classified into physiological and psychological cues [14, 15]. Physiological cues such as binocular disparity, convergence and accommodation produce information based on physical reaction of the HVS. On the other hand, psychological cues such as linear perspective and texture gradients are more related to learned experiences. The visual cues can be also divided into four categories by a finer classification [15–17]:

- Oculomotor cues—Vergence and accommodation constitute the two oculomotor functions that give rise to corresponding cues. Vergence is the rotation of the two eyes in the opposite direction to fixate on the object and obtain a single fused image. It is mainly driven by the binocular disparity stimulus [18].

Accommodation is the adjustment of the focal length of the crystalline lens in the human eye to focus (accommodate) on a given object and perceive sharp image. The primary stimulus that drives accommodation is the retinal blur [18]. The strength of corresponding contraction or relaxation in the eye muscles that control the focal length of the lens produce the depth information. The oculomotor cues are effective at short distances (typically up to 2 m).

- Binocular disparity—The positional difference (disparity) in the image locations of an object in the left and right retinal projections depends on the depth of the object. Thus, depth information is extracted based on the disparity of matched object points in those projections. Binocular disparity is a primary cue, which is utilized in a wide depth range (typically, from around 10 cm up to 100 m).
- Pictorial cues—Shadows, perspective, occlusion, texture scaling, gradient, etc. constitute pictorial monocular depth cues. HVS relies more on these cues especially at long distances, where other cues (e.g., binocular depth cues and/or motion parallax) cannot provide necessary information.
- Motion parallax (head parallax)—Closer objects appear to move faster than further objects. Motion parallax is a physiological monocular cue that is created by this relative motion of the objects at different depths, when the head is moved. It is especially effective at long distances, e.g., where the accommodation cue is not reliable.

In binocular viewing (stereopsis), the so-called *Panum's area* and *depth of focus* define 3D zones, with respect to the limits of binocular vision and accommodation function, respectively, within which the viewing is considered to be comfortable (i.e., with reduced visual fatigue or discomfort) [15, 17]. For a given position of the eyes, there exists a surface in 3D space called as horopter, for which the corresponding images in the left and right eyes produce zero retinal disparity [15]. Panum's area defines a 3D zone around the horopter that puts a limit for the allowable retinal disparity. The objects within the Panum's area can be fused to a single clear image (without double vision). On the other hand, the depth of focus is related to accommodation function and it defines a 3D zone around the focused depth within which the object points can be perceived sharp enough (in focus) without requiring reaccommodation. The comfort zone can be considered as the intersection of Panum's area and depth of focus [15].

The two oculomotor functions accommodation and vergence usually work in harmony [18]. Thus, the accommodation and vergence functions are actually coupled, i.e., one response evokes the other and vice versa. Such a coupling accelerates both accommodation and vergence, i.e., accommodation is faster in binocular viewing compared to monocular viewing and vergence is faster when also a blur signal (consistent with the disparity signal) is available and utilized [19].

3.2 From Ideal to Real Light Field Display

3D displays aim at reproducing a real 3D scene in a visually indistinguishable way. Thus, an ideal 3D display is expected to recreate all visual cues accurately so that the viewer perceives the 3D image of a scene as close as possible to its reality. An LF display is aimed at providing all the necessary cues (mainly vergence, binocular disparity, motion parallax, and accommodation) with sufficient accuracy by actually reconstructing the LF that includes a complete description of the scene. In other words, an LF display is mainly intended to address the continuous (smooth) motion parallax and accommodation-vergence conflict problem, which constitute the two main deficiencies of the conventional 3D displays such as stereoscopic and multiview displays.

While the stereoscopic displays do not provide motion parallax at all, the motion parallax provided by the multiview displays is usually discontinuous. On the other hand, since such conventional 3D displays mainly rely only on the binocular disparity, they cannot provide (correct) accommodation cue. The eyes of the viewer focus on the display surface, which is the location of the source of the light, while they converge at the depth addressed by the (simulated) disparity cue. The coupling between the vergence and accommodation cue is, thus, broken and the so-called accommodation-vergence conflict occurs [18]. This conflict has been reported to cause potentially serious visual discomfort in prolonged use of such displays [20, 21]. The *Percival's zone of comfort* defines a set of vergence and accommodation responses, which can be achieved without discomfort [19]. Its width is about one-third of the width the *zone of clear single binocular vision*, where the accommodation and vergence are possible without excessive error in either [19]. Figure 11 illustrates these zones in relation with Panum's area and depth of focus.

The design of an LF display is, thus, based on the motivations of providing smooth motion parallax and avoiding (or reducing) the accommodation-vergence conflict [15, 22]. Both are actually dictated by the characteristics of the HVS. The two main system parameters of an LF display are the spatial and angular resolutions of the emitted LF, which are characterized by the corresponding sampling steps Δx and $\Delta\theta$, respectively, as illustrated in Fig. 12. The reference plane represents the spatial sampling plane of the LF which can be either right on the display surface (e.g., in the case of super-multiview display) or separated from it (e.g., in the case of integral imaging). The angular resolution of LF determines the resolution of viewpoints at the observation distance d_v as $\Delta v = d_v \Delta\theta$ [22]. The relation between the eye pupil size W_e and Δv then dictates the motion parallax and accommodation cues. If $W_e \geq \Delta v$, then the motion parallax is continuously perceived. Regardless of this requirement, motion parallax will be also smooth when the reconstructed image is within one pixel disparity range, with respect to reference plane, for adjacent viewpoints, i.e., $\Delta x \geq |z_i| \Delta\theta$ with z_i being the distance of the image from the reference plane [22].

On the other hand, if the so-called super-multiview (SMV) condition is satisfied, i.e., there are two or more rays incident in the eye pupil, the accommodation cue

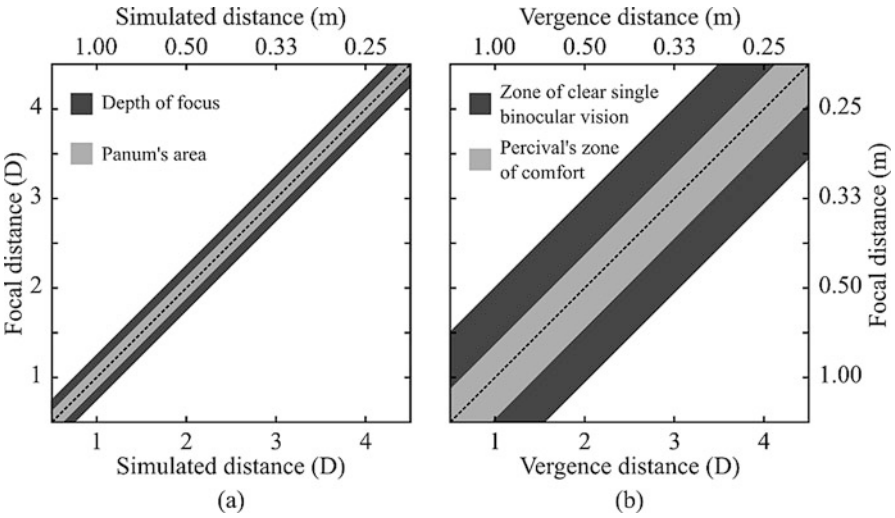


Fig. 11 Illustration of different comfort zones for the HVS with respect to simulated distance-vergence and accommodated distances (adapted from [19]). The points on the dashed diagonal line in (**a**) correspond to real objects at different depths. The points on the dashed diagonal line in (**b**) correspond to real world stimuli

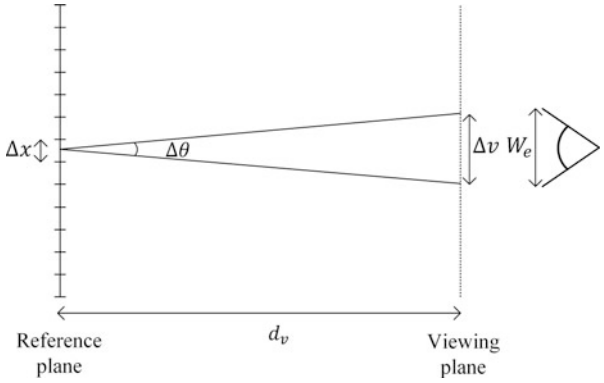


Fig. 12 The relation between the parametrization of a LF display and HVS

is invoked [23, 24], and the eye focuses on the reconstructed image even when it is separated from the reference plane. Therefore, the accommodation-vergence conflict is avoided (or reduced). Please note, however, that creation of the correct accommodation cue depends on several other factors, such as the distance of the reconstructed image from the reference plane [25, 26]. This issue is further addressed in the following sections.

In the following section, integral imaging, super-multiview displays, projection-based displays, tensor displays, and holographic stereograms are discussed as different examples of LF display techniques.

3.3 Overview of Current Light Field (Type) Displays

3.3.1 Integral Imaging

Integral imaging constitutes the oldest LF display technique, as it goes back to 1908, when Lipmann [27] invented it with the original name of integral photography. The idea was to capture and then reconstruct the LF by utilizing a two-dimensional lens array. Figure 13 illustrates the capture and reconstruction stages of integral imaging technique.

In the capture stage, the LF incident on the microlens array plane is spatially sampled by the microlenses. Then, the so-called elemental images behind each microlens record the local angular distribution of the LF. Thus, assuming that there is no cross-talk between the elemental images (this can be satisfied e.g., by putting physical barriers between the elemental images), the space-angle distribution of the LF incident on the microlens array is recorded by the sensor pixels. In the reconstruction stage, the 3D scene can be reconstructed by simply writing the recorded sensor image onto a display. In case the same microlens array is to be used, the pixel pitches of the sensor and display should be same. Otherwise, the system can be scaled. As seen in Fig. 13b, integral imaging actually reconstructs focused points in space by integrating several beams focused by different microlenses. Thus, such sets of beams create continuous angular intensity distributions from the focused points in space. When the viewer moves his/her head within such an LF, he/she will perceive continuous motion parallax. The accommodation cue of integral imaging has been addressed in several studies [24, 25, 28, 29]. As a result of a subjective test presented in [25], 73% of the viewers are reported to actually focus on the reconstructed image. Moreover, the conflict between the vergence and accommodation is relieved in the super-multiview region [28].

One critical problem with direct reconstruction technique shown in Fig. 13 is that the reconstructed images are pseudoscopic, i.e., reversed in depth. In order to provide orthoscopic images with correct depths, one should digitally recalculate the elemental images knowing the capture and display parameters [30]. Another approach is to use virtual image presentation technique proposed by [31], which is illustrated in Fig. 14.

In this technique, the captured elemental images are simply rotated around their centers by 180° and the distance between the microlens array and the display plane is chosen as $l_r = l_c - 2f^2(d_c - f)$. By this way, a virtual image is obtained at $d_r = d_c - f$ from the microlens array plane, where d_c and d_r are the image planes of sensor and display planes during capture and reconstruction, respectively, and f is the focal length of the microlenses [32].

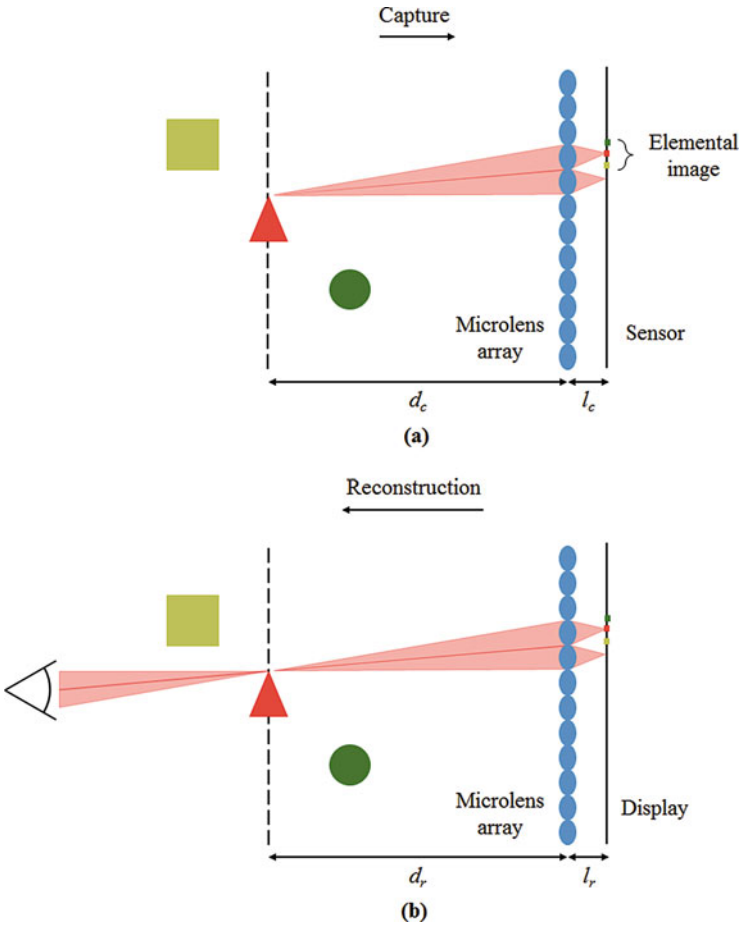


Fig. 13 Capture (**a**) and reconstruction (**b**) of a scene by integral imaging

In both the direct display technique, $l_c = l_r > f$, and the virtual image presentation technique, $l_c > f > l_r$, the scene regions inside the depth of field of the microlenses are sharply reconstructed, however blurred reconstructions are inevitable outside this region. An alternative approach to these "resolution-priority" cases is the "depth-priority" technique, where $l_c = l_r = f$ so that the resolution of the reconstructed images are now dependent on the microlens pitch, i.e., sharp reconstruction as in the resolution-priority case is not possible, but the resolution can be kept in a much larger depth range around the focal plane [22, 33, 34].

For more detailed analysis, advances and recent issues in integral imaging, we refer the reader to [32, 35].

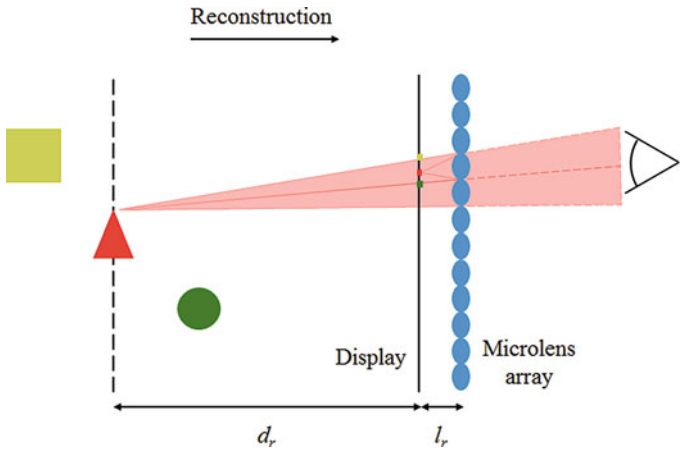


Fig. 14 Virtual image reconstruction technique by integral imaging

3.3.2 Super-Multiview Displays

Multiview displays (MVDs) usually employ a pair of flat-panel display (e.g., LCD) and a horizontal array of optical elements or openings on a surface that refract or direct the light in the horizontal direction, e.g., lenticular sheet [36, 37] or parallax barrier [38, 39]. Thus, unlike integral imaging, they provide only horizontal parallax. The red, green and blue sub-pixels of the flat-panel display create the full color range by emitting light in the corresponding color. The light emitted from each sub-pixel forms a vertical stripe of beam after being directed by the horizontal array of optical elements. An RGB beam triplet (corresponding to an RGB sub-pixel triplet) forms one color component to be perceived at the corresponding viewpoint. The set of such triplets (from different lenticules or slits) forms a parallax image when viewed at a given viewpoint at the intended viewing plane. The set of all those parallax images at different viewpoints constitutes the so-called multiview images. The vertical resolution of the perceived image is the same as the vertical resolution of the display panel, whereas the horizontal resolution is reduced by a factor of number of views, which is the (rounded) total horizontal number of pixels under a single optical element. Thus, there is an uneven resolution loss in the vertical and horizontal dimensions. The slanted lenticular approach proposed in [36] makes this loss more even via sub-pixel multiplexing technique. For instance, for an 18-view display the resolution loss can be chosen to be by a factor 3 in the vertical and 6 in the horizontal direction.

Super-multiview displays (SMVDs) can be seen as advanced types of MVDs that provide a very dense set of views (typically more than 50). In particular, the SMV condition, which requires that there should be at least two rays incident in the eye pupil of the viewer, is what separates a SMVD from MVDs. As shown in Fig. 15, when this constraint is satisfied, the viewer is able to focus on the reconstructed

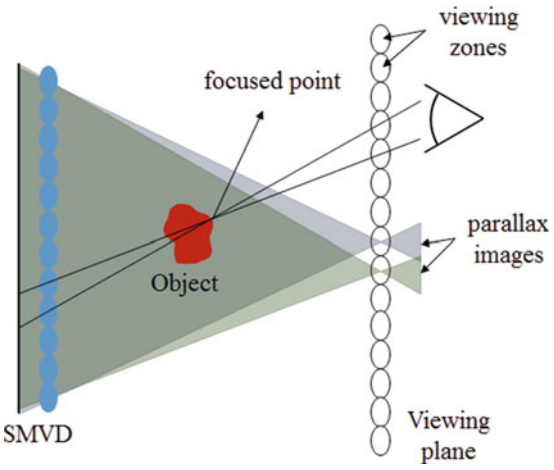


Fig. 15 Reconstruction of focused scene point by SMVD

image separated from the display surface and hence, the accommodation-vergence conflict may be avoided [23, 40]. Furthermore, the viewer experiences smooth motion parallax.

The condition for smooth motion parallax is actually less strict. Even when the distance between viewpoints is larger than the pupil size, smooth motion parallax can be perceived due to cross-talk between views. Both the smoothness of the motion parallax and the accuracy of the accommodation response depend not only on the SMV condition, but also on the crosstalk between the views, the depth range of the scene, etc. For instance, for scene points that are not close enough to the display surface, the accommodation response may not be accurate or the motion parallax may not be smooth, even if the SMV condition is satisfied. The corresponding depth range (around the display) is related to the capacity (bandwidth) of the 3D display device. This issue is analyzed in more detail in Sect. 4. A detailed analysis of motion parallax and accommodation aspects for the lenticular based SMVD is presented in [26, 41] for several application scenarios.

The typical flat-panel display technique used in MVDs has been also demonstrated to be effective for designing SMVD [37, 41]. However, the current available resolutions of flat panel displays are not good enough to deliver both the required number of views and high resolution 3D images. The number of views can be considerably reduced, and thus the resolution can be increased, by utilizing eye tracking algorithms and providing views only around the two eyes of the viewer [37]. Besides this, several other SMVD design techniques have been proposed such as focused light array, multi-projection, time-multiplexing and hybrid systems consisting of both flat-panel and multi-projection systems [42]. The focused light array technique was actually used in the first SMVD design, where a set of laser diodes are focused at the same viewpoint and then they are scanned in two dimensions to cover different viewpoints [42, 43]. The scanning requirement of this technique is removed in multi-projection type of systems [44–46], where an array of

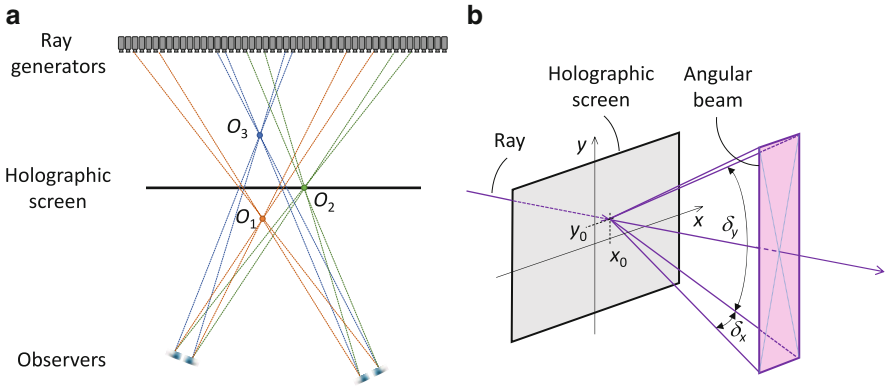


Fig. 16 Principle of operation of a projection-based LF display. (**a**) Forming a point in space (O) by light rays originating at different ray-generators. (**b**) The diffusing property of the holographic screen with a wide spread in vertical and a narrow spread in horizontal direction

projectors are used, in the expense of large space requirement. Time-multiplexing technique has been used to reduce the number of projectors in such multi-projection systems [47]. In the hybrid design, the images of all flat-panel displays are superimposed on the common screen using the projection lens array [42]. The total number of viewpoints is the product of the number of flat-panel systems and the number of viewpoints generated by the flat-panel displays. By using 16 flat-panel displays having 16 views, a SMVD having as high as 256 views has been created [48]. For more detailed information on different SMVD design techniques, the reader is referred to [44].

3.3.3 Projection-Based Displays

A projection-based display that recreates an approximation of the continuous plenoptic function out of a discrete set of rays consists, see Fig. 16a for illustration, of the following two parts [49]: First, a set of projection engines that act as ray generators (discrete sources of light), and second, a holographic screen that is a special optical element that performs the discrete to continuous conversion of light rays. The holographic screen, in its simplified form, for an HPO system can be interpreted as an anisotropic diffusor that converts (diffuses) each ray into an angular beam around the main direction of the ray, having a narrow horizontal angle δ_x and wide vertical angle δ_y, as illustrated in Fig. 16b. This ensures the visibility of a ray from all vertical positions (in the front of the display) but only a narrow horizontal range of positions. The display recreates an object in space by recombining rays from different projection engines depending on the position of the observer and the object itself. This is illustrated for three objects and two observers in Fig. 16a.

The display has no pixel structure since rays originate from different sources and hit the surface of the display on a non-regular grid. The number of rays determines the overall throughput of the display in terms of angular and spatial resolution. For the same number of rays, trade off can be made between spatial and angular details by the mechanical design of the display – as it will be discussed in more detail in Sect. 4.

Having a dense set of rays such displays are capable of maintaining continuous parallax thereby providing (e.g., by dynamic rendering) the correct perspective in e.g., free viewpoint video. This makes them one of the front-runners, among existing LF displays of today, for visualizing immersive 3D content. The drawback is the high hardware complexity of the system and the huge amount of data that the system has to process. This makes it challenging for large setups as well as makes the extension to full parallax very difficult, if not impossible.

3.3.4 Holographic Stereograms

The LF displays discussed above are based on ray reconstruction that is modelled via the ray-based LF paradigm. Holographic stereogram (HS), on the other hand, can be seen as a hybrid approach. It relies on holographic recording and reconstruction principles, nevertheless, it utilizes a set of 2D images as the information source [50–52]. The possibility of using real scene images recorded under white light illumination is essentially what make HSs attractive compared to other (coherent) holographic techniques. Nevertheless, their computational simplicity compared to other coherent techniques is also critical in computer-generated holography (CGH), especially for dynamic displays [53]. Display technologies utilizing HS technique consist of static displays such as holographic prints [54, 55] and dynamic displays which can be implemented e.g., via spatial light modulators (SLMs) [53] or rewritable holographic materials [56, 57].

For optically recorded HSs, a set of 2D (multiview) images are projected one-by-one on the hologram surface and the interference pattern of the projected images and a reference beam is recorded hogel-by-hogel, where hogel refers to a spatial holographic element on the hologram surface [58]. In the case of computer-generated HSs, the corresponding fringe patterns to be written in a hogel is calculated via physically simulating the interference process [50, 58]. More specifically, a hogel on the hologram surface contains the information about the 2D image that would be seen from a very narrow window (as of the same size of the hogel) at the location of the hogel [59]. This information is coded in the form of a holographic fringe pattern, which has varying spatial frequency components. The amplitudes of those components control the intensities and the spatial frequencies control the directions of the rays to be sent to a particular direction. Each such ray corresponds to one of the pixels of the corresponding 2D image that was utilized during recording or computation at the given hogel location. The collection of all those 2D images corresponds to a discrete LF representation (multiview images) defined on the hologram surface. The necessary data for obtaining this LF can be

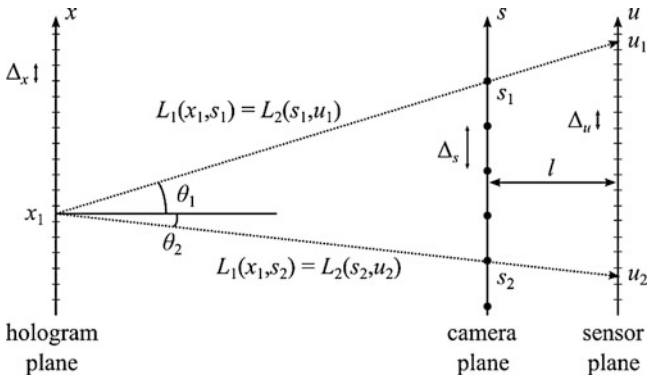


Fig. 17 The relation between the captured LF and HS parameters (adapted from [60])

collected by a scanning camera rig (or via computer graphics rendering for synthetic scenes) either on the hologram plane or on some other plane further away from the hologram surface. As the scene is usually confined in a region around the hologram surface, separation of capture plane from the hologram surface is usually preferred for practical reasons. In this case, a remapping from captured images is necessary to acquire the set of ray intensities to be utilized for a given hogel. The setup shown in Fig. 17 illustrates this case (for 2D space), where different directional ray components for a given hogel correspond to pixels from different view images captured on the camera plane.

The wavefield (amplitude) expression for the HS shown in Fig. 17 is given as [59, 60]

$$O_{HS}(x) = \sum\nolimits_m \text{rect}\left(\frac{x - m\Delta_x}{\Delta_x}\right) \sum\nolimits_i \sqrt{L_1\,[m,i]}\exp\left(j2\pi f_x^{mi} x\right), \tag{4}$$

where f_x^{mi} is the spatial frequency component on the x-axis, for hogel m and ray i. The spatial frequency is related to the direction of the corresponding ray via the grating equation [61]

$$f_x^{mi} = \frac{\sin\left(\theta_x^{mi}\right) - \sin\left(\theta_{\text{ref}}\right)}{\lambda} \tag{5}$$

where λ is the wavelength (color) of the light and θ_{ref} is the incidence angle of the planar reference beam $R(x)$ to be used in calculating the interference pattern, i.e., $|O_{HS}(x) + R(x)|^2 = |O_{HS}(x)|^2 + |R(x)|^2 + 2\,Re\,\{O_{HS}(x)R(x)^*\}$. Please note that the relevant holographic information is contained in the third term, which is called as the bipolar intensity [58]. Thus, in CGH, this intensity pattern is usually treated to be the actual HS. When illuminated with the same reference beam, the hogels of the HS reconstruct the recorded content as planar wavefront segments, propagating towards various directions with their intensities defined by the corresponding LF

samples. As seen in (4), the fringe pattern in each hogel can be obtained by inverse Fourier transform of the image segments [59]. When utilizing fast Fourier transform implementations, care need to be taken for accurate calculation. As the set of discretized spatial frequency values (f_x^{mi}, $i = 1, \ldots, N$) is fixed, the corresponding discrete LF samples $L_1[m, i]$ are to be appropriately calculated from the captured LF samples, e.g., via resampling from the captured LF $L_2[i, j]$ [60].

Assuming properly chosen hologram parameters, the HS is able to provide continuous motion parallax, and maximized perceived resolution with respect to the HVS [58]. In their original form described above, HSs can provide limited accommodation response in a shallow depth range around the hologram. This is one of their main drawbacks compared to (coherent) holographic display methods. However, correct accommodation cues can be provided in a much larger depth range via techniques that modulate the planar wavefront segments based on the depth information of the captured ray (i.e., the information of the corresponding point source in the scene) [62, 63].

3.3.5 Tensor Displays

The underlying idea of tensor displays is that the LF ray directionality can be manipulated through a (ratter small) number of layers of light modulators with varying transmittances. The joint multiplicative effect is a modulated light field with desired characteristics. The idea can be traced back to the first proposal to create directional views through parallax barriers [64]. In that early work, the directionality is maintained by a mask of pinholes installed in front of a 2D display, which blocks some rays and allows other rays coming from the 2D image source to go through and to be seen from a specific perspective. Thus, different groups of pixels are visible from different perspectives. This is somehow dual to using lenselets to direct the light, as in the case of auto-stereoscopic displays [36, 38, 39]. Parallax barriers have the apparent disadvantage of blocking some of the light rays thus reducing the image brightness. Furthermore, the mask reduces the spatial resolution per view as the light source behind it multiplexes all perspective views. The approach has been further extended toward content-adaptive parallax barriers, where a few layers of barriers have been multiplied in order to increase the degrees of freedom in manipulating different directions and the pinholes have been replaced by varying-transmittance elements (e.g. pixels of an LCD display), as illustrated in Fig. 18a, [65]. Very fast panels can be used to additionally introduce time multiplexing. That is, layer patterns are changed with a high rate above the rate at which the eye perceives temporal flickering, and the time-multiplexed images are perceptually averaged to perceive an LF with adapted directionality of rays (Fig. 18c). Eventually, directional backlighting has been proposed to further extend the light field generation fidelity [66] (Fig. 18d).

The elegancy of the approach comes from the fact that stacking panels with varying-transmittance pixels can be mathematically modelled by the multi-linear

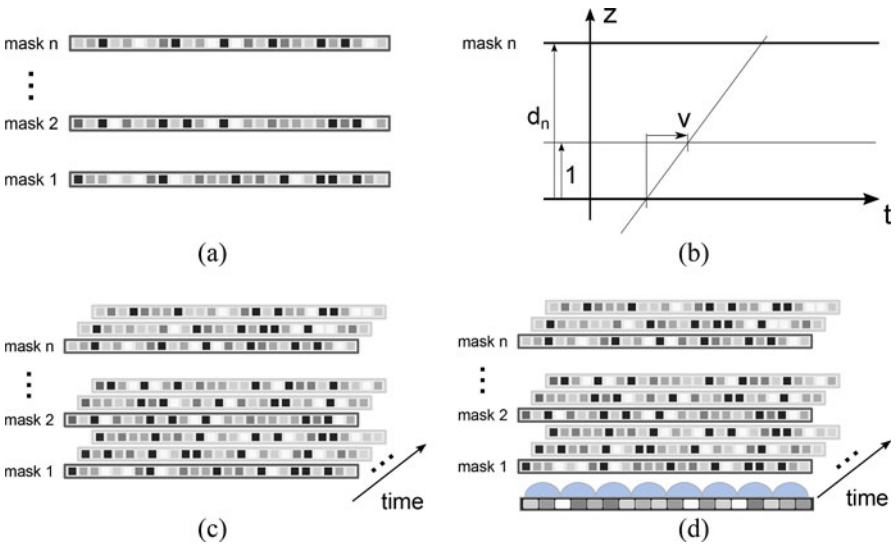


Fig. 18 Tensor displays. (**a**) n layers (masks); (**b**) LF parameterization; (**c**) combining layers and time multiplexing; (**d**) adding directional backlight. Adapted from [66]

algebra tools, which leads to effective solutions using tensors, thus giving the name of this class of LF displays: *tensor displays* [66].

Let's consider the general case of combining N stacked layers visualizing M temporal frames each, as given in Fig. 18c. The n-th layer is at distance d_n from the light field generating plane and the light field is parameterized with respect to some reference plane at unit distance (Fig. 18b, see also Fig. 2 for reference). Each layer has a corresponding transmittance $f^{(n)}$. Thus, the generated light field is formed by multiplying the layer images and averaging the temporal frames

$$\tilde{L}(t,v) = \frac{1}{M}\sum\nolimits_{m=1}^{M}\prod\nolimits_{n=1}^{N} f_m^{(n)}(t + d_n v) \tag{6}$$

Dropping the time multiplexing will model the simpler case of layered LF displays [67]. The case of directional backlight can be achieved by adding lenticular optics at the light source place (Fig. 18d) and modelled by adding the corresponding terms $b_m(t,v)$ [66]

$$\tilde{L}(t,v) = \frac{1}{M}\sum\nolimits_{m=1}^{M} b_m(t,v)\prod\nolimits_{n=1}^{N} f_m^{(n)}(t + d_n v) \tag{7}$$

Given a desired LF $L(t,v)$, one has to find the display transmittances $f_m^{(n)}$ and possibly the directed backlights $b_m(t,v)$. This can be solved by a least square optimization procedure, constrained by the requirement for non-negativity of all pixel values being optimized. For solving it, the representation of the layered

planes in the form of tensors comes into play. Taking into account the discrete form of pixel layers, the corresponding transmittances can be organized in vectors $\mathbf{f}^{(1)}, \mathbf{f}^{(2)}, \ldots \mathbf{f}^{(N)}$, then the temporal frames are organized in matrices $\mathbf{F}^{(n)} = \left[\mathbf{f}_1^{(n)}, \mathbf{f}_2^{(n)}, \ldots \mathbf{f}_M^{(n)}\right]$, and the combination of all transmittances takes the form of an N-th order, rank-M tensor $[\![\mathbf{F}^{(1)}, \mathbf{F}^{(2)}, \ldots \mathbf{F}^{(N)}]\!]$. The generated light field in (6) can be expressed in the explicit tensor form as

$$\widetilde{\mathcal{L}} = \mathcal{W} \circledast \left\{ \frac{1}{M} \sum_{m=1}^{M} \mathbf{f}_m^{(1)} \circ \mathbf{f}_m^{(2)} \circ \ldots \mathbf{f}_m^{(N)} \right\} = \mathcal{W} \circledast \left[\!\!\left[\mathbf{F}^{(1)}, \mathbf{F}^{(2)}, \ldots \mathbf{F}^{(N)} \right]\!\!\right] \tag{8}$$

where $\circ$ denotes the vector outer product, $\mathcal{W}$ is a binary weight tensor picking up the valid rays only through element-wise multiplication denoted by $\circledast$ [66]. In practice, one has to represent the desired LF $L(t, v)$ in a tensor form $\mathcal{L}$ by parametrizing all rays by their intersections with all layers and then to solve the minimization problem

$$\widetilde{\mathcal{L}} = \mathcal{W} \circledast \arg \min_{\{\mathbf{F}^{(n)}\}} \left\| \mathcal{L} - \widetilde{\mathcal{L}} \right\|, \quad \text{for } \mathbf{F}^{(n)} \in [0, 1] \tag{9}$$

The solution makes use of non-negative tensor factorization. For more details the reader is referred to [66].

As discussed in [18], the tensor approach to LF displays is a form of compressive display as it manages to factorize the LF into temporal and multiplicative light-modulating layers. A frequency analysis has revealed that the multiplicative, essentially non-linear manner of combining layers yields extended depth of field and field of view. Brightness can be also gracefully maintained. These benefits come for the price of increased computational complexity for solving the optimization problem (9) through non-negative tensor optimization. Aligning multiple display modules requires precise calibration and using multiple temporal frames requires high frame rate hardware. Artifacts, such as Moiré, color crosstalk, and interreflections can be expected though optical engineering solutions for those do exist [66]. The tensor concept has been investigated also for the case of near-eye displays [68] and projection based displays [69]. Tensor displays are potentially capable of supporting focus cues. High angular resolution (i.e., dense viewing zones) can be achieved either by increasing the pixel resolution of the display panels or by increasing the distance between layers [18]. However, in these cases the diffraction limit starts to play a significant role and has to be taken into account.

4 Display Specific Light Field Analysis

An LF display is a visualization system that strives to reproduce (approximate), from a dense (though finite) set of light rays (samples), the underlying continuous plenoptic function describing the scene that is visualized. This makes the display, in

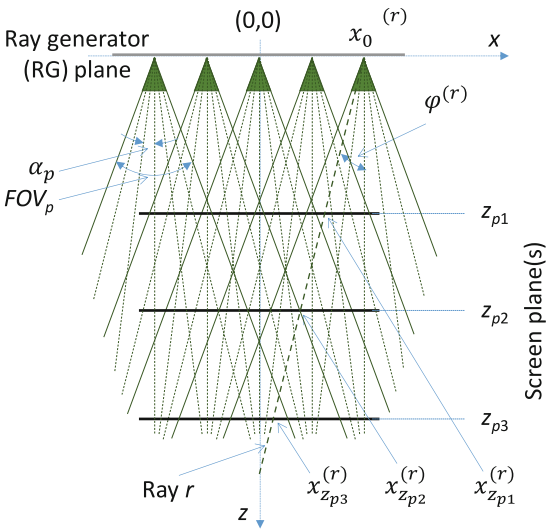


Fig. 19 Ray propagation in a light field display

its essence, a multidimensional sampling system with various means of generating rays (e.g., projectors, display panel) and possible implementations of the discrete-to-continuous (D/C) converter (e.g., directional diffusor, lenticular sheet) [36, 37, 49].

Based on the LF analysis presented so far, in this section it will be shown how the discrete nature of a typical LF display influences the LF reconstruction and more importantly how the multidimensional sampling theory can be applied to optimize the display setup (maximize the visual performance given a limited number of rays) and how to capture/prepare/pre-process content for a given display that will maximally utilize its capabilities. This will be done by using projection-based LF displays [49, 70], where the ray generators act as discrete sources of light rays and the holographic screen is the D/C converter that converts the set of samples (rays) into its continuous representation that is observed by a viewer (see Fig. 16). The presented analysis can be extended / to other types of LF displays as well.

4.1 Display-related Ray Propagation

A model of a typical LF display under consideration is presented in Fig. 19. Each of the N_p projection engines, uniformly distributed on the ray generators plane (p—plane) with the distance between two adjacent engines being x_p, generates N_x rays over its field of view FOV_p for a total of N_pN_x rays generated by the display. Those rays propagate along the z direction and at a certain distance hit the screen plane (s—plane) parallel to the ray generators plane. The screen of the display is where rays recombine to reconstruct the desired continuous LF function. Although not true for large angles, we assume here that rays from one ray generator hit the screen plane at

equidistant points and that the angular distribution of the rays is also uniform, that is, $\alpha_p = FOV_p/N_x$. Rays are parameterized by their spatial position and direction (x, φ) and thus represented as samples in the corresponding ray space with position typically expressed in mm and angle in degree. At the screen plane, those can be recalculated in the terms of equivalent spatial resolution (e.g., number of pixels per mm or per screen size) and its angular resolution (e.g., number of rays per degree or FOV of the display FOV_{disp}).

Following the discussion of Sect. 2.3, the propagation (position) of a ray r, originating at one of the ray generators at distance z from its origin, is given as

$$\begin{bmatrix} x_z^{(r)} \\ \varphi_z^{(r)} \end{bmatrix} = \begin{bmatrix} x_0^{(r)} + z \tan\left(\varphi^{(r)}\right) \\ \varphi^{(r)} \end{bmatrix}. \tag{10}$$

Here, $x_0^{(r)}$ is the position of the ray on the ray generators plane and $\varphi^{(r)}$ is the direction of the ray. As seen from the equation, as well as Fig. 19, the direction of the ray does not change with distance z. However, depending on the distance of the screen (several cases shown with black lines in the figure) the ray crosses the screen at different horizontal positions. This means that for different distances z, a different set of rays will contribute to forming of an equivalent multiview pixel originating at a given point on the screen. As it will be seen later, as a consequence, on the screen plane the uniform distribution of rays from the ray generators plane is lost.

For a fixed z, each ray is considered as a sample in the 2D (x, φ) space. The sampling pattern formed by those samples changes with the distance z and is illustrated by means of an example in Fig. 20. Two things can be observed. First, the propagation of rays is equivalent to shearing the ray space in x direction (on the figure, samples of one ray generator are marked with blue), and second, at every distance the sampling pattern might not be uniform, but it always will be regular.

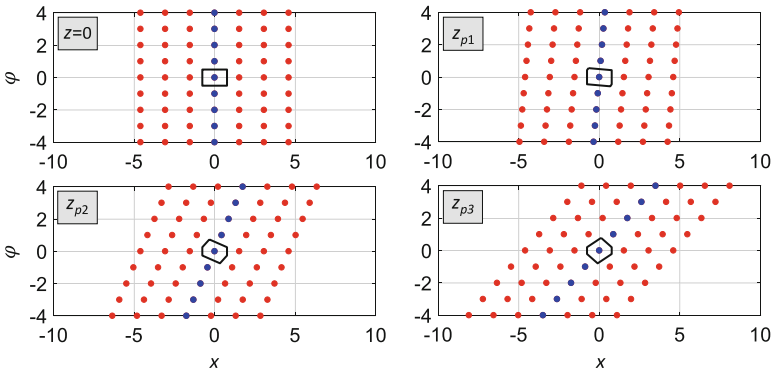


Fig. 20 Ray space spatial sampling patterns at different distances from the ray generators plane with $0 < z_{p1} < z_{p2} < z_{p3}$

This regularity will ensure a uniform performance of the display over the overall screen and simplify the follow-up analysis.

4.2 Display Bandwidth

As a user of an LF display, one is interested in its visualization capabilities, that is, what level of details one can visualize (see) on a given display? In order to answer this question, one has to convert the display's sampling pattern, which is determined by the design configuration of the display, into a more meaningful (user friendly) representation. One such representation is based around the concept of the display bandwidth.

For analyzing a regular sampling pattern, one can utilize the multi-dimensional sampling theory [71, 72] that can be summarized for the case under consideration as follows [6, 8]:

1. Any regular 2D pattern can be described through a sampling lattice Λ with the elements of the lattice being $\Lambda(\boldsymbol{V}) = \{n_1\boldsymbol{v}_1 + n_2\boldsymbol{v}_2 \mid n_1, n_2 \in \mathfrak{Z}\}$. Here, $\boldsymbol{v}_k = \begin{bmatrix} v_k^{(x)} & v_k^{(\varphi)} \end{bmatrix}^{\mathrm{T}}$ for $k = 1, 2$ are two linearly independent vectors typically referred to as basis vectors and $^{\mathrm{T}}$ is the transpose operator.
2. The associated sampling matrix $\boldsymbol{V}(\boldsymbol{v}_1, \boldsymbol{v}_2) = [\boldsymbol{v}_1 \quad \boldsymbol{v}_2] = \begin{bmatrix} v_1^{(x)} & v_2^{(x)} \\ v_1^{(\varphi)} & v_2^{(\varphi)} \end{bmatrix}$ is not unique. There are different vectors that can be associated with the same sampling pattern as shown in Fig. 21a, that is, $\Lambda(\boldsymbol{V}) = \Lambda(\boldsymbol{EV})$ for $\boldsymbol{E}$ being any integer matrix with $|\det \boldsymbol{E}| = 1$
3. In practice, the sampling matrix $\tilde{\boldsymbol{V}}$ with shortest basis vectors is preferred, that is, $\left\|\tilde{\boldsymbol{v}}_1\right\| + \left\|\tilde{\boldsymbol{v}}_2\right\| = \min\left(\|\boldsymbol{v}_1\| + \|\boldsymbol{v}_2\|\right)$. Finding the smallest basis vectors for a given lattice is known as the lattice basis reduction problem and can be done as discussed in [73].
4. A unit cell P, defined for a lattice Λ, is a set in $\mathbb{R}^2$ such that the union of all cells centred on each lattice sample covers the whole sampling space without overlapping or leaving empty space. Similar to the basis vectors, the unit cell is not unique, as illustrated in Fig. 21b.
5. The most compact unit cell (where all points in the cell are closer, based on the Euclidian distance, to the cell's sample than any other sample), is the Voronoi cell (also known as Wigner-Seitz cell) [74, 75].
6. Thinking in terms of reconstruction of the underlying bandlimited continuous function described by the lattice Λ, the periodicity and the baseband frequency support are defined through the reciprocal lattice $\Lambda^*(\boldsymbol{V}) = \Lambda((\boldsymbol{V}^T)^{-1})$ as discussed in [6, 71].
7. There are many different unit cells for a given lattice Λ^*. Consequently, each of them describes a bandlimited function that can be represented with

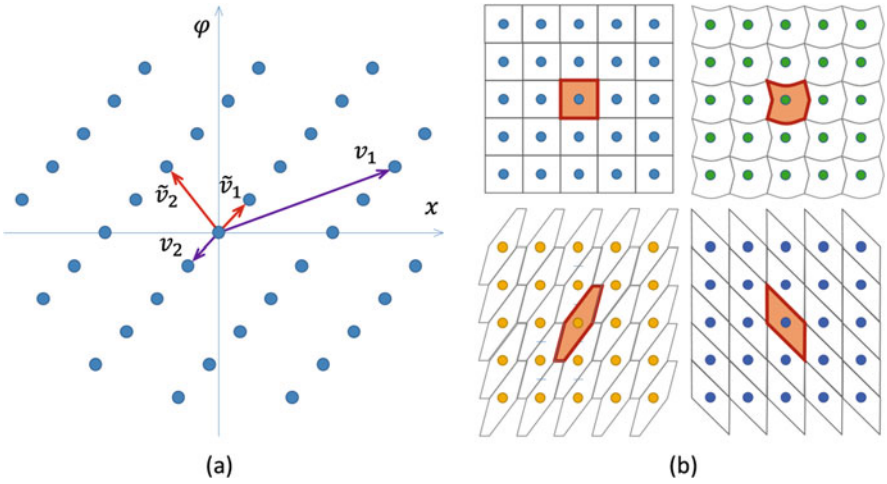


Fig. 21 (**a**) Sampling pattern and two possible sets of basis vectors. (**b**) Some possible unit cells P (shaded area) for a given lattice Λ (points)

(reconstructed from) the lattice samples. Out of all possible ones, in practice, the most interesting one is again the Voronoi cell, denoted in this chapter as P^*, since it treats equally the spatial and angular direction in ray space representation (this is beneficial from the HVS viewpoint) and represents, bandwidth-wise, the 'most low-pass' characteristic (support) provided by the pattern (this typically matches the possible physically implementable D/C converters [49]).

8. This Voronoi cell in the frequency domain is also referred to as the display passband since it specifies which spatio-angular frequencies the display is capable of reconstructing.

In summary, by performing a frequency domain analysis of a typical LF display it is possible to determine the throughput of the display in terms of its spatial and angular resolution, which in turn is determined by (from) the Voronoi cell of the sampling pattern in the Frequency domain. For this, one needs to know display setup, that is, enumerated rays at the ray generators in terms of position and angle and the distance between the ray generators and the screen. Alternatively, if one does not have access to display specifications, for estimating the display passband, one can also use the measurement based techniques as described in [76, 77]. The throughput of the display can be then expressed in terms of its spatio-angular bandwidth, also referred to as the display bandwidth. The display bandwidth enables one to calculate the optimal amount of data that has to be captured and sent to the display to maximally utilize its visual capability. Moreover, it gives the user a good idea on what to expect from the display in terms of visual quality.

4.3 Display-Camera Setup and Optimization

Based on the discussion presented in the previous two sections, one can estimate, from the display configuration, the display bandwidth and the corresponding optimal camera configuration for capturing the content or designing optimal filters for adapting any content to the display. One can also, based on given (desired) display bandwidth estimate the optimal display parameters (configuration) that would result with such bandwidth. The corresponding display-camera setup, with all adjustable parameters, covering those two cases, is illustrated in Fig. 22 and will be discussed in the following two sections.

Notation-wise, tilde ($\sim$), hat ($\wedge$) and bar ($\bar{}$), are used for denoting the parameters after the lattice basis reduction operation, estimated parameters, and optimized parameters, respectively.

4.3.1 Light Field Display Setup Optimization

In the ray space representation, the optimal sampling pattern on the screen plane can be defined by the following sampling matrix:

$$\boldsymbol{V}\,(x_s, \alpha_s) = \begin{bmatrix} x_s & 0 \\ 0 & \alpha_s \end{bmatrix}. \tag{11}$$

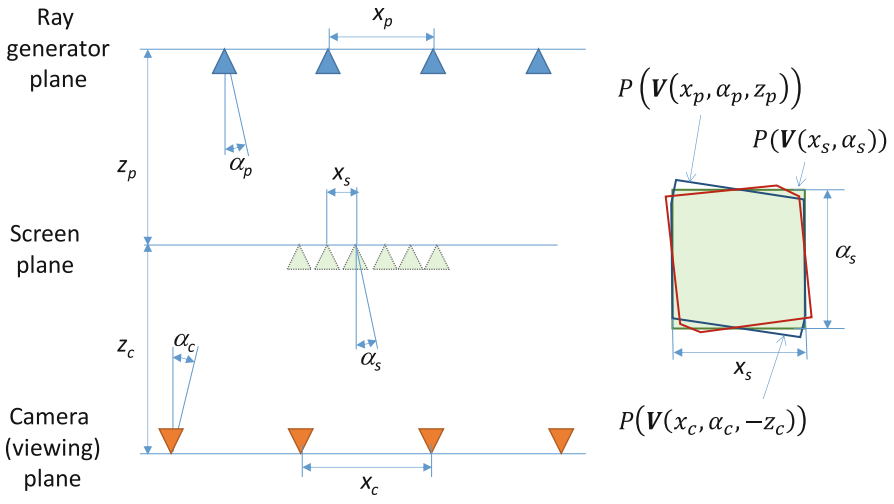


Fig. 22 Light field display–camera setup together with notations for expected sampling patterns. Subscripts p, s, and c are used to denote the parameters related to the ray generator, screen, and camera/viewer plane, respectively

The reason for this being optimal is two-fold. First, from the perspective of human visual system, the spatial and angular direction should be treated in a similar manner. Second, a diffusor (D/C converter) is much easier to implement for such configuration—this effectively is a separable rectangular (in practice typically gauss-shaped) low-pass reconstruction element.

For such desired sampling grid on the screen plane defined by (x_s, α_s), the LF display optimization problem is to determine optimal parameters of the ray generators (x_p,α_p) and the distance between the ray generator plane and the screen plane z_p for which the sampling pattern generated by ray generators $\boldsymbol{V}(x_p, \alpha_p)$ mapped to the screen plane

$$\boldsymbol{V}\left(x_p, \alpha_p, z_p\right) = \begin{bmatrix} x_p & z_p \tan \alpha_p \\ 0 & \alpha_p \end{bmatrix} \tag{12}$$

will match the desired one $\boldsymbol{V}(x_s, \alpha_s)$, that is, to minimize δ_p given as

$$\delta_p = \left\| \widetilde{\boldsymbol{V}}\left(x_p, \alpha_p, z_p\right) - \boldsymbol{V}\left(x_s, \alpha_s\right) \right\|. \tag{13}$$

Here, $\widetilde{\boldsymbol{V}}\left(x_p, \alpha_p, z_p\right)$ is the lattice basis reduced sampling matrix of $\boldsymbol{V}(x_p, \alpha_p, z_p)$. The obtained solution will ensure that the difference between unit cells generated by the ray generators and the desired one $\|P(\boldsymbol{V}(x_p, \alpha_p, z_p)) - P(\boldsymbol{V}(x_s, \alpha_s))\|$ is small, and consequently, the grids described by sampling lattices $\Lambda(\boldsymbol{V}(x_p, \alpha_p, z_p))$ and $\Lambda(\boldsymbol{V}(x_s, \alpha_s))$ match.

Although there are only three unknowns, the optimisation problem is highly non-linear with a lot of local optima. Following the analysis in [8], it can be shown that a good initial solution can be obtained by fixing one of the unknowns and estimating the other two using the following expressions:

$$\widehat{z}_p = \frac{x_s}{\tan \alpha_p} \quad \Longleftrightarrow \quad \widehat{\alpha}_p = \tan^{-1} \frac{x_s}{z_p} \tag{14}$$

$$\widehat{x}_p = x_s \frac{\alpha_s}{\alpha_p} \quad \Longleftrightarrow \quad \widehat{\alpha}_p = \alpha_s \frac{x_s}{x_p}. \tag{15}$$

Moreover, a good selection for α_p is

$$\widehat{\alpha}_p \approx \alpha_s / L \ \text{ for } L \in \mathbb{N}. \tag{16}$$

The optimal set of parameters $\left(\overline{x}_p, \overline{\alpha}_p, \overline{z}_p\right)$ can be found by refining the result using iterative search/general purpose optimization in range $\widehat{x}_p \pm x_s/2$.

Example: For illustration purpose the optimization will be demonstrated for a display with desired spatial and angular resolution at the screen plane with, $x_s = 1$ mm and $\alpha_s = 1°$. Following (16), the angular resolution is selected as

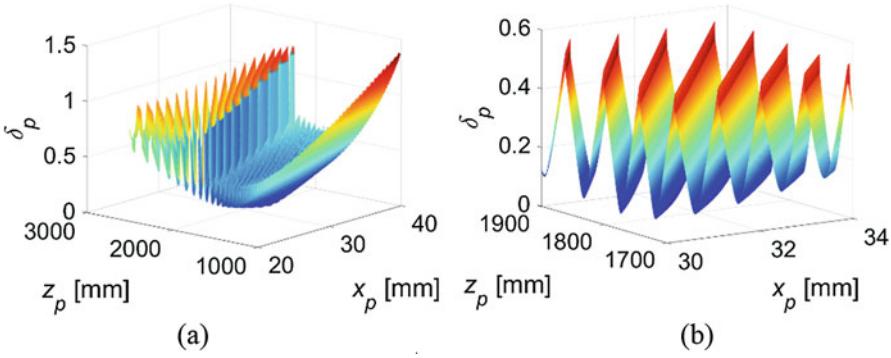


Fig. 23 Display optimization example: optimization error in the case of $x_s = 1$ mm, $\alpha_s = 1°$, and $\alpha_p = 0.0391°$. (**b**) is zoomed in version of (**a**) around the global minima

$\alpha_p = 0.03125^\circ = 60^\circ/1920px$. For those parameters, the matching error δ_p on the screen plane is calculated for various values of x_p (20 mm $\leq x_p \leq$ 40 mm) and z_p (1000 mm $\leq z_p \leq$ 2500 mm) and shown in Fig. 23. As seen on the figure, due to the non-convexity of the optimization problem, direct optimization will not find the minimum, that can be read from the curves as $(\overline{x}_p, \overline{z}_p) =$ (32.00 mm, 1832.51 mm). By applying the two-step optimization proposed above, one gets an estimate in the first step $(\widehat{x}_p, \widehat{z}_p) =$ (31.98 mm, 1833.00 mm) (see (14) and (15)) and after performing single gradient-based optimization from this estimate, ends up with the aforementioned values $(\overline{x}_p, \overline{z}_p)$ corresponding to the minimum. The values can be found in a fraction of a second instead of 10–15 min needed for the grid search approach. For comparison, the estimated, optimized, and desired Voronoi cells at the screen plane are shown in Fig. 24. As it can be seen, the match with the desired $P(V(x_s, \alpha_s))$ is almost perfect for the estimated and the optimized solution.

The importance of a proper selection for α_p is illustrated in Fig. 25. As seen in the figure, for a good match in the example under consideration, α_p has to be selected small enough, $\leq$0.01.

4.3.2 Camera Setup Optimization

In comparison to the display that is a band-limited device, a 3D scene (except a very simple one) is not [13]. This makes the data capture (processing) for visualization of a 3D scene on a display a two-fold problem. First, the scene must be recorded without (noticeable) aliasing, and second, the captured data has to be 'limited' to the reproduction capability of the display, that is, as discussed earlier, defined by its bandwidth.

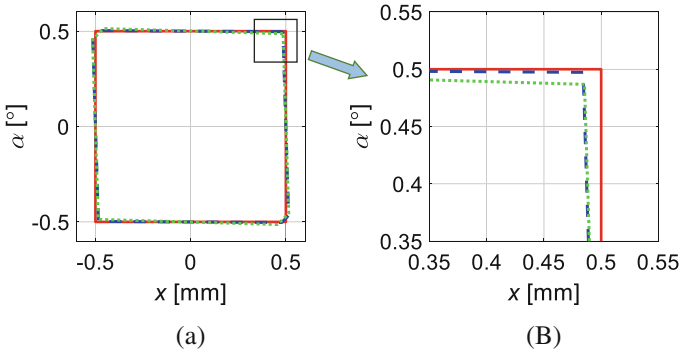


Fig. 24 Unit cells at screen plane for the optimized display setup solution for $x_s = 1$ mm, $\alpha_s = 1°$, and $\alpha_p = 0.0391°$. $P\left(\boldsymbol{V}\left(\overline{x}_p, \overline{\alpha}_p, \overline{z}_p\right)\right)$ dashed/blue, $P(\boldsymbol{V}(x_s, \alpha_s))$ solid/red, and $P\left(\boldsymbol{V}\left(\widehat{x}_p, \widehat{\alpha}_p, \widehat{z}_p\right)\right)$ green/dot. (**b**) is a zoomed in version of part of (**a**)

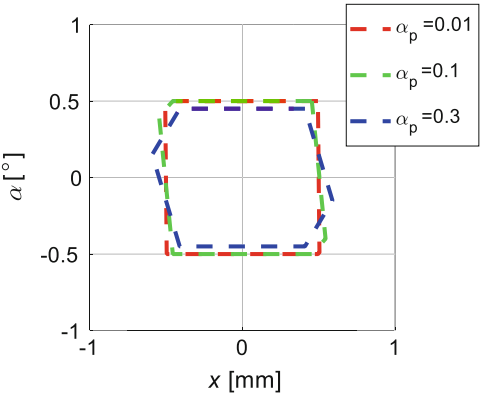


Fig. 25 Normalized ray-generator unit cells at the screen plane, $P\left(\boldsymbol{V}\left(\overline{x}_p, \overline{\alpha}_p, \overline{z}_p\right)\right)$ for various values of α_p

Similar to the discussion in the previous section, the optimal solution for matching the bandwidths of a capture system and the display is obtained by matching the sampling patterns of the display and cameras at the screen plane. This can be formulated as the following optimization problem (see also Fig. 22): For given display specifications described by (x_p, α_p, z_p) find (x_c, α_c, z_c) that minimizes

$$\delta_c = \left\| \widetilde{\boldsymbol{V}}\left(x_p, \alpha_p, z_p\right) - \widetilde{\boldsymbol{V}}\left(x_c, \alpha_c, -z_c\right) \right\|, \tag{17}$$

where $\widetilde{\boldsymbol{V}}\left(x_p, \alpha_p, z_p\right)$ and $\widetilde{\boldsymbol{V}}\left(x_c, \alpha_c, -z_c\right)$ are the lattice basis reduced sampling matrices of the ray generators and cameras mapped to the screen plane, respectively. The matching is done on the screen plane since this is the place where the D/C conversion happens. The problem can be solved by iterative optimization as described in Sect. 4.3.1. The optimized camera parameters are denoted as $(\overline{x}_c, \overline{\alpha}_c, \overline{z}_c)$.

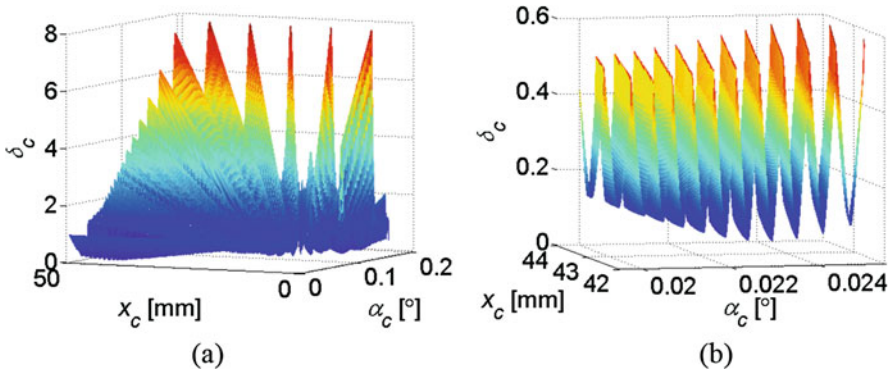


Fig. 26 Camera optimization example based on grid search for optimal $P\left(\boldsymbol{V}\left(\overline{x}_p, \overline{\alpha}_p\right)\right)$ optimized for $x_s = 1$ mm, $\alpha_s = 1^\circ$ and $\alpha_p = 0.0391^\circ$ with $z_c = 2500$. (**b**) is zoomed in version of (**a**) around the global minima

An important thing to point out is that this camera setup is optimal from the point of view of the display, that is, the scene can be captured with this setup only if it has the same or smaller bandwidth than the display. Otherwise, proper anti-aliasing has to be applied during the capture stage or the scene has to be captured with a higher density of cameras, pre-filtered and then decimated to the desired setup.

Example: Continuing the example from the previous section, for the obtained optimized display setup defined by $P\left(\boldsymbol{V}\left(\overline{x}_p, \overline{\alpha}_p, \overline{z}_p\right)\right)$ that is an approximation of the desired one $P(\boldsymbol{V}(x_s, \alpha_s))$, one can evaluate the optimal camera/viewer setup, that is, the optimal parameters (α_c, x_c, z_c) that would support the display bandwidth in the best possible way. After fixing the screen to viewer distance $z_c = 2500$ mm and performing a grid search, the result of the optimization is shown in Fig. 26 with the dominant minimum being at $(\overline{x}_c, \overline{\alpha}_c) = \left(43.80 \text{ mm}, 0.0544^\circ\right)$. As seen in Fig. 27, the obtained passband matches well with the one obtainable by the optimized ray-generator setup. Similar to the ray generator optimization, the algorithm can be made more efficient by performing the grid search only in the vicinity of a good initial estimation that can be obtained by assuming ideal unit cell at the screen plane, that is (x_s, α_s) and then performing the estimation using expressions similar to (14), (15), and (16) with assumption that $\widehat{z}_p, \widehat{x}_p, \widehat{\alpha}_p$ corresponds to $\widehat{z}_c, \widehat{x}_c, \widehat{\alpha}_c$.

The sampling pattern in the spatial domain can be converted to the frequency domain, see Sect. 4.2. By shearing the frequency domain unit cell belonging to the optimized display pattern from the screen plane to the camera plane, we obtain the bandwidth of the display—shown in blue in Fig. 28. As discussed before, one should sample the scene with wide enough bandwidth to avoid aliasing, then pre-filter and then downsample.

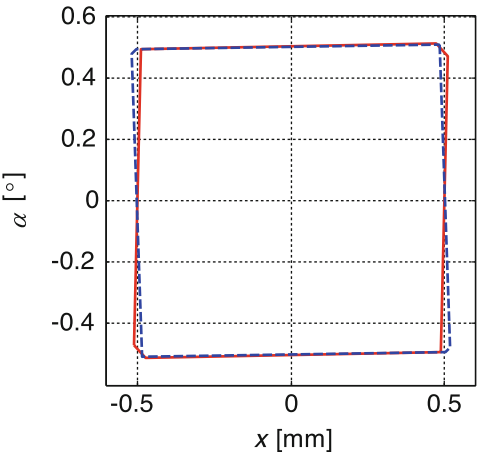


Fig. 27 Unit cells at screen plane for the optimized display and camera setup solution for $x_s = 1$ mm, $\alpha_s = 1^\circ$, $\alpha_p = 0.0391^\circ$ and $z_c = 2500$. $P\left(V\left(\overline{x}_p, \overline{\alpha}_p, \overline{z}_p\right)\right)$ dashed/blue and $P\left(V\left(\overline{x}_c, \overline{\alpha}_c, -\overline{z}_c\right)\right)$ solid/red

5 Reconstruction of Densely Sampled Light Field

This section discusses how one can capture and generate content, which is suitable for a wide range of light field displays. Our conceptual LF representation is the DSLF, as defined in Sect. 2.7. In DSLF, we require that the disparity between corresponding points in neighboring views is 1 pixel at most. Having such representation at hand, one can interpolate rays at arbitrary positions by simple (quadri) linear interpolation and the synthesized novel views are free of ghosting artifacts [78]. Thus, DSLF is instrumental in many applications, where an arbitrary set of rays is required. Beside LF displays creating continuous parallax, the list of such applications includes refocused image generation [79], dense depth estimation [80], object segmentation [81], novel view generation for FVT [82], and holographic stereography [83].

5.1 Plenoptic Modelling, Depth Layering and Rendering

A DSLF is captured by imposing the required distance between neighboring camera positions based on the minimal scene depth ($z_{\min}$) and the camera resolution [10]. The latter should be high enough to capture the desired spatial details. With reference to Fig. 3, consider cameras with focal distance f, having a horizontal sampling rate Δv satisfying the Nyquist sampling criterion for scene's highest texture frequency. The required sampling rate Δt along the camera axis t is

$$\Delta t \leq \frac{z_{\min}}{f} \Delta v. \tag{18}$$

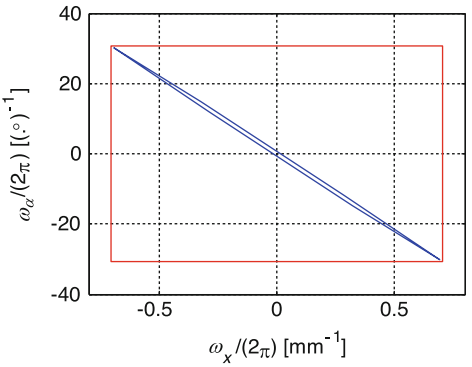


Fig. 28 Unit cells (bandwidths) at the camera (viewer) plane. $P^*\left(V\left(x_c^{BIG}, \alpha_c^{BIG}\right)\right)$ red and sheared $P^*\left(V\left(\overline{x}_p, \overline{\alpha}_p, \overline{z}_p\right)\right)$ to the camera plane blue (see [8] for description of shearing)

This rate imposes a quite high number of cameras, which is not feasible in practice. Therefore, the task is to synthesize the required number of intermediate views based on input multiview images taken by a sparse set of cameras. Methods of this type have been referred to as image-based rendering (IBR) [84]. Here, we use the term 'reconstruction', in order to emphasize the links between sampling and reconstruction of the underlying light field function in spatial (EPI) and Fourier domains.

The EPIs of a DSLF are transformed in Fourier domain into spectra, whose support is limited by the minimum and maximum depth and by the two sampling rates, Δv and Δt, as shown in Fig. 29a. The yellow line in the figure represents a particular scene depth layer. Sparse cameras capture an aliased version of the light field, as illustrated in Fig. 29b. Therefore, a direct bandlimited reconstruction of DSLF is not possible. One has to resort to methods using more insights about the geometry of the scene.

Unstructured Lumigraph Rendering [86] can be given as an example of a generic IBR technique, which utilizes a few perspective images augmented with an accurate geometric model. Then, a ray must intersect some point on a geometric proxy of a scene in order to estimate its radiance. There is a trade-off between the number of input camera views and accuracy of the geometric proxy: the less the number of input camera views, the more the rendering quality depends on the accuracy of the available geometry. An accurate and globally consistent estimation of the scene geometry can be sought in terms of depth maps, point clouds, oriented planes, using the given camera images and making use of methods for structure from motion and depth estimation from two or multiple images [87–91]. Having depth maps as a scene geometry model, one can render the required views also by perspective reprojections using a technique referred to as depth-image based rendering [92].

Depth layering has also been employed to effectively extend the plenoptic sampling model and improve the quality of rendered views [93]. Recall, that depth layers appear as directed lines in the Fourier representation of EPIs. Then, layering in a finite number of depth layers is equivalent to sectioning the Fourier spectrum in narrow sectors and a minimum number of equidistant layers can be

specified for a given camera sampling rate Δt, using (18). It has been shown in [93] that a non-uniform layer sectioning, i.e., selecting densely-spaced layers around intensive depth changes and coarsely-spaced layers in areas with no depth changes, substantially improves the rendering quality. Depth layering has been implemented through segmentation of spatial objects and estimating their evolution through the given perspective views, thus separating depth layer volumes. Intermediate images are then synthesized 'layer-by-layer' from the background to the foreground [93]. Depth layering is also in the core of our attempt to find and utilize a sparse LF representation instrumental in DSLF reconstruction.

5.2 Reconstruction of DSLF in Directional Transform Domain

In general, natural scenes are composed by clustered objects forming a finite, rather small number of depth layers. These depth layers appear as directional stripes in the EPIs of the continuous LF and as 'broken' stripes in the EPIs of the coarsely sampled LF. In frequency domain, directional filters should be able to analyze dominant directions and provide guidance to the DSLF reconstruction. This is equivalent to employing a proper frequency plane tiling. The case is illustrated in Fig. 29c where the Fourier plane is tiled by four depth layers, with 1 pixel disparity range in each layer. Given these layers, intermediate view interpolation is possible with no aliasing artifacts. One can additionally simplify the search of depth layers, by combining it with multiresolution analysis, as shown in Fig. 29d. The region L_1 in the figure is free from aliasing and therefore can be reconstructed by low-pass filtering. Noting, that the procedure of low-pass filtering followed by decimation can be interpreted as increasing the pixel size and thus decreasing the disparity between the given rows, one can reduce the sought depth layering directions, depending on the scale number. This gives rise to a frequency plane tiling with corresponding directional filters which are also scale dependent. Construction of such tilling for the DSLF reconstruction is discussed further in the section.

5.2.1 Directional Transforms

The interest in directional transforms comes from the observation that natural images are composed by objects delineated by edges. In their seminal work [94], Olshausen and Field have shown that natural images can be sparsely coded by oriented and localized multi-scale primitives (atoms). Such basis elements can be learned from training image datasets but can also be constructed in fixed dictionaries with certain properties, notably targeting directionality and anisotropy [95]. To formalize the problem, let's consider a class of piecewise-smooth functions $\varepsilon^2(\mathbb{R}^2)$ (also referred to as cartoon-like images), as discussed in [96–98]. A function $f \in \varepsilon^2(\mathbb{R}^2)$ consists of two components and has a form $f = f_0 + f_1 \aleph_B$, where f_0 and f_1 are C^2-smooth with support in $[0, 1]^2$ and $\aleph_B$ is characteristic function

of a set $B \subset [0,1]^2$ with bound δB being a closed C^2-curve with bounded curvature. The problem of reconstructing a function from $\varepsilon^2(\mathbb{R}^2)$ space using its given incomplete measurements can be addressed through sparse approximation in a basis or frame. The quality of the representation in a given frame is described by the asymptotic decay speed of the L^2 error of the approximation obtained using only N largest coefficients of the frame decomposition. Consider the wavelet-domain decomposition of functions in $\varepsilon^2(\mathbb{R}^2)$. For it, the approximation error rate is $O(N^{-1})$, where N is the number of best wavelet coefficients. In comparison, adaptive triangle-based approximation of the cartoon-like images provides $O(N^{-2})$ approximation rate [95], where N is the number of triangles used for image representation. In order to provide better approximation than the wavelet transform, the desirable transform should provide a good directional sensitivity due to approximation of singularities distributed over the C^2-smooth curve δB which is the border between smooth image pieces. Several frames and corresponding transforms have been constructed for sparse representations, among them, tight curvelet frames by Candes and Donoho [97] and countourlets by Do and Vetterli [99]. For the case of representing the light field, one can observe that the anisotropic property of the EPI is caused by a shear transform. This naturally leads to the idea of using a transform constructed with the same property, namely the shearlet transform.

The optimal sparse approximation property of the tight shearlet frame has been studied in [100]. Similar results for compactly supported shearlet frame have been reported in [101]. Both types of shearlet frames provide an optimal sparse approximation of $f \in \varepsilon^2(\mathbb{R}^2)$, in the sense that the N-term approximation f_N constructed by keeping N largest coefficients of the frame decomposition satisfies $\|f - f_N\|_2^2 = O\left(N^{-2}(\log N)^3\right)$.

5.2.2 Shearlet Transform

Our goal is to construct a frame with directed multi-scale elements, tiling the Fourier plane in the manner shown in Fig. 30a. This can be done by the so-called cone-adapted shearlet system [98]. Let partition the plane into two cone-like regions $C_\psi, C_{\underset{\psi}{\sim}}$ complemented by a low-pass region C_ϕ as drawn in Fig. 30b. For the effective tiling of the cones, one needs shearlet atoms generated by a scaling function $\phi \in L^2(\mathbb{R}^2)$ and two shearlets $\psi, \widetilde{\psi} \in L^2\left(\mathbb{R}^2\right)$. The shearlet system is generated by the translation of the scaling function, and translation, shearing and scaling of the shearlet transform

$$S := \begin{cases} \phi_m = \phi\left(\cdot - c_1 m\right), \quad m \in \mathbb{Z}^2, \\ \psi_{j,k,m} = 2^{\frac{j+\lfloor j/2 \rfloor}{2} j} \psi\left(S_k A_{2^j} \cdot - M_c m\right), \\ \widetilde{\psi}_{j,k,m} = 2^{\frac{j+\lfloor j/2 \rfloor}{2} j} \widetilde{\psi}(S_k^T \widetilde{A}_{2^j} \cdot - \widetilde{M}_c m), \end{cases} \tag{19}$$

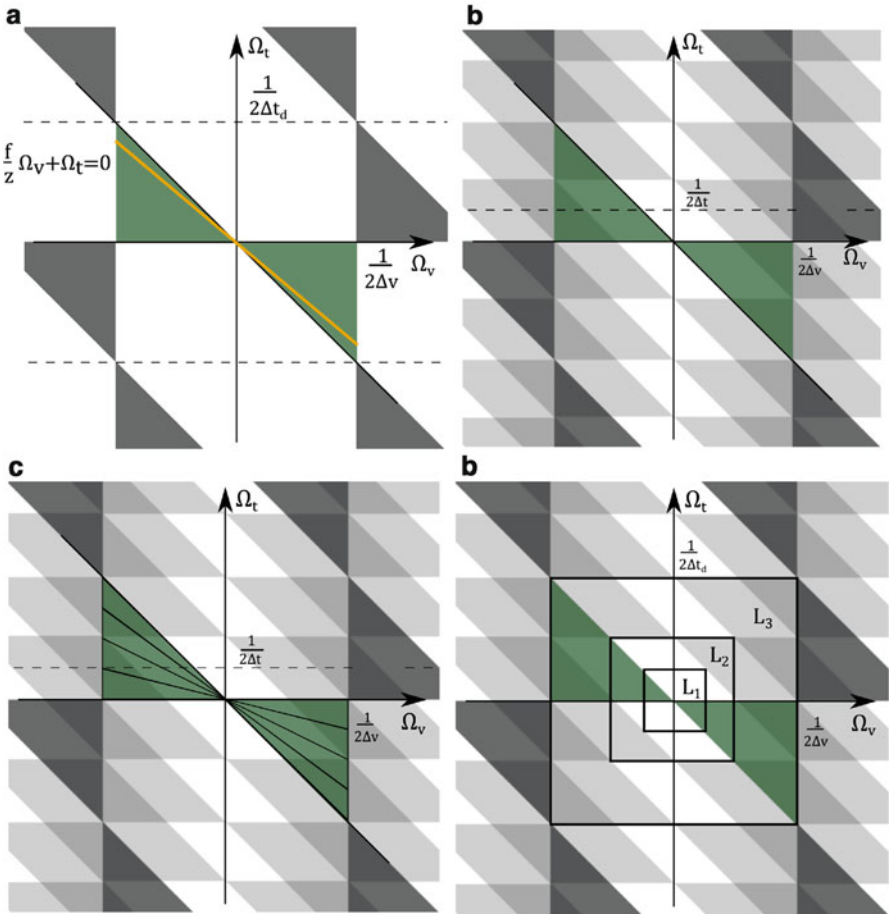


Fig. 29 DSLF in Fourier domain. (**a**) Baseband (in green) and its replicas. (**b**) Aliased replicas due to undersampling (sparser set of cameras, e.g., $\Delta t = 4$ px). (**c**) Discrete depth layers with 1 pixel disparity range. (**d**) Multiresolution analysis at three scales. © 2018 IEEE Reprinted with permission from [85]

where $S_k = \begin{pmatrix} 1 & k \\ 0 & 1 \end{pmatrix}$ is a shear matrix, $M_c = \begin{pmatrix} c_1 & 0 \\ 0 & c_2 \end{pmatrix}$ and $\widetilde{M}_c = \begin{pmatrix} c_2 & 0 \\ 0 & c_1 \end{pmatrix}$ are sampling densities of the translation grid and $A_{2^j} = \begin{pmatrix} 2^j & 0 \\ 0 & 2^{-1} \end{pmatrix}$ and $\widetilde{A}_{2^j} = \begin{pmatrix} 2^{-1} & 0 \\ 0 & 2^j \end{pmatrix}$ are scaling matrices, specifically tailored for the case of EPI, so to handle singularities over straight lines [85].

Few design and implementation remarks follow. First, it is desirable that the shearlet frames are compactly supported in both frequency and EPI domains.

The design of compactly-supported shearlets goes through the design of two 1D half-band filters and a directional non-separable filter [85]. Second, while the construction is in continuous domain, the input data comes from digital sensors in the form of discrete pixels $f_J^d(n), \quad n \in \mathbb{Z}^2$. This is handled by assuming that these are samples of a continuous function at some sufficiently large scale $J \in \mathbb{N}$

$$f(x) = \sum_{n \in \mathbb{Z}^2} f_J^d(n) 2^J \phi \left(2^J x - n\right). \tag{20}$$

The particular choice of J depends on the maximum disparity between input views and for dyadic scales it can be set as [85]

$$J = \left\lceil \log_2 d_{\max} \right\rceil. \tag{21}$$

Third, for an efficient implementation, the transform has to be discretized, that is to find the digital filters $\psi_{j,k,m}^d$ corresponding to $\psi_{j,\,k,\,m}$. This should be done by refining the regular grid $\mathfrak{Z}^2$ in order to make it invariant under the shear transforms [85]. The corresponding frames are not orthogonal and the dual shearlet filters have to be obtained as well. Furthermore, the shear operation is enforced to be with positive sign, i.e., $0 \leq k \leq 2^j + 1$ in order to apply it on EPIs (c.f. Fig. 30c).

Eventually, one gets analysis and synthesis frame elements represented through pairs of digital filters enabling the direct and inverse shearlet transforms [85]. Figure 30d represents the frequency-domain support of the elements obtained following the above design remarks for $J = 2$.

5.2.3 DSLF Reconstruction in Shearlet Domain

We consider the case of horizontal parallax first and make remarks about how to generalize the method for full parallax later in the section. Consider a setting of rectified cameras on a horizontal rig. The key starting point is to regard the given set of camera views as a downsampled version of the unknown DSLF. For the sake of simplicity, assume that the cameras are uniformly distributed over each $\lceil d_{\max} \rceil$-th view of DSLF, where $d_{\max}$ is the maximum disparity presented. An example is shown in Fig. 31a, where EPI representation of four views with 16 pixels maximum disparity is given. In Fig. 31b, the targeted densely sampled EPI is to be constructed in such a way that the available data appears in rows with 16 px distance. Figure 31c shows the same rows with respect to the fully reconstructed EPI, where the disparity less than or equal to 1 pixel is maintained. The task is to inpaint the empty areas by continuining the directional strips which only start to form in Fig. 31b.

Assume that the densely-sampled EPI is a square image reordered in a vector $y^* \in \mathbb{R}^{N^2}$, where $N = (K - 1)d_{\max} + 1$ and K is the number of input views. The samples $y \in \mathbb{R}^{N^2}$ of y^* are obtained by

$$y = Hy^*, \tag{22}$$

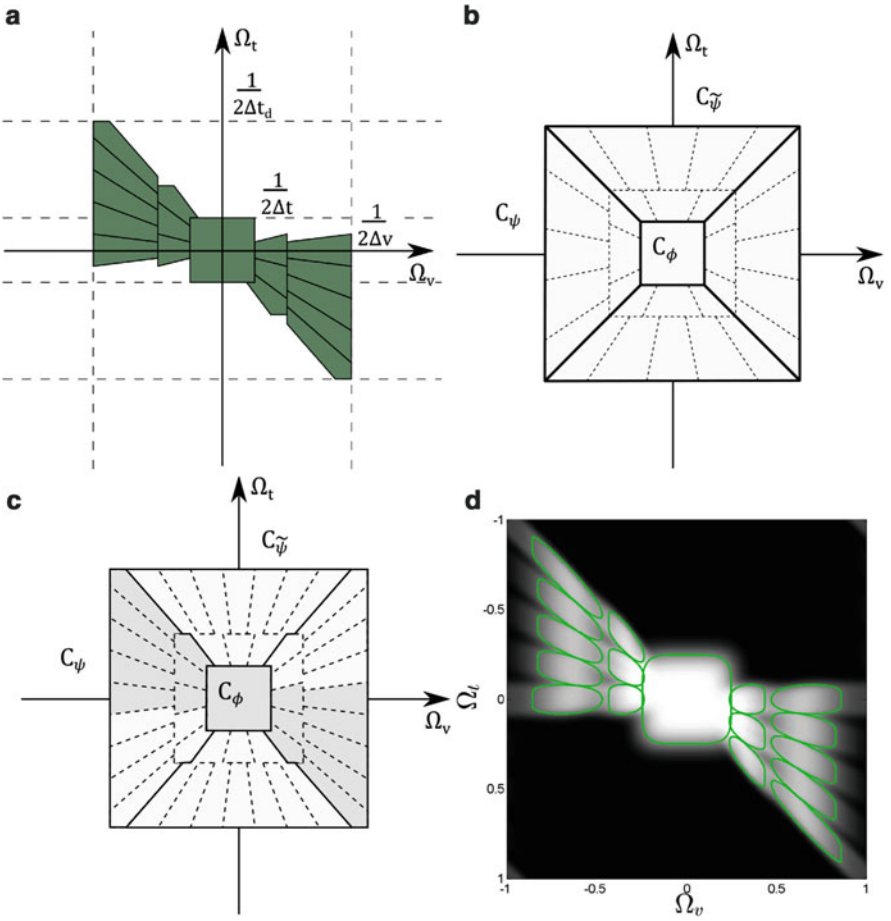


Fig. 30 Construction of shearlet frame. (**a**) Desired frequency plane tiling. (**b**) Low-pass region and two adjacent cones. (**c**) Frame elements corresponding to admissible disparity directions (in grey). (**d**) Frequency response of frame elements constructed by the use of particular filters. © 2018 IEEE Reprinted with permission from [85]

where $H \in \mathbb{R}^{N^2 \times N^2}$ is a diagonal sampling matrix, such that $H(kd_{\max}, kd_{\max}) = 1$, $k = 0, \ldots, K$ and 0 elsewhere. The measurements y form an incomplete EPI where only rows from the available images are presented, while everywhere else the EPI values are 0. The shearlet analysis and synthesis transforms are defined as $S : \mathbb{R}^{N^2} \rightarrow \mathbb{R}^{N^2 \times \eta}$, $S^* : \mathbb{R}^{N^2 \times \eta} \rightarrow \mathbb{R}^{N^2}$ where η is the number of all translation-invariant transform elements. The solution for y^* given the sampling matrix H and the measurements y is constrained by the sparsity requirement in the shearlet transform domain, i.e.,

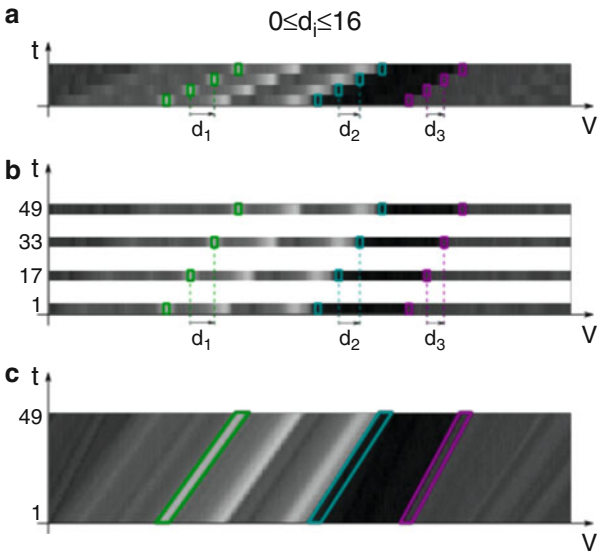


Fig. 31 Four given views of an undersampled LF. (**a**) All views stacked together. (**b**) Input views distributes with respect to $d_{\max} = 16$ and views to be synthesized in between. (**c**) The targeted DSLF. © 2018 IEEE Reprinted with permission from [85]

$$x^* = \underset{x \in \mathbb{R}^{N^2}}{\operatorname{argmin}} \|S(x)\|_1, \quad \text{subject to} \quad y = Hx, \tag{23}$$

The problem (23) can be solved e.g., by making use of the iterative procedure within the morphological component analysis approach [102, 103]. More specifically, the EPI y^* is obtained by performing shearlet-domain regularization through iterative thresholding

$$x_{n+1} = S^* \left(T_{\lambda_n} \left(S \left(x_n + \alpha_n \left(y - H x_n \right) \right) \right) \right), \tag{24}$$

where $\left(T_\lambda x\right)(n) = \begin{cases} x(n), |x(n)| \geq \lambda \\ 0, |x(n)| < \lambda \end{cases}$, is a hard thresholding operator and α_n is an adaptive acceleration parameter, which controls the convergence [85].

Note the influence of the parameter $d_{\max}$. It determines the number of scales J as in (21). For dyadic scales $j = 0, \ldots, J-1$, one gets $2^{j+1} + 1$ shears (disparities) in each scale $s_k = \frac{k}{2^{j+1}}, k = 0, \ldots, 2^{j+1}$.

Full-parallax imagery can be handled in a separable manner: The horizontal views of targeted DSLF are reconstructed first followed by the same procedure in vertical direction. This is illustrated in Fig. 32 and referred to as direct method. A computationally more efficient method, referred to as hierarchical reconstruction is presented in Fig. 32b. There, the reconstruction is performed in a specific

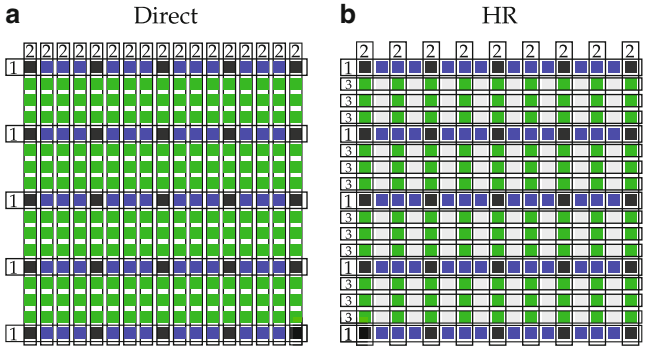


Fig. 32 Full-parallax DSLF reconstruction. (**a**) Direct reconstruction: input views in black, blue views reconstructed first, green views reconstructed second. (**b**) Hierarchical reconstruction: by alternating reconstructions in horizontal and vertical directions, one reduces the maximum disparity between views. © 2018 IEEE Reprinted with permission from [85]

order aimed at reducing the maximum disparity and thus reducing the number of directional shearlet frame elements [85].

DSLF reconstruction employing shearlet transform has shown superior results compared with state of the art depth-based methods [85]. This is to be attributed to the way the transform handles spatial and directional LF details. If attempting to estimate (a globally-consistent) depth and use it as a geometry guidance in view synthesis, one has always to associate a depth value to a pixel, thus compromising in cases of semi-transparent scenes when such association is not possible. In contrast, the reconstruction based on directional transform employs atoms which are natural for the LF imagery. Regularization is implemented in a linear space of functions, which yields a good reconstruction quality also for scenes where the depth layers are being fused in the captured views, as in the case of semi-transparent materials.

5.2.4 Other Sparsifying Transforms

While the shearlet transform is a suitable methodological example of directional transform being good for DSLF reconstruction, other sparsifying approaches should be mentioned as well.

In the work [104], the LF sparsity in the angular domain has been acknowledged and the corresponding sparse representation has been sought through continuous Fourier transform. The work takes 1D viewpoint trajectories as input and applies the Fourier projection slice theorem [79] to get a sparse representation. Both the magnitudes and positions of the continuous Fourier atoms are estimated. The algorithm demonstrates its power on non-Lambertian full-parallax scenes.

Instead of employing a fixed transform, a sparse LF decomposition can be obtained by learning a dictionary of atoms from LF data. A number of works have pursued this approach. In the work [105], 4D spatio-angular LF patches have been

used to learn a dictionary to be used for the reconstruction of LF captured by a single-sensor coded-mask optical system. In the work [106], LF patches have been used to form dictionaries to be used for joint denoising and spatial/angular super-resolution, which is essentially LF reconstruction. The problem of upsampling camera arrays has been cast as a directional super-resolution in 4D space in the work [107], where the generation of the desired perspective views has been performed through patch matching. In the work [108], the directionality in EPI domain has been employed to increase the dimensionality of image patches to 4D LF patches, which can learn then a dictionary with atoms preserving the orientation-depth relationship. The so-learned dictionaries have been used for depth estimation in light fields. In the work [109], a view synthesis technique has been developed by a learning-based approach using two convolutional neural networks for disparity and color estimation correspondingly. Four corner views from the light fields have been used to synthesize an intermediate view in attempt to increase the angular resolution of the light field captured by commercial plenoptic camera.

6 Conclusions

The ultimate goal of a 3D visualization system is to perfectly recreate a desired 3D scene. To achieve this, such system must be able to generate all rays radiating from the scene, that is, it must recreate the underlying continuous plenoptic function describing the scene. Since in practice one deals with discrete data, the scene to be visualized must be captured with a level of detail that is sufficient to avoid artefacts due to sampling (e.g., aliasing), processed if needed (e.g., filtered to display bandwidth) and finally reproduced by some means (converted back to its continuous form at a resolution better than the resolution of the human eye). The underlying concept for determining the requirements of visualization systems as well as analyzing scenes in a systematic manner is based on the notion of LF.

This chapter presented the state of the art in the area of LF processing and visualization. In the first half of the chapter, the basic LF concepts have been introduced, emphasizing the concept of continuous plenoptic function and its sampling for subsequent reconstruction. This has been followed by an overview of the existing 3D display technologies summarizing their pros and cons with respect to their ability to reproduce realistic 3D scenes, in terms of visual cues. In the second part of the chapter, signal processing methods have been described for analyzing LF displays and pre-processing data to be shown on such displays. These addressed the two main problems related with today's LF display, namely the display design (how rays recreate an LF) and data manipulation (how to represent sampled LF and process it for visualization).

It has been shown that the introduced concept of display bandwidth can be used to either evaluate the quality of an LF display and optimize (pre-process) data to be shown on the display or, if constructing the display, optimize the display configuration such that the display specifications match the desired requirements.

Equipped with this methodology, and with a good understanding of existing display technology, one can identify and quantify the technology limitations, which in turn, have to be addressed by proper data capture and processing. This part has been addressed in the last section which overviewed the challenge of reconstructing DSLF from a sparse set of captured images. The emphasis has been put on methods based on directional transforms (and more specifically the shearlet transform), since those are methodologically very elegant and result in the best LF reconstruction.

In conclusion, as illustrated in this chapter, although big leaps in the LF technology have been made during recent years, further advancements are needed to address e.g., the issues of display miniaturization (today's systems are bulky), real time processing of data to be visualized (today's algorithms are computationally demanding), and handling the storage and transmission of the large amount of data associated with the high-quality LF visualization.

References

1. A. Gershun, "The light field," *Journal of Mathematics and Physics*, vol. 18, no. 1-4, pp. 51-151, 1939.
2. E. H. Adelson and J. R. Bergen, "The plenoptic function and the elements of early vision," *Computational Models of Visual Processing*. MIT Press, pp. 3-20, 1991.
3. M. Levoy and P. Hanrahan, "Light field rendering," in *Proc. ACM SIGGRAPH*, 1996, pp. 31-42.
4. S.J.Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen, "The Lumigraph", in *Proc. ACM SIGGRAPH*, 1996, pp. 43-54.
5. P. Moon and D.E. Spencer, *The Photic Field*. MIT Press, 1981.
6. R. Bregović, P. T. Kovács, T. Balogh, and A. Gotchev, "Display-specific light-field analysis," in *Proc. SPIE* 9117, 911710, 2014.
7. C. K. Liang, Y. C. Shih, and H. H. Chen, "Light field analysis for modeling image formation," *IEEE Trans. Image Process*. Vol. 20, no. 2, 446–460, 2011.
8. R. Bregović, P. T. Kovács, and A. Gotchev, "Optimization of light field display-camera configuration based on display properties in spectral domain," *Optics Express*, vol. 24, no. 3, pp. 3067-3088, Feb. 2016.
9. R. Bolles, H. Baker, and D. Marimont, "Epipolar-plane image analysis: An approach to determine structure from motion," *Int. J. Comput. Vis*., vol. 1, no. 1, pp. 7-55, 1987.
10. J.-X. Chai, X. Tong, S.-C. Chan, and H.-Y. Shum, "Plenoptic sampling," in *Proc. ACM SIGGRAPH* 2000, pp. 307-318.
11. C. Zhang and T. Chen, "Spectral analysis for sampling image-based rendering data," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 1, pp 1038-1050, Nov. 2003.
12. J. Pearson, M. Brookes, and P. Dragotti, "Plenoptic layer-based modeling for image based rendering," *IEEE Trans. Image Process.*, vol. 22, no. 9, pp. 3405–3419, Sep. 2013.
13. C. Gilliam, P.L. Dragotti, and M. Brookes, "On the spectrum of the plenoptic function," *IEEE Trans. Image Proc.*, vol. 23, no. 2, pp. 502-516, Feb. 2014.
14. D. R. Proffitt and C. Caudek, "Depth perception and the perception of events," in *Handbook of Psychology*. New York, NY, 2002.
15. A. Stern, Y. Yitzhaky, and B. Javidi, "Perceivable Light Fields: Matching the Requirements Between the Human Visual System and Autostereoscopic 3-D Displays," *Proc IEEE* vol. 102, no. 10, pp. 1571-1587, 2014.
16. L. Goldmann and T. Ebrahimi, "Towards reliable and reproducible 3-D video quality assessment," in *Proc. SPIE Int. Soc. Opt. Eng.*, vol. 8043, 2011.

17. A. Boev, R. Bregović, and A. Gotchev, "Signal processing for stereoscopic and multi-view 3D displays," in *Handbook of Signal Processing Systems, 2nd edition*, S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, eds., Springer, 2013.
18. M. S. Banks, D. M. Hoffman, J. Kim, and G. Wetzstein, "3D Displays," *Annual Review of Vision Science*, vol. 2, pp. 397-435, 2016.
19. D. M. Hoffman, A. R. Girshick, K. Akeley, and M. S. Banks, "Vergence–accommodation conflicts hinder visual performance and cause visual fatigue," *Journal of Vision*, vol. 8, no. 33, 2008.
20. F. L. Kooi and M, Lucassen, "Visual comfort of binocular and 3D displays," in *Proc. SPIE 4299, Human Vision and Electronic Imaging VI*, 586, 2001.
21. P. A. Howarth, "Potential hazards of viewing 3-D stereoscopic television, cinema and computer games: a review," *Ophthalmic and Physiological Optics*, vol. 31, pp 111–122, 2011.
22. M. Yamaguchi, "Light-field and holographic three-dimensional displays [Invited]," *J. Opt. Soc. Am. A*, vol. 33, no. 12, 2348-2364, 2016.
23. Y. Kajiki, H. Yoshikawa, and T. Honda, "Ocular accommodation by super multi-view stereogram and 45-view stereoscopic display," in *Proc. of Third International Display Workshops (IDW)*, 1996.
24. H. Hiura, T. Mishina, J. Arai, and Y. Iwadate, "Accommodation response measurements for integral 3D image," in *Proc. SPIE* 9011, 90111H, 2014.
25. Y. Kim, et al., "Accommodative response of integral imaging in near distance," *J. Disp. Technol.* Vol. 8, no. 2, pp. 70–78, 2012.
26. Y. Takaki, Y. Urano, S. Kashiwada, H. Ando, and K. Nakamura, "Super multi-view windshield display for long-distance image information presentation," *Opt. Express* vol. 19, no. 2, pp. 704–716, 2011.
27. G. Lippmann, "Epreuves reversibles Photographies integrals," *Comptes Rendus Academie des Sciences,* vol. 146, pp. 446–451, 1908.
28. J.-H. Jung, K. Hong and B. Lee, "Effect of viewing region satisfying super multi-view Condition in Integral Imaging," *SID Symposium Digest of Technical Papers*, vol. 43, pp. 883–886, 2012.
29. H. Deng, Q.-H. Wang, C.-G. Luo, C.-L. Liu, and C. Li, "Accommodation and convergence in integral imaging 3D display," *J. SID*, vol. 22, no. 3, pp. 158–162, 2014.
30. H. Navarro, R. Martínez-Cuenca, G. Saavedra, M. Martínez-Corral, and B. Javidi, "3D integral imaging display by smart pseudoscopic-to-orthoscopic conversion (SPOC)," *Opt. Express,* vol. 18, no. 25, pp. 25573–25583, 2010.
31. F. Okano, H. Hoshino, J. Arai, and I. Yuyama, "Real-time pickup method for a three-dimensional image based on integral photography," *Appl. Opt.* vol. 36, pp. 1598–1603, 1997.
32. X. Xiao, B. Javidi, M. Martinez-Corral, and A. Stern, "Advances in three-dimensional integral imaging: sensing, display, and applications [Invited]," *Appl. Opt.* vol. 52, no. 4, pp. 546-560, 2013.
33. J. S. Jang, F. Jin, and B. Javidi, "Three-dimensional integral imaging with large depth of focus by use of real and virtual image fields," *Opt. Lett.* Vol. 28, no. 16, pp. 1421–1423, 2003.
34. S. W. Min, B. Javidi, and B. Lee, "Enhanced three-dimensional integral imaging system by use of double display devices," *Appl. Opt.* vol. 42, no. 20, pp. 4186–4195, 2003.
35. S.- Park, J. Yeom, Y. Jeong, N. Chen, J.-Y. Hong, and B. Lee, "Recent issues on integral imaging and its applications" *J. Inf. Disp.,* vol. 15, no. 1, pp. 37–46, 2014.
36. C. van Berkel and J. A. Clarke, "Characterization and optimization of 3D-LCD module design", in *Proc. SPIE*, 3012, pp.179-186, 1997.
37. Y. Takaki, K. Tanaka, and J. Nakamura, "Super multi-view display with a lower resolution flat-panel display", *Opt. Express,* vol. 19, no. 5, pp. 4129–4139, 2011.
38. B. Javidi, F. Okano, and J. Y. Son, *Three-Dimensional Imaging, Visualization, Display*. New York, NY, USA: Springer-Verlag, 2009.
39. J. Geng, "Three-dimensional display technologies," *Adv. Opt. Photon.*, vol. 5, no. 4, pp. 456–535, 2013.

40. T. Honda, Y. Kajiki, S. Susami, T. Hamaguchi, T. Endo, T. Hatada, and T. Fujii, "A display system for natural viewing of 3-D images," in *Three-Dimensional Television, Video and Display Technologies. Berlin, Germany*: Springer-Verlag, pp. 461–487, 2002.
41. Y. Takaki, Y. Urano, and H. Nishio, "Motion-parallax smoothness of short-, medium-, and long-distance 3D image presentation using multi-view displays," *Opt. Express,* vol. 20, no. 24, pp. 27180-27197, 2012.
42. Y. Takaki, "Development of super multi-view displays," *ITE Transactions on Media Technology and Applications*, vol. 2, no. 1, pp. 8–14, 2014.
43. Y. Kajiki, H. Yoshikawa and T. Honda: "Hologram-like video images by 45-view stereoscopic display", in *Proc. SPIE*, 3012, pp.154-166, 1997.
44. T. Honda, D. Nagai and M. Shimomatsu: "Development of 3-D display system by a fan-like array of projection optics", in *Proc. SPIE*, 4660, pp.191-199, 2001.
45. H. Nakanuma, H. Kamei, and Y. Takaki: "Natural 3D display with 128 directional images used for human-engineering evaluation", in *Proc. SPIE*, 5664, pp.28-35, 2005.
46. K. Kikuta and Y. Takaki: "Development of SVGA resolution 128-directional display", in *Proc. SPIE*, 6490, pp.64900U-1-8, 2007.
47. T. Kanebako and Y. Takaki: "Time-multiplexing display module for high-density directional display", in *Proc. SPIE*, 6803, pp.68030P-1-8, 2008.
48. Y. Takaki and N. Nago: "Multi-projection of lenticular displays to construct a 256-view super multi-view display", *Opt. Express*, vol. 18, no. 8, pp.8824-8835, 2010.
49. T. Balogh, "The HoloVizio system," in *Proc. SPIE* 6055, 12 pages, 2006.
50. J. T. McCrickerd and N. George, "Holographic stereogram from sequential component photographs," *Applied Physics Letters*, vol. 12, no. 1, pp. 10-12, 1968.
51. D. J. DeBitetto, "Holographic Panoramic Stereograms Synthesized from White Light Recordings," *Applied Optics*, vol. 8, no. 8, pp. 1740-1741, 1969.
52. M. W. Halle, "Holographic stereograms as discrete imaging systems," in *Proc. SPIE*, vol. 2176, pp. 73-84, 1994.
53. F. Yaraş, H. Kang, and L. Onural, "Real-time phase-only color holographic video display system using LED illumination," *Applied Optics*, vol. 48, no. 34, pp. H48-H53, 2009.
54. D. Brotherton-Ratcliffe, F. Vergnes, A. Rodin, and M. Grichine Holographic Printer. U.S. Patent 1999b; No. US7800803B2.
55. Zebra Imaging Inc. Company (2012) http://www.zebraimaging.com/.
56. X. Li, C. P. Chen, H. Gao, Z. He, Y. Xiong, H. Li, W. Hu, Z. Ye, G. He, J. Lu, and Y. Su, "Video-rate holographic display using azo-dyedoped liquid crystal," *J. Display Technol.*, vol. 10, pp. 438–443, 2014.
57. S. Tay, M. Yamamoto, and N. Peyghambarian, "An updateable holographic 3-D display based on photorefractive polymers," *SID Symp. Dig. Tech. Pap.* 39, pp. 356–357, 2008.
58. M. Lucente, *Diffraction-specific fringe computation for electro-holography*, Ph.D. dissertation, Cambridge, MA, USA, 1994.
59. T. Yatagai, "Three-dimensional displays using computer-generated holograms," *Optics Communications,* vol. 12, no. 1, pp. 43-45, 1974.
60. J. Mäkinen, *From light fields to wavefields: Hologram generation from multiperspective images,* Master's thesis, Tampere University of Technology, Finland, 2017.
61. B. E. A. Saleh and M. C. Teich, *Fundamentals of photonics*, 2nd ed. Hoboken, N.J: John Wiley & Sons, 2007.
62. Q. Y. J. Smithwick, J. Barabas, D. Smalley, and V. M. Bove, Jr., "Interactive Holographic Stereograms with Accommodation Cues," in *Proc. SPIE Practical Holography XXIV: Materials and Applications*, 7619, 761903, 2010.
63. H. Zhang, Y. Zhao, L. Cao, and G. Jin, "Fully computed holographic stereogram based algorithm for computer-generated holograms with accurate depth cues," *Opt. Express* vol. 23, no. 4, 3901-3913, 2015.
64. Ives FE. 1903. *Parallax stereogram and process of making same*. US Patent No. 725,567.
65. D. Lanman, M. Hirsch, Y. Kim Y, R. Raskar, "Content-adaptive parallax barriers: optimizing dual-layer 3D displays using low-rank light field factorization," *ACM Trans. Graph*. vol. 29,

no. 6, 163: 10 pages, 2010.
66. G. Wetzstein, D. Lanman, M. Hirsch, R. Raskar, "Tensor displays: compressive fight field synthesis using multilayer displays with directional backlighting," *ACM Trans. Graph.* vol. 31, 80: 11 pages, 2012.
67. G. Wetzstein, D. Lanman, W. Heidrich, R. Raskar, "Layered 3D: tomographic image synthesis for attenuation-based light field and high dynamic range displays," *ACM Trans. Graph*. vol. 30, 95: 11 pages, 2011.
68. F-C. Huang, K. Chen, G. Wetzstein, "The light field stereoscope: immersive computer graphics via factored near-eye light field displays with focus cues," *ACM Trans. Graph.,* vol. 34, 60: 12 pages, 2015.
69. M. Hirsch, G. Wetzstein, R. Raskar, "A Compressive Light Field Projection System," *ACM Trans. Graph.,* vol. 33, 4: 12 pages, 2014.
70. J.H. Lee, J. Park, D. Nam, S.Y. Choi, D.S. Park, and C.Y. Kim, "Optimal projector configuration design for 300-Mpixels multi-projection 3D display," *Opt. Express* vol. 21, no. 22, 26820–26835, 2013.
71. E. Dubois, "The sampling and reconstruction of time-varying imagery with application in video systems," in *Proc. IEEE* 73, 502–522, 1985.
72. E. Dubois, "Video sampling and interpolation," in *The Essential Guide to Video Processing*, J. Bovik, ed., Academic Press, 2009.
73. P.Q. Nguyen and D. Stehlé, "Low-dimensional lattice basis reduction revisited," *ACM Trans. Algorithms,* vol. 5, no. 4, 46 pages, 2009.
74. F. Aurenhammer, "Voronoi diagrams – A survey of a fundamental geometric data structure," *ACM Computing Surveys* vol. 23, pp. 245-405, 1991.
75. E.B. Tadmor and R.E. Miller, *Modeling Materials: Continuum, Atomistic and Multiscale Techniques*, Cambridge University, 2011.
76. P. T. Kovács, K. Lackner, A. Barsi, A. Balázs, A. Boev, R. Bregović, and A. Gotchev, "Measurement of perceived spatial resolution in 3D light-field displays," in *Proc. IEEE Int. Conf. Image Processing,* Paris, France, pp. 768–772, Oct. 2014.
77. P. T. Kovács, R. Bregović, A. Boev, A. Barsi, and A. Gotchev, "Quantifying spatial and angular resolution of 3D light-field displays," *IEEE Journal of Selected Topics in Signal Processing*, vol. 11, no. 7, pp. 1213-1222, Oct. 2017.
78. Z. Lin and H.-Y. Shum, "A geometric analysis of light field rendering," *Int'l J. of Computer Vision*, vol. 58, no. 2, pp. 121–138, 2004.
79. R. Ng, "Fourier Slice Photography," in *Proc. ACM SIGGRAPH*, pp. 735–744, July 2005.
80. I. Tosic and K. Berkner, "Light Field Scale-Depth Space Transform for Dense Depth Estimation," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition Workshops* (CVPRW), pp. 441–448, June 2014.
81. K. Yücer, A. Sorkine-Hornung, O. Wang, and O. Sorkine-Hornung, "Efficient 3D Object Segmentation from Densely Sampled Light Fields with Applications to 3D Reconstruction," *ACM Trans. on Graphics*, vol. 35, no. 3, 2016.
82. M. Tanimoto, "Overview of FTV (free-viewpoint television)," in *Proc. IEEE Conf. Multimedia and Expo* (ICME 2009), pp. 1552–1553, June 2009.
83. J. Jurik, T. Burnett, M. Klug, and P. Debevec, "Geometry-Corrected Light Field Rendering for Creating a Holographic Stereogram," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition Workshops* (CVPRW), pp. 9–13, 2012.
84. H. Shum, S. Chan, and S. Kang, *Image-Based Rendering*. Springer- Verlag, 2007.
85. S. Vagharshakyan, R. Bregovic, and A. Gotchev, "Light Field Reconstruction Using Shearlet Transform," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no.1, pp. 133-147, Jan. 2018.
86. C. Buehler, M. Bosse, L. McMillan, S. Gortler, and M. Cohen, "Unstructured lumigraph rendering," in *Proc. 28th Conf. on Computer Graphics and Interactive Techniques*, pp. 425-432, 2001.
87. S. Fuhrmann, F. Langguth and M. Goesele "MVE – A Multi-View Reconstruction Environment," in *Proc EUROGRAPHICS Workshops on Graphics and Cultural Heritage,* 2014.

88. S. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski "A Comparison and Evaluation of Multi-View Stereo Reconstruction Algorithms," in *Proc. Conference on Computer Vision and Pattern Recognition (CVPR),* 2006.
89. H. Hirschmuller, "Stereo Processing by Semiglobal Matching and Mutual Information," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 30, no. 2, pp. 328–341, Feb. 2008.
90. S. N. Sinha, D. Scharstein and R. Szeliski, "Efficient High- Resolution Stereo Matching Using Local Plane Sweeps," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pp. 1582–1589, June 2014.
91. G. Zhang, J. Jia, T. Wong, and H. Bao, "Consistent Depth Maps Recovery from a Video Sequence," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 31, no. 6, pp. 974–988, June 2009.
92. C. Fehn, "Depth-image-based rendering (DIBR), compression and transmission for a new approach on 3D-TV," in *Proc. Stereoscopic Displays Appl,* pp. 93-104, 2002.
93. J. Pearson, M. Brookes, and P. Dragotti, "Plenoptic Layer- Based Modeling for Image Based Rendering," *IEEE Trans. Image Processing*, vol. 22, no. 9, pp. 3405–3419, Sept. 2013.
94. B. Olshausen and D. Field "Emergence of simple-cell receptive field properties by learning a sparse code for natural images", *Nature* vol. 381, pp. 607-609, 1996.
95. D. Donoho, "Sparse Components Analysis and Optimal Atomic Decomposition", Technical Report, Statistics, Stanford, 1998.
96. E. J. Candes, D. L. Donoho, *Curvelets: A surprisingly effective nonadaptive representation for objects with edges*. Stanford University, 1999.
97. E. J. Candes and D. L. Donoho, "New tight frames of curvelets and optimal representations of objects with piecewise c^2 singularities," *Comm. Pure Appl. Math.*, vol. 57, no. 2, pp. 219–266, 2004.
98. G. Kutyniok, *Shearlets: Multiscale analysis for multivariate data*. Springer Science & Business Media, 2012.
99. M. Do and M. Vetterli, "The contourlet transform: an efficient directional multiresolution image representation," *IEEE Trans. Image Processing*, vol. 14, no. 12, pp. 2091–2106, Dec 2005.
100. G.Easley, D.Labate, and W.-Q.Lim, "Optimally sparse image representations using shearlets," in *Proc. Fortieth Asilomar Conf. Signals, Systems and Computers (ACSSC '06)*, pp. 974–978, Oct 2006.
101. G. Kutyniok and W.-Q. Lim, "Compactly supported shearlets are optimally sparse," *J. of Approximation Theory*, vol. 163, no. 11, pp. 1564 – 1589, 2011.
102. J.-L. Starck, Y. Moudden, J. Bobin, M. Elad, and D. L. Donoho, "Morphological Component Analysis," in *Proc. SPIE 5914 Wavelets XI*, 59140Q, May 2005.
103. J. Fadili, J.-L. Starck, M. Elad, and D. Donoho, "Mcalab: Reproducible Research in Signal and Image Decomposition and Inpainting," *IEEE Computing in Science & Engineering*, vol. 12, no. 1, pp. 44–63, 2010.
104. L. Shi, H. Hassanieh, A. Davis, D. Katabi, and F. Durand, "Light Field Reconstruction Using Sparsity in the Continuous Fourier Domain," *ACM Trans. on Graphics*, vol. 34, no. 1, 2014.
105. K. Marwah, G. Wetzstein, Y. Bando, and R. Raskar, "Compressive light field photography using overcomplete dictionaries and optimized projections," *ACM Transactions on Graphics*, vol. 32, no. 4, pp. 1-11, 2013.
106. Z. Li, *Image patch modeling in a light field*. PhD thesis, EECS Department, University of California, Berkeley, May 2014.
107. D. C. Schedl, C. Birklbauer, and O. Bimber, "Directional Super-Resolution by Means of Coded Sampling and Guided Upsampling," in *Proc. IEEE Conf. Computational Photography* (ICCP), pp. 1–10, 2015.
108. O. Johannsen, A. Sulc, and B. Goldluecke, "What Sparse Light Field Coding Reveals about Scene Structure," in *Proc. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, pp. 3262-3270, 2016.
109. N.K. Kalantari, T.-C. Wangand and R. Ramamoorthi, "Learning-Based View Synthesis for Light Field Cameras," *ACM Trans. on Graphics*, vol. 35, no. 6, 2016.

Inertial Sensors and Their Applications

Jussi Collin, Pavel Davidson, Martti Kirkko-Jaakkola, and Helena Leppäkoski

Abstract Due to the universal presence of motion, vibration, and shock, inertial motion sensors can be applied in various contexts. Development of the microelectromechanical (MEMS) technology opens up many new consumer and industrial applications for accelerometers and gyroscopes. The multiformity of applications creates different requirements to inertial sensors in terms of accuracy, size, power consumption and cost. This makes it challenging to choose sensors that are suited best for the particular application. In addition, development of signal processing algorithms for inertial sensor data require understanding on the physical principles of both motion generated and sensor operation principles. This chapter aims to aid the system designer to understand and manage these challenges. The principles of operation of accelerometers and gyroscopes are explained with examples of different applications using inertial sensors data as input. Especially, detailed examples of signal processing algorithms for pedestrian navigation and motion classification are given.

1 Introduction to Inertial Sensors

Inertial sensors measure motion parameters with respect to the inertial space. They generally fall into two categories: (a) instruments sensing linear inertial displacement, also known as accelerometers, (b) rotational inertial rate sensors, also called angular rate sensors or gyroscopes.

J. Collin (✉)
Laboratory of Pervasive Computing, Tampere University of Technology, Tampere, Finland
e-mail: jussi.collin@tut.fi

P. Davidson · H. Leppäkoski
Laboratory of Automation and Hydraulics, Tampere University of Technology, Tampere, Finland
e-mail: pavel.davidson@tut.fi; helena.leppakoski@tut.fi

M. Kirkko-Jaakkola
Finnish Geospatial Research Institute, National Land Survey of Finland, Helsinki, Finland
e-mail: martti.kirkko-jaakkola@nls.fi

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_2

1.1 Accelerometers

An accelerometer is a device that measures translational acceleration resulting from the forces acting on it. This acceleration is associated with the phenomenon of weight experienced by a mass that resides in the frame of reference inside accelerometer and can be described by Newton's second law of motion: "A force **F** acting on a body of mass *m* causes the body to accelerate with respect to inertial space." A typical accelerometer consists of a small mass, also known as a proof or seismic mass, connected via a spring to the case of the instrument as shown in Fig. 1.

When the instrument experiences acceleration along its sensitive axis, the proof mass is displaced with respect to the case of instrument; this is the scenario in Fig. 1b. Under steady state conditions, the force acting on the mass will be balanced by the tension in the spring. The extension (or contraction) of the spring creates a force which is proportional to the displacement. When there is no drag force to resist the movement of the proof mass, its displacement is directly proportional to the acceleration. This way the applied acceleration can be measured by measuring the displacement of the proof mass.

There are many different designs for accelerometer but most of them operate in a manner similar to the simple spring and mass system described above. In many applications, including navigation, the three dimensional vector of acceleration is required. Normally, three single-axis accelerometers are used. In recent years, tri-axis instruments have become very popular in the segment of low-cost accelerometers. It is a common practice to mount the three accelerometers with their sensitive axes mutually orthogonal, although any non-coplanar configuration is acceptable as long as the angles between the sensitive axes are known.

Accelerometers are insensitive to the gravitational acceleration and unable to separate the total acceleration from that caused by the presence of a gravitational field [18]. These sensors instead provide measurements of the difference between the true acceleration and the acceleration due to gravity. This quantity is the non-gravitational force per unit mass exerted on the instrument, and often called a

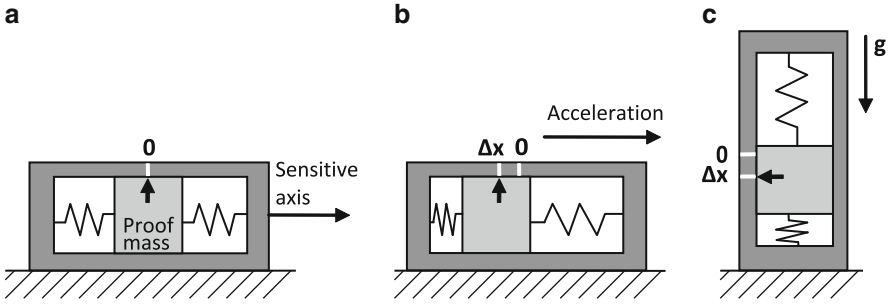


Fig. 1 A mass-and-spring accelerometer under different conditions: (**a**) at rest or in uniform motion; (**b**) accelerating; (**c**) at rest

specific force. For example, if we consider an accelerometer in free fall, the case and the proof mass will fall together. Therefore, there will be no displacement of the proof mass with respect to the case and the output of the instrument will remain at zero. In other words, the acceleration $\mathbf{a}$ of the instrument with respect to an inertially fixed set of axes equals the gravitational acceleration $\mathbf{g}$ and the specific force is zero. If the accelerometer is held stationary, i.e. $\mathbf{a} = \mathbf{0}$, it will measure the force which is counteracting to stop it from falling, $\mathbf{f} = -m\mathbf{g}$, as visualized in Fig. 1c. This specific force is required to offset the effect of gravitational attraction. Therefore, the measurements provided by the accelerometer must be combined with knowledge of the gravitational field in order to determine the acceleration of the sensor unit with respect to the inertial space.

The various accelerometer technologies include [60]: mechanical, surface acoustic waves, piezoelectric, fiber optic, vibrating beam and solid-state microelectromechanical (MEMS) accelerometers. Historically, mechanical accelerometers were the first type of accelerometers in mass production. All mechanical accelerometers are mass–spring type sensors. They can be implemented in open loop when a displacement of a proof mass with respect to its 'null' position is proportional to the specific force applied along its input axis. They can be also implemented as closed loop or force feedback pendulous accelerometer in which the spring is replaced by an electromagnetic device that produces force on the proof mass to maintain it at its 'null' position. The most precise mechanical force-feedback pendulous accelerometers are capable of measuring specific force with resolutions of micro-g or better. This class of mechanical accelerometers is used in very accurate (navigation grade) inertial navigation systems (INS).

Most of accelerometers nowadays are manufactured using MEMS technology that was developed for the military and aerospace markets in the 1970s. In 2016, the production volume of MEMS inertial sensors was about 7.5 billion units, dominated by consumer electronics and automotive applications. MEMS accelerometers can be fabricated in many different ways. The basic process modules include bulk micromachining, surface micromachining, wafer bonding, and deep reactive-ion etching (DRIE). In most cases, the fabrication involves a combination of two modules or more. The majority of the commercial accelerometers are surface micromachined. One advantage of surface micromachining is its potential of Complementary Metal-Oxide-Semiconductor (CMOS) integration. However, due to some technical challenges, two-chip solutions are still dominant in commercial products. Bulk micromachining is often combined with wafer bonding (glass–silicon or silicon–silicon) to produce high-performance accelerometers. A recent development in which single crystal silicon (SCS) sensing elements are created in CMOS substrate by using DRIE shows some promising results. In terms of materials, almost all MEMS accelerometers are made of silicon including silicon on insulator (SOI). More about MEMS accelerometers can be found in [20, Chapter 2.05].

1.2 Gyroscopes

Gyroscope (or gyro for short) is a device for measuring or maintaining angular orientation. It can measure turn rates caused by changes in attitude with respect to inertial space. Historically the first sensors of this kind were mechanical gyros. They exploit the inertial properties of a wheel spinning at high speed, which tends to keep the direction of its spin axis due to the principles of conservation of angular momentum. Although the axle orientation does not remain fixed, it changes in response to an external torque much less and in a different direction than it would without the large angular momentum associated with the disc's high rate of spin and moment of inertia. Since external torque is minimized by mounting the device in gimbals, its orientation remains nearly fixed, regardless of any motion of the platform on which it is mounted. There are several designs for mechanical gyros including: dynamically tuned gyroscope (DTG), flex gyro, and dual-axis rate transducer (DART) which is suitable only for low accuracy applications [60].

Following the development of spinning mass gyros, other kinds of angular rate sensors, such as optical and vibrating gyros, were developed [4]. These sensors are based on different physical principles than the conservation of angular momentum. Optical gyros are based on the Sagnac effect which causes a phase shift between two waves counter-propagating in a ring interferometer that is rotating; the shift is proportional to the rate of rotation. Vibrating gyros are based on Coriolis effect that induces a coupling between two resonant modes of a mechanical resonator. Optical gyros can be effectively implemented using different integrated optics technologies that generally fall into two categories: (a) ring laser gyroscopes (RLG) and (b) fiber optics gyroscopes (FOG). RLGs can be made very accurate to meet the requirements for navigation grade, but on the other hand, they are expensive, their size increases with performance, and they are high-voltage devices. FOGs are less accurate compared to RLGs, but they meet the requirements of medium accuracy (tactical grade), medium cost gyroscopes.

Vibrating gyros are usually manufactured using MEMS technology [20, Chapter 2.06]. From the accuracy point of view, MEMS gyros are of low to medium accuracy with their performance approaching FOG. They have low manufacturing costs, small physical size, and low power consumption; moreover, they can survive severe shocks and temperature changes. Therefore, MEMS technology is ideally suited for mass production.

1.3 Areas of Application

Due to the universal presence of motion, vibration, and shocks, inertial sensors can be applied almost everywhere, from aircraft and space navigation to underground drilling, from hard disk fall protection to airbags in vehicles, and from video games to performance improvement of athletes. The large variety of applications

creates different requirements to inertial sensors in terms of accuracy, size, power consumption, and cost. For example, the principal driving force for high-accuracy inertial sensors development has been inertial navigation for aircraft and submarines, precise aiming of telescopes, imaging systems, and antennas. For some applications, improved accuracy is not necessarily the most important issue, but meeting performance at reduced cost and size is. The major requirements to inertial sensors in automotive industry are low cost, high reliability, and possibility of mass production. In the following sections some examples of applications are given.

1.3.1 Navigation

An INS normally consists of three gyros and three accelerometers. The data from inertial sensors is processed to calculate the position, velocity, and attitude of the vehicle. Given the ability to measure the acceleration it would be possible to calculate the change in velocity and position by performing successive mathematical integrations of the acceleration with respect to time. In order to navigate with respect to the desired reference frame, it is necessary to keep track of the direction in which the accelerometers are pointing. Rotational motion of an INS with respect to the inertial reference frame may be sensed by gyroscopes that are used to determine the orientation of the accelerometers at all times. Given this information it is possible to resolve the accelerations into the reference frame before the integration process takes place.

High performance INSs require accurate sensors. Such systems are expensive, weigh several kilos, and have significant power consumption. However, not in every navigation application has a high-performance INS to be used. For example, land vehicle navigation systems can significantly reduce INS error growth by applying non-holonomic constraints[1] and using odometer measurements. Therefore, in many land vehicle applications a lower cost tactical grade INS can be used instead of a more expensive navigation grade INS. Pedestrian navigation systems take advantage of biomechanics of walking. Recognizing that people move one step at a time, the pedestrian mechanization restricts error growth by propagating position estimates in a stride-wise fashion, rather than on a fixed time interval. Inertial sensors are used to detect the occurrence of steps, and provide a means of estimating the distance and direction in which the step was taken. For step detection, accelerometers do not have to be of high accuracy. Pedestrian navigation is addressed more profoundly in Sect. 3.

[1] In short, non-holonomic constraints limit the lateral and vertical speeds of the vehicle and this knowledge is translated into a measurement [53].

1.3.2 Automotive

In modern cars, MEMS accelerometers are used in airbag deployment systems to detect a rapid negative acceleration of the vehicle, determine if a collision occurred, and estimate the severity of the collision. Another common automotive use of MEMS gyros and accelerometers is in electronic stability control systems. It compares the driver's intended direction which can be determined through the measured steering wheel angle to the vehicle's actual direction determined through measured lateral acceleration, vehicle yaw rotation, and individual wheel speeds.

Other automotive applications of MEMS accelerometers include monitoring of noise, vibration, harshness, and conditions that cause discomfort for drivers and passengers and may also be indicators of mechanical faults. Once the data has been collected during road tests it can be analyzed and compared to previous captures or against donor vehicles. Comparing data may highlight a problem within the vehicle allowing the technician to proceed to a repair with confidence supported by measurements taken.

1.3.3 Industrial

In industrial applications accelerometers are widely used to monitor machinery vibrations. Analysis of accelerometer based vibration data allows the user to detect conditions such as wear and tear of bearings, shaft misalignment, rotor imbalance, gear failure, or bearing fault in rotating equipment such as turbines, pumps, fans, rollers, compressors, and cooling towers. The early diagnosis of these faults can prevent costly repairs, reduce downtime, and improve safety of plants in such industries as automotive manufacturing, power generation, pulp and paper, sugar mills, food and beverage production, water and wastewater, hydropower, petrochemistry, and steel production.

1.3.4 Consumer Products

The availability of small size tri-axis accelerometers and gyroscopes with prices less than $2 has opened up new markets for inertial sensors in video game controllers, mobile phones, cameras, and other personal electronic devices. The applications of inertial sensors in consumer devices can be divided into the following categories: (a) orientation sensing, (b) gesture recognition, (c) motion input, (d) image stabilization, (e) fall detection, and (f) sport and healthy lifestyle applications.

The most common application of orientation sensing by accelerometers is converting the display to a horizontal or vertical format based on the way the device is being held. For example STMicroelectronics LSM6DSL inertial module provide configurable interrupts for change in orientation [59]. Third-party developers have created thousands of motion-sensitive games and other fanciful applications with orientation sensing features. With the use of the Global Positioning System (GPS) and a magnetic compass, location-based services are enabled, making it possible to identify special sales or lunch menus by just pointing a cell phone at a building.

Computer or video games can exploit gesture recognition techniques and make it possible to play the games or do virtual activities such as swinging a tennis racket or drive a vehicle by moving a hand-held controller. Nintendo's Wii video game console uses a controller called a Wii Remote that contains a tri-axis accelerometer and was designed primarily for motion input. The Sony PlayStation 4 uses the DualShock 4 remote with embedded inertial module that can be used, for example, to make steering more realistic in racing games.

Commonly used example of motion input application is darkening the display when not needed by detecting the motionless state. Some smartphones use accelerometers for user interface control, for example, make selections by scrolling down a list by tilting. The accelerometer-enabled wireless mouse makes it possible to move an object in space and have a corresponding object or cursor follow in a computer-generated visual model.

Cameras use inertial sensors for image stabilization to reduced blurring associated with the motion of a camera during exposure [24]. It compensates for angular yaw and pitch movement of the camera. There are two ways for images stabilization in cameras: (1) make adjustments to the image sensor or the lenses to ensure that the image remains as motionless as possible, (2) digital image stabilization in which the physical image is allowed to track the scene on the sensor by software to produce a stable image. The digital technique requires the pixel count to be increased to allow the image to move on the sensor while keeping reference points within the boundaries of the capture chip. Different companies have different names for their image stabilization technology: Image Stabilizer (Canon), Vibration Reduction (Nikon), Optical SteadyShot (Sony Cyber-Shot), Super SteadyShot (Sony), MEGA Optical Image Stabilizer (Panasonic and Leica), Optical Stabilizer (Sigma), Vibration Compensation (Tamron) and Shake Reduction (Pentax).

Fall detection is an important safety feature to protect hard disk drives in laptops and some other portable, "always on" devices like MP3 players [1]. Many of these devices feature an accelerometer which is used to detect drops. If a drop is detected, the heads of the hard disk are parked to avoid data loss and possible head or disk damage caused by the shock.

1.3.5 Sport

The advent of small low-cost inertial sensors caused the boom in sensor-laden sport equipment. Examples of MEMS inertial sensor application in sports include running, golf, tennis, basketball, baseball, soccer, boxing. Wearable electronics for running may include accelerometers, gyroscopes, magnetometer and pressure sensor located in waistband, running shorts or footpod. It can measure different running metrics, such as cadence, step length, braking, foot contact time, pelvic rotation, tilt, etc.

In ball games such as soccer and basketball inertial sensors are integrated in the ball. In soccer the equipment estimates how hard the ball has been struck, its speed, spin, and flight path [41]. In basketball the system detects shots made and missed as well as throw distance, speed, spiral efficiency, and whether a ball has been caught or dropped. In bat-and-ball games (baseball, softball, cricket) the equipment is embedded in the bat and computes different swing metrics, including power, speed, efficiency, and distance the bat travels in the hitting zone. In tennis the inertial sensors are usually embedded in racket's handle and they can detect the type of shot (forehand, backhand, serve, and smash), ball spin (topspin, slice), swing speed and ball impact spot. In golf the sensors are attached to the shaft of a club and track the position, speed, and angle of the club as it moves through a swing.

Concussion detection is important in contact sports of all kinds, especially in boxing, football and hockey [47]. MEMS accelerometers that are able to measure more than 100 g are usually embedded in helmets, headbands or mouth guards to measure the severity of an impact. In boxing a small device containing accelerometer can be attached to the boxer hand wraps or gloves to measure punch types and rate, power, hit/miss ratio.

Other examples of inertial sensors in sport include motion analysis such as figure skating jumps, and trajectory analysis in ski jumping and javelin. Xsens MVN Motion Capture [51, 67] is an interesting example of how inertial sensors can be used to record human movement. The motion capture suit includes 17 inertial trackers strapped to the different parts of the body. The data can be used in medical and sports applications to analyze human movement and gait. It can be also used to animate digital characters in movies, games, and virtual environments.

2 Performance of Inertial Sensors

Selection of the most suitable inertial sensors for a particular application is a difficult task. Among the parameters that have to be considered are resolution, dynamic range, accuracy, cost, power consumption, reliability, weight, volume, thermal stability, and immunity to external disturbances. Usually when sensors are examined for compliance, accuracy is the first parameter to start with; however, accuracy cannot be expressed as a single quantity because several factors contribute to it.

All accelerometers and gyros are subject to errors which limit their accuracy in the measurement of the applied acceleration or angular rate. The measurement error is defined as the difference between the measured and the true value of the physical quantity. Generally, inertial sensor errors fall into two broad categories: (a) systematic errors and (b) random errors. When measurement errors are analyzed, the same methodology can be applied to gyros and accelerometers.

Systematic errors are measurable and sensor type specific. They are caused by inaccuracy of system parameters and parasitic effects, streaming from the sensor design, its fabrication processes, and the readout electronics. In the context of

MEMS sensors, systematic errors apply to whole batches of sensors of a certain type produced by the same process.

Random errors are caused by interference, noise, instability etc. They can be divided into two groups: (a) wideband or uncorrelated noise and (b) colored or correlated noise. Examples of uncorrelated noise are thermal noise [39] and quantization errors in the analog-to-digital conversion of the output signal. These errors can be modeled as additive Gaussian white noise process. The effect of zero-mean white noise can be mitigated by averaging the signal over longer periods of time; since the output rate of inertial sensors is typically very high (e.g., 1000 Hz), the signals are usually down-sampled to a slower update rate by averaging.

Correlated noise is a more complicated and much more diverse phenomenon. Some examples of correlated noise are random walk, Markov processes, and flicker noise. Flicker or $1/f$ noise is a nonstationary, long-memory process (i.e., its autocorrelation decays slower than exponentially) [34]. The name stems from the fact that the power spectral density of $1/f$ noise is inversely proportional to the frequency; this implies that a major part of the power of the noise is located at low frequencies. In the context of inertial sensors, this noise process is also referred to as bias instability [28], but in this chapter, we will use the term $1/f$ noise to refer to this process and reserve the term "bias instability" for characterizing sensor quality (see Sect. 2.1).

$1/f$ noise has been observed in a wide range of different contexts, such as semiconductors, time standards, and highway traffic; even the ancient records of river Nile's flood levels have a $1/f$ power spectral density [64]. However, the origin of the phenomenon is not known, but it seems that there is no common physical mechanism to cause it in all these contexts [34]. Therefore, in order to model inertial sensor errors accurately, the contribution of $1/f$ noise must be handled carefully. A common tool for characterizing the contributions of the different noise types is the Allan variance which is described in Sect. 2.1.2. Other characterization methods do exist [37], but using Allan variance is recommended in [29].

2.1 *Effect of Different Sources of Error*

When analyzing the measurement errors of inertial sensors, it is a common practice to split the measurement error into several components that are mutually independent and specific to different modes of operation. For instance, even if the applied input signal is absent, the sensor output is not zero; this error source is called an offset or *bias*. Therefore, the bias is defined as the average of sensor output over a specified time interval that has no correlation with the input signal. Accelerometer bias is measured in $\mathrm{m/s^2}$ or fractions of g whereas gyro bias is measured in $^\circ/\mathrm{h}$ or $^\circ/\mathrm{s}$. In many cases the bias is not exactly constant but changes slowly in time. This phenomenon is also called *bias instability* and can be quantified as the peak-to-peak amplitude of the long-term bias drift.

The next important error component is the *scale factor error* which is defined as the error in the ratio relating the change in the output signal to a change in the applied input signal. Scale factor error is commonly expressed as a ratio of output error to input rate in parts per million (ppm), or, especially in the lower performance class, as a percentage figure.

Cross-axis sensitivity errors result from the sensor's sensitivity to signals applied about axes that are perpendicular to the sensitive axis. Such errors can be due to physical misalignments of the sensors' sensitive axes or, particularly in the case of MEMS sensors, electromagnetic interference between the channels. The cross-axis sensitivity is also expressed in ppm or a percentage of the applied acceleration or angular rate. *Linearity* (non-linearity) error is defined as the closeness of the calibration curve to a specified straight line. The *acceleration-dependent bias* (*g*-dependent bias) is an error which occurs in Coriolis vibratory gyros; it is proportional to the translational acceleration of the sensor. Sudden impacts and shocks may cause significant errors in the output of both accelerometers and gyroscopes in other ways as well, e.g., as a hysteresis effect.

All the error sources mentioned above consist of both systematic and random errors.

2.1.1 Calibration of Inertial Sensors

Calibration refers to correcting a measuring device by adjusting it to match reference values. Calibration of inertial sensors can significantly improve their performance. Long-term errors, i.e., those which remain constant for at least 3–5 years, can be corrected for in the factory. The factory calibration usually includes temperature compensation to guarantee good performance over the entire operational temperature range. This calibration eliminates a significant part of the measurement errors. The residual errors are much smaller than the initial errors and can be explained by the fact that the bias and scale factor errors can slightly change when the system is turned on next time—the so-called *day-to-day* error. Furthermore, the temperature compensation does not eliminate all errors caused by temperature variations.

Despite the fact that the residual errors are much smaller than the errors before the factory calibration, the sensors' performance can be improved even further if these residual errors are calibrated out. The approach for calibration of these errors depends on the application, the measurement scenario, and the type of error. From the system's perspective, one can approach the errors and their correction based on the sensor transfer characteristic (static and dynamic). With the emergence of digital signal processing and its use with sensors, this approach is becoming the standard. Keeping in mind that all sources of measurement error cumulatively affect the accuracy and resolution of a sensing system in a negative manner, the systems obey the principle of "a chain only being as strong as its weakest link". Errors such as interference, noise, and instability could be eliminated through chopping or dynamic amplification and division applied to individual sensors.

2.1.2 Allan Variance

Named after Dr. David W. Allan, the Allan variance [2] is a quantity to characterize the stability of oscillator systems. Although originally developed for frequency standards, the Allan variance is widely used to characterize the performance of inertial sensors; it reveals the contributions of uncorrelated and random walk type error processes on the measurement noise. The Allan variance σ_A^2 is a function of the averaging time τ, computed as

$$\sigma_A^2(\tau) = \frac{1}{2(N-1)} \sum_{i=1}^{N-1} (\bar{y}_\tau(i+1) - \bar{y}_\tau(i))^2 \tag{1}$$

where the data y have been partitioned into N disjoint bins of length τ, and $\bar{y}_\tau(i)$ is the average value of the ith such bin. The square root of Allan variance is known as the Allan deviation, which is in accordance with common statistical terminology.

Usually, the Allan variance function is visualized as a log–log graph; an example is shown in Fig. 2. Generally, the Allan variance curve is U-shaped. At short averaging times, quantization and uncorrelated noise dominate the output. The variance of independent and identically distributed data is inversely proportional to the averaging time, which causes a negative slope to the Allan variance at short averaging times. As the averaging time increases, after some point, $1/f$ noise starts to dominate over uncorrelated noise and the curve levels off—the Allan variance of $1/f$ noise is constant [64]. Eventually, the curve starts to increase due to rate random walk. There are also other phenomena that can be identified using Allan variance [29], but the three effects discussed above are usually the most significant.

Based on the Allan variance plot, it is possible to quantify certain characteristics of the sensor noise. The spectral density of white noise can be estimated as the

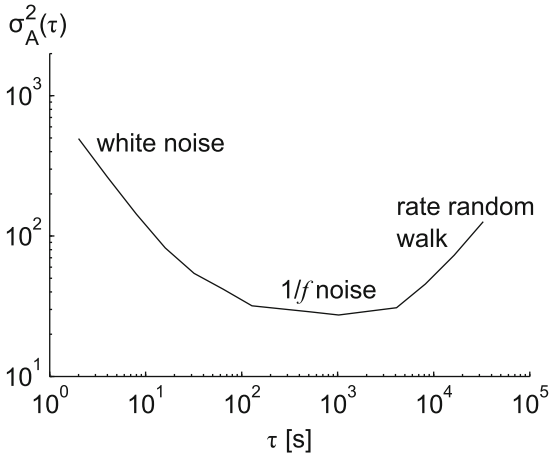


Fig. 2 An example Allan variance plot

value of the descending white noise slope at $\tau = 1\,\mathrm{s}$. The minimum value of the Allan variance between the white noise and rate random walk slopes corresponds to the square of the bias instability of the sensor; this value is directly related to the power of $1/f$ noise [64].

2.1.3 Modeling the Measurement Errors

A key to estimating and compensating for inertial sensor measurement errors is an accurate model of the evolution of the different error components with time. Some of the most commonly encountered models of sensor error time series $x(t)$ are

- *random constant*

$$x(t) = x(t-1); \tag{2}$$

- *first-order Gauss–Markov* (GM) models of the form [9]

$$x(t) = \mathrm{e}^{-\Delta t/\gamma} x(t-1) + \eta(t) \tag{3}$$

  where Δt is the time interval between steps, γ is the *correlation time* of the process, and $\eta(i)$ are independent zero-mean Gaussian random variables; and
- *random walk*

$$x(t) = x(t-1) + \eta(t) \tag{4}$$

  where the random increments $\eta(i)$ are independent and zero-mean (but not necessarily Gaussian).

These three models are closely related. It can be seen that when the correlation time γ tends to infinity, GM approaches the random walk process. On the other hand, with $\gamma \to 0$, GM tends to white noise. Random walk and GM processes are examples of *autoregressive* (AR) models which are more generally expressed as

$$x(t) = \sum_{i=0}^{t-1} a(i)x(i) + \eta(t) \tag{5}$$

where $a(i)$ are known coefficients and $\eta(i)$ are independent zero-mean random variables. Sometimes the noise process η is called the driving noise. Figure 3 shows an example realization of white noise along with the random walk and GM processes ($\gamma = 300$ samples) generated using the same noise. It can be seen that the correlated processes have significantly higher values than their driving noise.

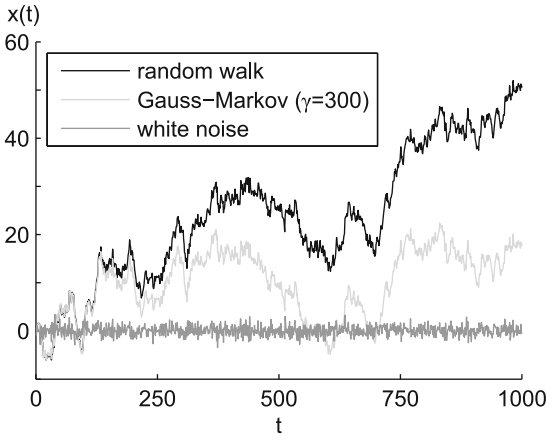


Fig. 3 Example realizations of white noise, random walk, and a first-order Gauss–Markov process

Usually, scale factor errors are quite stable over time and can be modeled as random constants.[2] In contrast, the bias of an inertial sensor can vary significantly during operation, particularly in the case of MEMS sensors. Therefore, sensor biases are often modeled as GM or random walk processes. It should be noted that they are *Markovian* processes, i.e., the value of the process at time t only depends on the state of the process at $t - 1$, not on other past or future states.[3] Thus, they are suboptimal for modeling the $1/f$ bias instability process which is known to have a long memory.

It is possible to model $1/f$ processes as AR processes [35]. However, optimal modeling of a long-memory process requires an infinite number of states to be memorized [56]; for this reason, many authors have fitted finite-order AR models on sequences of data in order to predict the future behavior of, e.g., a gyroscope's bias.

2.2 Sensor Quality Grade

Inertial sensors are used for various purposes and not all use cases demand similar performance. For instance, the requirements for the gyroscope of an automotive stability control system are significantly different from the requirements for full six-degrees-of-freedom inertial navigation. Traditionally, inertial sensors have been categorized into several grades based on their performance.

[2]Scale factors are not exactly constant: for instance, the scale factors of MEMS sensors depend strongly on the temperature.

[3]There exist higher-order Gauss–Markov process where the difference equation (3) contains older values of the process.

Navigation grade sensors are targeted for long-term autonomous navigation whereas *tactical grade* systems are manufactured for shorter intervals of navigation, usually a few minutes. Typically, the required performance for a navigation-grade system can be that the position error must not increase by more than one nautical mile (1.85 km) after 1 h of autonomous inertial navigation. For instance, navigation grade sensors can be needed for navigation systems in aircraft while a tactical grade unit can be sufficient for a missile. For examples of navigation grade IMUs, see, e.g., [27, 31]; examples of tactical grade IMUs include [26, 46].

Consumer or *automotive* grade sensors are not capable of autonomous navigation, but can be used for positioning temporarily, e.g., when satellite based positioning is not available, such as when driving through an underpass. Consumer grade sensors, e.g., [17, 59], are primarily installed for other purposes than navigation; examples of applications are given in Sect. 1.3.

Table 1 shows example specifications of different grades of inertial measurement units (IMUs); the values should be regarded as indicative orders of magnitude corresponding to the example devices referenced above, and should not be used as a definition of the different quality levels. Anyway, it is clear that the gap between consumer and navigation grades is large—the differences are in the order of many decades. Misalignment errors have not been specified for consumer-grade units because it is difficult, if not impossible, to separate their misalignment errors from other cross-coupling effects such as inter-channel electromagnetic interference; hence, the total cross-axis sensitivity is given for these IMUs instead. The consumer-grade performance figures represent low-cost bulk-manufactured MEMS sensors that are not individually calibrated by the manufacturer. When considering the size and power consumption of such a MEMS IMU, one needs to account for other

Table 1 Indicative specifications for IMUs of different quality grades

Component	Parameter	Unit	Navigation	Tactical	Consumer
Accelerometer	Pre-calibration bias	mg	0.03	1	30
	Noise density	$\mu g/\sqrt{Hz}$	10	50	100
	Scale factor error	%	0.01	0.03	1
	Misalignment	mrad	0.05	0.5	–
	Cross-axis sensitivity	%	–	–	1
Gyroscope	Pre-calibration bias	°/h	0.005	1	1000
	Bias instability	°/h	0.003	0.1	20
	Angular random walk	$°/\sqrt{h}$	0.002	0.1	0.5
	Scale factor error	%	0.0005	0.01	1
	Misalignment	mrad	0.01	0.5	–
	Cross-axis sensitivity	%	–	–	1
IMU assembly[a]	Weight	kg	5	1	0.01
	Volume	cm^3	1500	500	0.01
	Power consumption	W	10	5	0.01

[a]The figures given for MEMS IMUs correspond to the sensor chip only

necessary components such as the circuit board and readout electronics in addition to the sensor chip itself; these are not included in the example figures given for a consumer-grade IMU in Table 1. Nevertheless, it is not challenging to build a MEMS IMU into a package with size in the order of a few cubic centimeters.

When considering the performance parameters and requirements of sensors, it is important to distinguish between errors before calibration and residual errors [55]. For instance, the large bias of a consumer gyroscope can be mostly compensated for by frequent calibration (e.g., whenever the IMU is stationary), but the bias instability ultimately determines the attainable performance. On the other hand, with high-quality IMUs it may be possible to calibrate out misalignment errors to an accuracy better than the physically achievable sensor alignment precision.

3 Pedestrian Dead Reckoning

The term *dead reckoning* (DR) refers to the method where a new position estimate is computed by adding measured or estimated displacements to the coordinates of a known starting point. Inertial sensors are well known devices for providing the information on the direction and the distance traveled.

In inertial navigation, the data from three accelerometers and three gyroscopes are used to update position estimates. As described in Sect. 1.3.1, position estimation with INS involves the integration of gyroscope measurements to keep track of the attitude of the sensor unit, followed by double integration of acceleration measurements to obtain the velocity and position. The process of maintaining the attitude estimate and integrating the accelerations is called the strapdown INS mechanization. In this section, we will shortly discuss about the INS mechanisation and its challenges. This is followed by the detailed description of Pedestrian Dead Reckoning (PDR) and its accuracy analysis.

3.1 INS Mechanization

The traditional Inertial Navigation System (INS) mechanization includes the following tasks [60]:

1. Integration of the outputs of gyros to obtain the attitude of the system in the desired coordinate reference frame
2. Using the obtained attitude of the system, transformation of the specific force measurements to the chosen reference frame
3. Computing the local gravity in the chosen reference frame and adding it to the specific force to obtain the device acceleration in space
4. If required by the chosen reference frame, the Coriolis correction is applied
5. Double-integration of the acceleration to obtain the velocity and the position of the device

For the first task, parameterization for rotations in three-dimensional space is required. The ones selected in here are direction cosine matrix $\mathrm{C}_{\mathrm{A}_2}^{\mathrm{A}_1}$ and rotation vector $\mathbf{p}$ with notation from [54]. Many other attitude parameterizations can be used [45]. For example, identical presentation would be possible by switching direction cosine matrices to quaternions. A 3×3 direction cosine matrix transforms a 3×1 vector from reference frame A_2 to frame A_1

$$\mathrm{C}_{\mathrm{A}_2}^{\mathrm{A}_1}\mathbf{v}^{\mathrm{A}_2} = \mathbf{v}^{\mathrm{A}_1} \tag{6}$$

The rotation vector $\mathbf{p}$ defines an axis of rotation and its magnitude defines an angle to be rotated. Similarly as direction cosine matrix, rotation vector can be used to define attitude between frames A_2 and A_1. If frame A_1 is rotated about the rotation vector $\mathbf{p}$ through the angle $p = \sqrt{\mathbf{p}^T\mathbf{p}}$ the new attitude can be uniquely used to define frame A_2. Conversely, for arbitrary frames A_2 and A_1 we can find rotation vector that defines the relative attitude, although not uniquely. The relationship between direction cosine matrix and rotation vector is [5]

$$\mathrm{C}_{\mathrm{A}_2}^{\mathrm{A}_1}(\mathbf{p}) = \begin{cases} \mathrm{I} + \frac{\sin(p)}{p}(\mathbf{p}\times) + \frac{1-\cos(p)}{p^2}(\mathbf{p}\times)(\mathbf{p}\times) & \text{if } p \neq 0 \\ \mathrm{I} & \text{otherwise} \end{cases} \tag{7}$$

and this can be used to transform any rotation vector to uniquely defined direction cosine matrix. In Eq. (7) $(\mathbf{p}\times)$ denotes 3×3 skew symmetric form of 3×1 vector $\mathbf{p}$.

In inertial navigation the orientation estimation beings with finding an initial orientation $\mathrm{A}_{\mathrm{t}=0}$ of the sensor unit with respect to some locally level frame L. Then gyro triad measurements $\boldsymbol{\omega}_{\mathrm{IA_t}}^{\mathrm{A_t}}$ which satisfy

$$\dot{\mathrm{C}}_{\mathrm{A_t}}^{\mathrm{A_0}} = \mathrm{C}_{\mathrm{A_t}}^{\mathrm{A_0}}(\boldsymbol{\omega}_{\mathrm{IA_t}}^{\mathrm{A_t}}\times) \tag{8}$$

can be used to update the orientation. In Eq. (8) I refers to inertial (non-accelerating, non-rotating) reference frame. With sufficiently short time update interval dt an approximation $\mathbf{p}_t \approx \boldsymbol{\omega}_{\mathrm{IA_t}}^{\mathrm{A_t}} dt$ can be used and then Task 1 is completed by updating $\mathrm{C}_{\mathrm{A_t}}^{\mathrm{L}}$ at each computer cycle:

$$\mathrm{C}_{\mathrm{A_t}}^{\mathrm{L}} \leftarrow \mathrm{C}_{\mathrm{A_{t-1}}}^{\mathrm{L}}\mathrm{C}_{\mathrm{A_t}}^{\mathrm{A_{t-1}}}(\mathbf{p}_t) \tag{9}$$

In Task 2 the accelerometer triad measurement

$$\mathbf{a}_{\mathrm{SF}}^{\mathrm{A_t}} = \ddot{\mathbf{r}}^{\mathrm{A_t}} - \mathbf{g}^{\mathrm{A_t}}, \tag{10}$$

is transformed to L frame using Eq. (9), which leads to differential equation for position to be solved

$$\ddot{\mathbf{r}}^{\mathrm{L}} = \mathrm{C}_{\mathrm{A_t}}^{\mathrm{L}}\mathbf{a}_{\mathrm{SF}}^{\mathrm{A_t}} + \mathbf{g}^{\mathrm{L}}, \tag{11}$$

where $\mathbf{g}^{\mathrm{L}}$ is result from Task 3. Solving Eq. (11) completes Task 5. In this compact introduction the Task 4 was neglected in Eq. (9). In the double-integration of accelerations even a small error in acceleration measurement yields a large position error drift in the output. Because the accelerometers measure the specific force instead of the true acceleration of the sensor unit, as explained in Sect. 1.1, the gravitational acceleration is added to the vertical acceleration component; this is straightforward when the accelerations are first transformed to a local level frame (Eq. (11)). However, because the gravity compensation of accelerations require the coordinate transformation, any error in gyroscope output causes errors in the transformed accelerations, which in turn introduces increasing errors to the computed accelerations through the errors in the gravity compensation. As the gyro outputs are integrated to form the coordinate transformation and the transformed accelerations are double-integrated for position estimate, the gyro errors produce a position error which increases with time cubed. Therefore the gyro performance is very critical in INS implementations. Effect of gyro errors can be reduced with GNSS integration but this is quite difficult with consumer-grade sensors due to linearization problems [42].

As the requirements for sensor accuracies are very strict for the strapdown INS mechanization, requiring very high-quality and expensive sensor units, the developers of mass-market applications are looking for solutions where multiple integrations of sensor errors can be avoided. In pedestrian applications, the cyclic nature of the human gait can be utilized to enable navigation with low-cost inertial sensors. Two approaches have become popular in the literature: mounting the sensors to the user's shoe and evaluating the INS mechanization equations in a stepwise manner; and Pedestrian Dead Reckoning (PDR) where the position estimate is propagated by detecting steps and estimating their length, and keeping track of the heading using body-mounted sensors.

The concept of foot-mounted inertial navigation hinges on the idea that when the sensor unit is known to be stationary, the velocity errors can be observed [19]; this condition holds regularly for a pedestrian's foot when walking. In addition to resetting velocity this allows to estimate and compensate for other errors that are correlated with the velocity errors, e.g. position and attitude offsets and sensor biases. The most important benefit of foot-mounted inertial navigation is the fact that it is insensitive to the direction of the step and gait characteristics as long as the foot stance periods can be properly detected. However, detecting the stance phase is not trivial especially when the user is running or moving in stairs [50, 57]. In addition, the foot is subject to higher dynamics than the rest of the body; the sensors are subject to a significant shock whenever the foot hits the ground, which can lead to temporary measurement errors.

In PDR, instead of double-integration of the accelerations, the speed of the walk is estimated from the periodical acceleration waveform produced by pedestrian movements. The speed can be estimated either from the main frequency of the periodic signal or by detecting individual steps and estimating their lengths and

durations from the acceleration waveform. This information along with estimated heading is used to propagate the estimate of user position. It can be shown that PDR mechanization is superior to the traditional INS mechanization for a person on foot when the sensors are mounted on the user's torso [44]. The main drawback of PDR is the limitation to one motion mode; the mechanization works only when walking while the general strapdown INS mechanization works without any assumptions about the user motion. In addition, while foot-mounted inertial navigation is 3-dimensional by nature, PDR is 2-dimensional and requires height information from other sources such as map [66] or barometric altimeter.

3.2 Step Detection with Accelerometers

In this section step detection with torso mounted sensors are considered in detail. With *step* we mean the displacement of one foot during walking movement, i.e. the distance between two consecutive foot prints. The occurrence of a step can be easily detected from the signal pattern of the vertical acceleration component [40]. However, this approach is sensitive to orientation errors of the sensor unit, as it is assumed that one axis is aligned with vertical or that the transformation to the vertical is known. Other possibility it to compute the magnitude of the measured acceleration vector, i.e. the norm of acceleration [33]. Most commonly the step detection is based on accelerometers but also gyroscopes can be used [14]. The signal pattern varies according to where the user attaches the sensor unit [38]. Typical choices to wear the sensor unit are on the belt, e.g. on the side of the user or on lower back, or onto upper parts of the torso, e.g. attach it to the shoulder strap of a backpack or wear it in a chest pocket. Step detection is often based on the detection of signal peaks [38] or crossings of the signal with its average [33] or some other reference level [43]. Often the detection algorithm combines both peak detection and detection of reference level crossings. For example, step detection from acceleration norm may consists of the following steps:

1. Low pass filtering and resampling the signal; sampling frequency in the range 20–25 Hz is high enough.
2. Computation of the norm of current acceleration sample, i.e.,

$$a_n(t) = \sqrt{a_x^2(t) + a_y^2(t) + a_z^2(t)}, \tag{12}$$

   where $a_n(t)$ is the acceleration norm and $a_x(t)$, $a_y(t)$, and $a_z(t)$ are the filtered components of the measured acceleration.
3. Instances of step starts $t_s(k)$ are detected by observing the g-crossings of the acceleration norm that are followed by a rise rate and a peak height that exceed the preset limits, and requiring that the time between the current and previous g-crossings is long enough.

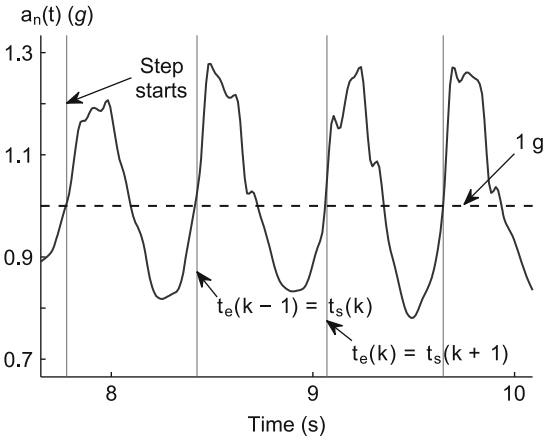


Fig. 4 Detection of steps from acceleration norm

4. The step end $t_e\,(k)$ is considered to be found when the next step starts or when a predefined time, considered as the maximum duration of one step, has passed after the start of the current step.

An example with acceleration norm and the detected step starts is shown in Fig. 4. The data for the figure were recorded using a sensor unit that was attached to the belt and positioned to the back of the test walker. Other methods that can be used to detect individual steps include the correlating of sensor signal with predefined stride template [7]. The template is formed offline, e.g., by recording it from sample walk [25]. The correlation method can be improved by using dynamic time warping (DTW) which allows non-linear mapping between the template and the online signal [52].

There are applications and devices, such as mobile phones, where the orientation of the sensor unit cannot be assumed to be predetermined and constant. If the methods for step detection and step length estimation require e.g. vertical acceleration component, the phone orientation need to be tracked or the motion classification can be used to allow adapting different algorithms for different motion modes [13].

3.3 Step Length Estimation

There are two main categories for methods to estimate step length. The first category includes models that are based on the biomechanical principles whereas the models in the second category are based on empirical relationships between acceleration signal pattern and step length. With biomechanical models, certain user-related parameters, such as leg length, are needed in addition to the empirically determined scaling parameters [32]. In empirical models, the acceleration norm $a_n(t)$ or the

vertical acceleration component $a_v(t)$ are typically used for step length estimation. The signal patterns that have been found to correlate well with step length include the following:

$$\text{Main frequency} \quad p_1(k) = 1/(t_e(k) - t_s(k)) \tag{13}$$

$$\text{Variance, } a_n \quad p_2(k) = \text{var}(a_n(t)), \quad t_s(k) \le t < t_e(k) \tag{14}$$

$$\text{Variance, } a_v \quad p_3(k) = \text{var}(a_v(t)), \quad t_s(k) \le t < t_e(k) \tag{15}$$

$$\text{Area integral} \quad p_4(k) = \int_{t_s(k)}^{t_e(k)} |a_n(t) - g| \, dt \tag{16}$$

$$\text{Maximum difference, } a_n \quad p_5(k) = \max a_n(t) - \min a_n(t), \quad t_s(k) \le t < t_e(k) \tag{17}$$

$$\text{Maximum difference, } a_v \quad p_6(k) = \max a_v(t) - \min a_v(t), \quad t_s(k) \le t < t_e(k) \tag{18}$$

Instead of (13), the main frequency of the periodical signal can be obtained using Fast Fourier Transformation (FFT) [38, 40]. In (14)–(15) the variance of the acceleration signal (e.g., norm or vertical component) is computed over a time window comparable to some step durations [38], e.g. over one step. The area integral (16) is obtained by integrating over one step duration the absolute value of the acceleration norm where the local gravity has been subtracted [33]. In (17)–(18) the difference between the maximum and minimum acceleration (e.g., norm or vertical component) of a detected step is used [32].

Also the use of combinations of these signal patterns has been proposed [32, 38], as well as slightly different patterns from these [43]. The empirical step length model often includes at least one empirically determined parameter. In many cases a non-linear function, such as raising to a power or extraction of root, has to be applied to the signal pattern. It is also common to add constant offsets to the pattern or the function [23, 38]. A generic form of the step length model can be written as

$$\Delta s_k = K_{j,q} \; p_j(k)^q + b \tag{19}$$

where Δs_k is the distance traveled and $p_j(k)$ is the signal pattern, both computed for the kth step. $K_{j,q}$ is the scaling factor, b is the offset, and q is the exponent that defines the function to be applied on p_j. The performance of step length estimation with different functions applied on different signal patterns were demonstrated with real pedestrian data in [11]. With the best combinations, the relative error in the estimated distance traveled was 2–3%.

The step length models discussed here are applicable in flat floor or terrain. In stairs, the step length is forced to be shorter. A method based on analysis of accelerometer and gyro signal patterns can be used to detect forward direction and going up or down in stairs [36].

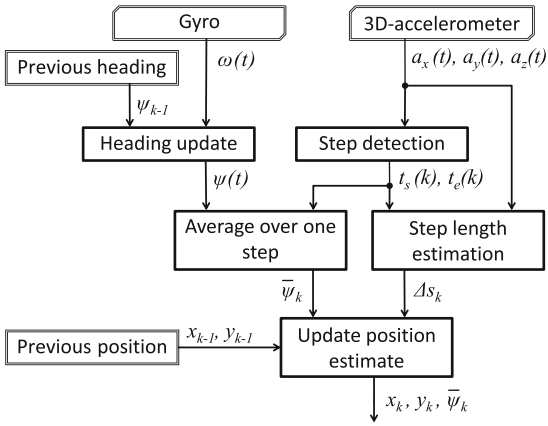


Fig. 5 Block diagram of the PDR algorithm

3.4 PDR Mechanization

In PDR mechanization, the dead reckoning process involves step detection and step length estimation, as shown in the diagram of Fig. 5. The PDR position estimate is computed by starting from initial coordinates, x_0, y_0, and initial heading angle ψ_0. As the DR method is not able to determine absolute positions, these initial estimates have to determined using alternative positioning methods, such as radio navigation or satellite based positioning.

While the position in PDR algorithm is updated only when step ends are detected, the heading is updated every Δt_g seconds, i.e., at the sampling frequency of the gyro:

$$\psi_\lambda = \psi_{\lambda-1} + \omega_\lambda \Delta t_g, \tag{20}$$

where ω_λ is the angular rate measurement by the gyro at the sampling instance $\lambda \Delta t_g$. In position estimation, a heading estimate representative of the whole step duration is needed. Therefore the heading is averaged over the step duration:

$$\bar{\psi}_k = \frac{1}{n_k} \sum_{\lambda \in \Lambda_k} \psi_\lambda, \qquad \Lambda_k = \left\{ \lambda : \lambda \text{ is an integer}, \frac{t_s(k)}{\Delta t_g} \leq \lambda < \frac{t_e(k)}{\Delta t_g} \right\}, \tag{21}$$

where n_k is the number of samples in Λ_k. The heading and horizontal coordinates are propagated by

$$\begin{aligned} x_k &= x_{k-1} + \Delta s_k \cos \bar{\psi}_k \\ y_k &= y_{k-1} + \Delta s_k \sin \bar{\psi}_k, \end{aligned} \tag{22}$$

where Δs_k is the estimated step length, i.e., the distance traveled during the step with index k (Fig. 6). Position estimates that are based on step detection and step length estimation are available at step intervals Δt_k, which vary according to the walking style and the speed of the pedestrian.

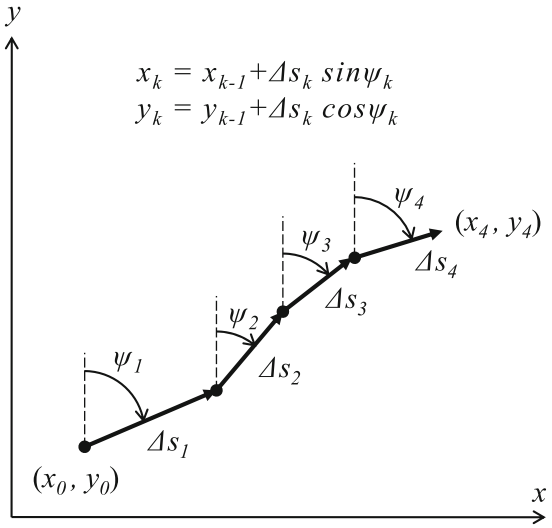


Fig. 6 Dead reckoning in two dimensions

The orientation of the sensor unit with respect to the direction of pedestrian travel is not fixed in smart phones and many other mobile devices. To determine the step direction, the knowledge about the orientation of the device with respect to the environment is required but it is not enough [13, 36]. Methods to estimate the unknown alignment between the mobile device and the pedestrian (and the step direction) are compared in [12].

3.5 Effect of Sensor Quality Grade to the Accuracy of PDR

Although PDR mechanization is not as sensitive to sensor errors as the traditional INS mechanization, the grade of sensors still has an effect to the performance of the PDR. In this section, the accumulation of errors in PDR is studied based on simple test cases.

From (13)–(18) it can be seen that the step length estimate is not sensitive to accelerometer bias: in p_2, p_3, p_5, and p_6 the bias is totally canceled out and in p_1 and p_4 its effect is small. Contrary to the bias error, the effect of the scale factor error on all other signal patterns except p_1 is directly proportional to the sensor error. However, taking square root, cube root or the fourth root of the signal pattern decreases the effect of accelerometer scale factor error on the step length estimate, as can be seen in Table 2.

If the scale factor error of the accelerometer is constant, its effect can be taken into account in the scaling factor of the step length model (19). In practice the scale factor error of a consumer grade accelerometer based on MEMS technology is slowly changing as a function of internal conditions of the sensing element, such as

Table 2 Effect of 1% scale factor error in accelerometer to functions of signal patterns for step length estimation

Function	Raw $p_j(k)$	Square root $p_j(k)^{1/2}$	Cube root $p_j(k)^{1/3}$	Fourth root $p_j(k)^{1/4}$
Step length error (%)	1.00	0.50	0.33	0.25

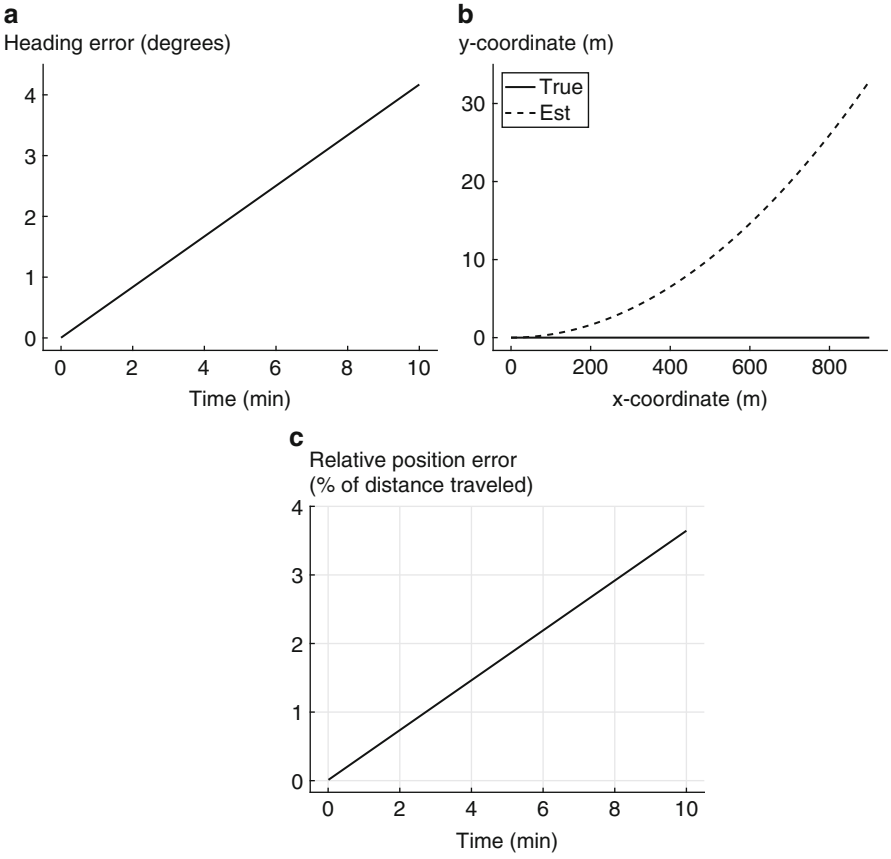


Fig. 7 Effect of 25°/h gyro bias when the pedestrian is walking with constant speed along the positive x-axis: (**a**) heading error; (**b**) true and estimated coordinates; (**c**) relative position error

the temperature. If the temperature effect on the sensor scale factor at its maximum is 1%, then the effect on the estimated distance traveled is the same as the relative error of the evaluated function (Table 2) at the most. These values are small when compared with step length modeling errors reported in literature [11, 32].

The effect of the gyro quality to PDR estimates can be analyzed by the simulation of a PDR system defined by (20)–(22). The effect of the gyro bias is simulated by using a scenario where the pedestrian walks with constant step length of 0.75 m and

Table 3 Comparison of gyro grade with respect to the effect of uncompensated bias to the PDR error build-up

	Navigation	Tactical	Consumer
Bias instability (°/h)	0.0035	1	25
Time to 2% relative position error	27 days	2.3 h	5.5 min
Time to 3% relative position error	41 days	3.4 h	8.3 min
Time to 3° heading error	35 days	3.0 h	7.2 min

constant frequency of 2 steps/s along the positive x-axis. The gyro bias is assumed to be 25°/h, which is a typical bias instability of consumer grade gyros (Table 1). The development of heading error, error in estimated position and the position error relative to the distance traveled is shown in Fig. 7. The heading error grows linearly (Fig. 7a), the error in the y-coordinate grows quadratically[4] with respect to the x-coordinate and time (Fig. 7b), and the relative position error with respect to the distance traveled grows almost linearly (Fig. 7c). With the best step length models, the long term average in the relative positioning error is about 2–3% [11]. With the given simulation parameters, the relative positioning error introduced by the gyro bias is smaller in the beginning, but exceeds 2% in less than 6 min and 3% in less than 9 min.

To compare the gyro grades described in Table 1, the simulations were also run with gyro instabilities typical to navigation and tactical grade gyros. The results are shown in Table 3.

The effect of the gyro scale factor error is simulated by using a scenario where the pedestrian first makes a 180° turn and then walks with a constant step length of 0.75 m and a constant frequency of 2 steps/s along the positive x-axis. The gyro scale factor error is assumed to be 1%, which corresponds to the scale factor uncertainty due to the temperature sensitivity over 50 K in a consumer grade gyro [6]. The heading error, the error in the estimated position, and the position error relative to the distance traveled are shown in Fig. 8. In this simulation, the heading error grows in the turn to 1.8° and then stays constant, as the scale factor error has an effect only when the gyro senses a non-zero anular rate (Fig. 8a). Due to the constant heading error, the position error grows linearly with respect of time and x-coordinate (Fig. 8b). In the initial turn, the position error relative to the distance traveled jumps directly to more than 3% (Fig. 8c). That is, with the parameters used in this simulation and after a 180° turn, the error due to the gyro scale factor error is larger than the error introduced by the best step length models in [11].

To compare the gyro grades described in Table 1, the simulations were also run with gyro scale factor errors typical to navigation and tactical grade gyros. The results are shown in Table 4.

[4]The growth is almost quadratic with small heading errors; however, with larger heading errors, the sine and cosine functions in (22) bound the error growth.

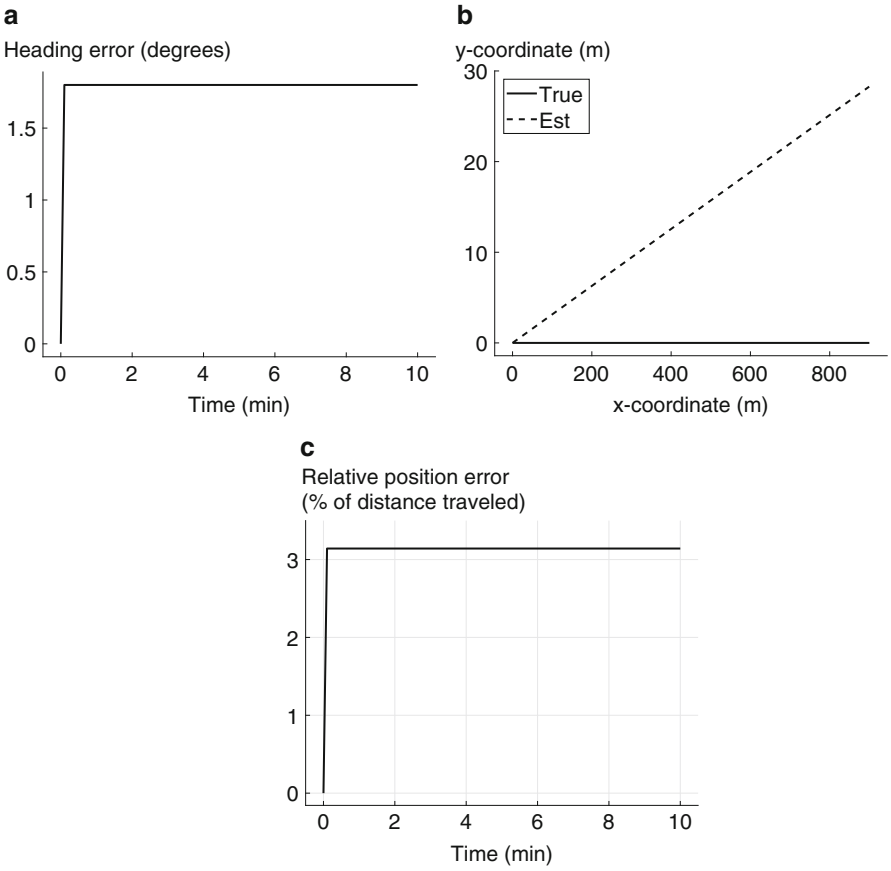


Fig. 8 Effect of 1% gyro scale factor error when the pedestrian has made a 180° turn from −180° and then walks with constant speed along positive x-axis: (**a**) heading error; (**b**) true and estimated coordinates; (**c**) relative position error

Table 4 Comparison of gyro grade with respect to the effect of uncompensated scale factor error to the PDR error build-up

	Navigation	Tactical	Consumer
Scale factor error (%)	0.001	0.015	1
Constant heading error (degrees)	0.0018	0.027	1.8
Constant relative position error (%)	0.00314	0.047	3.14

It should be noted that the simulation results given in this section apply only on PDR mechanization of inertial sensors. The growth of position error is much faster with traditional INS mechanization, partly due to the low speed of the pedestrian and partly due to the algorithm simplifications allowed by the characteristics of pedestrian movements. Another important remark considers the effect of the tilt

error of the heading gyro: the simulations assume that the sensitive axis of the gyro is aligned with vertical. However, in practice the sensor unit easily gets tilted by a couple of degrees, which introduces a scaling error to the gyro output.

4 Infering Context with Inertial Sensors

In addition to providing data for navigation purposes, inertial sensors can be used to increase the context awareness of a device. One widely used application is motion mode classification. In Fig. 9 the waveform of the norm of accelerometer measurements, as defined in Eq. (12), is shown. The different characteristics in waveform depending on motion mode is clearly seen. When walking, foot impacts clearly increase the variability of the signal. When driving a car, engine vibrations, vehicle accelerations, and road imperfections cause variations which are smaller than those occurring during walking. Yet, these variations are distinguishable from the case of a stationary device where the only source of variation is measurement noise.

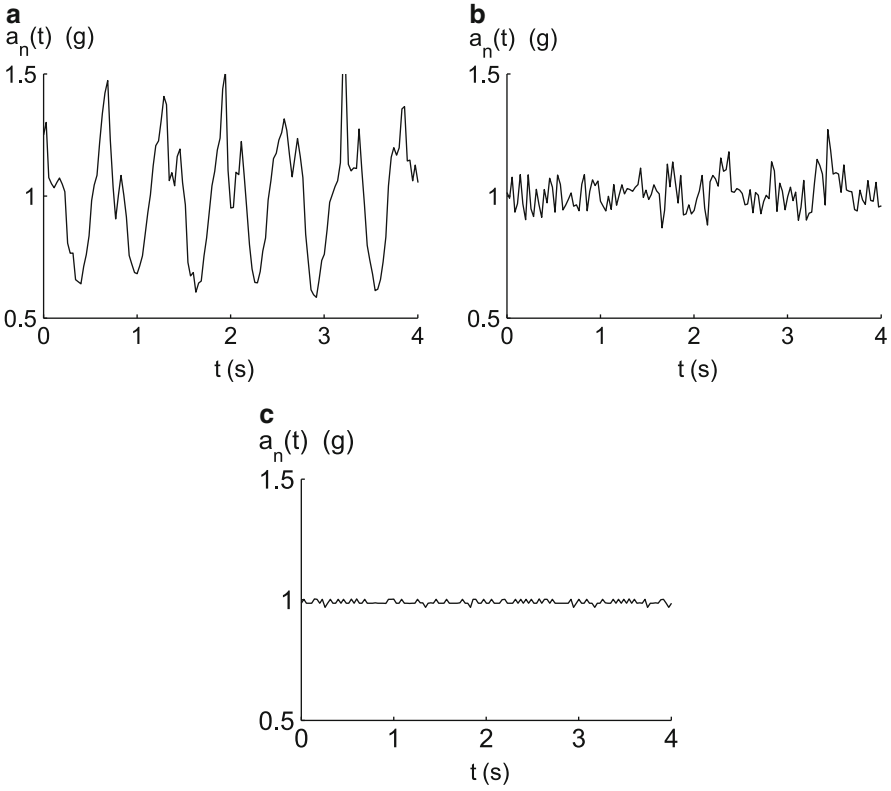


Fig. 9 Norm of accelerometer output in different motion modes: (**a**) walking, $\sigma = 0.24$ g; (**b**) driving, $\sigma = 0.071$ g; (**c**) stationary, $\sigma = 0.0084$ g

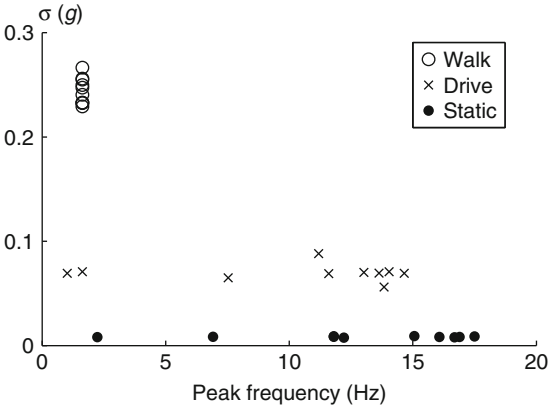


Fig. 10 Standard deviation and peak frequency as features

In order to have a computer to identify these motion modes, features such as sample variance or peak frequency need to be extracted from the acceleration data. Figure 10 shows two such features: the sample standard deviation σ and the peak frequency from non-overlapping 5-s windows. In this example, the classification is relatively easy, as the characteristics are clearly distinguishable and there is only one label to learn. In practice, the classification problems are more complex, with overlapping features and multiple labels [69]. Thus, proper algorithms and statistical tools are needed to obtain useful classification results. In this section a brief introduction to such tools is given.

4.1 Pattern Recognition

As a simplified statistical example, pattern recognition problem can be considered as discrimination between r multivariate normal populations. The Bayes theorem is applied to obtain the probability of the originating population class (e.g. motion mode) given the statistics (e.g. features) obtained from the sensor data. A training data set with labeled motion modes is needed to obtain the class means μ_j and covariances Σ_j for each class j. Then, according to the model, the future observations collated to a q-dimensional feature vector $\mathbf{z}$ are distributed as

$$\mathbf{z}_j \sim N(\mu_j, \Sigma_j). \tag{23}$$

It should be stressed that due to limited size of the training data set the mean vector $\mu_j \in \mathbb{R}^q$ and the covariance matrix $\Sigma_j \in \mathbb{R}^{q\times q}$ are actually estimates of the true model parameters. Further simplification is made by assuming that the prior probability $P(C = j)$, where $C = j$ denotes an event that the correct class is j is known. Under these assumptions, Bayes' theorem can be applied to obtain

$$P(C = j|\mathbf{z}) = \frac{p_{\mathbf{z}_j} P(C = j)}{p_{\mathbf{z}}}, \tag{24}$$

where

$$p_{\mathbf{z}} = \sum_{i=1}^{r} p_{\mathbf{z}_i} P(C = i). \tag{25}$$

The actual classification result is obtained by finding the class that maximizes the posterior probability $P(C = j|\mathbf{z})$. Adding inference for sequential data can be done, for example, using Markov model for state transition probabilities $P(C_t = h)|P(C_{t-1} = k)$ for all possible states $h, k = 1, \ldots, r$. In practice, the assumption that distributions and correlations between features are known is often invalid as the feature set may include binary features, multimodally distributed features and Wishart distributed features (due to sampling in training phase). Thus, in modern machine learning more generalizable and scalable methods, such as gradient tree boosting are popular [10]. Even though the new methods in machine learning require less assumptions for the inputs, there is still a need to understand what kind of data and features should be included. When inertial sensors are used for classification there are many options for feature engineering if the basic principles of inertial sensors are understood well. Features can be extracted from raw data (angular rates, specific force) or from integrated data (position, velocity, orientation). When characteristics of sensor noise are identified the effectiveness of high frequency versus low frequency features may become apparent. Such examples of advanced features are given in the following section.

4.2 Feature Extraction

Two very important features for classification of motion modes shown in Fig. 10 were examples of statistical (variance) and frequency domain (peak frequency) features. Using windowed raw sensor data there are many other features easily obtainable such as [16]:

- Skewness
- Mean absolute deviation
- Zero-crossing rate
- Sub band energies and their ratios
- Change in the peak frequency over 4 sub-frames
- Frequency domain entropy

To make the classification more efficient, there exist efficient algorithms that can be used reduce the dimensionality of feature space by utilizing the correlation between features [65].

To show how knowledge of inertial navigation theory may help in classification the effect known as coning is introduced. The relation between gyroscope measurements and device orientation with fast processing rate was shown in Eqs. (7) and (9). However, the exact relation between rotation vector and gyroscope measurements is [5]

$$\dot{\mathbf{p}} = \boldsymbol{\omega}_{\mathrm{IA_t}}^{\mathrm{A_t}} + \frac{1}{2}\mathbf{p} \times \boldsymbol{\omega}_{\mathrm{IA_t}}^{\mathrm{A_t}} + \frac{1}{p^2}(1 - \frac{p\sin(p)}{2(1-\cos(p))})\mathbf{p} \times (\mathbf{p} \times \boldsymbol{\omega}_{\mathrm{IA_t}}^{\mathrm{A_t}}) \tag{26}$$

and the last two terms, describing non-commutativity rate, begin to play a role if the attitude update rate is too slow. Important feature in this equation is that the cross product terms remain zero if gyro signal vector ($\boldsymbol{\omega}_{\mathrm{IA_t}}^{\mathrm{A_t}}$) keeps its direction. If the gyro signal is constant or the object can rotate only about one fixed axis, then there is no problem of non-commutativity. The problem arises if gyro data is averaged, assuming

$$\dot{\mathbf{p}} \approx \boldsymbol{\omega}_{\mathrm{IA_t}}^{\mathrm{A_t}} \tag{27}$$

and the true rotation is lost due to non-commutativity of rotations. Typically in inertial sensor processing this is avoided by performing the direction cosine update with fast rate with respect to motion (or applying coning correction terms). In this context we loosely define the error due to approximation in Eq. (27) as coning motion. To see why this is important in motion classification, consider following scenarios for time period $n \rightarrow m$

- Unit is in smartwatch attached to wrist of a pedestrian
- Unit is fixed to a vehicle that is cornering
- Unit is stationary on table, gyros have constant bias

Gyroscope data samples at 20 Hz from these scenarios is shown in Fig. 11. To see the effect of coning errors, this data is resampled by averaging to 1 Hz and maximum angle error with respect to 20 Hz reference is plotted in Fig. 12. By combining amount of coning error in each case we will see that first example has quite large non-commutativity rate, vehicular motion clearly less and the in the static case the coning error is negligible. The coning effect computed this way is a direct measure of complexity of angular motion experienced, and thus an useful, acceleration independent feature for motion mode classification. It may also help obtaining more insight on how the user experiences the motion [61]. It should be noted that orientation with fast rate is already computed by the inertial processing algorithm, so the only extra work for deriving this feature is to take direct average of gyro data, multiply it by time interval and apply it in Eq. (7) ($\mathbf{p} \leftarrow \frac{n-m}{N}\sum\omega$). Extension this method to specific coning correction algorithms [30] and accelerometer processing (velocity rotation compensation) is also straightforward. This illustrates the importance of feature engineering in machine learning, to build effective feature it is important to know what the sensors actually measure.

Combination of bias and moderate non-coning motion (such as in vehicle mode) may also result in large coning error. This is because direction of apparent rotation vector is changing when another component changes its magnitude. Thus the motion mode recognition and orientation estimation are not necessarily independent tasks. The quality of sensors affects input features, but on the other hand, known motion mode can be used to infer gyroscope biased, for example.

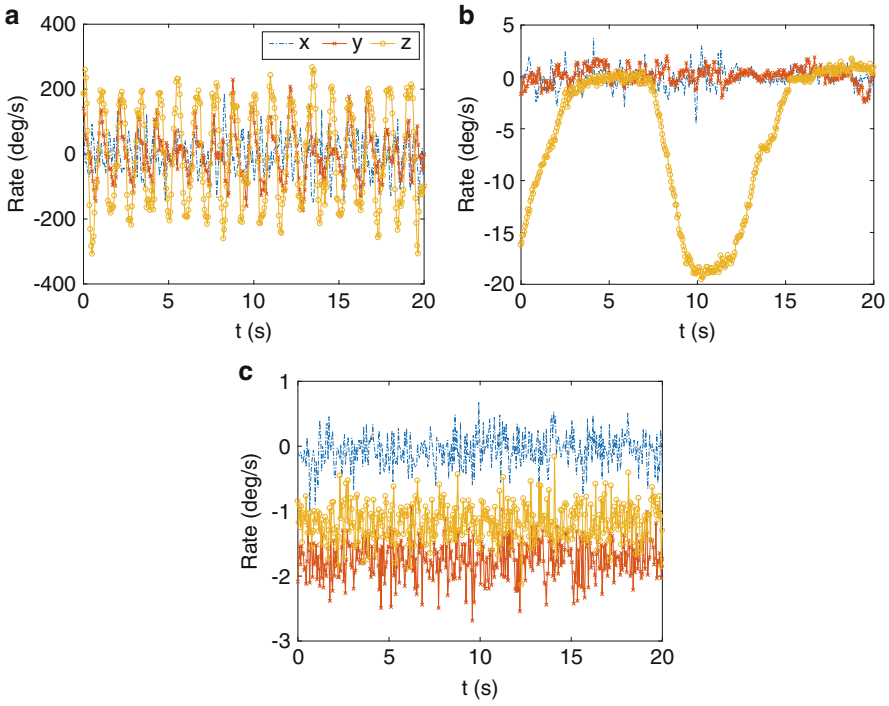


Fig. 11 Gyro data in different cases: (**a**) Smartwatch; (**b**) Driving; (**c**) Biased, stationary. Note the y-axis scales

4.3 Classification Accuracy

In practice it is impossible to implement a classifier that makes no mistakes; misclassifications will occur from time to time. From the viewpoint of the application designer, the classification accuracy can be evaluated for two different cases: (a) the expected misclassification rate prior to observing the features, and (b) the probability of misclassification given the observed feature vector. For the former, the overlap in the training data is a good indicator. For the latter, (24) directly gives such probability, but as mentioned the multinormal model for features is rarely valid. Often the system designer has no other choice than to collect sufficient amount of independent data for cross-validation to obtain realistic values for misclassification rates. In addition, one approach to tolerate misclassification is to apply partial classification methods, where the option of not classifying a situation at all is reserved [8]. In motion mode classification the number of classes can vary a lot, which affects the classification accuracy, but generally the reported misclassification error rate is almost always below 10% [16].

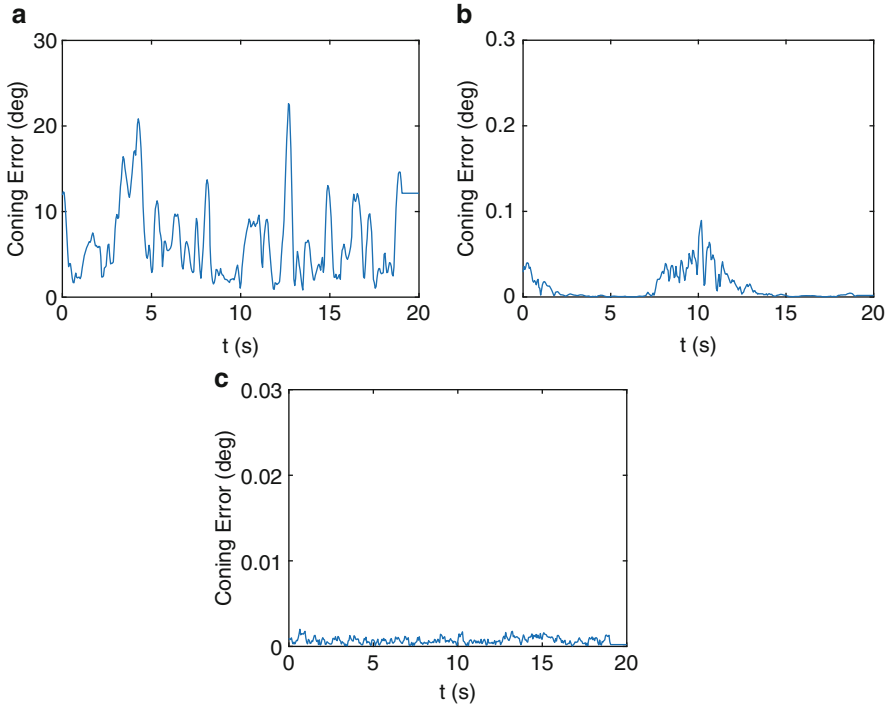


Fig. 12 Coning error in degrees in the three cases: (**a**) Smartwatch; (**b**) Driving; (**c**) Biased, stationary. Note the y-axis scales

4.4 Areas of Application

Motion mode using sensors in smartphones is already generally available [21] and applications are growing in number continuously. For example, detailed motion mode information would be valuable for remote monitoring of elderly people [62]. The modern machine learning tools such as XGBoost seems to be very effective in this [63]. For navigation applications the detection of Walking-mode allows using PDR, and many other context-dependent mechanizations or filter profiles have been proposed [15]. The sequential nature of navigation problem has to be taken into account in recognition [16, 49]. In principle, motion mode classification methods can be used in any area where human motion is involved and the subjects exhibit distinct signatures [3]. The list of available applications is not limited to human motion mode, as the market for Internet of Things devices is growing in the industrial side as well and general tendency is to include inertial sensors in all kinds devices that are experiencing motion, without forgetting the increasing amount of smartphone applications [68].

5 Summary

With the development of low-cost MEMS accelerometers and gyroscopes, more and more motion-aware applications become achievable. Since inertial sensors measure motion parameters the input is based on physical properties specific to the application; emerging applications are usually significantly different from the original use of inertial sensors, i.e., navigation. Novel applications typically are less strict in sensor accuracy requirements than traditional inertial navigation systems. However, imprecision may cause the application to perform poorly in certain situations. Common methods to improve the performance is to calibrate the inertial sensors and to filter the sensor data appropriately. Understanding of physical principles of inertial sensor measurements is essential in designing systems that involve motion measurement. In this chapter, an introduction to inertial sensor applications was provided. Such a concise presentation did not permit in-depth treatment of inertial navigation system algorithms and other applications. More information about these topics and future trends can be found, e.g., in [13, 16, 22, 48, 58, 60].

References

1. Aguiar, B., Rocha, T., Silva, J., Sousa, I.: Accelerometer-based fall detection for smartphones. In: Medical Measurements and Applications (MeMeA), 2014 IEEE International Symposium on, pp. 1–6. IEEE (2014)
2. Allan, D.W.: Statistics of atomic frequency standards. Proc. IEEE **54**(2), 221–230 (1966)
3. Altun, K., Barshan, B., Tunçel, O.: Comparative study on classifying human activities with miniature inertial and magnetic sensors. Pattern Recogn. **43**, 3605–3620 (2010)
4. Armenise, M.N., Ciminelli, C., Dell'Olio, F., Passaro, V.: Advances in Gyroscope Technologies. Springer Verlag (2010)
5. Bortz, J.E.: A new mathematical formulation for strapdown inertial navigation. IEEE Transactions on Aerospace and Electronic Systems **AES-7**(1), 61–66 (1971). https://doi.org/10.1109/TAES.1971.310252
6. Bosch Sensortec: BMI160 small, low power inertial measurement unit. rev. 08. Doc.Nr. BST-BMI160-DS000-07, Data sheet (2015)
7. Brajdic, A., Harle, R.: Walk detection and step counting on unconstrained smartphones. In: Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp '13, pp. 225–234. ACM, New York, NY, USA (2013)
8. Broffit, J.D.: Nonparametric classification. In: P.R. Krishnaiah, L.N. Kanal (eds.) Handbook of Statistics 2. North-Holland (1990)
9. Brown, R.G., Hwang, P.Y.C.: Introduction to Random Signals and Applied Kalman Filtering, 3rd edn. John Wiley & Sons (1997)
10. Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, pp. 785–794. ACM, New York, NY, USA (2016). http://doi.acm.org/10.1145/2939672.2939785
11. Collin, J., Davidson, P., Kirkko-Jaakkola, M., Leppäkoski, H.: Inertial sensors and their applications. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems. Springer New York (2013)

12. Combettes, C., Renaudin, V.: Comparison of misalignment estimation techniques between handheld device and walking directions. In: 2015 International Conference on Indoor Positioning and Indoor Navigation (IPIN), pp. 1–8 (2015). https://doi.org/10.1109/IPIN.2015.7346766
13. Davidson, P., Piche, R.: A survey of selected indoor positioning methods for smartphones. IEEE Communications Surveys Tutorials **19**(2), 1347–1370 (2017). https://doi.org/10.1109/COMST.2016.2637663
14. Diaz, E.M., Gonzalez, A.L.M.: Step detector and step length estimator for an inertial pocket navigation system. In: 2014 International Conference on Indoor Positioning and Indoor Navigation (IPIN), pp. 105–110 (2014)
15. Dixon, R., Bobeye, M.: Performance differentiation in a tightly coupled gnss/ins solution. In: Proc. ION GNSS+ 2016 (2016)
16. Elhoushi, M., Georgy, J., Noureldin, A., Korenberg, M.J.: A survey on approaches of motion mode recognition using sensors. IEEE Transactions on Intelligent Transportation Systems (2016). https://doi.org/10.1109/TITS.2016.2617200
17. Fairchild Semiconductor Corporation: FIS1100 6D Inertial Measurement Unit with Motion Co-Processor and Sensor Fusion Library (2016). Data sheet rev. 1.2
18. Farrell, J.: Aided navigation: GPS with high rate sensors. McGraw-Hill, Inc. (2008)
19. Foxlin, E.: Pedestrian tracking with shoe-mounted inertial sensors. IEEE Computer Graphics and Applications **25**(6), 38–46 (2005). https://doi.org/10.1109/MCG.2005.140
20. Gianchandani, Y.B., Tabata, O., Zappe, H.P.: Comprehensive microsystems. Elsevier (2008)
21. Google: DetectedActivity API for Android. https://developers.google.com/android/reference/com/google/android/gms/location/DetectedActivity (2017). [Online; accessed 29-March-2017]
22. Groves, P.: Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems, Second Edition:. GNSS/GPS. Artech House (2013). https://books.google.fi/books?id=t94fAgAAQBAJ
23. Gusenbauer, D., Isert, C., Krösche, J.: Self-contained indoor positioning on off-the-shelf mobile devices. In: 2010 International Conference on Indoor Positioning and Indoor Navigation, pp. 1–9 (2010)
24. Hanning, G., Forslöw, N., Forssén, P.E., Ringaby, E., Törnqvist, D., Callmer, J.: Stabilizing cell phone video using inertial measurement sensors. In: Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on, pp. 1–8. IEEE (2011)
25. Harle, R.: A survey of indoor inertial positioning systems for pedestrians. IEEE Communications Surveys Tutorials **15**(3), 1281–1293 (2013)
26. Honeywell Aerospace, Phoenix, AZ, USA: HG1700 Inertial Measurement Unit (2016). Data sheet N61-1619-000-000 I 09/16
27. Honeywell Aerospace, Phoenix, AZ, USA: HG9900 Inertial Measurement Unit (2016). Data sheet N61-1638-000-000 I 10/16
28. IEEE standard for inertial sensor terminology. IEEE Std 528-2001 (2001)
29. IEEE standard specification format guide and test procedure for single-axis laser gyros. IEEE Std 647-1995 (1996)
30. Ignagni, M.: Optimal strapdown attitude integration algorithms. Journal of Guidance, Control, and Dynamics **13**(2), 363–369 (1990)
31. iMAR Navigation GmbH, St. Ingbert, Germany: iNAT-RQH400x (2017). Data sheet rev. 1.13
32. Jahn, J., Batzer, U., Seitz, J., Patino-Studencka, L., Gutiérrez Boronat, J.: Comparison and evaluation of acceleration based step length estimators for handheld devices. In: Proc. Int. Conf. on Indoor Positioning and Indoor Navigation, pp. 1–6. Zurich, Switzerland (2010)
33. Käppi, J., Syrjärinne, J., Saarinen, J.: MEMS-IMU based pedestrian navigator for handheld devices. In: Proc. ION GPS, pp. 1369–1373. Salt Lake City, UT (2001)
34. Keshner, M.S.: $1/f$ noise. Proc. IEEE **70**(3), 212–218 (1982)
35. Kirkko-Jaakkola, M., Collin, J., Takala, J.: Bias prediction for MEMS gyroscopes. IEEE Sensors J. (2012). https://doi.org/10.1109/JSEN.2012.2185692

36. Kourogi, M., Kurata, T.: Personal positioning based on walking locomotion analysis with self-contained sensors and a wearable camera. In: The Second IEEE and ACM International Symposium on Mixed and Augmented Reality, 2003. Proceedings., pp. 103–112 (2003). https://doi.org/10.1109/ISMAR.2003.1240693
37. Krobka, N.I.: Differential methods of identifying gyro noise structure. Gyroscopy and Navigation **2**, 126–137 (2011)
38. Ladetto, Q.: On foot navigation: continuous step calibration using both complementary recursive prediction and adaptive Kalman filtering. In: Proc. ION GPS, pp. 1735–1740. Salt Lake City, UT (2000)
39. Leland, R.P.: Mechanical-thermal noise in MEMS gyroscopes. IEEE Sensors J. **5**(3), 493–500 (2005)
40. Levi, R.W., Judd, T.: Dead reckoning navigational system using accelerometer to measure foot impacts. U.S. Patent 5,583,776 (1996)
41. Lightman, K.: Silicon gets sporty. IEEE Spectrum **53**(3), 48–56 (2016)
42. Martin, H., Groves, P., Newman, M.: The limits of in-run calibration of mems inertial sensors and sensor arrays. Navigation **63**(2), 127–143 (2016). http://dx.doi.org/10.1002/navi.135. Navi.135
43. Meriheinä, U.: Method and device for measuring the progress of a moving person. U.S. Patent 7,962,309 (2007)
44. Mezentsev, O., Collin, J., Lachapelle, G.: Pedestrian Dead Reckoning – A Solution to Navigation in GPS Signal Degraded Areas. Geomatica **59**(2), 175–182 (2005)
45. Nitschke, M., Knickmeyer, E.H.: Rotation parameters-a survey of techniques. Journal of surveying engineering **126**(3), 83–105 (2000)
46. Northrop Grumman LITEF GmbH, Freiburg, Germany: LCI-100C Inertial Measurement Unit (2013). Data sheet
47. Pellman, E.J., Viano, D.C., Withnall, C., Shewchenko, N., Bir, C.A., Halstead, P.D.: Concussion in professional football: helmet testing to assess impact performance–part 11. Neurosurgery **58**(1), 78–95 (2006)
48. Prikhodko, I.P., Zotov, S.A., Trusov, A.A., Shkel, A.M.: What is mems gyrocompassing? comparative analysis of maytagging and carouseling. Journal of Microelectromechanical Systems **22**(6), 1257–1266 (2013). https://doi.org/10.1109/JMEMS.2013.2282936
49. Read, J., Martino, L., Hollmén, J.: Multi-label methods for prediction with sequential data. Pattern Recognition **63**, 45–55 (2017). http://dx.doi.org/10.1016/j.patcog.2016.09.015. http://www.sciencedirect.com/science/article/pii/S0031320316302758
50. Ren, M., Pan, K., Liu, Y., Guo, H., Zhang, X., Wang, P.: A novel pedestrian navigation algorithm for a foot-mounted inertial-sensor-based system. Sensors **16**(1) (2016). https://doi.org/10.3390/s16010139
51. Roetenberg, D., Luinge, H., Slycke, P.: Xsens MVN: Full 6DOF human motion tracking using miniature inertial sensors. Tech. rep., Xsens Motion Technologies BV (2009)
52. Rong, L., Zhiguo, D., Jianzhong, Z., Ming, L.: Identification of individual walking patterns using gait acceleration. In: 2007 1st International Conference on Bioinformatics and Biomedical Engineering, pp. 543–546 (2007)
53. Rothman, Y., Klein, I., Filin, S.: Analytical observability analysis of ins with vehicle constraints. Navigation **61**(3), 227–236 (2014). http://dx.doi.org/10.1002/navi.63. NAVI-2014-006.R1
54. Savage, P.: Strapdown inertial navigation integration algorithm design. Journal of Guidance, Control and Dynamics **21**(1-2) (1998)
55. Savage, P.G.: Laser gyros in strapdown inertial navigation systems. In: Proc. IEEE Position, Location, and Navigation Symp. San Diego, CA (1976)
56. Sierociuk, D., Tejado, I., Vinagre, B.M.: Improved fractional Kalman filter and its application to estimation over lossy networks. Signal Process. **91**(3), 542–552 (2011)
57. Skog, I., Händel, P., Nilsson, J.O., Rantakokko, J.: Zero-velocity detection—an algorithm evaluation. IEEE Transactions on Biomedical Engineering **57**(11), 2657–2666 (2010). https://doi.org/10.1109/TBME.2010.2060723

58. Skog, I., Nilsson, J.O., Händel, P., Nehorai, A.: Inertial sensor arrays, maximum likelihood, and Cramer-Rao bound. IEEE Transactions on Signal Processing **64**(16), 4218–4227 (2016). https://doi.org/10.1109/TSP.2016.2560136
59. STMicroelectronics: LSM6DSL iNEMO inertial module: always-on 3D accelerometer and 3D gyroscope (2017). Data sheet rev. 7
60. Titterton, D.H., Weston, J.L.: Strapdown Inertial Navigation Technology, 2nd edn. American Institute of Aeronautics and Astronautics, Reston, VA (2004)
61. Tweed, D.B., Haslwanter, T.P., Happe, V., Fetter, M.: Non-commutativity in the brain. Nature **399**(6733), 261–263 (1999)
62. Twomey, N., Diethe, T., Kull, M., Song, H., Camplani, M., Hannuna, S., Fafoutis, X., Zhu, N., Woznowski, P., Flach, P., Craddock, I.: The SPHERE challenge: Activity recognition with multimodal sensor data. arXiv preprint arXiv:1603.00797 (2016)
63. Voisin, M., Dreyfus-Schmidt, L., Gutierrez, P., Ronsin, S., Beillevaire, M.: Dataiku's solution to sphere's activity recognition challenge (2016)
64. Voss, R.F.: $1/f$ (flicker) noise: A brief review. In: Proc. 33rd Ann. Symp. Frequency Control, pp. 40–46 (1979)
65. Webb, A.: Statistical Pattern Recognition, 2nd edn. John Wiley & Sons, LTD (2002)
66. Woodman, O., Harle, R.: Pedestrian localisation for indoor environments. In: Proceedings of the 10th International Conference on Ubiquitous Computing, UbiComp '08, pp. 114–123. ACM, New York, NY, USA (2008). http://doi.acm.org/10.1145/1409635.1409651
67. Xsens MVN – inertial motion capture. http://www.xsens.com/en/general/mvn
68. Yu, J., Chen, Z., Zhu, Y., Chen, Y., Kong, L., Li, M.: Fine-grained abnormal driving behaviors detection and identification with smartphones. IEEE Transactions on Mobile Computing (2016). https://doi.org/10.1109/TMC.2016.2618873
69. Zhang, M.L., Zhou, Z.H.: A review on multi-label learning algorithms. IEEE Transactions on Knowledge and Data Engineering **26**(8), 1819–1837 (2014). https://doi.org/10.1109/TKDE.2013.39

Finding It Now: Networked Classifiers in Real-Time Stream Mining Systems

Raphael Ducasse, Cem Tekin, and Mihaela van der Schaar

Abstract The aim of this chapter is to describe and optimize the specifications of signal processing systems, aimed at extracting in real time valuable information out of large-scale decentralized datasets. A first section will explain the motivations and stakes and describe key characteristics and challenges of stream mining applications. We then formalize an analytical framework which will be used to describe and optimize distributed stream mining knowledge extraction from large scale streams. In stream mining applications, classifiers are organized into a connected topology mapped onto a distributed infrastructure. We will study linear chains and optimise the ordering of the classifiers to increase accuracy of classification and minimise delay. We then present a decentralized decision framework for joint topology construction and local classifier configuration. In many cases, accuracy of classifiers are not known beforehand. In the last section, we look at how to learn online the classifiers characteristics without increasing computation overhead. Stream mining is an active field of research, at the crossing of various disciplines, including multimedia signal processing, distributed systems, machine learning etc. As such, we will indicate several areas for future research and development.

R. Ducasse (✉)
The Boston Consulting Group, Boston, MA, USA
e-mail: ducasse.raphael@bcg.com

C. Tekin
Bilkent University, Ankara, Turkey
e-mail: cemtekin@ee.bilkent.edu.tr

M. van der Schaar
Oxford-Man Institute, Oxford, UK
University of California, Los Angeles, Los Angeles, CA, USA
e-mail: mihaela.vanderschaar@oxford-man.ox.ac.uk

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_3

Fig. 1 Nine examples of high volume streaming applications

1 Defining Stream Mining

1.1 Motivation

The spread of computing, authoring and capturing devices along with high bandwidth connectivity has led to a proliferation of heterogeneous multimedia data including documents, emails, transactional data, digital audio, video and images, sensor measurements, medical data, etc. As a consequence, there is a large class of emerging stream mining applications for knowledge extraction, annotation and online search and retrieval which require operations such as classification, filtering, aggregation, and correlation over high-volume and heterogeneous data streams. As illustrated in Fig. 1, stream mining applications are used in multiple areas, such as financial analysis, spam and fraud detection, photo and video annotation, surveillance, medical services, search, etc.

Let us deep-dive into three illustrative applications to provide a more pragmatic approach to stream mining and identify key characteristics and challenges inherent to stream mining applications.

Fig. 2 Semantic concept detection in applications

1.1.1 Application 1: Semantic Concept Detection in Multimedia; Processing Heterogeneous and Dynamic Data in a Resource-Constrained Setting

Figure 2 illustrates how stream mining can be used to tag concepts on images or videos in order to perform a wide set of tasks, from search to ad-targeting. Based upon this stream mining framework, designers can construct, instrument, experiment with, and optimize applications that automatically categorize image and video data captured by various cameras into a list of semantic concepts (e.g., skating, tennis, etc.) using various chains of classifiers.

Importantly, such stream mining systems need to be highly adaptive to the dynamic and time-varying multimedia sequence characteristics, since the input stream is highly volatile. Furthermore, they must often be able to cope with limited system resources (e.g. CPU, memory, I/O bandwidth), working on devices such as smartphones with increasing power restrictions. Therefore, applications need to cope effectively with system overload due to large data volumes and limited system resources. Commonly used approaches to dealing with this problem in resource constrained stream mining are based on load-shedding, where algorithms determine when, where, what, and how much data to discard given the observed data characteristics, e.g. burst, desired Quality of Service (QoS) requirements, data value or delay constraints.

1.1.2 Application 2: Online Healthcare Monitoring; Processing Data in Real Time

Monitoring individual's health requires handling a large amount of data, coming from multiple sources such as biometric sensor data or contextual data sources. As shown in Fig. 3, processing this raw information, filtering and analyzing it are key challenges in medical services, as it allows real time census and detection of irregular condition. For example, monitoring pulse check enables to identify if patient is in critical condition.

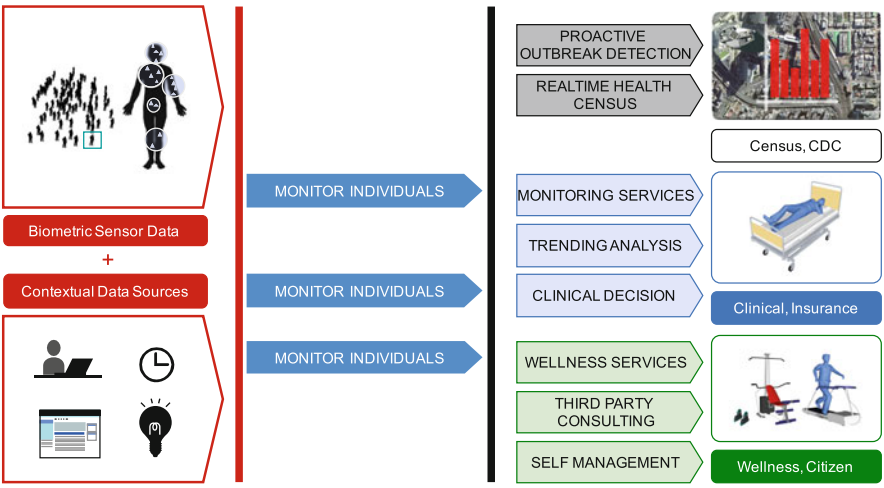


Fig. 3 Online healthcare monitoring workflow

In such application, being able to process data in real time is essential. Indeed, the information must be extracted and analyzed early enough to either take human decision or have an automatic control action. As an example, high concentration of calcium (happening under pain) could lead to either alerting medical staff or even automatic delivery of pain-killers, and the amount of calcium in the blood would determine the amount of medicine delivered. This control loop is only possible if the delay between health measurements (e.g. concentration of calcium in blood) and adaptation of treatment (e.g. concentration of pain-killer) is minimized.

1.1.3 Application 3: Analysis of Social Graphs; Coping with Decentralized Information and Setup

Social networks can be seen as a graph where nodes represent people (e.g. bloggers) and links represent interactions. Each node includes a temporal sequence of data, such as blog posts, tweets, etc. Numerous applications require to manage this huge amount of data: (1) selecting relevant content to answer keyword search, (2) identifying key influencers with page rank algorithms or SNA measures, and characterizing viral potential using followers' statistics, (3) recognizing objective vs. subjective content through lexical and pattern-based models, (4) automatically classifying data into topics (and creating new topics when needed) by observing work co-occurrence and using clustering techniques and classifying documents according to analysis performed on a small part of the document.

These applications are all the more challenging since the information is often decentralized across a very large set of computers, which is dynamically evolving over time. Implementing decentralized algorithms is therefore critical, even with

only partial information about other nodes. The performance of these algorithms can be greatly increased by using learning techniques, in order to progressively improve the pertinence of the analysis performed: at start, analysis is only based on limited data; over time, parameters of the stream mining application can be better estimated and the model used to process data is more and more precise.

1.2 From Data Mining to Stream Mining

1.2.1 Data Mining

Data mining can be described as the process of applying a ***query*** to a set of data, in order to select a sub-set of this data on which further action or analysis will be performed. For example, in Semantic Concept Detection, the query could be: "Select images of skating".

A data mining application may be viewed as a processing pipeline that analyzes data from a set of raw data sources to extract valuable information. The pipeline successively processes data through a set of filters, referred to as ***classifiers***. These classifiers can perform simple tests, and the query is the resultant of the answer of these multiple tests. For example, the query "Select images of skating" could be decomposed in the following tests: "Is it a team sport?"/"Is a Winter sport?"/"Is it a Ice sport?"/"Is it skating?"

Figure 4a provides an example of data mining application for sports image classification. Classifiers may be trained to detect different high-level semantic features, e.g. sports categories. In this example, the "Team Sports" classifier is used to filter the incoming data into two sets, thereby shedding a significant volume of data before passing it to the downstream classifiers (negatively identified team sports

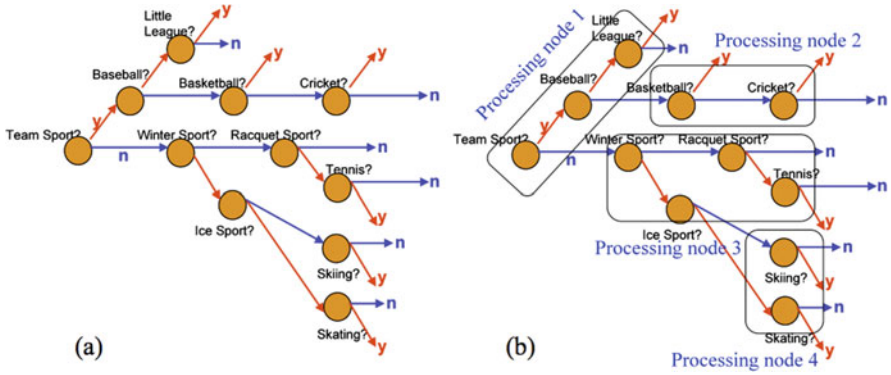


Fig. 4 A hierarchical classifier system that identifies several different sports categories and subcategories (**a**) at the same node, (**b**) across different nodes indicated in the figure as autonomous processing nodes

data is forwarded to the "Winter" classifier, while the remaining data is not further analyzed). Deploying a network of classifiers in this manner enables successive identification of multiple features in data, and provides significant advantages in terms of deployment costs. Indeed, decomposing complex jobs into a network of operators enhances scalability, reliability, and allows cost-performance tradeoffs to be performed. As a consequence, less computing resources are required because data is dynamically filtered through the classifier network. For instance, it has been shown that using classifiers operating in series with the same model (boosting [23]) or classifiers operating in parallel with multiple models (bagging [13]) can result in improved classification performance.

In this chapter, we will focus on mining applications that are built using a topology of low-complexity ***binary classifiers*** each mapped to a specific concept of interest. A binary classifier performs feature extraction and classification leading to a yes/no answer. However, this does not limit the generality of our solutions, as any M-ary classifiers may be decomposed into a chain of binary classifiers. Importantly, our focus will not be on the operators' or classifiers' design, for which many solutions already exist; instead, we will focus on configuring[1] the networks of distributed processing nodes, while trading off the processing accuracy against the available processing resources or the incurred processing delays. See Fig. 4b.

1.2.2 Changing Paradigm

Historically, mining applications were mostly used to find facts with data at rest. They relied on static databases and data warehouses, which were submitted to queries in order to extract and ***pull*** out valuable information out of raw data.

Recently, there has been a paradigm change in knowledge extraction: data is no longer considered static but rather as an inflowing stream, on which to dynamically compute queries and analysis in real time. For example, in Healthcare Monitoring, data (i.e., biometric measurements) is automatically analyzed through a batch of queries, such as "Verify that the calcium concentration is in the correct interval", "Verify that blood pressure is not too high", etc. Rather than applying a single query to data, the continuous stream of medical data is by default ***pushed*** through a predefined set of queries. This enables to detect any abnormal situation and react accordingly. See Fig. 5.

Interestingly, stream mining could lead to performing automatic action in response to a specific measurement. For example, a higher dose of pain killers could be administrated when concentration of calcium becomes too high, thus enabling real-time control. See Fig. 6.

[1] As we will discuss later, there are two types of configuration choices we must make: the topological ordering of classifiers and the local operating points at each classifier.

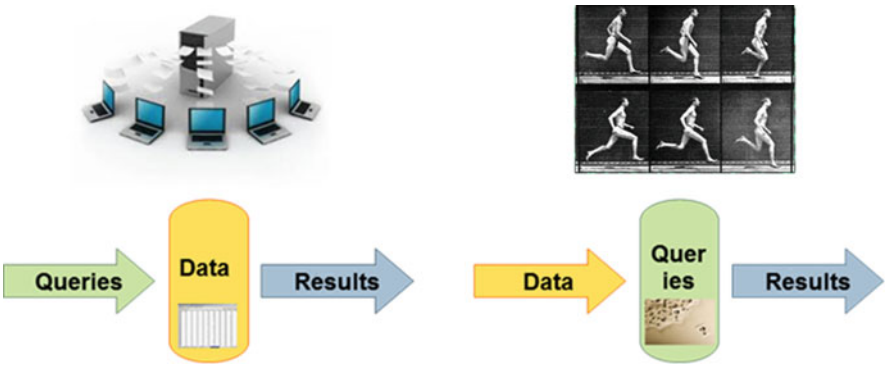


Fig. 5 A change of paradigm: continuous flow of information requires real-time extraction of insights

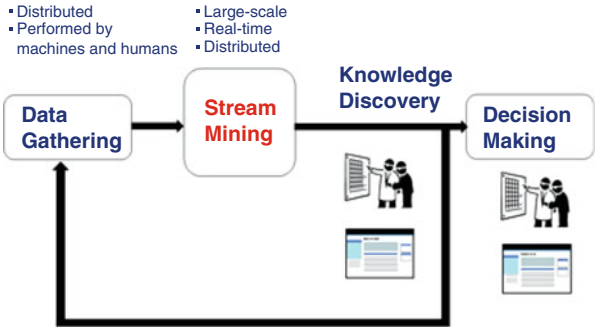


Fig. 6 Representation of knowledge extraction process in data mining system

1.3 Problem Formulation

1.3.1 Classifiers

A stream mining system can be seen as a set of binary classifiers. A binary classifier divides data into two subsets—one containing the object or information of interest (the "Positive" Set), and one not containing such objects or information (the "Negative" Set)—by applying a certain classification rule. For instance, the 'Team sport' classifiers separates images into those who represent a team sport and those who do not represent a team sport. This can be done using various classification techniques, such as Support Vector Machine (SVM), or K-nearest neighbor.

These algorithms are based on learning techniques, built upon test data and refined over time: they look for patterns in data, images, etc. and make decisions based on the resemblance of data to these patterns. As such, they are not fully accurate. A classifier can introduce two types of errors:

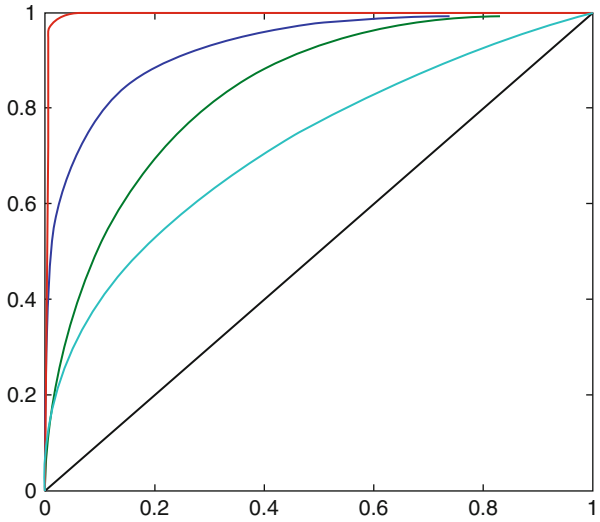


Fig. 7 ROC curves: $p^F = f(p^D)$. X axis is probability of misdetection error. Y axis is probably of false alarm error. We call sensitivity the factor that slides the operating point along the ROC curve

- Misdetection errors: Missing objects or data of interest by tagging it as belonging to the Negative Set rather than the Positive Set. We will note p^D the probability of detecting a data unit: $1 - p^D$ is the probability of misdetection.
- False alarm errors: Wrongly tagging objects or data which are not of interest as belonging to the Positive Set. We will note p^F this probability of false alarm.

Naturally, there is a trade-off between misdetection and false alarm errors: to avoid misdetections, the classifier could tag all data as positive, which would generate a high false alarm rate.

We will call **operating point** the couple (p^D, p^F). In Fig. 7, the operating points of various classifiers are plotted and form what is referred as ROC curves. The accuracy of the classifier depends on the concavity of the ROC curve, the more concave, the more precise.

The operating points' choice has two consequences on the performance of the stream mining system. First, it affects the precision of each classifier (both misdetection and false alarms) and of the system as a whole. Secondly, it defines the amount of data which is going to be transmitted through the classifiers and therefore the delay required for the system to process the data stream.

1.3.2 Axis for Study

This chapter focuses on developing a new systematic framework for knowledge extraction from high-volume data streams using a network of classifiers deployed

over a distributed computing infrastructure. It can be decomposed into four sub-problems which we will develop in the following sections:

1. **Stream Mining System Optimization:** In Sect. 2, we develop optimization techniques for tuning the operating points of individual classifiers in order to improve the stream mining performance, in terms of accuracy and delay. We formalize the problem of large-scale knowledge extraction by defining appropriate local and end-to-end objective functions, along with resource and delay constraints. They will guide the optimization and adaptation algorithms used to improve the stream mining performance.
2. **Stream Mining System Topology Optimization:** As shown in Fig. 4, a stream mining system is a topology of classifiers mapped onto a distributed infrastructure. These classifiers can be organized in one single chain, or in multiple parallel chains, thus forming a tree topology. In Sect. 3, we investigate the impact of the classifiers' topology on the performance, scalability and dynamic behavior of the stream mining system. We will focus on the study of linear chains of classifiers and determine how to jointly choose the order of classifiers in the chain and the operating point of each classifier in order to maximize accuracy and minimize delays.
3. **Decentralized Solutions Based on Interactive Multi-Agent Learning:** For large scale stream mining systems, where the classifiers are distributed across multiple nodes, the choice of operating point and topology of the classifiers would require heavy computational resources. Furthermore, optimizing the overall performance requires interactive multi-agent solutions to be deployed at each node in order to determine the effect of each classifiers' decisions on the other classifiers and hence, the end to end performance of the stream mining applications. In the fourth section of this chapter, we develop a decentralized decision framework for stream mining configuration and propose distributed algorithms for joint topology construction and local classifier configuration. This approach will cope with dynamically changing environments and data characteristics and adapt to the timing requirements and deadlines imposed by other nodes or applications.
4. **Online Learning for Real-Time Stream Mining:** In Sect. 5, we consider the stream mining problems in which the classifier accuracies are not known beforehand and needs to be learned online. Such cases frequently appear in real applications due to the dynamic behavior of heterogeneous data streams. We explain how the best classifiers (or classifier configurations) can be learned via repeated interaction, by driving the classifier selection process using meta-data. We also model the loss due to not knowing the classifier accuracies beforehand using the notion of regret, and explain how the regret can be minimized while ensuring that memory and computation overheads are kept at reasonable levels.

1.4 Challenges

Several key research challenges drive our analysis and need to be tackled: These are discussed in the following sections.

1.4.1 Coping with Complex Data: Large-Scale, Heterogeneous and Time-Varying

First, streaming data supposes that have high volume of timeless information flows in continuously. Stream mining systems thus need be scalable to massive data source and be able to simultaneously deal with multiple queries.

Both structured and unstructured data may be mined. In practice, data is wildly heterogeneous in terms of formats (documents, emails, transactions, digital video and/or audio data, RSS feeds) as well as data rates (manufacturing: 5–10 Mbps, astronomy: 1–5 Gbps, healthcare: 10–50 Kbps per patient). Furthermore, data sources and sensors may eventually be distributed on multiple processing nodes, with little or no communication in between them.

Stream mining systems need to be adaptive in order to cope with data and configuration dynamics: (1) heterogeneous data stream characteristics, (2) classifier dependencies, (3) congestion at shared processing nodes and (4) communication delays between processing nodes. Additionally, several different queries (requiring different topological combinations of classifiers) may need to be satisfied by the system, requiring reconfiguration as queries change dynamically.

1.4.2 Immediacy

Stream mining happens **now**, in real time. The shift from data mining to stream mining supposes that data cannot be stored and has to be processed on the fly.

For instance, in healthcare monitoring, minimizing delay between health measurements (e.g. concentration of calcium in blood) and adaptation of treatment (e.g. concentration of pain-killer) is critical. For some applications such as high-frequency trading, being real time may even be more important than minimizing misclassification costs. otherwise historic data would become obsolete and lead to phrased-out investment decisions.

Delay has seldom been analyzed in existing work on stream mining systems and, when it has been [1], it has always been analyzed in steady-state, at equilibrium, after all processing nodes are configured. However, the equilibrium can often not be reached due to the dynamic arrival and departure of query applications. Hence, this reconfiguration delay out of equilibrium must be considered when designing solutions for real-time stream mining systems.

Delay constraints are all the more challenging in a distributed environment, where the synchronization among nodes may not be possible or may lead to sub-optimal designs, as various nodes may experience different environmental dynamics and demands.

1.4.3 Distributed Information and Knowledge Extraction

To date, a majority of approaches for constructing and adapting stream mining applications are based on centralized algorithms, which require information about each classifier's analytics to be available at one node, and for that node to manage the entire classifier network. This limits scalability, creates a single point of failure, and provide limits in terms of adaptivity to dynamics.

Yet, data sources and classifiers are often distributed over a set of processing nodes and each node of the network may exchange only limited and/or costly message with other interconnected nodes to. Thus, it may be impractical to develop centralized solutions [4, 7, 18, 32, 33].

In order to address this naturally distributed setting, as well as the high computational complexity of the analytics, it is required to formally define local objectives and metrics and to associate inter-node message exchanges that enable the decomposition of the application into a set of autonomously operating nodes, while ensuring global performance. Such ***distributed mining systems*** have recently been developed [5, 19]. However, they do not encompass the accuracy and delay objectives described earlier.

Depending on the system considered, classifiers can have strong to very limited communication. Thus, classifiers may not have sufficient information to jointly configure their operating points. In such distributed scenarios, optimizing the end-to-end performance requires interactive, multi-agent solutions in order to determine the effect of each classifier's decisions on the other classifiers. Nodes need to learn online the effect of both their experienced dynamics as well as the coupling between classifiers.

Besides, for classifiers instantiated on separate nodes (possibly over a network), the communication time between nodes can greatly increase the total time required to deal with a data stream. Hence, the nodes will not be able to make decisions synchronously.

1.4.4 Resource Constraints

A key research challenge [1, 12] in distributed stream mining systems arises from the need to cope effectively with system overload, due to limited system resources (e.g. CPU, memory, I/O bandwidth etc.) while providing desired application performance. Specifically, there is a large computational cost incurred by each classifier (proportional to the data rate) that limits the rate at which the application can handle input data. This is all the more topical in a technological environment where low-power devices such as smartphones are becoming more and more used.

2 Proposed Systematic Framework for Stream Mining Systems

2.1 Query Process Modeled as Classifier Chain

Stream data analysis applications pose queries on data that require multiple concepts to be identified. More specifically, a query q is answered as a conjunction of a set of N classifiers $\mathcal{C}(q) = \{C_1, \ldots, C_N\}$, each associated with a concept to be identified (e.g. Fig. 4 shows a stream mining system where the concepts to be identified are sports categories).

In this chapter, we focus on binary classifiers: each binary classifier C_i labels input data into two classes $\mathcal{H}_i$ (considered without loss of generality as the class of interest) and $\overline{\mathcal{H}_i}$. The objective is to extract data belonging to $\bigcap_{i=1}^{N} \mathcal{H}_i$.

Partitioning the problem into this ensemble of classifiers and filtering data successively (i.e. discarding data that is not labelled as belonging to the class of interest), enables to control the amount of resources consumed by each classifier in the ensemble. Indeed, only data labelled as belonging to $\mathcal{H}_i$ is forwarded, while data labelled as belonging to $\overline{\mathcal{H}_i}$ is dropped. Hence, a classifier only has to process a subset of the data processed by the previous classifier. This justifies using a chain topology of classifiers, where the output of one classifier C_{i-1} feeds the input of classifier C_i, and so on, as shown in Fig. 8.

2.1.1 A-Priori Selectivity

Let X represent the input data of a classifier C. We call *a-priori* selectivity $\phi = \mathbf{P}(X \in \mathcal{H})$ the *a-priori* probability that the data X belongs to the class of interest. Correspondingly $1 - \phi = \mathbf{P}(X \in \overline{\mathcal{H}})$. Practically speaking, the *a-priori* selectivity ϕ is computed on a training and cross-validation data set. For well-trained classifiers, it is reasonable to expect that the performance on new, unseen test data is similar to that characterized on training data. In practice, there is potential train-test mismatch in behavior, but this can be accounted for using periodic reevaluation of the classifier performance (e.g. feedback on generated results).

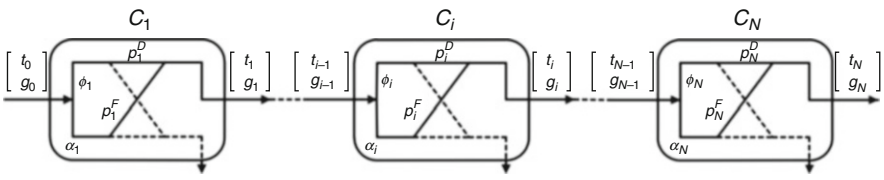


Fig. 8 Representation of analytical framework to evaluate classifier chain performance

For a chain of classifiers $\mathcal{C} = \{C_1, \ldots, C_N\}$, the *a-priori* selectivity of a classifier corresponds to the conditional probability of data belonging to classifier C_i's class of interest, given that it belongs to the class of interest of the previous $i-1$ classifiers: $\phi_i = \mathbf{P}(X \in \mathcal{H}_i | X \in \bigcap_{k=1}^{i-1} \mathcal{H}_k)$. Similarly, we define the negative *a-priori* selectivity as $\overline{\phi_i} = \mathbf{P}(X \in \mathcal{H}_i | X \notin \bigcap_{k=1}^{i-1} \mathcal{H}_k)$. Since *a-priori* selectivities depend on classifiers higher in the chain, $\overline{\phi_i} \neq 1 - \phi_i$.

2.1.2 Classifier Performance

The output $\hat{X}$ of a classifier C can be modeled as a probabilistic function of its input X. The proportion of correctly classified samples in $\mathcal{H}_k$ is captured by the probability of correct detection $p_k^D = \mathbf{P}(\hat{X} \in \mathcal{H}_k | X \in \mathcal{H}_k)$, while the proportion of falsely classified samples in $\mathcal{H}_k$ is $p_k^F = \mathbf{P}(\hat{X} \in \mathcal{H}_k | X \in \overline{\mathcal{H}_k})$.

The performance of the classifier C is characterized by its ROC curve that represents the tradeoff between the probability of detection p^D and probability of false alarm p^F. We represent the ROC curve as a function $f : p^F \mapsto p^D$ that is increasing, concave and lies over the first bisector [11]. As a consequence, an operating point on this curve is parameterized uniquely by its false alarm rate $x = p^F$. The operating point is denoted by $(x, f(x)) = (p^F, p^D)$.

We model the average time needed for classifier C to process a stream tuple as α (in seconds). The order of magnitude of α depends on the data characteristics, as well as the classification algorithm, and can vary from microseconds (screening text) to multiple seconds (complex image or video classification).

2.1.3 Throughput and Goodput of a Chain of Classifiers

The forwarded output of a classifier C_i consists of both correctly labelled data from class $\mathcal{H}_i$ as well as false alarms from class $\overline{\mathcal{H}_i}$. We use g_i to represent the goodput (portion of data correctly labelled) and t_i to represent the throughput (total forwarded data, including mistakes). And we will note t_0 to represent the input rate of data.

Using Bayes formula, we can derive t_i and g_i recursively as

$$\begin{bmatrix} t_i \\ g_i \end{bmatrix} = \underbrace{\begin{bmatrix} a_i & b_i \\ 0 & c_i \end{bmatrix}}_{T_i^{i-1}} \begin{bmatrix} t_{i-1} \\ g_{i-1} \end{bmatrix}, \qquad \text{where} \qquad \begin{cases} a_i = p_i^F + (p_i^D - p_i^F)\overline{\phi_i} \\ b_i = (p_i^D - p_i^F)(\phi_i - \overline{\phi_i}) \\ c_i = p_i^D \phi_i \end{cases} \tag{1}$$

For a set of independent classifiers, the positive and negative *a-priori* selectivities are equal: $\phi_i = \overline{\phi_i} = \mathbf{P}(X \in \mathcal{H})$. As a consequence, the transition matrix is diagonal: $T_i^{i-1} = \begin{bmatrix} p_i^D \phi_i + (1-\phi_i) p_i^F & 0 \\ 0 & p_i^D \phi_i \end{bmatrix}$.

2.2 Optimization Objective

The global utility function of the stream mining system can be expressed as a function of misclassification and delay cost, under resource constraints.

2.2.1 Misclassification Cost

The misclassification cost, or error cost, may be computed in terms of the two types of accuracy errors—a penalty c^M per unit rate of missed detection, and a penalty c^F per unit rate of false alarm. These are specified by the application requirements. Noting $\Phi = \prod_{h=1}^{N} \phi_h$, the total misclassification cost is

$$c_{err} = c^M \underbrace{(\Phi t_0 - g_N)}_{misseddata} + c^F \underbrace{(t_N - g_N)}_{wronglyclassifieddata} . \tag{2}$$

2.2.2 Processing Delay Cost

Delay may be defined as the time required by the chain of classifiers in order to process a stream tuple. Let α_i denote the expected processing time of classifier C_i. The average time required by classifier C_i to process a stream tuple is given by $\delta_i = \alpha_i P_i$, where P_i denotes the fraction of data which has not been rejected by the first $i-1$ classifiers and still needs to be processed through the remaining classifiers of the chain. Recursively, $P_i = \prod_{k=1}^{i-1} \frac{t_k}{t_{k-1}} = \frac{t_{i-1}}{t_0}$. After summation across all classifiers, the average end-to-end processing time required by the chain to process stream data is

$$c_{delay} = t_0 \sum_{i=1}^{N} \delta_i = t_0 \sum_{i=1}^{N} \alpha_i P_i = \sum_{i=1}^{N} \alpha_i t_{i-1}. \tag{3}$$

2.2.3 Resource Constraints

Assume that the N classifiers are instantiated on M processing nodes, each of which has a given available resource r_j^{max}. We can define a location matrix $M \in \{0, 1\}^{M \times N}$ where $M_{ji} = 1$ if C_i is located on node j and 0 otherwise. The resource constraint at node j can be written as $\sum_{i=1}^{N} M_{ji} r_i \leq r_j^{max}$.

The resource r_i consumed at node j by classifier C_i is proportional to the throughput t_i, i.e. $r_i \propto t_i$.

2.2.4 Optimization Problem

Stream mining system configuration involves optimizing both accuracy and delay under resource constraints. The utility function of this optimization problem may be defined as the negative weighted sum of both the misclassification cost and the processing delay cost: $U = -c_{err} - \lambda\, c_{delay}$, where the parameter λ controls the tradeoff between misclassification and delay. This utility is a function of the throughputs and goodputs of the stream within the chain, and therefore implicitly depends on the operating point $x_i = p_i^F \in [0, 1]$ selected by each classifier.

Let $\mathbf{x} = \left[x_1, \ldots, x_N\right]^T$, $K = \frac{c^F}{c^F + c^M} \in [0, 1]$ and $\rho = \frac{\lambda}{c^F + c^M}\boldsymbol{\alpha} \in \mathbf{R}^{+N}$. The optimization problem can be reformulated under a canonic format as follows:

$$\begin{cases} \underset{\mathbf{x}\in[0\ 1]^N}{\text{maximize}}\ U(\mathbf{x}) = g_N(\mathbf{x}) - K t_N(\mathbf{x}) - \sum_{i=1}^{N} \rho_i t_{i-1}(\mathbf{x}) \\ \text{subject to } \mathbf{0} \le \mathbf{x} \le \mathbf{1} \quad \text{and } M\mathbf{r} \le \mathbf{r}^{\max} \end{cases}. \tag{4}$$

2.3 Operating Point Selection

Given a topology, the resource-constrained optimization problem defined in Eq. (4) may be formulated as a network optimization problem (NOP) [16, 20]. This problem has been well studied in [11, 21, 31] and we refer the interested reader to the corresponding literature.

The solutions proposed involve using iterative optimization techniques based on Sequential Quadratic Programming (SQP) [3]. SQP is based on gradient-descent, and models a nonlinear optimization problem as an approximate quadratic programming subproblem at each iteration, ultimately converging to a locally optimal solution.

Selecting the operating point can be done by applying the SQP-algorithm to the Lagrangian function of the optimization problem in (4):

$$\mathcal{L}(\mathbf{x}, \nu_1, \nu_2) = U(\mathbf{x}) - \nu_1^T(\mathbf{x} - 1) + \nu_2^T\mathbf{x}.$$

Because of the gradient-descent nature of the SQP algorithm, it is not possible to guarantee convergence to the global maximum and the convergence may only be locally optimal. However, the SQP algorithm can be initialized with multiple starting configurations in order to find a better local optimum (or even the global optimum). Since the number and size of local optima depend on the shape of the various ROC curves of each classifier, a rigorous bound on the probability to find the global optimum cannot be proven. However, certain start regions are more likely to converge to better local optimum.[2]

[2]For example, since the operating point $p^F = 0$ corresponds to a saddle point of the utility function, it would achieve steepest utility slope. Furthermore, the slope of the ROC curve is

2.4 Further Research Areas

Further research areas are the following:

- **Communication delay between classifiers:** The model could be further refined to explicitly consider communication delays, i.e. the time needed to send stream tuples from one classifier to another. This is all the more true in low-delay settings where classifiers are instantiated on different nodes.
- **Queuing delay between classifiers:** Due to resource constraints, some classifiers may get congested, and the stream will hence incur additional delay. Modeling these queuing delays would further improve the suitability of the framework for real-time applications.
- **Single versus multiple operating points per classifier:** Performance gains can be achieved by allowing classifiers to have different operating points x_i and $\overline{x_i}$ for their positive and negative classes. If the two thresholds overlap, low-confidence data will be duplicated across both output edges, thereby increasing the end-to-end detection probability. If they do not overlap, low-confidence data is shed, thus reducing congestion at downstream classifiers.
- **Multi-query optimization:** Finally, a major research area would consist in studying how the proposed optimization and configuration strategies adapt to multi-query settings, including mechanisms for admission control of queries.

3 Topology Construction

In the previous section, we have determined how to improve performance of a stream mining system—both in terms of accuracy and delays—by selecting the right operating point for each classifier of the chain. This optimization was however performed given a specific topology of classifiers: classifiers were supposed arranged as a chain and the order of the classifiers in the chain was fixed.

In this section, we study the impact of the topology of classifiers on the performance of the stream mining system. We start by focusing on a chain topology and study how the order of classifiers on the chain alters performance.

3.1 Linear Topology Optimization: Problem Formulation

Since classifiers have different *a-priori* selectivities, operating points, and complexities, different topologies of classifiers will lead to different classification and delay costs.

maximal at $p^F = 0$ (due to concavity of the ROC curve), such that high detection probabilities can be obtained under low false alarm probabilities near the origin.

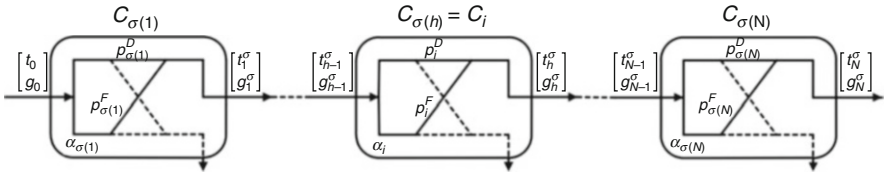


Fig. 9 Representation of σ-ordered classifier chain

Consider N classifiers in a chain, defined as in the previous section. An order $\sigma \in \mathcal{P}erm(N)$ is a permutation such that input data flows from $C_{\sigma(1)}$ to $C_{\sigma(N)}$. We generically use the index i to identify a classifier and h to refer to its depth in the chain of classifiers. Hence, $C_i = C_{\sigma(h)}$ will mean that the hth classifier in the chain is C_i. To illustrate the different notations used, a σ-ordered classifier chain is shown in Fig. 9.

Using the recursive relationship defined in Eq. (1), we can derive the end-to-end throughput t_i and goodput g_i of classifier $C_i = C_{\sigma(h)}$ recursively as

$$\begin{bmatrix} t_i \\ g_i \end{bmatrix} = \underbrace{\begin{bmatrix} p_i^F + \overline{\phi_h^\sigma}(p_i^D - p_i^F) & (\phi_h^\sigma - \overline{\phi_h^\sigma})(p_i^D - p_i^F) \\ 0 & \phi_h^\sigma p_i^D \end{bmatrix}}_{T_i^{i-1} = T_h^\sigma} \begin{bmatrix} t_{h-1}^\sigma \\ g_{h-1}^\sigma \end{bmatrix}. \tag{5}$$

The optimization problem can be written as:

$$\begin{cases} \underset{\sigma \in \mathcal{P}erm(N), \mathbf{x} \in [0\ 1]^N}{\text{maximize}} & U(\sigma, \mathbf{x}) = g_N^\sigma(\mathbf{x}) - K t_N^\sigma(\mathbf{x}) - \sum_{i=1}^{N} \rho_i t_{i-1}^\sigma(\mathbf{x}) \\ \text{subject to} & \mathbf{0} \leq \mathbf{x} \leq \mathbf{1} \end{cases}. \tag{6}$$

3.2 Centralized Ordering Algorithms for Fixed Operating Points

In this section, we consider a set of classifiers with fixed operating points $\mathbf{x}$. Since transition matrices T_i^σ are lower triangular, the goodput does not depend on the order of classifiers.[3] As a consequence, the expression of the utility defined in Eq. (4) can be simplified as:

[3]Furthermore, when classifiers are independent, the transition matrices T_i^σ are diagonal and therefore commute. As a consequence the end throughput $t_N(\mathbf{x})$ and goodput $g_N(\mathbf{x})$ are independent of the order. However, intermediate throughputs do depend on the ordering—leading to varying expected delays for the overall processing.

$$\underset{\sigma \in \mathcal{P}erm([1,N])}{\text{maximize}} \; U_{ord} = -\left(\sum_{h=1}^{N} \rho_{\sigma(h)} t_{h-1}^{\sigma} + K t_N^{\sigma} \right). \tag{7}$$

3.2.1 Optimal Order Search

The topology construction problem involves optimizing the defined utility by selecting the appropriate order σ. In general, there exist $N!$ different topologic orders, each with a different achieved utility and processing delay. Furthermore, the relationship between order and utility cannot be captured using monotonic or convex analytical functions. Hence, any search space for order selection increases combinatorially with N. This problem is exacerbated in dynamic settings where the optimal order has to be updated online; in settings with multiple chains, where each chain has to be matched with a specific optimal order; and, in settings with multiple data streams corresponding to the queries of multiple users.

3.2.2 Greedy Algorithm

Instead of solving the complex combinatorial problem, we suggest to design simple, but elegant and powerful, order selection algorithms—or Greedy Algorithms—with provable bounds on performance [2, 6].

The Greedy Algorithm is based on the notion of *ex-post* selectivity. For a given order σ, we define the *ex-post* selectivity as the conditional probability of classifier $C_{\sigma(h)}$ labelling a data item as positive given that the previous $h-1$ classifiers labelled the data as positive,[4] i.e. $\psi_h^{\sigma} = \frac{t_h^{\sigma}}{t_{h-1}^{\sigma}}$. The throughput at each step can be expressed recursively as a product of *ex-post* selectivities: $t_h^{\sigma} = \psi_h^{\sigma} t_{h-1}^{\sigma} = \ldots = \left(\prod_{i=1}^{h} \psi_i^{\sigma} \right) t_0$.

The Greedy Algorithm then involves ordering classifiers in increasing order of $\frac{\psi}{\mu}$ where $\mu_i^{\sigma} = \begin{cases} \rho_{\sigma(i+1)} = \frac{\lambda}{c^M + c^F} \alpha_{\sigma(i+1)} & \text{if } i \leq N-1 \\ K = \frac{c^F}{c^F + c^M} & \text{if } i = N \end{cases}$. Note that this fraction depends on the selected order.

Since this ratio depends implicitly on the order of classifiers in the chain, the algorithm may be implemented iteratively, selecting the first classifier, then selecting the second classifier given the fixed first classifier, and so on:

[4] Observe that for a perfect classifier ($p_{\sigma(h)}^{D} = 1$ and $p_{\sigma(h)}^{F} = 0$), the *a-priori* conditional probability ϕ_h^{σ} and the *ex-post* conditional probabilities ψ_h^{σ} are equal.

Centralized Algorithm 1 Greedy ordering

- Calculate the ratio $\psi_1^\sigma/\mu_1^\sigma$ for all N classifiers. Select $C_{\sigma(1)}$ as the classifier with lowest weighted non-conditional selectivity $\psi_1^\sigma/\mu_1^\sigma$. Determine $\begin{bmatrix} t_1^\sigma \\ g_1^\sigma \end{bmatrix}$.
- Calculate the ratio $\psi_2^\sigma/\mu_2^\sigma$ for all remaining $N-1$ classifiers. Select $C_{\sigma(2)}$ as the classifier with lowest weighted conditional selectivity $\psi_2^\sigma/\mu_2^\sigma$. Determine $\begin{bmatrix} t_2^\sigma \\ g_2^\sigma \end{bmatrix}$.
- Continue until all classifiers have been selected.

In each iteration we have to update $O(N)$ selectivities and there are $O(N)$ iterations, making the complexity of the algorithm $O(N^2)$ (compared to $O(N!)$ for the optimal algorithm). Yet, it can be shown that the performance of the Greedy Algorithm can be bound:

$$\frac{1}{\kappa} U_{ord}^{opt} \leq U_{ord}^{G} \leq U_{ord}^{opt} \qquad \text{with } \kappa = 4.$$

The value U_{ord}^{G} of the utility obtained with the Greedy Algorithm's order is at least 1/4th of the value of the optimal order U_{ord}^{opt}. Furthermore, the approximation factor $\kappa = 4$ corresponds to a system with infinite number of classifiers [34]. In practice, this constant factor is smaller. Specifically, we have $\kappa = 2.35, 2.61, 2.8$ for 20, 100 or 200 classifiers respectively.

The key of the proof of this result is to show that the Greedy Algorithm is equivalent to a greedy 4-approximation algorithm for pipelined set-cover. We refer the interested reader to the demonstration made by Munagala and Ali in [2] and let him show that our problem setting is equivalent to the one formulated in their problem.

3.3 Joint Order and Operating Point Selection

Further system performance can be achieved by both optimizing the order of the chain of classifiers and the operating point configuration.

To build a joint order and operating point selection strategy, we propose to combine the SQP-based solution for operating point selection with the iterative Greedy order selection. This iterative approach, or SQP-Greedy algorithm, is summarized as follows:

Centralized Algorithm 2 SQP-Greedy algorithm for joint ordering and operating point selection

- **Initialize** $\sigma^{(0)}$.
- **Repeat** until greedy algorithm does not modify order.
  1. Given order $\sigma^{(j)}$, compute locally optimal $\mathbf{x}^{(j)}$ through SQP.
  2. Given operating points $\mathbf{x}^{(j)}$, update order $\sigma^{(j+1)}$ using (A-)Greedy algorithm.

Each step of the SQP-Greedy algorithm is guaranteed to improve the global utility of the problem. Given a maximum bounded utility, the algorithm is then guaranteed to converge. However, it may be difficult to bound the performance gap between the SQP-Greedy and the optimal algorithm with a constant factor, since the SQP only achieves local optima. As a whole, identification and optimization of algorithms used to compute optimal order and operating points represents a major roadblock to stream mining optimization.

3.3.1 Limits of Centralized Algorithms for Order Selection

We want to underline that updating the *ex-post* selectivities requires strong coordination between classifiers. A first solution would be for classifiers to send their choice of operating point (p^F, p^D) to a central agent (which would also have knowledge about the *a-priori* conditional selectivities ϕ^σ, $\overline{\phi^\sigma}$) and would compute the *ex-post* conditional selectivities. A second solution would be for each classifier C_i to send their rates t_i and g_i to the classifiers C_j which have not yet processed the stream for them to compute ψ^i_j. In both cases, heavy message exchange is required, which can lead to system inefficiency (cf. Sect. 4.1). We will propose in Sect. 4 a decentralized solution with limited message exchanges, as an alternative to this centralized approach.

3.4 Multi-Chain Topology

3.4.1 Motivations for Using a Multi-Chain Topology: Delay Tradeoff Between Feature Extraction and Intra-Classifier Communication

In the previous analysis, we did not take into consideration the time α^{com} required by classifiers to communicate with each other. If classifiers are all grouped on a single node, such communication time α^{com}_{inter} can be neglected compared to the time α^{feat} required by classifiers to extract data features. However for classifiers instantiated on separate nodes, this communication time α^{com}_{ext} can greatly increase the total time required to deal with a stream tuple.

As such, we would like to limit the communication between nodes, i.e. (1) avoid sending the stream back and forth from one node to another and (2) limit message exchanges between classifiers. To do so, a solution would be to process the stream in parallel on each node and to intersect the output of each node-chain.

3.4.2 Number of Chains and Tree Configuration

Suppose that instead of considering classifiers in a chain, we process the stream through R chains, where chain r has N_r classifiers with the order σ_r. The answer of the query is then obtained by intersecting the output of each chain r (we assume that this operation incurs zero delay).

We can show that, as a first approximation, the end-to-end processing time can be written as

$$c^{\sigma}_{delay} = \underbrace{\sum_{r=1}^{R}\sum_{h=1}^{N_r} \alpha^{feat}_{\sigma_r(h)} t^{\sigma_r}_{h-1}}_{feature extraction} + \underbrace{\sum_{r=1}^{R}\sum_{h=1}^{N_r-1} \alpha^{com}_{\sigma_r(h),\sigma_r(h+1)} t^{\sigma_r}_{h}}_{intra-classifier communication}. \quad (8)$$

Intuitively, the feature extraction term increases with the number of chains R, as each chain needs to process the whole stream, while the intra-classifier communication term decreases with R, since using multiple chains enables classifiers instantiated on the same node to be grouped together in order to avoid time-costly communication between nodes (cf. Fig. 4b).

Configuring stream mining systems as tree topologies (i.e. determining the number of chains to use in order to process the stream in parallel, as well as the composition and order of each chain) represents a major research theme. The number of chains R and the choice of classifiers per chain illustrate the tradeoff between feature extraction and intra-classifier communication and will depend on the values of α^{feat} and α^{com}.

4 Decentralized Approach

4.1 Limits of Centralized Approaches and Necessity of a Decentralized Approach

The centralized approach presented in the previous sections has six main limitations:

1. **System and Information Bottlenecks:** Centralized approaches require a central agent that collects all information, generates optimal order and operating points per classifier, and distributes and enforces results on all classifiers. This creates a

bottleneck, as well as a single point of failure, and is unlikely to scale well as the number of classifiers, topologic settings, data rates, and computing infrastructure grow.

2. **Topology Specificity:** A centralized approach is designed to construct one topology for each user application of interest. In practice the system may be shared by multiple such applications—each of which may require the reuse of different subsets of classifiers. In this case, the centralized algorithm needs to design multiple orders and configurations that need to be changed dynamically as application requirements change, and applications come and go.
3. **Resource Constraints:** Currently designed approaches minimize a combination of processing delay and misclassification penalty. However, in general we also need to satisfy the resource constraints of the underlying infrastructure. These may in general lead to distributed non-convex constraints in the optimization, thereby further increasing the sub-optimality of the solution, and increasing the complexity of the approach.
4. **Synchronization Requirements:** The processing times vary from one classifier to the other. As a result, transmission from one classifier to another is not synchronized. Note that this asynchrony is intrinsic to the stream mining system. Designing one centralized optimization imposes synchronization requirements among classifiers and as the number of classifiers and the size of the system increases may reduce the overall efficiency of the system.
5. **Limited Sensitivity to Dynamics:** As an online process, stream mining optimization must involve algorithms which take into account the system's dynamics, both in terms of the evolving stream characteristics and classifiers' processing time variations. This time-dependency is all the more true in a multi-query context, with heterogeneous data streams for which centralized algorithms are unable to cope with such dynamics.
6. **Requirement for Algorithms to Meet Time Delay Constraints:** These dynamics require rapid adaptation of the order and operating points, often even at the granularity of one tuple. Any optimization algorithm thus needs to provide a solution with a time granularity finer than the system dynamics. Denote by τ the amount of time required by an algorithm to perform one iteration, i.e. to provide a solution to the order and configuration selection problem. The solution given by an algorithm will not be obsolete if $\tau \leq \mathfrak{C}\tau^{dyn}$ where τ^{dyn} represents the characteristic time of significant change in the input data and characteristics of the stream mining system and $\mathfrak{C} \leq 1$ represents a buffer parameter in case of bursts.

To address these limitations, we propose a decentralized approach and design a decentralized stream mining framework based on reinforcement learning techniques.

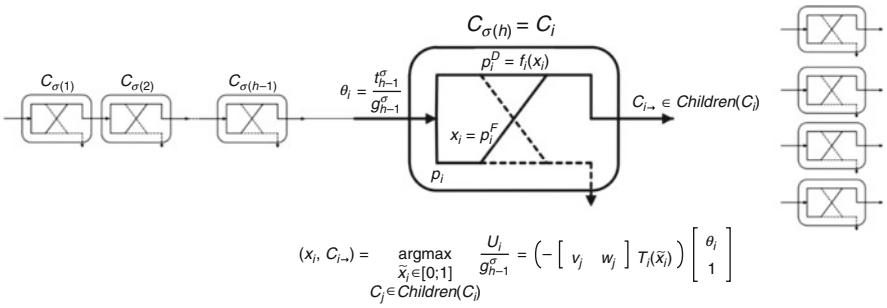


Fig. 10 Stochastic decision process: at each node, optimisation of local utilisation of select operating point and child classifier

4.2 Decentralized Decision Framework

The key idea of the decentralized algorithm is to replace centralized order selection by local decisions consisting in determining to which classifier to forward the stream. To describe this, we set up a stochastic decision process framework $\{\mathcal{C}, \mathcal{S}, \mathcal{A}, \mathcal{U}\}$ [15], illustrated in Fig. 10, where

- $\mathcal{C} = \{C_1, \ldots, C_N\}$ represents the set of classifiers
- $\mathcal{S} = \underset{i \leq N}{\times} \mathcal{S}_i$ represents the set of states
- $\mathcal{A} = \underset{i \leq N}{\times} \mathcal{A}_i$ represents the set of actions
- $\mathcal{U} = \{U_1, \ldots, U_N\}$ represents the set of utilities

4.2.1 Users of the Stream Mining System

Consider N classifiers $\mathcal{C} = \{C_1, \ldots, C_N\}$. The classifiers are autonomous: unless otherwise mentioned, they do not communicate with each other and take decisions independently. We recall that the hth classifier will be referred as $C_i = C_{\sigma(h)}$. We will also refer to the stream source as $C_0 = C_{\sigma(0)}$.

4.2.2 States Observed by Each Classifier

The set of states can be decomposed as $\mathcal{S} = \underset{i \leq N}{\times} \mathcal{S}_i$. The local state set of classifier $C_i = C_{\sigma(h)}$ at the hth position in the classifier chain is defined as $\mathcal{S}_i = \{(Children(C_i), \theta_i)\}$:

- $Children(C_i) = \left\{C_k \in \mathcal{C} | C_k \notin \{C_{\sigma(1)}, C_{\sigma(2)} \ldots, C_i\}\right\} \subset \mathcal{C}$ represents the subset of classifiers through which the stream still needs to be processed after

it passes classifier C_i. This is a required identification information to be included in the header of each stream tuple such that the local classifier can know which classifiers still need to process the tuple.

- The throughput-to-goodput ratio $\theta_i = \frac{t^\sigma_{h-1}}{g^\sigma_{h-1}} \in [1, \infty]$ is a measure of the accuracy of the ordered set of classifiers $\{C_{\sigma(1)}, C_{\sigma(2)}, \ldots, C_i\}$. Indeed, $\theta_i = 1$ corresponds to perfect classifiers $C_{\sigma(1)}, C_{\sigma(2)}, \ldots, C_i$, (with $p^D = 1$ and $p^F = 0$), while larger θ_i imply that data has been either missed or wrongly classified.

  The state θ_i can be passed along from one classifier to the next in the stream tuple header. Since $\theta_i \in [1, \infty]$, the set of states $\mathcal{S}_i$ is of infinite cardinality. For computational reasons, we would require a finite set of actions. We will therefore approximate the throughput-to-goodput ratio by partitioning $[1, \infty]$ into L bins $S_l = [b_{l-1}, b_l]$ and approximate $\theta_i \in S_l$ by some fixed value $s_l \in S_l$.

4.2.3 Actions of a Classifier

Each classifier C_i has two independent actions: it selects its operating point x_i and it chooses among its children the trusted classifier $C_{i\rightarrow}$ to which it will transmit the stream. Hence $\mathcal{A}_i = \{(x_i, C_{i\rightarrow})\}$, where

- $x_i \in [0, 1]$ corresponds to the operating point selected by C_i.
- $C_{i\rightarrow} \in Children(C_i)$ corresponds to the classifier to which C_i will forward the stream. We will refer to $C_{i\rightarrow}$ as the trusted child of classifier C_i.

Note that the choice of trusted child $C_{i\rightarrow}$ is the local equivalent of the global order σ. The order is constructed classifier by classifier, each one selecting the child to which it will forward the stream: $\forall h \in [1, N]$, $C_{\sigma(h)} = C_{\sigma(h-1)\rightarrow}$.

4.2.4 Local Utility of a Classifier

We define the local utility of a chain of classifiers by backward induction:

$$U_{\sigma(h)} = -\rho_{\sigma(h)} t^\sigma_{h-1} + U_{\sigma(h+1)} \qquad \text{and} \qquad U_{\sigma(N)} = -\rho_{\sigma(N)} t^\sigma_{N-1} + g^\sigma_N - K t^\sigma_N. \tag{9}$$

The end-to-end utility of the chain of classifiers can then be reduced to $U = U_{\sigma(1)}$.

The key result of this section consists in the fact that the global optimum can be achieved locally with limited information. Indeed, each classifier $C_i = C_{\sigma(h)}$ will globally maximize the system's utility by autonomously maximizing its local utility $U_i = \underbrace{\begin{bmatrix} v^\sigma_h & w^\sigma_h \end{bmatrix}}_{=[v_i \; w_i]} \begin{bmatrix} t^\sigma_{h-1} \\ g^\sigma_{h-1} \end{bmatrix}$ where the local utility parameters $\begin{bmatrix} v^\sigma_h & w^\sigma_h \end{bmatrix}$ are defined recursively:

$$\begin{aligned}\left[\, v_N^\sigma \;\; w_N^\sigma \,\right] &= -\left[\, \rho_{\sigma(N)} \;\; 0 \,\right] + \left[\, -K \;\; 1 \,\right] T_N^\sigma \\ \left[\, v_h^\sigma \;\; w_h^\sigma \,\right] &= -\left[\, \rho_{\sigma(h)} \;\; 0 \,\right] + \left[\, v_{h+1}^\sigma \;\; w_{h+1}^\sigma \,\right] T_h^\sigma .\end{aligned}$$

This proposition can easily be proven recursively.

Therefore, the local utility of classifier C_i can now be rewritten as

$$U_i = \left(-\left[\, \rho_i \;\; 0 \,\right] + \left[\, v_{h+1}^\sigma \;\; w_{h+1}^\sigma \,\right] T_i^\sigma(x_i) \right) \begin{bmatrix} t_{h-1}^\sigma \\ g_{h-1}^\sigma \end{bmatrix}. \tag{10}$$

As such, the decision of classifier C_i only depends on its operating point x_i, on the state θ_i which it observes[5] and on the local utility parameters $\left[\, v_j \;\; w_j \,\right]$ of its children classifiers $C_j \in Children(C_i)$. Once it knows the utility parameters of all its children, classifier C_i can then uniquely determine its best action (i.e. its operating point x_i and its trusted child $C_{i\rightarrow}$) in order to maximize its local utility.

4.3 Decentralized Algorithms

At this stage, we consider classifiers with fixed operating points. The action of a classifier C_i is therefore limited to selecting the trusted child $C_{i\rightarrow} \in Children(C_i)$ to which it will forward the stream.

4.3.1 Exhaustive Search Ordering Algorithm

We will say that a classifier C_i *probes* a child classifier C_j when it requests its child utility parameters $\left[\, v_j \;\; w_j \,\right]$.

To best determine its trusted child, a classifier only requires knowledge on the utility parameters of all its children. We can therefore build a recursive algorithm as follows: all classifiers are probed by the source classifier C_0; to compute their local utility, each of the probed classifiers then probes its children for their utility parameters $\left[\, v \;\; w \,\right]$. To determine these, each of the probed children needs to probe its own children for their utility parameter, etc. The local utilities are computed in backwards order, from leaf classifiers to the root classifier C_0. The order yielding the maximal utility is selected.

Observe that this decentralized ordering algorithm leads to a full exploration of all $N!$ possible orders at each iteration. Achieving the optimal order only requires one iteration, but this iteration requires $O(N!)$ operations and may thus

[5] t_{i-1} and g_{i-1} are not required since: $\operatorname{argmax} U_i = \operatorname{argmax} \frac{U_i}{g_{i-1}} = \left(-\left[\, \rho_i \;\; 0 \,\right] + \left[\, v_{i+1} \;\; w_{i+1} \,\right] T_i^\sigma \right) \begin{bmatrix} \theta_i \\ 1 \end{bmatrix}$.

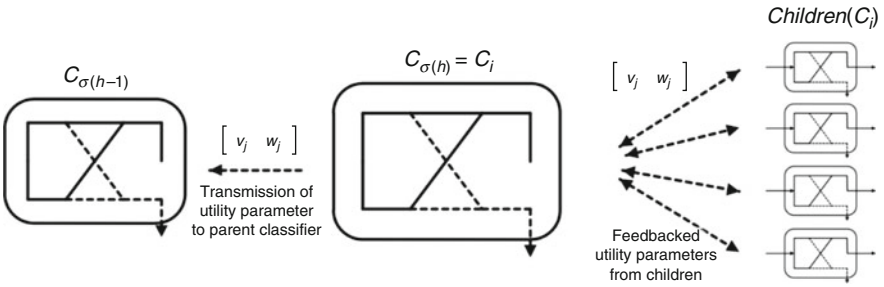


Fig. 11 Feedback information for decentralized algorithms

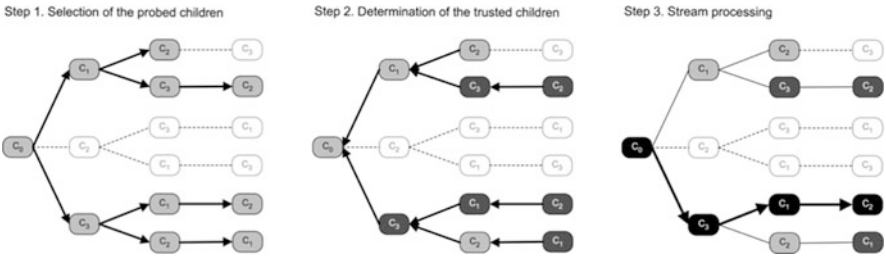


Fig. 12 Global Partial Search Algorithm only probes a selected subset of classifier orders

require substantial time, since heavy message exchange is required (Fig. 11). For quasi-stationary input data, the ordering could be performed offline and such computational time requirement would not affect the system's performance. However, in bursty and heterogeneous settings, we have to ensure that the optimal order calculated by the algorithm would not arrive too late and thus be completely obsolete. In particular, the time constraint $\tau \leq \mathfrak{C}\tau^{dyn}$, defined in Sect. 4.1 must not be violated.

We therefore need algorithms capable of quickly determining a good order, though convergence may require more than one iteration. In this way, it will be possible to reassess the order of classifiers on a regular basis to adapt to the environment.

4.3.2 Partial Search Ordering Algorithm

The key insight we want to leverage is to screen only through a selected subset of the $N!$ orders at each iteration. Instead of probing all its children classifiers systematically, the hth classifier will only request the utility parameters $\left[\ v\ w\ \right]$ of a subset of its $N - h$ children.

From a *global* point of view, one iteration can be decomposed in three major steps, as shown on Fig. 12:

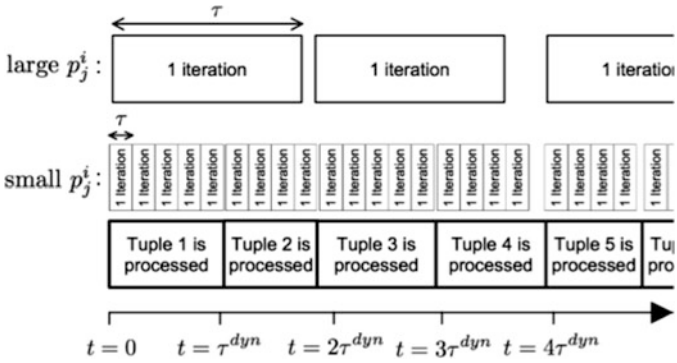


Fig. 13 Time scales for decentralized algorithms

Step 1: Selection of the Children to Probe A partial tree is selected recursively (light grey on Fig. 12). A subset of the N classifiers are probed as first classifier of the chain. Then, each of them selects the children it wants to probe, each of these children select the children which it wants to probe, etc.

Step 2: Determination of the Trusted Children The order to be chosen is determined backwards: utilities are computed from leaf classifiers to the source classifier C_0 based on feedback utility parameters. At each node of the tree, the child classifier which provides its parent with the greatest local utility is selected as the trusted child (dark grey on Fig. 12).

Step 3: Stream Processing The stream is forwarded from one classifier to its trusted child (black on Fig. 12).

If we want to describe Step 1 more specifically, classifier C_i will probe its child C_j with probability p_j^i. As will be shown in Sect. 4.5, adjusting the values of p_j^i will enable to adapt the number of operations and the time τ required per iteration, as shown on Fig. 13. Indeed, for low values of p_j^i, few of the $N!$ orders will be explored, and since each classifier only probes a small fraction of its children, one iteration will be very rapid. However, if the values of p_j^i are close to 1, each iteration requires a substantial amount of probing and one iteration will be long.

In the Partial Search Ordering Algorithm, one classifier may appear at multiple depths and positions in the classifiers' tree. Each time, it will realize a *local* algorithm described in the flowchart in Fig. 14.

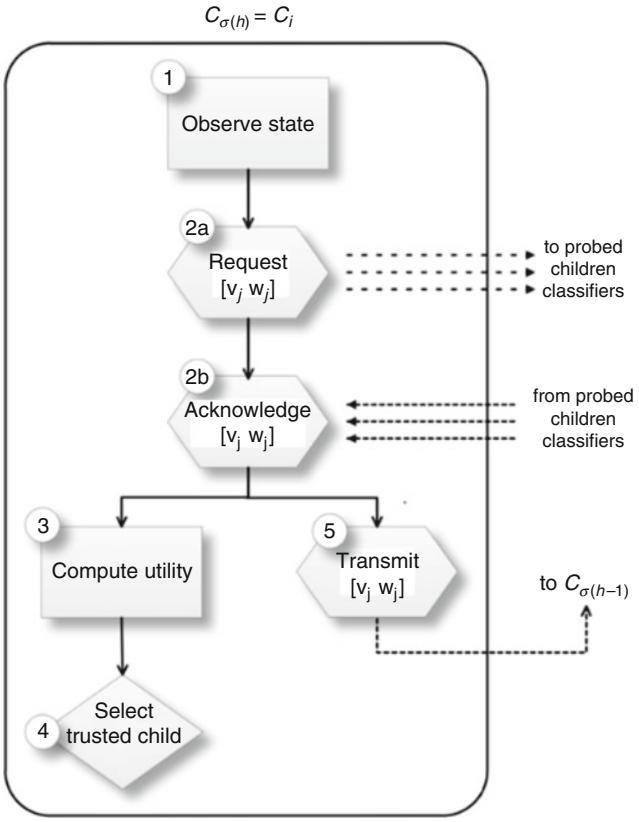


Fig. 14 Flowchart of local algorithm for partial search ordering

Decentralized Algorithm 3 Partial Search Ordering Algorithm—*for classifier* $C_i = C_{\sigma(h)}$

1. **Observe state** $(\theta_i, Children(C_i))$
2. With probability p^i_j, **request utility parameters** $\left[\, v_{\sigma(h+1)} \;\; w_{\sigma(h+1)} \,\right] = \left[\, v_j \;\; w_j \,\right]$ for any of the $N - h$ classifiers $C_j \in Children(C_i)$
3. For each child probed, **compute** corresponding **utility**

$$U_i(C_j) = \left(-\left[\, \rho_{\sigma(i)} \;\; 0 \,\right] + \left[\, v_j \;\; w_j \,\right] T_i^{\cdot 0}\right) \begin{bmatrix} t^{\sigma}_{h-1} \\ g^{\sigma}_{h-1} \end{bmatrix}$$

4. **Select** the child classifier with the highest U_i as **trusted child**.
5. Compute the corresponding $\left[\, v_i \;\; w_i \,\right]$ and **transmit** it to a previous classifier who requested it.

4.3.3 Decentralized Ordering and Operating Point Selection

In case of unfixed operating points, the local utility of classifier $C_i = C_{\sigma(h)}$ also depends on its local operating point x_i—but it does not directly depend on the operating points of other classifiers[6]:

$$U_i = \left(- \left[\rho_i \ 0 \right] + \left[v_{h+1}^{\sigma} \ w_{h+1}^{\sigma} \right] T_i^{\sigma}(x_i) \right) \begin{bmatrix} t_{h-1}^{\sigma} \\ g_{h-1}^{\sigma} \end{bmatrix}.$$

As a consequence, we can easily adapt the Partial Search Ordering Algorithm into a Partial Search Ordering and Operating Point Selection Algorithm by computing the maximal utility (in terms of x_i) for each child:

$$U_i(C_j) = \max_{x_i} \left(- \left[\rho_{\sigma(i)} \ 0 \right] + \left[v_j \ w_j \right] T_i^{\sigma}(x_i) \right) \begin{bmatrix} t_j \\ g_j \end{bmatrix}. \tag{11}$$

To solve the local optimization problem defined in Eq. (11), each classifier can either derive the nullity of the gradient if the ROC curve function $f_i : p^F \mapsto p^D$ is known, or search for optimal operating point using a dichotomy method (since $U_i(C_j)$ is concave).

4.3.4 Robustness of the Partial Search Algorithm and Convergence Speed

It can be shown that under stable conditions the Partial Search Algorithm converges and the equilibrium point of the stochastic decision process. For fixed operating point the Partial Search Algorithm converges to the optimal order if $p_j^i > 0 \ \forall \ i, j$.

In case of joint ordering and operating point selection, there exist multiple equilibrium points, each corresponding to a local minimum of the utility function. The selection of the equilibrium point among the set of possible equilibria depends on the initial condition (i.e. order and operating points) of the algorithm. To select the best equilibrium, we can perform the Partial Search Algorithm for multiple initial conditions and keep only the solution which yielded the maximum utility.

In practice, stable stream conditions will not be verified by the stream mining system, since the system's characteristics vary at a time scale of τ^{dyn}. Hence, rather than achieving convergence, we would like the Partial Search Algorithm to reach near-equilibrium fast enough for the system to deliver solution to the accuracy and delay joint optimization on a timely basis.

In analogy to [9], we first discuss how model-free Safe Experimentation, a heuristic case of Partial Search Algorithm can be used for decentralized stream mining and leads to a low-complexity algorithm, however with slow convergence

[6]The utility parameters $\left[v_j \ w_j \right]$ fed back from classifier C_j to classifier C_i are independent of any classifiers' operating points.

rate. Fortunately, the convergence speed of the Partial Search Algorithm can be improved by appropriately selecting the screening probabilities p_j^i. In Sect. 4.5, we will construct a model-based algorithm which enables to control the convergence properties of the Partial Search Algorithm, and lead to faster convergence.

4.4 Multi-Agent Learning in Decentralized Algorithm

We aim to construct an algorithm which would maximize as fast as possible the global utility of the stream mining system expressed in Eq. (4). We want to determine whether it is worthwhile for a classifier C_i to probe a child classifier C_j for its utility parameters and determine search probabilities p_j^i of the Partial Search Algorithm accordingly.

4.4.1 Tradeoff Between Efficiency and Computational Time

Define an experiment $E_{i\to j}$ as classifier C_i's action of probing a child classifier C_j by requesting its utility parameter $\left[\, v_j \;\; w_j \,\right]$. Performing an experiment can lead to a higher utility, but will induce a cost in terms of computational time:

- Denote by $\hat{U}(E_{i\to j}|s_k)$ the expected additional utility achieved by the stream mining system if the experiment $E_{i\to j}$ is performed under state s_k.
- Let τ^{ex} represent the expected amount of time required to perform an experiment. This computational time will be assumed independent of the classifiers involved in the experiment performed and the state observed.

Then, the total expected utility per iteration is given by $\hat{U}(p_j^i) = \sum p_j^i \hat{U}(E_{i\to j}|s_k)$ and the time required for one iteration is $\tau(p_i^j) = \hat{n}(p_j^i)\tau^{ex}$, where $\hat{n}(p_j^i)$ represents the expected number of experiments performed in one iteration of the Partial Search Algorithm and will be defined precisely in the next paragraph.

The allocation of the screening probabilities p_j^i aims to maximize the total expected utility within a certain time:

$$\begin{cases} \underset{p_j^i \in [0,1]}{\text{maximize}}\ \hat{U}(p_j^i) \\ \text{subject to}\ \tau(p_i^j) \le \mathfrak{C}\tau^{dyn} \end{cases}. \tag{12}$$

4.4.2 Safe Experimentation

We will benchmark our results on Safe Experimentation algorithms as cited in [9]. This low-complexity, model-free learning approach was first proposed for large-

scale, distributed, multi-agent systems, where each agent is unable to observe the actions of all other agents (due to informational or complexity constraints) and hence cannot build a model of other agents [17]. The agent therefore adheres to a "trusted" action at most times, but occasionally "explores" a different one in search of a potentially better action.

Safe Experimentation is a reinforcement learning algorithm where each classifier learns by observing the payoff with which its past actions were rewarded. As such, it does not consider the interactions between agents, or in our case, the actions of other autonomous classifiers. In particular, there is no explicit message exchange among classifiers required (i.e. no $\left[\, v \;\; w \,\right]$ exchanged), though each classifier needs to know the reward of its action (and computing this reward might yet require some form of implicit communication between agents).

The Safe Experimentation algorithm is initialized by selecting an initial order σ_0 of classifier. $C_{\sigma_0(h+1)}$ will be referred as the "trusted" child of the hth classifier $C_{\sigma_0(h)}$. At each time slot, $C_{\sigma(h)}$ will either forward the stream to its "trusted" child $C_{\sigma(h+1)}$ with probability $(1 - \varepsilon)$ or, with probability ε, will explore a classifier C_j chosen randomly among its children. In the case where a higher reward is achieved through exploration, C_j will become the new "trusted" child of $C_{\sigma(h)}$. Note that so long as $\epsilon > 0$, all possible orders will ultimately be explored, such that Safe Experimentation converges to the optimal order [9].

Instead of considering a fixed exploration rate ϵ, we can consider a diminishing exploration rate ϵ_t. In this way, the algorithm will explore largely for first iterations and focus on exploited orders near convergence. $\epsilon_t \rightarrow 0$ and $\prod_{t=1}^{\infty}\left(1 - \frac{{\epsilon_t}^{N-1}}{(N-1)!}\right) \rightarrow 0$ are sufficient conditions for convergence, typically verified for $\epsilon_t = (1/t)^{1/n}$.

Two majors limits of Safe Experimentation can be identified:

- **Slow convergence**: One iteration of Safe Experimentation is very rapid ($O(N)$), since only one order is experienced. However, the expected number of iterations required to converge to optimal order is bounded below by $N!$ (corresponding to uniform search: $\epsilon_t = 1$). As a consequence, the time required to reach the optimal solution might be infinitely long, since the optimal order could be experimented after an infinitely large number of iterations.
- **General approach**: This slow convergence can be explained by the fact that Safe Experimentation, as a model-free approach, does not leverage the structure of the problem studied. In particular, one major constraint fixed by Safe Experimentation is to try only one classifier among all its children, while the stream mining optimization problem allows to probe multiple children simultaneously by requesting their utility parameters $\left[\, v \;\; w \,\right]$ and selecting the trusted child based on these fed back values. This capacity to try multiple orders per iterations will enable to build a parameterized algorithm to speed-up the convergence to optimal order by choosing screening probabilities p^i_j appropriately.

4.5 Parametric Partial Search Order and Operating Point Selection Algorithm

As expressed in (12), the screening probabilities p^i_j can be used to tradeoff the expected utility and the computational cost. In this final section, we frame a general methodology aiming to determine the optimal tradeoff. In order to be adaptable to the setting considered, we construct our learning algorithm in three steps, each step representing a certain granularity level, and each step being controllable by one "macroscopic" state variable. Doing so, we put forward three tradeoffs corresponding to three independent questions: (1) how much to search?, (2) how deep to search?, (3) where to search?

This enables the construction of a parametric learning algorithm, extensively detailed in [8]. This article shows that the probability $p^i_j(p, \xi, \beta)$ that the hth classifier $C_{\sigma(h)} = C_i$ probes its children C_j, given that it received data with throughput-to-goodput ratio $\theta_i \in S_k$ can be expressed as:

$$p^i_j(p, \xi, \beta) = \underbrace{p}_{howmuch?} \times \underbrace{\frac{C'}{1 + e^{-\frac{h - [Np]}{\xi}}}}_{howdeep?} \times \underbrace{\frac{e^{\beta U_i(j,k)}}{\sum_{C_l \in Children(C_i)} e^{\beta U_i(l,k)}}}_{where?}.$$

The reader will find a justification of the formalism of p^i_j in [8].

4.5.1 Controlling the Screening Probability

Using this expression of p^i_j is meant to be able to control key characteristics of the screening probability by tuning parameters p,ξ and β.

The first parameter $p = \underset{i,j}{Av}(p^i_j)$ is used to arbitrate between rapid but inflexible search and slower but system-compliant search. Its value will impact the time τ required for one iteration and has to be selected small enough in order to ensure that $\tau \leq \mathfrak{C}\tau^{dyn}$, thus, coping with environment dynamics.

The second control parameter ξ is used to arbitrate between rapid but less secure search and slower but exhaustive search. It is a refinement parameter, which dictates how much more extensive search should be performed in the lower classifiers than in the upper ones.

- $\xi = 0$ corresponds to searching only for last classifiers and violates the exhaustivity of the search (no optimal convergence ensured).
- $0 < \xi < \infty$ corresponds to searching more exhaustively for last classifiers than for first classifiers.
- $\xi = \infty$ corresponds to searching uniformly at any depth with probability p.

The third parameter β balances the options of probing unexplored children versus exploiting already-visited orders, by weighting the propensity of classifier $C_i = C_{\sigma(h)}$ to probe one of its children classifier C_j, based on past experiments' reward. In most learning scenarios where the tradeoff between exploration and exploitation arises, both exploitation and exploration cannot be performed at the same time: the algorithm will either exploit well-known tracks or explore new tracks [14]. This is due to the absence of immediate feedback. In our setting, each classifier requests the information (i.e. utility parameters) of a subset of its children and is then able to base its decision on their feedback information.

In practice, the weight associated to a specific child based on its past reward could be determined using any increasing function f. Using $f_\beta(U) = \frac{e^{\beta U}}{\sum_V e^{\beta V}}$ is motivated by the analogy of a classifiers utility U to an energy [22]. In this case, $\frac{e^{\beta U}}{\sum_V e^{\beta V}}$ represents the equilibrium probability of being at an energy level U. As such, the parameter β can be interpreted as the inverse of a temperature, i.e. it governs the amount of excitation of the system:

- $\beta = 0$ corresponds to a very excited system with highly time-varying characteristics. In this case, since characteristics change very quickly, random exploration: $p^i_j = p^i$ is recommended by the algorithm.
- $\beta = \infty$ corresponds to a non-varying system. Then, full-exploitation of past rewards is recommended (given that all states were explored at least once) and weight should be concentrated only on the child which provides the maximum utility.
- $0 < \beta < \infty$ is a tradeoff between exploration ($\beta = 0$) and exploitation ($\beta = \infty$) and corresponds to settings where algorithmic search and environment evolution are performed at the same time scale.

4.5.2 Comparison of Ordering and Operating Point Selection Algorithms

Our preliminary results in Table 1 compare the performance of several joint ordering and operating point selection algorithms based on important criteria in the considered stream mining system.

4.5.3 Order Selected by Various Classifiers for Different Ordering Algorithms

The performance of the different ordering algorithms are shown in Table 2 for seven classifiers with fixed operating points per classifier. The classifier's characteristics (p^F, p^D) (i.e. the ROC curve), ψ (i.e. the *ex-post* selectivities), and α (i.e. the resource requirements) were generated randomly. The misdetection cost $c^M = 10$,

Table 1 Comparison of ordering and operating point selection algorithms

Ordering and operating point selection algorithm	System compliance	Utility achieved	Message exchange	Speed of convergence	Adaptability	Control
SQP-Greedy	Low	Bound; local opt.	Heavy	Medium	Little	∅
Safe experimentation	High	Local opt.	∅	Medium	∅	∅
Partial search	Complete	Local opt.	Light	Rapid	Total	Yes

Table 2 Utilities and computational time achieved for different ordering algorithms

	Algorithm	Order obtained	Utility	Comp. time
Centralized	Optimal	[C_6 C_2 C_1 C_4 C_3 C_7 C_5]	100	>5 min
	Greedy	[C_4 C_1 C_6 C_7 C_2 C_3 C_5]	95	0.002 s
Decentralized	Safe experimentation	[C_6 C_2 C_1 C_4 C_3 C_7 C_5]	100	2.09
	Partial search	[C_6 C_2 C_1 C_4 C_3 C_7 C_5]	100	1.2 s

false alarm cost $c^F = 1$, and $\lambda = 0.1$. The input data rate $t_0 = g_0$ was selected to normalize the optimal utility to 100.

As expected, the globally optimal centralized solution requires too much computation time, while the centralized Greedy algorithm does not lead to the optimal order, but results in very little computational time. The Parametric Partial Search Algorithm (here with $p = 0, 1$, $T = 1$ and $\beta = 0$) converges quicker than Safe Experimentation (here with $\epsilon = 0, 1$), to the optimal order. Decentralized algorithms converge to the optimal order, given that they ultimately probe all the possible orders, but they require longer computational time than the centralized greedy solutions. However, as shown in [8], convergence to a near-optimal order requires only a few iterations.

5 Online Learning for Real-Time Stream Mining

Mining dynamic and heterogeneous data streams using optimization tools may not always be feasible due to the unknown and time-varying distributions of these streams. Since classification of the streaming data needs to be done immediately, and invoking a classifier is costly, choosing the right classifier at run time is an important problem. In this section we review numerous methods that learn which classifiers to invoke based on the streaming meta-data, which is also called the context. All the algorithms studied in this section are able to mine big data streams in real-time. To accomplish this task, they are designed to have the following key properties:

- After a data instance is classified, the result is used to update the parameters of the learning algorithms. Then, the data instance is discarded. Therefore, it is not necessary to keep the past data instances in the memory.
- Usually, a single classifier is selected to classify the current data instance. As a result, only the accuracy estimate for the selected classifier is updated. While all the algorithms discussed in this section are capable of simultaneously updating the accuracies of all of the classifiers, this can lead to significant computational overhead due to the fact that it requires predictions from all of the classifiers for each data instance. Nevertheless, the performance bounds discussed in this section also holds for the case above.

5.1 Centralized Online Learning

This subsection is devoted to the study of centralized online learning algorithms for stream mining. We introduce several challenges related to real-time stream mining, various performance measures and the algorithms that address each one of the introduced challenges.

5.1.1 Problem Formulation

Each classifier takes as input a data instance and outputs a prediction. The data stream also includes a stream of meta-data, which is also referred to as the context stream. At time t, the data instance $s(t)$ is observed together with the context $x(t) \in \mathcal{X}$. The label $y(t) \in \{0, 1\}$ is not observed. The context can be categorical, real-valued and/or multi-dimensional. For instance, in a medical diagnosis application, the data instance can be an MRI image of a tissue, while the context can be resolution, type of the scanner, age of the patient and/or radius of the tumor. Depending on the particular application, the set of all possible contexts $\mathcal{X}$ can be very large and even infinite. The dimension of the context space is denoted by D. Each dimension of the context is called a *context type*. For instance, for the medical diagnosis example given above "age" is a context type, while the specific value that this type takes is the context.

No statistical assumptions are made on the context stream. However, the data and the label is assumed to be drawn from a fixed distribution given the context. This departs from the majority of the supervised learning literature, which assumes that the data is i.i.d. over time. Based on this, the accuracy of a classifier C given context x is defined to be $\pi_C(x) \in [0, 1]$. The classifier accuracies are not known beforehand and need to be learned online.

It is common to assume that the accuracy of a classifier is similar for similar contexts [25, 30]. For instance, in a social network users with similar age, income and geographic location will have a tendency to click on similar ads, which will result in a similar accuracy for a classifier that tries to predict the ad that the user

will click to. This assumption, which is also called the *similarity assumption*, is mathematically formalized as Hölder continuity of the accuracy of classifier c as a function of the context:

$$|\pi_C(x) - \pi_C(x')| \leq L \times \text{dist}(x, x')^{\alpha} \tag{13}$$

where L is the Hölder constant, α is the Hölder exponent and $\text{dist}(\cdot, \cdot)$ is a distance metric for the contexts. For most of the cases α is set to be 1, which makes the accuracy of classifier f Lipschitz continuous in the context [24].

The standard performance measure for online learning is the regret, which is defined as

$$\text{Reg}(T) := \sum_{t=1}^{T} \pi^*(x(t)) - \text{E}\left[\sum_{t=1}^{T} \pi_{a(t)}(x(t))\right] \tag{14}$$

where $\pi^*(x(t)) := \max_{C \in \mathcal{C}} \pi_C(x(t))$ and $a(t)$ denotes the classifier selected at time t. Hence, minimizing the regret is equivalent to selecting the best classifier as many times as possible. The time-averaged regret is defined as $\overline{\text{Reg}}(T) = \text{Reg}(T)/T$. $\overline{\text{Reg}}(T) \to 0$ implies that the average performance is (asymptotically) as good as the average performance of the best classifier selection policy given the complete knowledge of classifier accuracies. In order for the time-averaged regret to converge to zero, the regret must grow at most sublinearly over time, i.e., $\text{Reg}(T) \leq K T^{\gamma}$ for some constants $K > 0$ and $\gamma \in [0, 1)$ for all T.

5.1.2 Active Stream Mining

Online learning requires knowing the labels, in order to update the accuracy of the selected classifier. In most of the stream mining applications, such as medical diagnosis, acquiring the label is costly. Hence, a judicious mechanism that decides when to acquire the label based on the confidence on the accuracy of the selected classifier needs to be developed. The performance measure, i.e., the regret, also needs to be re-defined to capture the cost of label acquisition; hence, it becomes

$$\text{Reg}(T) := \sum_{t=1}^{T} \pi^*(x(t)) - \text{E}\left[\sum_{t=1}^{T} \pi_{a(t)}(x(t)) - J r(t)\right] \tag{15}$$

where $r(t)$ is 1 if label is acquired at time t and 0 otherwise, and J is a constant that represents the tradeoff between accuracy and label acquisition cost.

Since the number of possible contexts is usually very large, it is very inefficient to learn the classifier accuracies for each context separately. Therefore, the learning algorithms developed for stream mining learn the classifier accuracies for groups of similar contexts, where the groups are formed by partitioning the context space based on the similarity assumption given in (13). Then, the estimated accuracy of

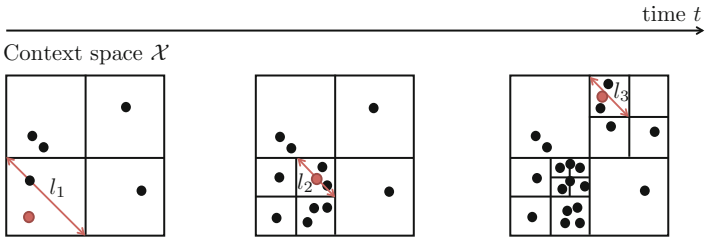


Fig. 15 Evolution of context space partition over time for $D = 2$. Red dots represent the most recent contexts and black dots represent the past contexts for which the label was acquired at the time of decision. Based on the similarity assumption, the type 2 error is proportional to l_j^{α}, $j = 1, 2, 3$, where l_j denotes the *diameter* of the group that the most recent context belongs to. On the other hand, the type 1 error decreases with the number of dots which belong to the group (square) that the current context belongs to

classifier C for context $x(t)$ is computed as $\hat{\pi}_C(x(t)) := \hat{\pi}_C(p(t))$ where $p(t)$ is the group that contains $x(t)$ in the partition of the context space. Here, $\hat{\pi}_C(p(t))$ is the sample mean of the correct predictions averaged over all past context arrivals to $p(t)$ for which the label was acquired. As shown in Fig. 15, this partition is adapted based on how the contexts arrive in order to balance the two sources of error in estimating the classifier accuracies: (1) type 1 error that arises from the number of past labeled data instances belonging to a group; (2) type 2 error that arises from the dissimilarity of the contexts that belong to the same group.

The label acquisition decision is also made to balance this tradeoff. Specifically, each label acquisition decreases the type 1 error of the selected classifier for the group that the current context belongs to. If accuracy of the selected classifier is known with a high confidence, then label acquisition is not necessary. In order to achieve this, the learning algorithm indefinitely alternates between two phases: exploration phase and exploitation phase, which are described below.

- **Exploration phase**: Select a classifier that the algorithm has a low confidence on its accuracy. After performing classification by the selected classifier, acquire the label of the data instance and update the accuracy estimate of the selected classifier.
- **Exploitation phase**: Select the classifier with the highest estimated accuracy, i.e., $a(t) = \arg\max_{C \in \mathcal{C}} \hat{\pi}_C(x(t))$.

After an exploration phase, the confidence on the accuracy of the selected classifier increases. Thus, classifier accuracies are learned in exploration phases. On the other hand, in an exploitation phase the prediction accuracy is maximized by classifying the data instance based on the empirically best classifier. In [25], it is shown that sublinear in time regret can be achieved by acquiring labels only sublinearly many times. While this regret bound holds uniformly over time, its dependence on T can be captured by using the asymptotic notation, which implies that $\mathrm{Reg}(T) = \tilde{O}(T^{(\kappa+D)/(\kappa'+D)})$, for some constants $\kappa' > \kappa > 0$. The specific

implementation keeps a control function $D(t)$, and explores only when the number of times the label is acquired by time t is less than or equal to $D(t)$. $D(t)$ increases both with t and the inverse of the diameter of the group that $x(t)$ belongs to. For this algorithm, the number of groups in the partition of the context space is also a sublinear function of time, which implies that the memory complexity of the algorithm is also sublinear in time. Moreover, identifying both the group that the current context belongs to and the empirically best classifier are computationally simple operations, which makes this algorithm suitable for real-time stream mining.

5.1.3 Learning Under Accuracy Drift

Since the data stream is dynamic, its distribution conditioned on the context can also change over time. This is called the concept drift [35]. It is straightforward to observe that the concept drift will also cause a change in the accuracy of the classifiers. For this setting, the time-varying accuracy of a classifier C for context x is denoted by $\pi_C(t, x)$. It is assumed that the accuracy *gradually drifts* over time, which can be written as

$$|\pi_C(t, x) - \pi_C(t', x')| \leq L \times \text{dist}(x, x')^{\alpha} + \frac{|t - t'|}{T_s}$$

where T_s denotes the *stability* of the concept. If T_s is large the drift is slow, while if T_s is small the drift is fast. Note that this assumption does not introduce any explicit restrictions on the data stream distribution. Hence, the accuracy drift is more general than the concept drift and can also model scenarios in which there is a change in the classifiers. For instance, in an application with SVM classifiers, some of the classifiers might be re-trained on-the-fly as more data instances arrive, which will result in a change in their decision boundaries, and hence their accuracies, even though the stream distribution remains the same.

An algorithm that learns and tracks the best classifier when there is accuracy drift is proposed in [26]. This algorithm estimates the classifier accuracies by using a recent time window of observations from similar contexts as opposed to using the entire past history of observations. In this work, the optimal window size is computed to be a sublinear function of T_s.

In general, it is not possible to achieve sublinear in time regret when there is accuracy drift due to the fact that the classifier accuracies are continuously changing. A constant rate of exploration is required in order to track the best classifier. A suitable performance measure for this setting is the time-averaged regret $\overline{\text{Reg}}(T)$. The algorithm proposed in [26] achieves a time-averaged regret of $\tilde{O}(T_s^{-\gamma})$, where $\gamma \in (0, 1)$ is a parameter that depends on α and the dimension of the context space. This implies that the time-averaged regret decreases as T_s increases, which is expected since it is easier to learn the classifier accuracies when the drift is slow.

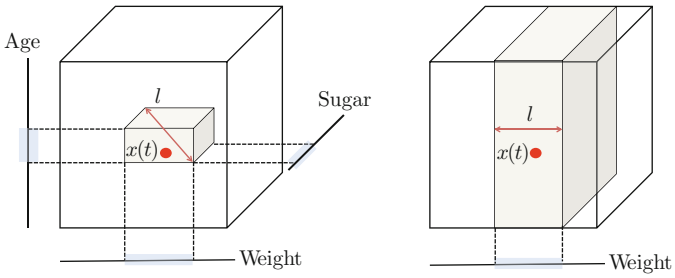


Fig. 16 A medical diagnosis example with three dimensional context space. Shaded areas represent the set of past observations that are used to estimate the classifier accuracies for the current context $x(t)$. On the left figure all context types are relevant, while on the right figure only the context type "weight" is relevant. The shaded areas on the left and right figures have the same type 2 error. However, the type 1 error for the shaded area on the right figure is much less than the one on the left figure since it includes many more past observations

5.1.4 Learning the Relevant Contexts

When the dimension D of the context space is large, the methods proposed in the previous sections which rely on partitioning the context space suffer from the curse of dimensionality as shown in Fig. 16. As a result, the regret bound given in Sect. 5.1.2 becomes almost linear in time.

It is possible to avoid the curse of dimensionality when the classifier accuracies depend only on a small subset of the set of all possible context types. In stream mining, this implies that there are many irrelevant context types which do not affect the outcome of the classification.[7] If the relevant context types were known, online learning could be easily performed by partitioning the context space restricted to the relevant context types. However, identifying these relevant context types on-the-fly without making any statistical assumptions on how the contexts arrive is a challenging task. Nevertheless, it is possible to identify the relevant context types through a sophisticated *relevance test*. This test identifies relevance assumptions that are consistent with the classifier accuracies estimated so far. The only requirements for this test are (13) and an upper bound on the number of relevant context types.

Here, we explain the relevance test that identifies one relevant context type for every classifier. The extension to more than one relevant context type can be found in [28]. It is important to note that the relevance test is only performed in exploitation phases as it requires confident accuracy estimates. First, for each context type i, the variation of the estimated accuracy of classifier C over all pairs of context types that include context type i is calculated. The resulting vector is called the *pairwise variation vector* of context type i. Then, a bound on the variation of the estimated accuracy of classifier C due to type 2 errors, which is called *natural variation*, is

[7]The definition of irrelevant context types can be relaxed to include all context types which have an effect that is less than $\epsilon > 0$ on the classifier accuracies.

calculated. The set of candidate relevant context types are identified as the ones for which the pairwise variation is less than a linear function of the natural variation. Finally, a context type i^* is selected from the set of candidate relevant types to be the estimated relevant context type, and the accuracies of the classifiers are re-calculated by averaging over all past observations whose type i^* contexts are *similar* to the current type i^* context. In [29] it is shown that online learning with relevance test achieves regret whose time order does not depend on D (but depends on the number of relevant context types). Hence, learning is fast and efficient, given that the number of relevant context types is much smaller than D.

5.2 Decentralized Online Learning

In this subsection, we consider how online learning can be performed in distributed classifier networks. We review two methods: cooperative contextual bandits in which local learners (LLs) cooperate to learn the best classifier to select within the network; hedged bandits in which an ensemble learner (EL) fuses the predictions of the LLs to maximize the chance of correct classification.

5.2.1 Problem Formulation

Most of the definitions and notations are the same as in Sect. 5.1.1. There are M data streams, each of which is processed locally by its LL. Each LL has a set of classifiers $\mathcal{C}_i$ that it can use to classify its incoming data stream. The set of all classifiers is denoted by $\mathcal{C} = \cup_{i=1}^{M} \mathcal{C}_i$. The context that is related to the ith data stream is denoted by $x_i(t)$, where $t \in \{1, 2, \ldots\}$.

5.2.2 Cooperative Contextual Bandits

In this part we describe a cooperative online learning framework that enables an LL to use other LLs' classifiers to classify its own data stream. For cooperative contextual bandits, the assumption on the context arrival process is the same as the assumption in Sect. 5.1.1.

In order to understand the benefit of cooperation, first we consider the case in which each LL acts individually. In this case, the highest accuracy that LL i can get for context x is $\pi_i^*(x) := \max_{C \in \mathcal{C}_i} \pi_C(x)$. On the other hand, if all LLs cooperate and share their classifiers with each other, the highest accuracy LL i can get for context x is $\pi_G^*(x) := \max_{C \in \mathcal{C}} \pi_C(x)$. Clearly, $\pi_G^*(x) \geq \pi_i^*(x)$. However, it is not straightforward for LLs to achieve a classification accuracy that is equal to $\pi_G^*(x)$ for all $x \in \mathcal{X}$ due to the following reasons:

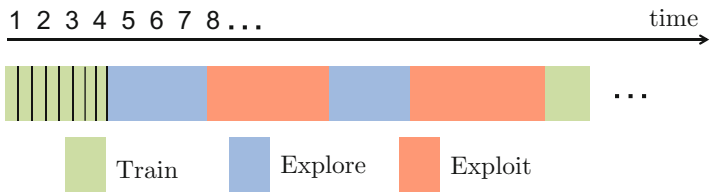


Fig. 17 Interleaving of the training, exploration and exploitation phases over time for a particular LL and context arrival process in cooperative contextual bandits

- LL i does not known the accuracies of its classifiers $\pi_C(x)$, $C \in \mathcal{C}_i$, $x \in \mathcal{X}$ a priori, and needs to learn these accuracies on-the-fly.
- LL i does not know the classifiers available to the other LLs. Moreover, other LLs may be reluctant to share such an information due to privacy constraints.
- LL i cannot observe the data streams arriving to other LLs due to both privacy and communication constraints. However, LL i is able to send selected data instances to other LLs (possibly by incurring some communication cost) and is able to receive selected data instances from the other LLs.

The challenging decentralized learning problem stated above is solved by designing a decentralized learning algorithm that promotes cooperation among the learners [27]. In this algorithm, each learner alternates between three different phases over time as given in Fig. 17. The sequencing of these phases is adapted online based on the context arrival process. In each phase the learning algorithm takes an *action* for a different purpose:

- **Training phase**: LL i selects another LL and sends its context and data instance to the selected LL. Then, the selected LL is asked to classify the data instance. After the classification is performed and the true label is received by LL i, this true label is also send to the selected LL in order for it to update the accuracy of the classifier that it had selected on behalf of LL i. Hence, the purpose of the training phase is to train other LLs such that they learn the accuracies of their own classifiers with a high confidence for contexts that arrive to LL i.
- **Exploration phase**: LL i selects one of its own classifiers or another LL for the purpose of learning the accuracy.
- **Exploitation phase**: LL i selects one of its own classifiers or another LL for the purpose of maximizing the probability of correct classification.

Due to the heterogeneity of the data streams, usually it is not possible for a single LL to learn its classifier accuracies well for all possible contexts by just observing its own data stream. This can happen because a context that is rare for one LL can be frequent for another LL. While this results in asymmetric learning, it is solved by the training phase.

Note from Fig. 17 that exploration is performed only when there is no need for training. This is to ensure that if another LL is selected to make a classification, it performs classification based on its best classifiers. Otherwise, LL i might learn the

accuracy of the other LLs incorrectly, which might results in failure to identify an LL, whose accuracy is higher than the accuracies of the classifiers in $\mathcal{C}_i$. Similarly, exploitation is performed only when there is no need to train any other LL or to explore any other LL or classifier.

One important question is how much training and exploration is needed. This can be analytically solved by defining confidence intervals for the sample mean (empirical) estimates of the classifier accuracies, and adjusting these confidence intervals over time to achieve a certain performance goal. In cooperative contextual bandits, the regret of LL i is defined as

$$\mathrm{Reg}_i(T) := \sum_{t=1}^{T} \pi^*_G(x_i(t)) - \mathrm{E}\left[\sum_{t=1}^{T} \pi_{a_i(t)}(x_i(t))\right] \tag{16}$$

where $\pi_{a_i(t)}(x_i(t))$ denotes the accuracy of the classifier selected by LL $a_i(t)$ on behalf of LL i for $a_i(t) \notin \mathcal{C}_i$. Again, we seek to achieve sublinear in time regret, which implies that the learning algorithm's average number of correct predictions converges to that of the $\pi^*_G(x)$.

Specifically, it is proven in [27] that sublinear regret can be achieved by an algorithm that uses sublinear number of training and exploration phases. In order to achieve sublinear regret, the classifier accuracies must also be learned together for similar contexts by a context space partitioning method such as the one given in Fig. 15.

5.2.3 Hedged Bandits

Hedged Bandits model decentralized stream mining applications in which all data streams are related to the same event. Hence, it is assumed that the contexts, data instances and labels are drawn according to an i.i.d. process. LLs produce predictions by choosing classifiers according to their own learning algorithms, classifiers and data streams, and then, send these predictions to an EL, which fuses the predictions together to produce a final prediction. The learning algorithm used by the LLs is similar to the learning algorithms discussed in Sect. 5.1. On the other hand, the EL uses a variant of the Hedge algorithm [10] that does not require the time horizon to be known in advance. This guarantees that the ELs prediction is as good as the predictions of the best LL given the context [30].

6 Conclusion

Adapting in real time the topology of classifiers and their configuration (operating point) enables to significantly improve the performance of stream mining systems, by optimally trading up accuracy and delay, under limited resources.

However, the emergence of new stream mining applications, with specific requirements and characteristics, widens the spectrum of possibilities and leaves room for further improvements. Today, more and more data is available to be processed, and more and more classification, filtering, analysis or sorting can be performed on this data. As such, a major challenge lies in identifying, prioritizing and planning mining tasks. Until now, the mapping between queries and a set of corresponding classifiers was considered as given. Yet, this mapping should be decided jointly with the topology construction and the system configuration for an optimal stream mining design.

An upstream consideration would be to decide whether streams should be systematically classified or only identified upon request. Indeed, a stream mining system must not only be seen as a query-centric processing system aiming to identify which subset of data answers a given set of queries. Instead of defining the set of classifiers on the basis of the set of queries ($\mathcal{C} = \bigcup_{q \in \mathcal{Q}} \mathcal{C}(q)$), we could determine what are all the queries which can be answered given a set of classifiers ($\mathcal{Q} = \{q \mid \mathcal{C}(q) \subset \mathcal{C}\}$). Since such classifier-centric approach leads to an explosion of the number of queries which can be processed (N binary classifiers can potentially process $\sum_{k=1}^{N} \frac{N!}{k!(N-k)!} 2^k$ different queries) it is critical to be able to identify or to learn online which classification to perform.

Indeed, classifier design is an expensive process and determining which feature to extract represents a major topic in the data mining community. Hence, given a data stream and a query, deciding which classifiers should match which queries has not yet been analytically studied. Underlying this issue resonates the exploration versus exploitation tradeoff, where we need to train the stream mining system to detect the classifiers which are critical to stream identification.

Acknowledgements This work is based upon work supported by the National Science Foundation under Grant No. 1016081. We would like to thank Dr. Deepak Turaga (IBM Research) for introducing us to the topic of stream mining and for many productive conversation associated with the material of this chapter as well as providing us with Figs. 1 and 3 of this chapter. We also would like to thank Dr. Fangwen Fu and Dr. Brian Foo, who have been PhD students in Prof. van der Schaar group and have made contributions to the area of stream mining from which this chapter benefited. Finally, we thank Mr. Siming Song for kindly helping us with formatting and polishing the final version of the chapter.

References

1. Babcock, B., Babu, S., Datar, M., Motwani, R.: Chain: Operator scheduling for memory minimization in data stream systems. In: Proc. ACM International Conference on Management of Data (SIGMOD), pp. 253–264 (2003)
2. Babu, S., Motwani, R., Munagala, K., Nishizawa, I., Widom, J.: Adaptive ordering of pipelined stream filters. In: ACM SIGMOD International Conference on Management of Data (2004)
3. Boyd, S.P., Vandenberghe, L.: Convex Optimization. Cambridge University Press (1994)
4. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., Zdonik, S.: Scalable distributed stream processing. In: Proc. of Conference on Innovative Data Systems Research, Asilomar (2003)

5. Cherniack, M., Balakrishnan, H., Carney, D., Cetintemel, U., Xing, Y., Zdonik, S.: Scalable distributed stream processing. In: Proc. CIDR (2003)
6. Condon, A., Deshpande, A., Hellerstein, L., Wu, N.: Flow algorithm for two pipelined filter ordering problems. In: ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (2006)
7. Douglis, F., Branson, M., Hildrum, K., Rong, B., Ye, F.: Multi-site cooperative data stream analysis. ACM SIGOPS **40**(3) (2006)
8. Ducasse, R., Turaga, D.S., van der Schaar, M.: Adaptive topologic optimization for large-scale stream mining. IEEE Journal on Selected Topics in Signal Processing **4**(3), 620–636 (2010)
9. Foo, B., van der Schaar, M.: Distributed classifier chain optimization for real-time multimedia stream-mining systems. In: Proc. IS&T / SPIE Multimedia Content Access, Algorithms and Systems II (2008)
10. Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. In: Proc. European Conference on Computational Learning Theory, pp. 23–37 (1995)
11. Fu, F., Turaga, D.S., Verscheure, O., van der Schaar, M., Amini, L.: Configuring competing classifier chains in distributed stream mining systems. IEEE Journal on Selected Topics in Signal Processing (2007)
12. Gaber, M., Zaslavsky, A., Krishnaswamy, S.: Resource-aware knowledge discovery in data streams. In: Proc. First Intl. Workshop on Knowledge Discovery in Data Streams (2004)
13. Garg, A., Pavlovic, V.: Bayesian networks as ensemble of classifiers. In: Proc. 16th International Conference on Pattern Recognition (ICPR), pp. 779–784 (2002)
14. Gupta, A., Smith, K., Shalley, C.: The interplay between exploration and exploitation. Academy of Management Journal (2006)
15. Hu, J., Wellman, M.: Multiagent reinforcement learning: Theoretical framework and an algorithm. In: Proceedings of the Fifteenth International Conference on Machine Learning (1998)
16. Low, S., Lapsley, D.E.: Optimization flow control I: Basic algorithm and convergence. IEEE/ACM Trans. Networking **7**(6), 861–874 (1999)
17. Marden, J., Young, H., Arslan, G., Shamma, J.: Payoff based dynamics for multi-player weakly acyclic games. SIAM Journal on Control and Optimization, special issue on Control and Optimization in Cooperative Networks (2007)
18. Merugu, S., Ghosh, J.: Privacy-preserving distributed clustering using generative models. In: Proc. of 3rd International Conference on Management of Data, pp. 211–218 (2003)
19. Olston, C., Jiang, J., Widom, J.: Adaptive filters for continuous queries over distributed data streams. In: Proc. ACM SIGMOD Intl. Conf. Management of Data, pp. 563–574 (2003)
20. Palomar, D., Chiang, M.: On alternative decompositions and distributed algorithms for network utility problems. In: Proc. IEEE Globecom (2005)
21. Park, H., Turaga, D.S., Verscheure, O., van der Schaar, M.: Foresighted tree configuring games in resource constrained distributed stream mining systems. In: Proc. IEEE Int. Conf. Acoustics Speech and Signal Process. (2009)
22. Saul, L., Jordan, M.I.: Learning in Boltzmann trees. Neural Computation (1994)
23. Schapire, Y.: A brief introduction to boosting. In: Proc. International Conference on Algorithmic Learning Theory (1999)
24. Slivkins, A.: Contextual bandits with similarity information. Journal of Machine Learning Research **15**(1), 2533–2568 (2014)
25. Tekin, C., van der Schaar, M.: Active learning in context-driven stream mining with an application to image mining. IEEE Transactions on Image Processing **24**(11), 3666–3679 (2015)
26. Tekin, C., van der Schaar, M.: Contextual online learning for multimedia content aggregation. IEEE Transactions on Multimedia **17**(4), 549–561 (2015)
27. Tekin, C., van der Schaar, M.: Distributed online learning via cooperative contextual bandits. IEEE Transactions Signal Processing **63**(14), 3700–3714 (2015)

28. Tekin, C., van der Schaar, M.: RELEAF: An algorithm for learning and exploiting relevance. IEEE Journal of Selected Topics in Signal Processing **9**(4), 716–727 (2015)
29. Tekin, C., Van Der Schaar, M.: Discovering, learning and exploiting relevance. In: Advances in Neural Information Processing Systems, pp. 1233–1241 (2014)
30. Tekin, C., Yoon, J., van der Schaar, M.: Adaptive ensemble learning with confidence bounds. IEEE Transactions on Signal Processing **65**(4), 888–903 (2017)
31. Turaga, D., Verscheure, O., Chaudhari, U., Amini, L.: Resource management for networked classifiers in distributed stream mining systems. In: Proc. IEEE ICDM (2006)
32. Vaidya, J., Clifton, C.: Privacy-preserving k-means clustering over vertically partitioned data. In: Proc. of 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 206–215 (2003)
33. Varshney, P.: Distributed Detection and Data Fusion. Springer (1997). ISBN: 978-0-387-94712-9
34. Vazirani, V.: Approximation Algorithms. Springer Verlag, Inc., New York, NY, USA (2001)
35. Žliobaitė, I.: Learning under concept drift: an overview. arXiv preprint arXiv:1010.4784 (2010)

Deep Neural Networks: A Signal Processing Perspective

Heikki Huttunen

Abstract Deep learning has rapidly become the state of the art in machine learning, surpassing traditional approaches by a significant margin for many widely studied benchmark sets. Although the basic structure of a deep neural network is very close to a traditional 1990s style network, a few novel components enable successful training of extremely deep networks, thus allowing a completely novel sphere of applications—often reaching human-level accuracy and beyond. Below, we familiarize the reader with the brief history of deep learning and discuss the most significant milestones over the years. We also describe the fundamental components of a modern deep neural networks and emphasize their close connection to the basic operations of signal processing, such as the convolution and the Fast Fourier Transform. We study the importance of pretraining with examples and, finally, we will discuss the real time deployment of a deep network; a topic often dismissed in textbooks; but increasingly important in future applications, such as self driving cars.

1 Introduction

The research area of artificial intelligence (AI) has a long history. The first ideas of intelligent machines were raised shortly after the first computers were invented, in the 1950s. The excitement around the novel discipline with great promises led into one of the first technological hypes in computer science: In particular, military agencies such as ARPA funded the research generously, which led into a rapid expansion of the area during the 1960s and early 1970s.

As in most hype cycles, the initial excitement and high hopes were not fully satisfied. It turned out that intelligent machines able to seamlessly interact with the natural world are a lot more difficult to build than initially anticipated. This led into a

H. Huttunen (✉)
Tampere University of Technology, Tampere, Finland
e-mail: heikki.huttunen@tut.fi

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_4

period of recession in artificial intelligence often called "The AI Winter"[1] during the end of 1970s. In particular, the methodologies built on top of the idea of modeling the human brain had been the most successful ones, and also the ones that suffered the most during the AI winter as the funding essentially ceased to exist for topics such as neural networks. However, the research of learning systems still continued under different names—machine learning and statistics.

The silent period of AI research was soon over, as the paradigm was refocused to study less ambitious topics than the complete human brain and seamless human-machine interaction. In particular, the rise of expert systems and the introduction of a closely related topic *data mining* led the community towards new, more focused topics.

At the beginning of 1990s the research community had already accepted that there is no silver bullet that would solve all AI problems at least in the near future. Instead, it seemed that more focused problems could be solved with tailored approaches. At the time, several successful companies had been founded, and there were many commercial uses for AI methodologies, such as the neural networks that were successful at the time. Towards the end of the century, the topic got less active and researchers directed their interest to new rising domains, such as kernel machines [35] and big data.

Today, we are in the middle of the hottest AI summer ever. Companies such as Google, Apple, Microsoft and Baidu are investing billions of dollars to AI research. Top AI conferences—such as the NIPS[2]—are rapidly sold out. The consultancy company Gartner has machine learning and deep learning at the top of their hype curve.[3] And AI even made its way to a perfume commercial.[4]

The definitive machine learning topic of the decade is *deep learning*. Most commonly, the term is used for referring to neural networks having a large number of layers (up to hundreds; even thousands). Before the current wave, neural networks were actively studied during the 1990s. Back then, the networks were significantly smaller and in particular more shallow. Adding more than two or three hidden layers only degraded the performance. Although the basic structure of a deep neural network is very close to a traditional 1990s style network, a few novel components enable successful training of extremely deep networks, thus allowing a completely novel sphere of applications—often reaching human-level accuracy and beyond.

After a silent period of the 2000s, neural networks returned to the focus of machine intelligence after Prof. Hinton from University of Toronto experimented with unconventionally big networks using unsupervised training. He discovered that training of large and deep networks was indeed possible with an unsupervised pretraining step that initializes the network weights in a layerwise manner. In the unsupervised setup, the model first learns to represent and synthesize the data

[1] https://en.wikipedia.org/wiki/History_of_artificial_intelligence.

[2] http://nips.cc/.

[3] http://www.gartner.com/newsroom/id/3784363.

[4] http://www.gq.com/story/alexandre-robicquet-ysl-model.

without any knowledge on the class labels. The second step then transforms the unsupervised model into a supervised one, and continues learning with the target labels. Another key factor to the success was the rapidly increased computational power brought by recent Graphics Processing Units (GPU's).

For a few years, different strategies of unsupervised weight initialization were at the focus of research. However, within a few years from the breakthroughs of deep learning, the unsupervised pretraining became obsolete, as new discoveries enabled direct supervised training without the preprocessing step. There is still a great interest in revisiting the unsupervised approach in order to take advantage of large masses of inexpensive unlabeled data. Currently, the fully supervised approach together with large annotated data produces clearly better results than any unsupervised approach.

The most successful application domain of deep learning is *image recognition*, which attempts to categorize images according to their visual content. The milestone event that started this line of research was the famous Alexnet network winning the annual Imagenet competition [24]. However, other areas are rising in importance, including sequence processing, such as natural language processing and machine translation.

This chapter is a brief introduction to the essential techniques behind deep learning. We will discuss the standard components of deep neural network, but will also cover some implementation topics from the signal processing perspective.

The remainder of the chapter is organized as follows. In Sect. 2, we will describe the building blocks of a modern neural network. Section 3 discusses the training algorithms and objective functions for the optimization. Finally, Sect. 4 discusses the tools for training and compares popular training platforms. We also present an example case where we compare two design strategies with examples using one of the most popular deep learning packages. Finally, Sect. 5 considers real time deployment issues in a framework where deep learning is used as one component of a system level deployment.

2 Building Blocks of a Deep Neural Network

2.1 Neural Networks

Neural networks are the core of modern artificial intelligence. Although they originally gained their inspiration from biological systems—such as the human brain—there is little in common with contemporary neural networks and their carbon-based counterparts. Nevertheless, for the sake of inspiration, let us take a brief excursion to the current understanding of the operation of biological neural networks.

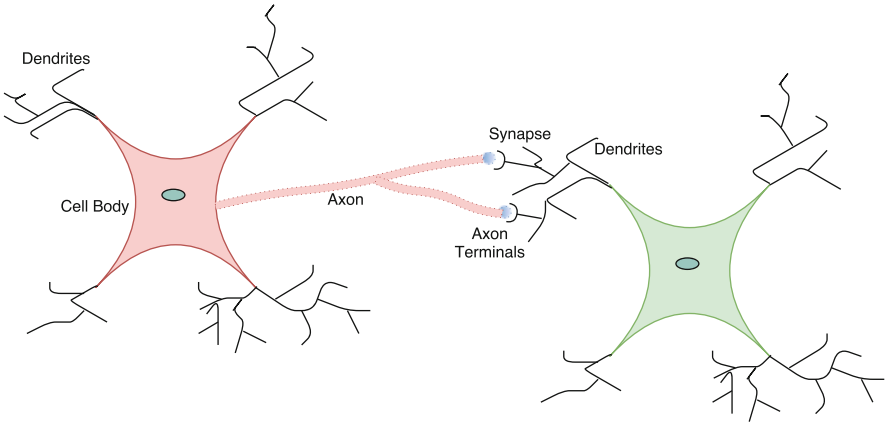


Fig. 1 A simple biological neuron network. Reprinted with permission from [45]

Figure 1 illustrates a simple biological neural network consisting of two nerve cells. The information propagates between the cells essentially through two channels: the *axon* is the transmitting terminal forwarding the level of activation to neighboring nerve cells. On the other hand, a *dendrite* serves as the receiving end, and the messages are passed through a synaptic layer between the two terminals.

Historically, the field of neural network research started in its simplest form in the 1950s, when researchers of electronics got excited about recent advances in neurology, and started to formulate the idea of an electronic brain consisting of *in silico* nerve cells that propagate their state of activity through a network of artificial cells. A landmark event of the time was the invention of Rosenblatt's *perceptron*, which uses exactly the same generalized linear model as any two-class linear classifier, such as Linear Discriminant Analysis, Logistic Regression, or the Support Vector Machine:

$$\text{Class}(\mathbf{x}) = \begin{cases} 1, & \text{if} \quad \mathbf{w}^T\mathbf{x} + b \geq 0, \\ 0, & \text{if} \quad \mathbf{w}^T\mathbf{x} + b < 0, \end{cases}$$

where $\mathbf{x} \in \mathbb{R}^N$ is the test vector to be classified, $\mathbf{w} \in \mathbb{R}^N$ and $b \in \mathbb{R}$ are the model parameters (weight vector and the bias) learned from training data. More compactly, we can write the model as $\sigma(\mathbf{w}^T\mathbf{x} + b)$ with $\sigma()$ the threshold function at zero.

The only thing differentiating the perceptron from the other generalized linear models is the training algorithm. Although not the first, nor the most powerful training algorithm, it emphasizes the idea of *iterative* learning, which presents the model training samples one at the time. Namely, the famous linear discriminant algorithm was proposed by Fisher already in the 1930s [12], but it was not used in an iterative manner due to existence of a closed form solution. When the perceptron algorithm was proposed, time was ready for exploitation of recently appeared digital computing. The training algorithm has many things in common with modern deep learning training algorithms, as well:

1. Initialize the weight vector $\mathbf{w}$ and the bias b at random.
2. For each sample $\mathbf{x}_i$ and target y_i in the training set:
   a. Calculate the model output $\hat{y}_i = \sigma(\mathbf{w}^T \mathbf{x}_i + b)$.
   b. Update the network weights by

$$\mathbf{w} := \mathbf{w} + (y_i - \hat{y}_i)\mathbf{x}_i .$$

The steps 2a and 2b correspond to the *forward pass* and *backward pass* of contemporary networks, where the samples are first propagated forward through the model to produce the output, and the error terms are pushed back as weight updates through the network in the backward pass.

For signal processing researchers, the idea of perceptron training is familiar from the field of adaptive signal processing and the Least Mean Squares (LMS) filter in particular. Coincidentally, the idea of the LMS filter was inspired by the famous Darthmouth AI meeting in 1957 [43], exactly the same year as Rosenblatt first implemented his perceptron device able to recognize a triangle held in front of its camera eye.

Fast-forwarding 30 years brings us to the introduction of the *backpropagation algorithm* [32], which enabled the training of *multilayer* perceptrons, i.e., layers of independent perceptrons stacked into a network. The structure of a 1980s multilayer perceptron is illustrated in Fig. 2. In the left, the m-dimensional input vector is fed to the first hidden layer of processing nodes (blue). Each hidden layer output is then fed to the next layer and eventually to the output layer, whose outputs are considered as class likelihoods in the classification context. The structure of each processing node is in turn illustrated in Fig. 3. Indeed, the individual neuron of Fig. 3 is very close to the 60-year-old perceptron, with a dot product followed by an activation function. This is still the exact neuron structure today, with the exception that there now exists a large library of different activations apart from the original hard thresholding, as later discussed in Sect. 2.4.

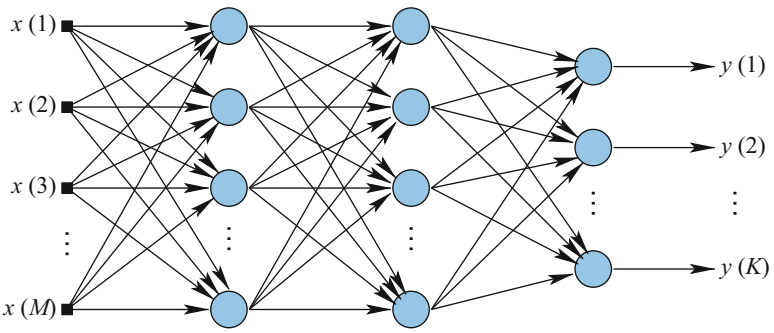


Fig. 2 A multilayer perceptron model

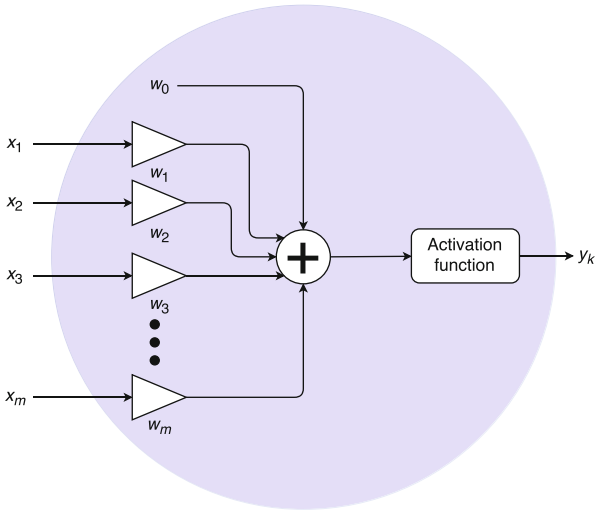


Fig. 3 A single neuron of the feedforward network of Fig. 2

2.2 Convolutional Layer

The standard layers of the 80s are today called *dense* or *fully connected* layers, reflecting their structure where *each* layer output is fed to *each* next layer node. Obviously, dense layers require a lot of parameters, which is expensive from both computational and learning point of view: High number of model coefficients require a lot of multiplications during training and deployment, but also their inference in training time is a nontrivial task. Consider, for example, the problem of image recognition (discussed later in Sect. 4.2), where we feed 64×64 RGB images into a network. If the first hidden layer were a dense layer with, say, 500 neurons, there would be over six million connections ($64 \times 64 \times 3$-dimensional input vector fed to 500 nodes requiring $500 \times 64 \times 64 \times 4 = 6{,}144{,}000$ connections). Moreover, the specific inputs would be very sensitive to geometric distortions (translations, rotations, scaling) of the input image, because each neuron can only see a single pixel at the time.

Due to these reasons, the *convolutional layer* is popular particularly in image recognition applications. As the name suggests, the convolutional layer applies a 2-dimensional convolution operation to the input. However, there are two minor differences to the standard convolution of an image processing textbook.

First, the convolution operates on multichannel input; i.e., it can see all channels of the three-channel (RGB) input image. In other words, denote the input to a convolutional layer as $\mathbf{X} \in \mathbb{R}^{M \times N \times C}$ and the convolution window as $\mathbf{W} \in \mathbb{R}^{J \times K \times C}$. Then, the output $y_{m,n}$ at spatial location (m, n) is given as

$$y_{m,n} = \sum_c \sum_j \sum_k \mathbf{W}_{j,k,c} \mathbf{X}_{m+j,n+k,c}, \tag{1}$$

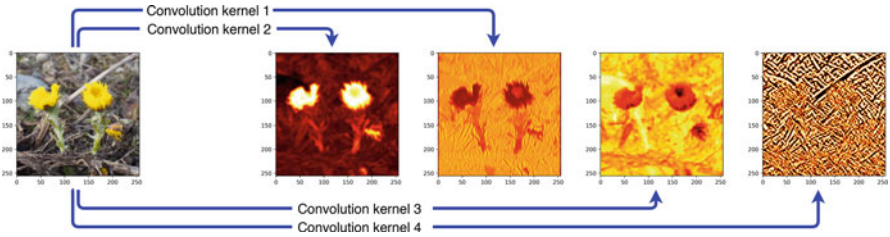


Fig. 4 Four feature maps produced from the input image by different convolution kernels

with the summation indices spanning the local window, i.e., $c = 1, 2, \ldots, C$; $j = -\lfloor \frac{J}{2} \rfloor, \ldots, \lfloor \frac{J}{2} \rfloor$ and $k = -\lfloor \frac{K}{2} \rfloor, \ldots, \lfloor \frac{K}{2} \rfloor$ assuming odd J and K. Alternatively, the convolution of Eq. (1) can also be thought of as a 3D convolution with window spanning all channels: The window only moves in the spatial dimensions, because there is no room for sliding in the channel dimension.

Second, the deep learning version of convolution does not reflect the convolution kernel with respect to the origin. Thus, we have the expression $\mathbf{X}_{m+j,n+k,c}$ in Eq. (1) instead of $\mathbf{X}_{m-j,n-k,c}$ of a standard image processing textbook. The main reason for this difference is that the weights are learned from the data, so the kernel can equally well be defined either way, and the minus is dropped out due to simplicity. Although this is a minor detail, it may cause confusion when attempting to re-implement a deep network using traditional signal processing libraries.

The role of the convolutional layer can be understood from the example of Fig. 4, where we have applied four $3 \times 3 \times 3$ convolutions to the input image. In this case, the convolution kernels highlight different features: yellow regions, green regions, diagonal edges and so on. With real convolutional networks, the kernels are learned from the data, but their role is nevertheless to extract the features essential for the application. Therefore, the outputs of the convolutions are called *feature maps* in the deep learning terminology.

In summary, the convolutional layer receives a stack of C channels (e.g., RGB), filters them with D convolutional kernels of dimension $J \times K \times C$ to produce D feature maps. Above, we considered the example with 64×64 RGB images fed to a dense layer of 500 neurons, and saw that this mapping requires 6,144,500 coefficients. For comparison, suppose we use a convolutional layer instead, producing 64 feature maps with 5×5 spatial window. In this case, each kernel can see all three channels within the local 5×5 spatial window, which requires $5 \times 5 \times 3$ coefficients. Together, all 64 convolutions are defined by $64 \times 5 \times 5 \times 3 = 4800$ parameters—over 1200 times less than for a dense layer. Moreover, the parameter count of the convolutional layer *does not depend on the image size* unlike the dense layer.

The computation of the convolution is today highly optimized using the GPU. However, the convolution can be implemented in several ways, which we will briefly discuss next. The trivial option is to compute the convolution directly using Eq. (1). However, in a close-to-hardware implementation, there are many special cases that would require specialized optimizations [6], such as small/large

spatial window, small/large number of channels, or small/large number of images in batch processing essential in the training time. Although most cases can be optimized, maintaining a large number of alternative implementations soon becomes burdensome.

The second alternative is to use the Fast Fourier Transform (FFT) via the convolution theorem:

$$w(n,m) * x(n,m) = \mathrm{w.f}^{-1}\left(W(n,m)^* \odot X(n,m)\right), \tag{2}$$

where $W(n,m) = \mathrm{F}(w(n,m))$ and $X(n,m) = \mathrm{F}(x(n,m))$ are the discrete Fourier transforms of the convolution kernel $w(n,m)$ and one channel of the input $x(n,m)$, respectively. Moreover, F^{-1} denotes the inverse discrete Fourier transform, and * the complex conjugation that reflects the kernel about the origin in spatial domain. The obvious benefit of this approach is that the convolution is transformed to low-cost elementwise (Hadamard) product in the Fourier domain, and the computation of the FFT is faster than the convolution ($O(N \log N)$ vs. $O(N^2)$). However, the use of this approach requires that $w(n,m)$ and $x(n,m)$ are zero-padded to same size, which consumes a significant amount of temporary memory, when the filter and image sizes are far from each other. Despite these challenges, the FFT approach has shown impressive performance improvement with clever engineering [40].

The third widely used approach transforms the convolution into matrix multiplication, for which extremely well optimized implementations exist. The approach resembles the use of the classic `im2col` function in Matlab. The function rearranges the data by mapping each filter window location into a column in the result matrix. This operation is illustrated in Fig. 5, where each 3×3 block of the input (left) is vectorized into a 9×1 column of the result matrix. After this rearrangement, the convolution is simply a left multiplication with the vectorized weight matrix,

$$\mathbf{y} = \mathbf{v}^T \mathbf{C},$$

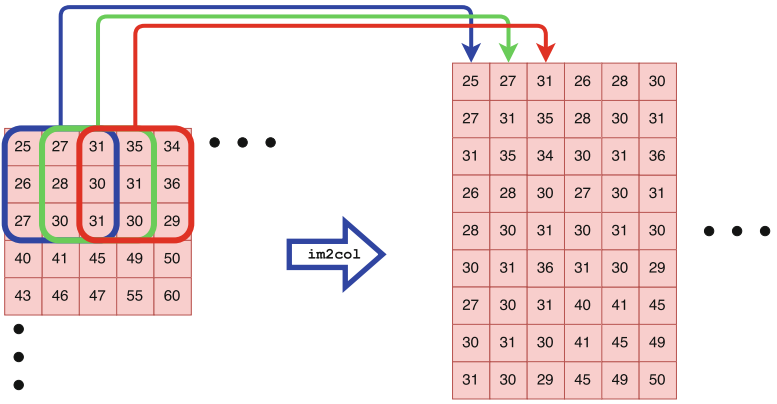


Fig. 5 The `im2col` operation

with $\mathbf{C} \in \mathbb{R}^{9\times NM}$ the result of `im2col` and $\mathbf{v} \in \mathbb{R}^{9\times 1}$ the vectorized 3×3 weight matrix. The result $\mathbf{y} \in \mathbb{R}^{1\times NM}$ can then be reshaped into to match the size of the original image. The drawback of this approach is the memory consumption of matrix $\mathbf{C}$, as the data is duplicated nine times (or more for larger window size). However, the idea provides a unified framework between convolutional layers and dense layers, since both can be implemented as matrix multiplications.

2.3 Pooling Layer

Convolutional layers are economical in terms of the number of coefficients. The parameter count is also insensitive to the size of the input image. However, the amount of *data* still remains high after the convolution, so we need some way to reduce that. For this purpose, we define the *pooling layer*, which shrinks the feature maps by some integer factor. This operation is extremely well studied in the signal processing domain, but instead of high-end decimation-interpolation process, we resort to an extremely simple approach: *max-pooling*.

Max-pooling is illustrated in Fig. 6, where we decimate the large image on the left by a factor of 2 along both spatial axes. The operation retains the largest value within each 2×2 window. Each 2×2 block is distinct (instead of sliding), so the resulting image will have half the size of the original both horizontally and vertically.

Apart from the max operation, other popular choices include taking the average, the L_2 norm, or a Gaussian-like weighted mean of the rectangular input [14, p. 330]. Moreover, the blocks may not need to be distinct, but may allow some degree of overlap. For example, [24] uses a 3×3 pooling window that strides spatially with step size 2. This corresponds to the pool window locations of Fig. 6, but the window would be extended by 1 pixel to size 3×3.

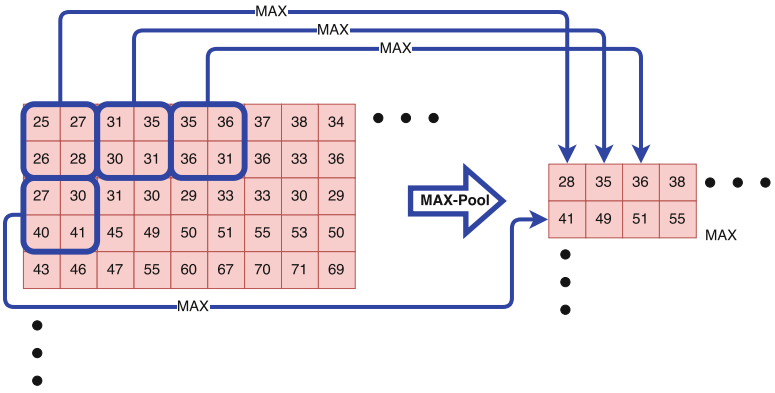


Fig. 6 The maxpooling operation

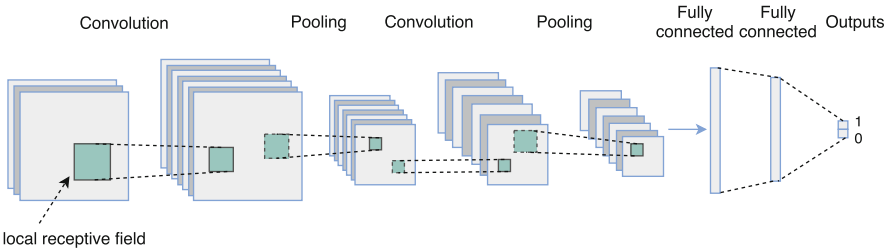


Fig. 7 Architecture of convolutional neural network (modified from [26])

The benefit of using max-pooling in particular, is its improved invariance to small translations. After the convolutions have highlighted the spatial features of interest, max-pooling will retain the largest value of the block regardless of small shifts of the input, as long as the maximum value ends up inside the local window. Translation invariance is usually preferred, because we want the same recognition result regardless of any geometric transformations.

Convolutional and pooling layers follow each other in a sequence. The convolutional layers learn to extract the essential features for the task at hand, while the pooling layers shrink the data size, together attempting to distill the essentials from the data. An example of their co-operation is illustrated in Fig. 7. In this case, the input in the left is an RGB-image (three channels). The pipeline starts with convolutions producing a number of feature maps. The feature maps are fed to the pooling layer, which shrinks each channel, but otherwise retains each map as it is. The same combination is repeated, such that the pooled feature maps are convolved with next level of convolution kernels. Note that the convolution kernel is again 3-dimensional, spanning all input channels within a small spatial window. The usual structure alternates between convolution and pooling until the amount of data is reasonable in size. At that point, the feature map channels are *flattened* (i.e., vectorized) into a vector that passes through a few dense layers. Finally, in a classification task, the number of output nodes equals the number of categories in the data; with each output interpreted as a class likelihood. As an example, a common benchmark for deep learning is the recognition of handwritten MNIST digits [25], where the inputs are 28×28 grayscale handwritten digits. In this case, there are ten classes, and the network desired output (target) is a 10-dimensional binary indicator vector—all zeros, except 1 indicating the correct class label.

2.4 Network Activations

If using only linear operations (convolution and dense layers) in a cascade, the end result could be represented by a single linear operation. Thus, the expression power will increase by introducing nonlinearities into the processing sequence; called

activation functions. We have already seen the nonlinear thresholding operation during the discussion of the perceptron (Sect. 2.1), but the use of hard thresholding as an activation is very limited due to challenges with its gradient: The function is not differentiable everywhere and the derivative bears no information on how far we are from the origin, both important aspects for gradient based training.

Traditionally, popular network activations have been the *logistic sigmoid*,

$$\text{logsig}(x) = \frac{1}{1 + e^{-x}}, \tag{3}$$

and the *hyperbolic tangent*,

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. \tag{4}$$

However, the challenge with both is that they tend to decrease the magnitude of the gradient when backpropagation is passing the weight updates through layers. Namely, the derivative of both activations is always bound to the interval $[-1, 1]$, and when backpropagation applies the chain rule, we are multiplying by a number within this range—a process that will eventually converge to zero. Thus, deep networks will encounter extremely small gradient magnitudes at the lower (close to input) layers.

While there are other approaches to circumvent this *vanishing gradient* problem, the most popular ones simply use an alternative activation function without this problem. The most widely used function is the *rectified linear unit* (ReLU) [24],

$$\text{ReLU}(x) = \max(0, x). \tag{5}$$

In other words, the function clips the input from below at zero. The benefits of the ReLU are clear: The gradient is always either 0 or 1, the computation of the function and its gradient are trivial, and experience has shown its superiority to conventional activations with many datasets. The three activation functions are illustrated in Fig. 8.

The arrangement of activation functions usually starts by setting all activations to ReLU—with the exception of output layer. The ReLU is probably not suitable for the output layer, unless our targets actually happen to fall in the range of positive reals. Common choices for output activation are either linear activation (identity mapping) in regression tasks, or logistic sigmoid in classification tasks. The sigmoid squashes the output range into the interval $[0, 1]$, where they can be conveniently interpreted as class likelihoods. However, a more common choice is to set the ultimate nonlinearity as the *softmax* function, which additionally scales the sum of outputs $\hat{\mathbf{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_K)$ to unity:

$$\left[\text{softmax}(\hat{\mathbf{y}})\right]_j = \frac{\exp(\hat{y}_j)}{\sum_{k=1}^{K} \exp(\hat{y}_k)}, \quad \text{for } j = 1, 2, \ldots, K. \tag{6}$$

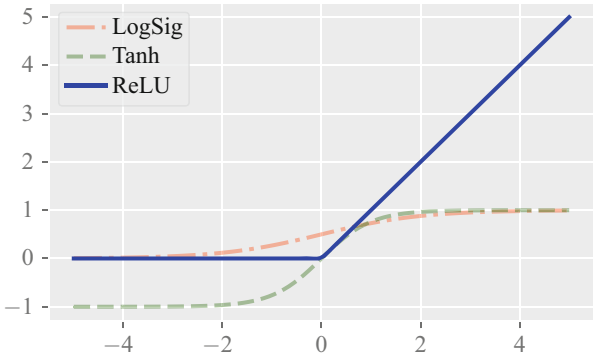


Fig. 8 Popular network activation functions

In other words, each input to the softmax layer is passed through the exponential function and normalized by their sum.

3 Network Training

The coefficients of the network layers are learned from the data by presenting examples and adjusting the weights towards the negative gradient. This process has several names: Most commonly it is called backpropagation—referring to the forward-backward flow of data and gradients—but sometimes people use the name of the optimization algorithm—the rule by which the weights are adjusted, such as *stochastic gradient descent*, *RMSProp*, *AdaGrad* [11], *Adam* [23], and so on. In [33], the good performance of backpropagation approach in several neural networks was discussed, and its importance got widely known after that. Backpropagation has two phases: propagation (forward pass) and weights update (backward pass), which we will briefly discuss next.

Forward Pass When the neural network is fed with an input, it pushes the input through the whole network until the output layer. Initially, the network weights are random, so the predictions are as good as a random guess. However, the network updates should soon push the network towards more accurate predictions.

Backward Pass Based on the prediction, the error between predictions $\hat{y}$ and target outputs y from each unit of output layer is computed using a loss function, $L(\mathbf{y}, \hat{\mathbf{y}})$, which is simply a function of the network output $\hat{\mathbf{y}} = (\hat{y_1}, \ldots, \hat{y_N})$ and the corresponding desired targets $\mathbf{y} = (y_1, \ldots, y_N)$. We will discuss different loss functions more in detail in Sect. 3.1, but for now it suffices to note that the loss should in general be smaller when $\mathbf{y}$ and $\hat{\mathbf{y}}$ are close to each other. It is also worth noting that the network outputs are a function of the weights $\mathbf{w}$. However, in order to avoid notational clutter, we omit the explicit dependence from our notation.

Based on the loss, we compute the partial derivative of the loss $\frac{\partial L(\mathbf{y},\hat{\mathbf{y}})}{\partial \mathbf{w}}$ with respect to the weights in the network. Since the network consists of sequence of layers, the derivatives of the lower (close-to-input) layers depends on that of the upper (close-to-output) layers, and the chain rule of differentiation has to be used. A detailed discussion on how the chain rule is unrolled can be found, e.g., in [15]. Nevertheless, in order to compute the partial derivative of the loss with respect to the parameters of any of the lower layers, we need to know the derivatives of the upper layers first. Therefore, the weight update progresses from the output layer towards the input layer, which coins the name, *backpropagation*. In essence, backpropagation simply traverses the search space by updating the weights in the order admitted by the chain rule. The actual update rule then adjusts each weight towards the negative gradient with the step size specified by the parameter $\eta \in \mathbb{R}_+$:

$$w_{ij} := w_{ij} - \eta \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial w_{ij}} \tag{7}$$

This equation is indeed exactly the same as that of the least mean square filter, familiar from adaptive signal processing.

There are various strategies for choosing the detailed weight update algorithm, as well as various possibilities for choosing the loss function $L(\mathbf{y}, \hat{\mathbf{y}})$ to be minimized. We will discuss these next.

3.1 Loss Functions

Ideally, we would like to minimize the classification error, or maximize the AUROC (area under the receiver operating characteristics curve) score, or optimize whatever quantity we believe best describes the performance of our system. However, most of these interesting performance metrics are not differentiable or otherwise intractable in closed form (for example, the derivative may not be informative enough to guide the optimization towards the optimum). Therefore, we have to use a *surrogate* target function, whose minimum matches that of our true performance metric.

Examples of commonly used loss functions are tabulated in Table 1 and plotted in Fig. 9 for a binary recognition case. In the table, we assume that the network targets $y_j \in \{0, 1\}$ for $j = 1, 2, \ldots, N$, with the exception of hinge loss, where the targets are assumed to be $y_j \in \{-1, 1\}$ for $j = 1, 2, \ldots, N$. This is the common practice in support vector machine literature where the hinge loss is most commonly used.

If we wish to maximize the classification accuracy, then our objective is to minimize the number of incorrectly classified samples. In terms of loss functions, this corresponds to the *zero-one loss* shown in Fig. 9. In this case, each network output $\hat{y}$ (a real number, higher values mean higher confidence of class membership) is rounded to the nearest integer (0 or 1) and compared to the desired target y:

Table 1 Loss functions

Loss function	Definition	Notes
Zero-one loss	$\delta\left(\langle\hat{y}\rangle, y\right)$	$\delta(\cdot,\cdot)$ is the indicator function (see text) $\langle\cdot\rangle$ denotes rounding to nearest integer
Squared loss	$\left(\hat{y}-y\right)^2$	
Absolute loss	$\lvert\hat{y}-y\rvert$	
Logistic loss	$-\ln\left(y\log(\hat{y})-(1-y)\log(1-\hat{y})\right)$	
Hinge loss	$\max\left(0, 1-y\hat{y}\right)$	Label encoding $y\in\{-1,1\}$

In all cases, we denote the network output by $\hat{y}$ and the corresponding desired targets by y. All except hinge loss assume labels $y_j \in \{0,1\}$ for all $j = 1, 2, \ldots, N$

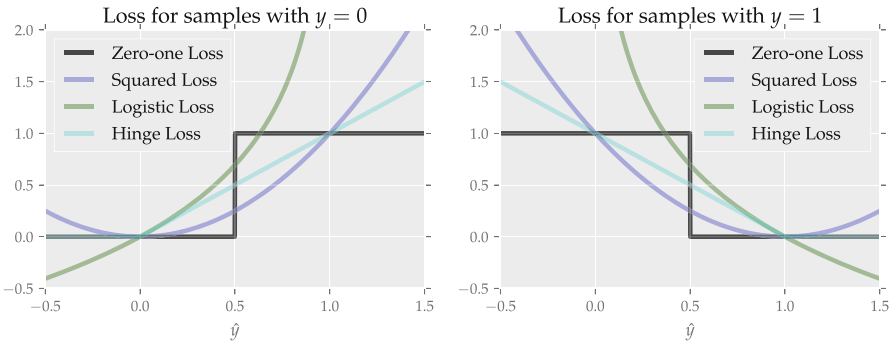


Fig. 9 Commonly used loss functions for classification. Note, that all hinge loss implementations in fact assume labels $y_j \in \{-1, 1\}$, but is scaled here to labels $y_j \in \{0, 1\}$ for visualization

$$L(\hat{y}, y) = \delta\left(\langle\hat{y}\rangle, y\right), \quad \text{with} \quad \delta(p,q) = \begin{cases} 1, & \text{if } p \neq q, \\ 0, & \text{otherwise,} \end{cases} \tag{8}$$

and $\langle x\rangle$ denotes $x \in \mathbb{R}$ rounded to the nearest integer.

Figure 9 plots selected loss functions for the two cases: $y = 0$ and $y = 1$ as a function of the network output $\hat{y}$. The zero-one loss (black) is clearly a poor target for optimization: The derivative of the loss function is zero almost everywhere and therefore conveys no information about the location of the loss minimum. Instead all of its surrogates plotted in Fig. 9 clearly direct the optimization towards the target (either 0 or 1).

In most use cases, the particular choice of loss function is less influential to the result than the optimizer used. A common choice is to use the logistic loss together with the sigmoid or softmax nonlinearity at the output.

3.2 Optimization

At training time, the network is shown labeled examples and the network weights are adjusted according to the negative gradient of the loss function. However, there are several alternative strategies on how the gradient descent is implemented.

One possibility would be to push the full training set through the network and compute the average loss over all samples. The benefit of this *batch gradient* approach would be that the averaging would give us a very stable and reliable gradient, but the obvious drawback is the resulting long waiting time until the network weights can actually be adjusted.

Similarly to the famous LMS algorithm, we obtain a similar averaging effect by using the instantaneous gradient after every sample presented to the network. This approach is called the *stochastic gradient*,

$$\mathbf{w} \leftarrow \mathbf{w} - \eta L(\hat{y}, y), \tag{9}$$

with $\eta > 0$ denoting the *step size*. Although individual gradient estimates are very noisy and may direct the optimization to a globally incorrect direction, the negative gradient will—on the average—point towards the loss minimum.

A common variant of the SGD is the so called *minibatch gradient*, which is a compromise between the batch gradient and stochastic gradient. Minibatch gradient computes the predictions (forward passes) for a *minibatch* of $B \in \mathbb{Z}_+$ samples before propagating the averaged gradient back to the lower layers:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \left(\frac{1}{B} \sum_{j=1}^{B} L(\hat{y}_j, y_j) \right). \tag{10}$$

The minibatch approach has the key benefit of speeding up the computation compared to pure SGD: A minibatch of samples can be moved to the GPU as a single data transfer operation, and the average gradient for the minibatch can be computed in a single batch operation (which parallelizes well). This will also avoid unnecessary data transfer overhead between the CPU and the GPU, which will only happen after the full minibatch is processed.

On the other hand, there is a limit to the speedup of using the minibatch. Sooner or later the GPU memory will be consumed, and the minibatch size can not be increased further. Moreover, large minibatches (up to training set size) may eventually slow down the training process, because the weight updates are happening less frequently. Although increasing the step size may compensate for this, it does not circumvent the fact that path towards the optimum may be nonlinear, and convergence would require alternating the direction by re-evaluating the local gradient more often. Thus, the sweet spot is somewhere between the stochastic gradient ($B = 1$ in Eq. (10)) and the batch gradient ($B = N$ in Eq. (10)).

Apart from the basic gradient descent, a number of improved optimization strategies have been introduced in the recent years. However, since the choice among them is nontrivial and beyond the scope of this chapter, we recommend the interested reader to study the 2012 paper by Bengio [4] or Chapter 8 of the book by Goodfellow et al. [14].

4 Implementation

4.1 Platforms

There exists several competing deep learning platforms. All the popular ones are open source and support GPU computation. They provide functionality for the basic steps of using a deep neural network: (1) Define a network model (layer structure, depth and input and output shapes), (2) train the network (define the loss function, optimization algorithm and stopping criterion), and (3) deploy the network (predict the output for test samples). Below, we will briefly discuss some of the most widely used platforms.

Caffe [21] is a deep learning framework developed and maintained by the Berkeley University Vision and Learning Center (BVLC). Caffe is written in both C++ and NVidia CUDA, and provides interfaces to Python and Matlab. The network is defined using a *Google Protocol Buffers* (prototxt) file, and trained using a command-line binary executable. Apart from the traditional manual editing of the prototxt definition file, current version also allows to define the network in Python or Matlab, and the prototxt definition will be generated automatically. A fully trained model can then be deployed either from the command line or from the Python or Matlab interfaces. Caffe is also known for the famous *Caffe Model Zoo*, where many researchers upload their model and trained weights for easy reproduction of the results in their research papers. Recently, Facebook has actively taken over the development, and released the next generation **caffe2** as open source. Caffe is licensed under the BSD license.

Tensorflow [1] is a library open sourced in 2015 by Google. Before its release, it was an internal Google project, initially under the name *DistBelief*. Tensorflow is most conveniently used through its native Python interface, although less popular C++, Java and Go interfaces exist, as well. Tensorflow supports a wide range of hardware from mobile (Android, iOS) to distributed multi-CPU multi-GPU server platforms. Easy installation packages exist for Linux, iOS and Windows through the Python *pip* package manager. Tensorflow is distributed under the Apache open source license.

Keras [7] is actually a front end for several deep learning computational engines, and links with Tensorflow, Theano [2] and Deeplearning4j backends. Microsoft is also planning to add the CNTK [44] engine into the Keras supported backends. The library is considered easy to use due to its high-level object-oriented Python

interface, and it also has a dedicated *scikit-learn* API for interfacing with the extremely popular Python machine learning library [29]. The lead developer of Keras works as an engineer at Google, and it was announced that Keras will be part of the Tensorflow project since the release of Tensorflow 1.0. Keras is released under the MIT license. We will use Keras in the examples of Sect. 4.2.

Torch [9] is a library for general machine learning. Probably the most famous part of Torch is its *nn* package, which provides services for neural networks. Torch is extensively used by Facebook AI Research group, who have also released some of their own extension modules as open source. The peculiarity of Torch is its interface using *Lua* scripting language for accessing the underlying C/CUDA engine. Recently, a Python interface for Torch was released with the name **pyTorch**, which has substantially extended the user base. Torch is licensed under the BSD license.

MXNet [5] is a flexible and lightweight deep learning library. The library has interfaces for various languages: Python, R, Julia and Go, and supports distributed and multi-GPU computing. The lightweight implementation also renders it very interesting for mobile use, and the functionality of a deep network can be encapsulated into a single file for straightforward deployment into Android or iOS devices. Amazon has chosen MXNet as its deep learning framework of choice, and the library is distributed under the Apache license.

MatConvNet [41] is a Matlab toolbox for convolutional networks, particularly for computer vision applications. Although other libraries wrap their functionality into a Matlab interface, as well, MatConvNet is the only library developed as a native Matlab toolbox. On the other hand, the library can *only* be used from Matlab, as the GPU support builds on top of Matlab Parallel computing toolbox. Thus, it is the only one among our collection of platforms, that requires the purchase of proprietary software. The toolbox itself is licensed under the BSD library.

Comparison of the above platforms is challenging, as they all have their own goals. However, as all are open source projects, the activity of their user base is a critical factor predicting their future success. One possibility for estimating the popularity and the size of the community is to study the activity of their code repositories. All projects have their version control in Github development platform (http://github.com/), and one indicator of project activity is the number of *issues* raised by the users and contributors. An issue may be a question, comment or bug report, but includes also all *pull requests*, i.e., proposals for additions or changes to the project code committed by the project contributors.

The number of new issues for the above deep learning frameworks are illustrated in Fig. 10, where the curves show the number of issues per quarter since the beginning of 2015. If our crude estimate of popularity reflects the real success of each platform, then the deep learning landscape is dominated by three players: Tensorflow, Keras and the MXNet, whose combined share of issues in our graph is over 75% for Q1 of 2017.

It is also noteworthy that the pyTorch is rising its popularity very fast, although plain Torch is not. Since their key difference is the interface (Lua vs. Python), this suggests that Python has become the *de facto* language for machine learning, and every respectable platform has to provide a Python interface for users to link with their legacy Python code.

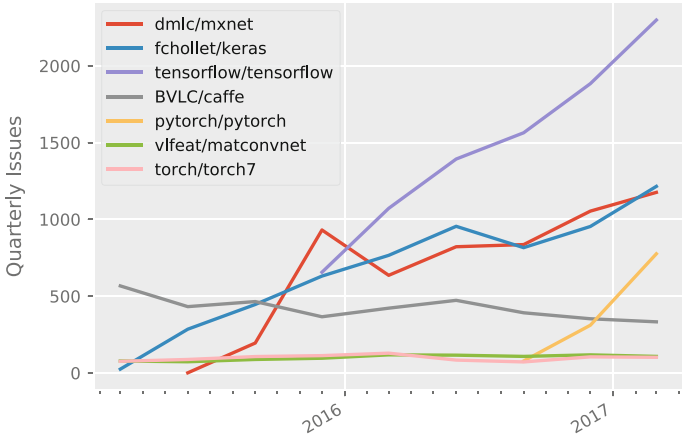


Fig. 10 Number of Github issues for popular deep learning platforms

4.2 Example: Image Categorization

Image categorization is probably the most studied application example of a deep learning. There are a few reasons for this. First, the introduction of the Imagenet dataset [10] in 2009 provided researchers access to a large scale heterogeneous annotated set of millions of images. Only very recently, other domains have reached data collections of equal magnitude; a recent example is the Google AudioSet database of acoustic events [13]. Large image databases were collected first, because their construction by crowdsourcing is relatively straightforward compared to, for example, annotation of audio files. The Imagenet database was collected using the Amazon Mechanical Turk crowdsourcing platform, where each user was presented an image and asked whether an object of certain category was shown in the picture. A similar human annotation for other domains is not so straightforward.

The second reason for the success of deep learning in image categorization are the ILSVRC (Internet Large Scale Visual Recognition Challenge) competitions organized annually since 2010 [34]. The challenge uses the Imagenet dataset with over one million images from 1000 categories, and different teams compete with each other in various tasks: categorization, detection and localization. The competition provides a unified framework for benchmarking different approaches, and speeds up the development of methodologies, as well. Thirdly, image recognition is a prime example of a task which is easy for humans but was traditionally difficult for machines. This raised also academic interest on whether machines can beat humans on this task.

As an example of designing a deep neural network, let us consider the Oxford Cats and Dogs dataset [28], where the task is to categorize images of *cats* and *dogs* into two classes. In the original pre-deep-learning era paper, the authors reached accuracy of 95.4% for this binary classification task. Now, let's take a look at how to

```
# Import the network container and the three types of layers
from keras.models import Sequential
from keras.layers import Conv2D, Dense, DropOut

# Initialize the model
model = Sequential ()

# Add six convolutional layers. Maxpool after every second convolution.
model.add (Conv2D (filters=32, kernel_size=3, padding=' same' , activation=' relu' ,
        input_shape =shape))
model.add (Conv2D (filters=32, kernel_size=3, padding=' same' , activation=' relu' ))
model.add (MaxPooling2D (2, 2)) # Shrink feature maps to 32x32

model.add (Conv2D(filters=48, kernel_size =3, padding =' same' , activation=' relu' ))
model.add (Conv2D(filters=48, kernel_size =3, padding =' same' , activation=' relu' ))
model.add (MaxPooling2D (2,2)) # Shrink feature maps to 16x16

model.add (Conv2D( filters=64, kernel_size=3, padding='same', activation='relu' ))
model.add (Conv2D(filters=64, kernel_size=3, padding='same', activation=' relu' ))
model.add (MaxPooling2D (2,2)) # Shrink feature maps to 8x8

# Vectorize the 8x8x64 representation to 4096x1 vector
model.add (Flatten())

# Add a dense layer with 128 nodes
model.add (Dense(128, activation='relu' ))
model.add (Dropout (0.5))

# Finally, the output layer has 1 output with logistic sigmoid nonlinearity
model.add(Dense(1, activation ='sigmoid' ))
```

Listing 1 Keras code for creating a small convolutional network with randomly initialized weights.

design a deep network using the Keras interface and how the result would compare with the above baseline.

We use a subset of 3687 images of the full dataset (1189 cats; 2498 dogs) for which the ground truth location of the animal's head is available. We crop a square shaped bounding box around the head and train the network to categorize based on this input. The bounding box is resized to fixed size 64×64 with three color channels. We choose the input size as a power of two, since it allows us to downsample the image up to six times using the maxpooling operator with stride 2.

We consider two approaches to network design:

1. Design a network from scratch,
2. Fine tune the higher layers of a pretrained network for this task.

Since the amount of training data is relatively small, the first option necessarily limits the network size in order to avoid overlearning. In the second case, the network size can be larger as it has been trained with a larger number of images before.

Small Network The structure of the network trained from scratch is shown in Fig. 11. The network consists of six convolutional layers followed by one dense

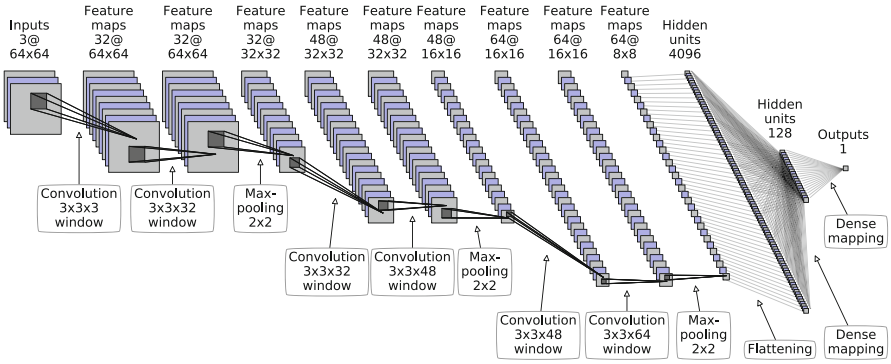


Fig. 11 Structure of the dogs and cats classification network

layer and the output layer. The input of the network is a $96 \times 96 \times 3$ array, and the output is a scalar: The probability of a dog (we encode dog as target $y_i = 1$ and cat as target $y_i = 0$). The network is created in Keras using the code on Listing 1, and the result is illustrated in Fig. 11.

The input at the left of the figure is the image to be categorized, scaled to 64×64 pixels with three color channels. The processing starts by convolving the input with a kernel with spatial size 3×3 spanning all three channels. Thus, the convolution window is in fact a cube of size $3 \times 3 \times 3$: It translates spatially along image axes, but can see all three channels at each location. This will allow the operation to highlight, e.g., all red objects by setting the red channel coefficients larger than the other channels. After the convolution operation, we apply a nonlinearity in a pixel-wise manner. In our case this is the ReLU operator: $\mathrm{ReLU}(x) = \max(0, x)$.

Since a single convolution can not extract all the essential features from the input, we apply several of them, each with a different $3 \times 3 \times 3$ kernel. In the first layer of our example network, we decide to learn altogether 32 such kernels, each extracting hopefully relevant image features for the subsequent stages. As a result, the second layer will consist of equally many *feature maps*, i.e., grayscale image layers of size 64×64. The spatial dimensions are equal to the first layer due to the use of zero padding at the borders.

After the first convolution operation, the process continues with more convolutions. At the second layer, the $64 \times 64 \times 32$ features are processed using a convolution kernel of size $3 \times 3 \times 32$. In other words, the window has spatial dimensions 3×3, but can see all 32 channels at each spatial location. Moreover, there are again 32 such kernels, each capturing different image features from the $64 \times 64 \times 32$ image stack.

The result of the second convolution is passed to a *maxpooling* block, which resizes each input layer to 32×32—half the original size. As mentioned earlier, the shrinking is the result of retaining the largest value of each 2×2 block of each channel of the input stack. This results in a stack of 32 grayscale images of size 32×32.

The first three layers described thus far highlight the basic three-layer block that is repeated for the rest of the convolutional layer sequence. The full convolutional pipeline consists of three *convolution–convolution–maxpooling* blocks; nine layers in total. In deep convolutional networks, the block structure is very common because manual composition of a very deep network (e.g., ResNet with 152 layers [16]) or even a moderately deep network (e.g., VGG net with 16 layers [37]) is not a good target for manual design. Instead, deep networks are composed of *blocks* such as the *convolution–convolution–maxpooling* as in our case.

The network of Fig. 11 repeats the *convolution–convolution–maxpooling* block three times. After each maxpooling, we immediately increase the number of feature maps by 16. This is a common approach to avoid decreasing the data size too rapidly at the cost of reduced expression power. After the three *convolution–convolution–maxpooling* blocks, we end up with 64 feature maps of size 8×8.

The 64-channel data is next fed to two dense (fully connected) layers. To do this, we *flatten* (i.e., vectorize) the data from a $64 \times 8 \times 8$ array into a 4096-dimensional vector. This is the input to the first fully connected layer that performs the mapping

```
# Import the network container and the three types of layers
from keras.applications.vgg16 import VGG16
from keras.models import Model
from keras.layers import Conv2D

# Initialize the VGG16 network. Omit the dense layers on top.
base_model = VGG16 (include_top = False, weights = 'imagenet' ,
                                input_shape = (64,64,3))

# We use the functional API, and grab the VGG16 output here:
w = base_model.output

# Now we can perform operations on w. First flatten it to 2048-dim vector:
w = Flatten () (w)

# Add dense layer :
w = Dense (128, activation = 'relu' ) (w)

# Add output layer:
output = Dense (1, activation = 'sigmoid' ) (w)

# Prepare the full model from input to output :
model = Model (input=base_model. input, output=output)

# Also set the last Conv block (3 layers) as trainable.
# There are four layers above this block, so our indices
# start at -5 (i.e., last minus five) :
model.layers [-5]. trainable = True
model.layers [-6]. trainable = True
model.layers [-7]. trainable = True
```

Listing 2 Keras code for instantiating the pretrained VGG16 network with dense layers appended on top

$\mathbb{R}^{4096} \mapsto \mathbb{R}^{128}$ by multiplying by a 128×4096-dimensional matrix followed by an elementwise ReLU nonlinearity. Finally, the result is mapped to a single probability (of a dog) by multiplying by a 1×128-dimensional matrix followed by the sigmoid nonlinearity. Note that the output is only a single probability although there are two classes: We only need one probability Prob($''$DOG$''$) as the probability of the second class is given by the complement Prob($''$CAT$''$) $= 1 -$ Prob($''$DOG$''$). Alternatively, we could have two outputs with the softmax nonlinearity, but we choose the single-output version due to its relative simplicity.

Pretrained Large Network For comparison, we study another network design approach, as well. Instead of training from scratch, we use a pretrained network which we then fine-tune for our purposes. There are several famous pretrained networks easily available in Keras, including VGG16 [37], Inception-V3 [38] and the ResNet50 [16]. All three are re-implementations of ILSVRC competition winners and pretrained weights trained with Imagenet data are available. Since the Imagenet dataset contains both cats and dogs among the 1000 classes, there is reason to believe that they should be effective for our case as well (in fact the pretrained net approach is known to be successful also for cases where the classes are not among the 1000 classes—even visually very different classes benefit from the Imagenet pretraining).

We choose the VGG16 network as our template because its 16 layers with 5 maxpoolings allow smaller input sizes than the deeper networks. The network structure follows the *convolution-convolution-maxpooling* block composition as in our own network design earlier, and is as follows.

1. **Conv block 1.** Two convolutional layers and a maxpooling layer with mapping $64 \times 64 \times 3 \mapsto 32 \times 32 \times 64$.
2. **Conv block 2.** Two convolutional layers and a maxpooling layer with mapping $32 \times 32 \times 64 \mapsto 16 \times 16 \times 128$.
3. **Conv block 3.** Three convolutional layers and a maxpooling layer with mapping $16 \times 16 \times 128 \mapsto 8 \times 8 \times 256$.
4. **Conv block 4.** Three convolutional layers and a maxpooling layer with mapping $8 \times 8 \times 256 \mapsto 4 \times 4 \times 512$.
5. **Conv block 5.** Three convolutional layers and a maxpooling layer with mapping $4 \times 4 \times 512 \mapsto 2 \times 2 \times 512$.

Additionally, the original network has three dense layers atop the five convolutional blocks. We will only use the pretrained convolutional pipeline, because the convolutional part is usually considered to serve as the feature extractor, while the dense layers do the actual classification. Therefore, the upper dense layers may be very specialized for the Imagenet problem, and would not work well in our case.

More importantly, the convolutional part is invariant to the image shape. Since we only apply convolution to the input, we can rather freely choose the input size, as long as we have large enough data to accommodate the five maxpoolings—at least 32×32 spatial size. The input shape only affects the data size at the output: for $32 \times 32 \times 3$ input we would obtain 512 feature maps of size 1×1 at the end, with

$128 \times 128 \times 3$ input the convolutional pipeline output would be of size $4 \times 4 \times 512$, and so on. The original VGG16 network was designed for $224 \times 224 \times 3$ input size, which becomes $7 \times 7 \times 512$ after five maxpooling operations. In our case the output size $2 \times 2 \times 512$ becomes 2048-dimensional vector after flattening, which is incompatible with the pretrained dense layers assuming 25,088-dimensional input.

Instead of the dense layers of the original VGG16 model, we append two layers on top of the convolutional feature extraction pipeline. These layers are exactly the same as in the small network case (see Fig. 11): One 128-node dense layer and 1-dimensional output layer. These additional layers are initialized at random.

In general, the lower layers (close to input) are less specialized to the training data than the upper layers. Since our data is not exactly similar to the Imagenet data (fewer classes, smaller spatial size, animals only), the upper convolutional layers may be less useful for us. On the other hand, the lower layers extract low level features and may be well in place for our case as well. Since our number of samples is small compared to the Imagenet data, we do not want to overfit the lower layers, but will retain them in their original state.

More specifically, we apply the backpropagation step only to the last convolutional block (and the dense layers) and keep the original pretrained coefficients for the four first convolutional blocks. In deep learning terms, we *freeze* the first four convolutional blocks. The fine-tuning should be done with caution, because the randomly initialized dense layers may feed large random gradients to the lower layers rendering them meaningless. As a rule of thumb, if in doubt, rather freeze too many layers than too few layers.

The code for instantiating the pretrained network in Keras is shown in Listing 2. Note that Keras automatically downloads the pretrained weights from the internet and keeps a local copy for the future. Listing 2 uses Keras *functional* API (in Listing 1 we used Sequential API), where each layer is defined in a functional manner, mapping the result of the previous layer by the appropriate layer type.

We train both networks with 80% of the Oxford cats and dogs dataset samples (2949 images), and keep 20% for testing (738 images). We increase the training set size by *augmentation*. Augmentation refers to various (geometric) transformations applied to the data to generate synthetic yet realistic new samples. In our case, we only use *horizontal flipping*, i.e., we reflect all training set images left-to-right. More complicated transformations would include rotation, zoom (crop), vertical flip, brightness distortion, additive noise, and so on.

The accuracy of the two network architectures is plotted in Fig. 12; on the left is the accuracy of the small network and on the right is the accuracy of the pretrained network for 50 epochs. Based on the figures, the accuracy of the pretrained network is better. Moreover, the accuracy reaches the maximum immediately after the very first epochs. The main reason for this is that the pretraining has prepared the network to produce meaningful representation for the data regardless of the input type. In essence, the pretrained classifier very close to a two layer dense network, which trains very rapidly compared to the small network with several trainable convolutional layers.

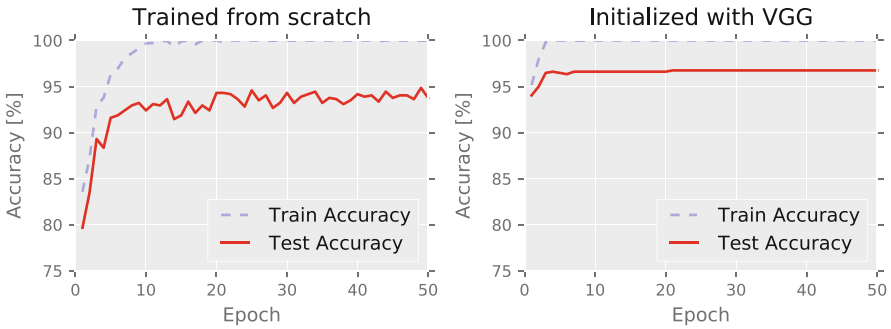


Fig. 12 The accuracy of classification for the Oxford cats and dogs dataset. Left: Learning curve of the small network initialized at random. Right: Learning curve of the fine-tuned VGG network

5 System Level Deployment

Deep learning is rarely deployed as a network only. Instead, the developer has to integrate the classifier together with surrounding software environment: Data sources, databases, network components and other external interfaces. Even in the simplest setting, we are rarely in an ideal position, where we are given perfectly cropped pictures of cats and dogs.

The *TUT Live Age Estimator* is an example of a full deep learning demo system designed to illustrate the human level abilities of a deep learning system.[5] The live video at https://youtu.be/Kfe5hKNwrCU illustrates the functionality of the demo. A screen shot of the video is also shown in Fig. 13.

The system uses three deep networks in real time:

1. An age estimator network [30, 31]
2. A gender recognizer network [30, 31]
3. An expression recognizer network

All the networks receive the cropped face, which needs to be located first. To this aim, we use the OpenCV implementation of the famous Viola-Jones object detection framework [42] with readily available face detection cascades. Moreover, the input video frames are acquired using the OpenCV VideoCapture interface.

Most of the required components are available in open source. The only thing trained by ourselves was the expression recognizer network, for which a suitable pretrained network was not available. However, after the relatively straightforward training of the one missing component, one question remains: How to put everything together?

[5]The full Python implementation is available at https://github.com/mahehu/TUT-live-age-estimator.

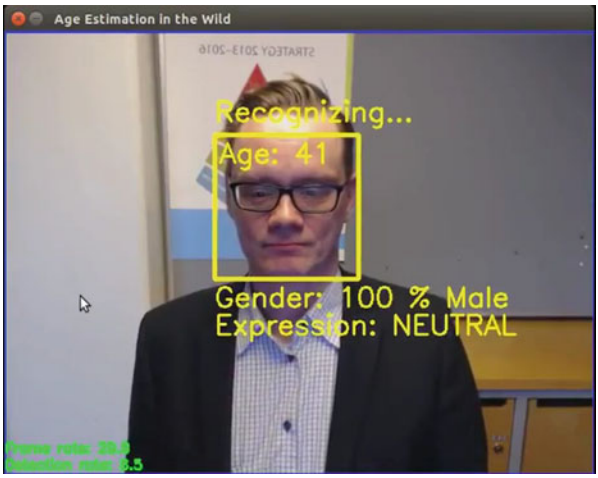


Fig. 13 Screen shot of the TUT live age estimation demo

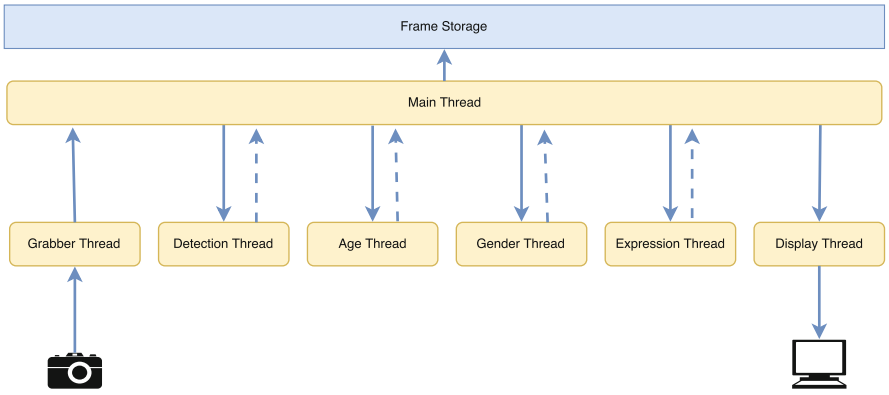


Fig. 14 Schematic diagram of the TUT age estimator

One of the challenges of the real time implementation is in the concurrency: How do we control the interplay of blocks that require different amount of computation? To this aim, we use asynchronous threads that poll for new frames to be processed. The schematic diagram of the system is shown in Fig. 14. Each stage of processing is implemented within a thread.

1. **Grabber thread** accesses the camera and requests video frames. The received frames are time stamped and pushed to the frame storage through the main thread.
2. **Detection thread** polls the frame storage for most recent frame not detected yet. When a frame is received, the OpenCV cascade classifier is applied to localize all faces. The location of the face (or *None* if not found) is added to the frame object, which also indicates that the frame has been processed.

3. **Age thread** polls the frame storage for most recent frame which has passed the detection stage but not age-recognized yet. When a frame is received, the age estimation network is applied to the cropped face. The age estimate is added to the frame object, which also indicates that the frame has been processed.
4. **Gender thread** polls the frame storage for most recent frame which has passed the detection stage but not gender-recognized yet. When a frame is received, the gender recognition network is applied to the cropped face. The gender result is added to the frame object, which also indicates that the frame has been processed.
5. **Expression thread** polls the frame storage for most recent frame which has passed the detection stage but not expression-recognized yet. When a frame is received, the expression recognition network is applied to the cropped face. The expression result is added to the frame object, which also indicates that the frame has been processed.
6. **Display thread** polls the frame storage for most recent frame not locked by any other thread for processing. The thread also requests the most recent age, gender and expression estimates and the most recent face bounding box from the main thread.
7. **Main thread** initializes all other threads and sets up the frame storage. The thread also locally keeps track of the most recent estimates of face location, age, gender and expression in order to minimize the delay of the display thread.
8. **Frame storage** is a list of frame objects. When new objects appear from the grabber thread, the storage adds the new item at the end of the list and checks whether the list is longer than the maximum allowed size. If this happens, then the oldest items are removed from the list unless locked by some processing thread. The storage is protected by mutex object to disallow simultaneous read and write.
9. **Frame objects** contain the actual video frame and its metadata, such as the timestamp, bounding box (if detected), age estimate (if recognized), and so on.

The described structure is common to many processing pipelines, where some stages are independent and allow parallel processing. In our case, the dependence is clear: Grabbing and detection are always required (in this order), but after that the three recognition events and the display thread are independent of each other and can all execute simultaneously. Moreover, if some of the processing stages needs higher priority, we can simply duplicate the thread. This will instantiate two (or more) threads each polling for frames to be processed thus multiplying the processing power.

6 Further Reading

The above overview focused on *supervised* training only. However, there are other important training modalities that an interested reader may study: *unsupervised learning* and *reinforcement learning*.

The amount of data is crucial to modern artificial intelligence. At the same time, data is often the most expensive component while training an artificial intelligent system. In particular, this is the case with annotated data used within supervised learning. Unsupervised learning attempts to learn from unlabeled samples, and the potential of unsupervised learning is not fully discovered. There is a great promise in learning from inexpensive unlabeled data instead of expensive labeled data. Not only the past of deep learning was coined by unsupervised pretraining [18, 19]; unsupervised learning may be the future of AI, as well. Namely, some of the pioneers of the field have called unsupervised learning as the future of AI,[6] since the exploitation on unlabeled data would allow exponential growth in data size.

Reinforcement learning studies problems, where the learning target consists of a sequence of operations—for example, a robot arm performing a complex task. In such cases, the entire sequence should be taken into account when defining the loss function. In other words, also the *intermediate* steps of a successful sequence should be rewarded in order to learn to solve the task successfully. A landmark paper in modern reinforcement learning is the 2015 Google DeepMind paper [27], where the authors introduce a *Deep Q-Learning* algorithm for reinforcement learning with deep neural network. Remarkably, the state-of-the-art results of the manuscript have now been obsoleted by a large margin [17], emphasizing the unprecedented speed of development in the field.

Another topic in AI with growing importance is *recurrent neural network* (RNN), which processes *sequences* using architectures that remember their past states. This enables the concept of *memory*, which allows storage of either temporal or otherwise sequential events for future decisions. Recurrent networks can have multiple configurations depending on the problem input/output characteristics, and Fig. 15 illustrates a few common ones. The particular characteristic of a recurrent network is that it can process sequences with applications such as *image captioning* [22], *action recognition* [36] or *machine translation* [3]. The most widely used RNN structures include the *Long Short-Term Memory* (LSTM) networks [20] and *Gated Recurrent Unit* (GRU) networks [8].

7 Conclusions

Deep learning has become a standard tool in any machine learning practitioner's toolbox surprisingly fast. The power of deep learning resides in the layered structure, where the early layers distill the essential features from the bulk of data, and the upper layers eventually classify the samples into categories. The research on the field is extremely open, with all important papers openly publishing their code along with the submission. Moreover, the researchers are increasingly aware of the

[6] http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/facebook-ai-director-yann-lecun-on-deep-learning.

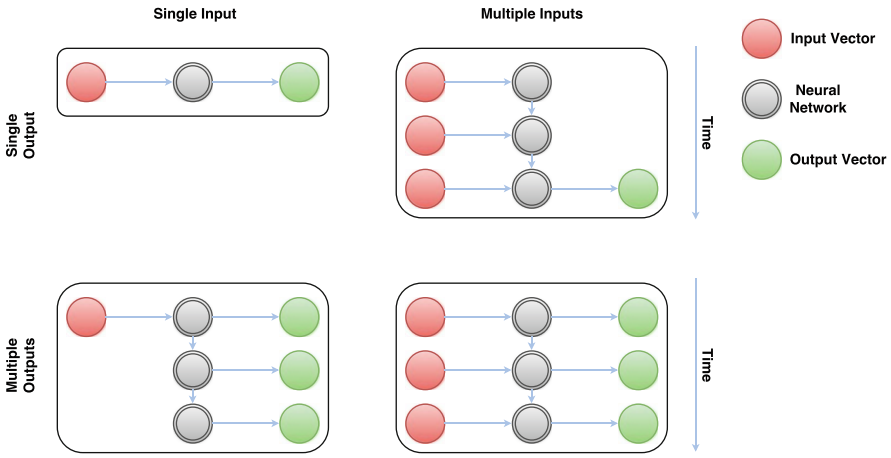


Fig. 15 Configurations of recurrent neural networks. *Top left:* A non-recurrent network, with a single input (e.g., facial image) and single output (e.g., age). *Top right:* A recurrent network with sequence input (e.g., video frames) and a single output (e.g., action of the user in the sequence). *Bottom left:* A recurrent network with single input (e.g., an image) and a sequence output (e.g., the image caption text). *Bottom right:* a recurrent network with a sequence input (e.g. text in Swedish) and sequence output (e.g., text in English)

importance of publishing open access; either in gold open access journals or via preprint servers, such as the ArXiv. The need for this kind of reproducible research was noted early in the signal processing community [39] and has luckily become the standard operating principle of machine learning.

The remarkable openness of the community has led to democratization of the domain: Today everyone can access the implementations, the papers, and other tools. Moreover, cloud services have brought also the hardware accessible to almost everyone: Renting a GPU instance from Amazon cloud, for instance, is affordable. Due to the increased accessibility, standard machine learning and deep learning have become a bulk commodity: Increased number of researchers and students possess the basic abilities in machine learning. So what's left for research, and where the future will lead us?

Despite the increased supply of experts, also the demand surges due to the growing business in the area. However, the key factors of tomorrow's research are twofold. First, *data* will be the currency of tomorrow. Although large companies are increasingly open sourcing their code, they are very sensitive to their business critical data. However, there are early signs that this may change, as well. Companies are opening their data as well: One recent surprise was the release of Google AudioSet—a large-scale dataset of manually annotated audio events [13]—which completely transformed the field of sound event detection research.

Second, the current wave of deep learning success has concentrated on the virtual world. Most of the deep learning is done in server farms using data from the cloud. In other words, the connection to the *physical world* is currently very slim. This

is about to change; as an example, deep learning is rapidly steering the design of self driving cars, where the computers monitor their surroundings via dashboard mounted cameras. However, most of the current platforms are at a prototype stage, and we will see more application-specific deep learning hardware in the future. We have also seen that most of the deep learning computation operations stem from basic signal processing algorithms, embedded DSP design expertise may be in high demand in the coming years.

Acknowledgements The author would like to acknowledge *CSC - IT Center for Science Ltd.* for computational resources.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)
2. Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., et al.: Theano: A python framework for fast computation of mathematical expressions. arXiv preprint arXiv:1605.02688 (2016)
3. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. Proceedings of ICLR2015 (2015)
4. Bengio, Y.: Practical recommendations for gradient-based training of deep architectures. In: Neural networks: Tricks of the trade, pp. 437–478. Springer (2012)
5. Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., Zhang, Z.: Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274 (2015)
6. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014)
7. Chollet, F.: Keras. https://github.com/fchollet/keras (2015)
8. Chung, J., Gulcehre, C., Cho, K., Bengio, Y.: Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. Proceedings of NIPS conference (2014)
9. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: A matlab-like environment for machine learning. In: BigLearn, NIPS Workshop (2011)
10. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: ImageNet: A Large-Scale Hierarchical Image Database. In: CVPR09 (2009)
11. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research **12**(Jul), 2121–2159 (2011)
12. Fisher, R.A.: The use of multiple measurements in taxonomic problems. Annals of eugenics **7**(2), 179–188 (1936)
13. Gemmeke, J.F., Ellis, D.P.W., Freedman, D., Jansen, A., Lawrence, W., Moore, R.C., Plakal, M., Ritter, M.: Audio set: An ontology and human-labeled dataset for audio events. In: Proc. IEEE ICASSP 2017. New Orleans, LA (2017)
14. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016). http://www.deeplearningbook.org
15. Haykin, S., Network, N.: A comprehensive foundation. Neural Networks **2**(2004), 41 (2004)
16. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016). https://doi.org/10.1109/CVPR.2016.90

17. Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., Silver, D.: Rainbow: Combining Improvements in Deep Reinforcement Learning. ArXiv e-prints (2017). Submitted to AAAI2018
18. Hinton, G.E.: Learning multiple layers of representation. Trends in Cognitive Sciences **11**(10), 428–434 (2007)
19. Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural Computation **18**(7), 1527–1554 (2006)
20. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)
21. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the ACM International Conference on Multimedia, pp. 675–678. ACM (2014)
22. Karpathy, A., Fei-Fei, L.: Deep visual-semantic alignments for generating image descriptions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 3128–3137 (2015)
23. Kingma, D., Ba, J.: Adam: A method for stochastic optimization. International Conference on Learning Representations (2015)
24. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: F. Pereira, C.J.C. Burges, L. Bottou, K.Q. Weinberger (eds.) Advances in Neural Information Processing Systems 25, pp. 1097–1105. Curran Associates, Inc. (2012)
25. LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D.: Backpropagation applied to handwritten zip code recognition. Neural computation **1**(4), 541–551 (1989)
26. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE pp. 2278–2324 (1998)
27. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
28. Parkhi, O.M., Vedaldi, A., Zisserman, A., Jawahar, C.V.: Cats and dogs. In: IEEE Conference on Computer Vision and Pattern Recognition (2012)
29. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
30. Rothe, R., Timofte, R., Gool, L.V.: Dex: Deep expectation of apparent age from a single image. In: IEEE International Conference on Computer Vision Workshops (ICCVW) (2015)
31. Rothe, R., Timofte, R., Gool, L.V.: Deep expectation of real and apparent age from a single image without facial landmarks. International Journal of Computer Vision (IJCV) (2016)
32. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature **323**(6088), 533–538 (1986)
33. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Cognitive modeling **5**(3), 1 (1988)
34. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L.: ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV) **115**(3), 211–252 (2015). https://doi.org/10.1007/s11263-015-0816-y
35. Schölkopf, B., Smola, A.J.: Learning with kernels. The MIT Press (2001)
36. Simonyan, K., Zisserman, A.: Two-stream convolutional networks for action recognition in videos. In: Advances in neural information processing systems, pp. 568–576 (2014)
37. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
38. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2818–2826 (2016)

39. Vandewalle, P., Kovacevic, J., Vetterli, M.: Reproducible research in signal processing. IEEE Signal Processing Magazine **26**(3) (2009)
40. Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., LeCun, Y.: Fast convolutional nets with fbfft: A gpu performance evaluation. arXiv preprint arXiv:1412.7580 (2014)
41. Vedaldi, A., Lenc, K.: Matconvnet – convolutional neural networks for matlab. In: Proceeding of the ACM Int. Conf. on Multimedia (2015)
42. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features. In: Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on, vol. 1, pp. I–I. IEEE (2001)
43. Widrow, B.: Thinking about thinking: the discovery of the lms algorithm. IEEE Signal Processing Magazine **22**(1), 100–106 (2005). https://doi.org/10.1109/MSP.2005.1407720
44. Yu, D., Eversole, A., Seltzer, M., Yao, K., Huang, Z., Guenter, B., Kuchaiev, O., Zhang, Y., Seide, F., Wang, H., et al.: An introduction to computational networks and the computational network toolkit. Microsoft Technical Report MSR-TR-2014–112 (2014)
45. Zhu, L.: Gene expression prediction with deep learning. M.Sc. Thesis, Tampere University of Technology (2017)

High Dynamic Range Video Coding

Konstantinos Konstantinides, Guan-Ming Su, and Neeraj Gadgil

Abstract Methods for the efficient coding of high-dynamic range (HDR) still-images and video sequences are reviewed. In dual-layer techniques, a base layer of standard-dynamic range data is enhanced by additional image data in an enhancement layer. The enhancement layer may be additive or multiplicative. If there is no requirement for backward compatibility, adaptive HDR-to-standard dynamic range (SDR) mapping schemes in the encoder allow for improved coding efficiency versus the backward-compatible schemes. In single-layer techniques, a base layer is complemented by metadata, such as supplementary enhancement information (SEI) data or color remapping information (CRI) data, which allow a decoder to apply special "reshaping" or inverse-mapping functions to the base layer to reconstruct an approximation of the original HDR signal. New standards for exchanging HDR signals, such as SMPTE 2084 and BT. 2100, define new mapping functions for translating linear scene light captured by a camera to video and are replacing the traditional "gamma" mapping. The effect of those transforms to existing coding standards, such as high efficiency video coding (HEVC) and beyond, are reviewed, and novel quantization and coding schemes that take these new mapping functions into consideration are also presented.

1 Introduction

In digital imaging, the term dynamic range refers to the ratio of the highest over the lowest luminance values in an image or a scene. For example, the human visual system (HVS) can perceive the brightness of an object from the darkest shadows

K. Konstantinides (✉) · G.-M. Su · N. Gadgil
Dolby Laboratories, San Francisco, CA, USA
e-mail: k.konstantinides@ieee.org; guanmingsu@ieee.org; njgadg@dolby.com

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_5

or faint starlight (about 10^{-6} cd/m^2 or nits),[1] all the way to direct sunlight (about 10^8 cd/m^2). These intensities may be referred to as "scene-referred" intensities. Thus, the physical word represents approximately 14 orders of dynamic range, typically referred to as high dynamic range (HDR). In contrast, the dynamic range that humans may simultaneously perceive is approximately 5–6 orders of magnitude. While this dynamic range, sometimes referred to as visual dynamic range or extended dynamic range, is significantly lower than HDR, it may still be referred to as a high dynamic range.

In display technology, dynamic range (say, 5000:1) refers to the ratio of the brightest white (e.g., 500 nits) over the darkest black (e.g., 0.1 nits) a display can render. These intensities are referred to as "display-referred" intensities. Most consumer desktop displays and high definition televisions (HDTVs) currently support peak luminance of 200–500 nits. Such conventional displays thus typify a lower dynamic range (LDR), also referred to as a standard dynamic range (SDR). Modern, HDR-branded, displays support peak luminance of 800–1000 nits; however, studio HDR monitors are known to support peak luminance values exceeding 4000 nits.

Traditional, 24-bit, digital photography represents a low dynamic range of about 2–3 orders of dynamic range and has dominated both software and hardware architectures in both the consumer electronics and the broadcast industry. However, many display manufacturers are now entering the market with HDR displays and thus there is increased interest and need for the efficient representation, compression, and transmission of HDR content. This Chapter provides a brief overview of past efforts in backwards-compatible coding of HDR video and presents some recent developments using both dual-layer and single-layer architectures. More specifically, Sect. 2 describes early work for coding still HDR images. Before covering HDR video coding, in Sect. 3, we review non-linear transfer functions commonly being used to translate linear light to video signals. Backward-compatible video coding methods are reviewed in Sect. 4, and non-backward compatible methods are reviewed in Sect. 5. Both single-layer and multi-layer coding schemes are examined.

2 Early Work: HDR Coding for Still Images

The image processing community has shown interest in HDR images since the late 1980s, using a variety of logarithmic-based or floating-point encodings for intensity, such as the Radiance RGBE format, LogLuv TIFF, and OpenEXR [1]. None of these formats allowed for backward compatibility, typically a necessary (but not always sufficient) requirement for wider adoption. In 2004, encouraged by the wide adoption of the JPEG image compression standard, Ward and Simmons

[1] Candela per square meter (cd/m^2), also referred to as *nit*, is the international standard unit of luminance.

[1] introduced JPEG-HDR, a JPEG-backwards-compatible format that includes a baseline image representing a tone-mapped version of the input HDR image,[2] and a ratio image representing pixel by pixel the ratio of luminance values in the input HDR image over those in the tone-mapped image. The ratio image is log-encoded and compressed as an 8-bit grayscale image, and it is stored as part of the JPEG image using a dedicated JPEG application marker. The format allows legacy JPEG decoders to ignore the ratio image and simply decode the tone-mapped version of the JPEG image. JPEG-HDR-enabled decoders can reconstuct the HDR image by a simple multiplication with the ratio image. JPEG-HDR was later enhanced to include chroma residuals, and the new format was standardized as Profile A in JPEG XT, which includes also two alternative Profiles (B and C) for coding HDR still images [1–3]. Following the notation by Richter [3], the three profiles may be expressed as follows:

Profile A
In this profile, based on JPEG-HDR, the reconstructed HDR image is generated as

$$HDR = \mu(r)\,(\Phi(SDR) + \chi)\,, \tag{1}$$

where SDR is the base JPEG image (typically encoded in a gamma-corrected space), Φ denotes an inverse-gamma correction, χ denotes a function of chroma (e.g., CbCr) residuals, and $\mu(r) = \exp(r)$ denotes an exponential function of the ratio log-image r (that is, the logarithm of the ratio of luminance in the original HDR image over luminance in the tone-mapped image).

Profile B
Profile B is similar to Profile A, except that the ratio is expressed for each color channel ($i = 1, 2, 3$). Ratios are also coded using a logarithmic representation, thus

$$HDR_i = \sigma\ \exp\left(\log\left(\Phi\left(HDR_i\right)\right) - \log\left(\Psi\left(RES_i\right)\right)\right), \tag{2}$$

where Φ denotes an inverse-gamma correction, Ψ is typically selected to be a gamma correction derived by the encoder, and σ is a scaler.

Profile C
In profile C, the floating-point residuals between the HDR image and the SDR image, in each color component (RES_i, $i = 1, 2, 3$), are expressed as integers using a pseudo-log2 representation. Then, in the decoder, one may apply a pseudo-exponent to recover them. Thus, while the reconstructed image may be represented as

$$HDR_i = \Phi\left(SDR_i\right) RES_i,$$

[2] "Tone mapping" refers to the process of mapping luminance values in a high dynamic range to luminance values in a lower dynamic range.

in a decoder, under Profile C,

$$HDR_i = \psi \exp\left(\widehat{\Phi}\left(SDR_i\right) + RES_i - o\right), \tag{3}$$

where ψexp represents a pseudo-exponential function, o is an offset that ensures the residual image is not negative, and $\widehat{\Phi}$ represents an inverse gamma followed by an inverse log approximation, typically optimized by the encoder and passed to the decoder.

Despite the adoption of HDR image capture in many cameras and smart-phones (typically, by combining three separate exposures), to the best of our knowledge, by mid-2017, none of the camera manufacturers had adopted any of the JPEG XT Profiles. All captured HDR images are simply represented and stored as tone-mapped JPEG images. Before we continue with HDR video coding schemes, it is worth revisiting another important topic: the role of cathode ray tube (CRT) technology in traditional SDR coding, especially as it is related to quantizing linear scene light (e.g., light as captured by a camera sensor) to a non-linear signal for efficient processing of captured images in a video pipeline.

3 Signal Quantization: Gamma, PQ,[3] and HLG[4]

Due to signal-to-noise constraints in analog and digital video signals and the characteristics of the traditional CRT display, scene light was never represented in a linear form in the video processing pipeline. Captured images are quantized using a non-linear opto-electrical transfer function (OETF) which converts linear scene light into the camera's (non-linear) video signal. Then, after encoding and decoding, on the receiver, the signal would be processed by an electro-optical transfer function (EOTF), which would translate the input video signal to output screen color values (e.g., screen luminance) produced by the display. Such non-linear functions include the traditional "gamma" curve documented in Recommendations ITU-R BT.709, BT.1886, and BT.2020 [4]. The combination of an OETF, the EOTF, and any artistic adjustments (either during content creation or content display) is referred to as the system opto-optical transfer function or OOTF [5].

Currently, most digital interfaces for video delivery, such as the serial digital interface (SDI) are limited to 12 bits per pixel per color component. Furthermore, most compression standards, such as H.264 (or AVC) and H.265 (or HEVC), are limited, at least in practical implementations, to 10-bits per pixel per component. Therefore, efficient encoding and/or quantization is required to support HDR content, with dynamic range from approximately 0.001–10,000 cd/m^2 (or nits), within existing infrastructures and compression standards.

[3]PQ stands for "Perceptual Quantizer" EOTF, as defined by Miller et al. [7].

[4]HLG stands for a non-linear transfer function known as "Hybrid Log-Gamma."

Gamma encoding was satisfactory for delivery of SDR (e.g., 8–10 bits) content, but has been proven rather inefficient when coding HDR content. The human visual system responds to increasing light levels in a very non-linear way. A human's ability to see a stimulus is affected by the luminance of that stimulus, the size of the stimulus, the spatial frequencies making up the stimulus, and the luminance level that the eyes have adapted to at the particular moment one is viewing the stimulus [6, 7]. In 2013, Miller et al. [7] proposed an alternative EOTF to the gamma function, commonly referred to as "PQ" (for perceptual quantizer). PQ maps linear input gray levels to output gray levels that better match the contrast sensitivity thresholds in the human visual system. Compared to the traditional gamma curve, which represents the response curve of a physical CRT device and coincidently may have a very rough similarity to the way the human visual system responds, the PQ curve imitates the true visual response of the human visual system using a relatively simple functional model, shown in Eq. (4) [8].

$$V = EOTF^{-1}[L] = \left(\frac{c1 + c2 * L^{m1}}{1 + c3 * L^{m1}}\right)^{m2}, \tag{4}$$

where, V represents the result non-linear signal (say, (R′, G′, B′)) in a range [0,1], L represents luminance of a displayed linear components in cd/m^2 (assuming a peak luminance of 10,000 cd/m^2), and the constants are:

$$m1 = 2610/16384 = 0.1593017578125,$$

$$m2 = 2523/4096 \times 128 = 78.84375,$$

$$c1 = 3424/4096 = 0.8359375 = c3 - c2 + 1,$$

$$c2 = 2413/4096 \times 32 = 18.8515625,$$

$$c3 = 2392/4096 \times 32 = 18.6875.$$

For comparison, according to Rec. BT. 709 [9], the traditional gamma curve on an encoder is given by

$$V = \begin{cases} 1.099L^{0.45} - 0.099; & 1 \geq L \geq 0.018 \\ 4.500\,L; & 0.018 > L > 0 \end{cases}, \tag{5}$$

where L denotes input luminance in [0,1].

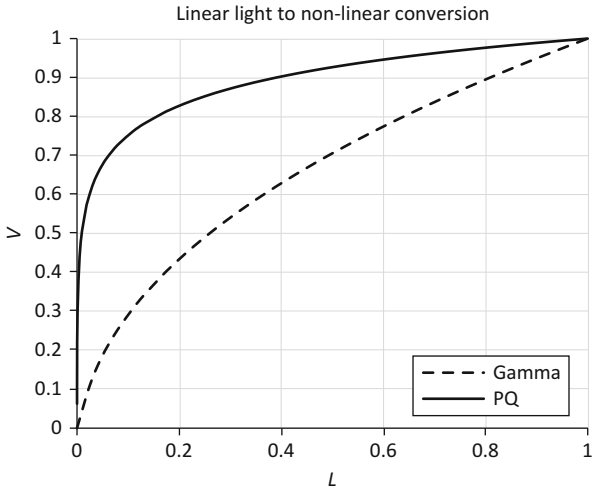


Fig. 1 BT. 709 (gamma) versus SMPTE 2084 (PQ) encoding for HDR signals

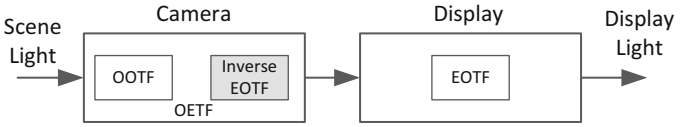


Fig. 2 PQ-oriented system, OOTF is in the camera

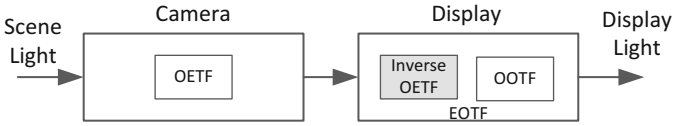


Fig. 3 HLG-oriented system, OOTF is in the display

The PQ mapping functions were adopted in SMPTE ST 2084 and Rec. ITU-R BT.2100 [8]. Figure 1 provides an example of the gamma and PQ coding for L in (0, 1].

The PQ EOTF was designed assuming a camera-centric OOTF that applies an inverse PQ EOTF to generate the signal to be processed (see Fig. 2). In Rec. BT. 2100, an alternative, display-centric nonlinear transfer function is also presented, commonly referred to as Hybrid Log-Gamma (HLG) (see Fig. 3). HLG was designed with backward-compatibility in mind, especially as related to ITU-R BT. 2020 color displays. PQ and HLG may co-exist, and Annex 2 of BT. 2100 provides examples of converting between PQ-coded signals and HLG-coded signals. Given a video stream, information about its EOTF is typically embedded in the bit stream as metadata.

4 Backward-Compatible HDR Coding

By the end of 2016, there was a variety of HDR content available for consumers with branded HDR displays; however, the majority of consumers around the world have only SDR displays. Backward compatibility; that is, support for both SDR and HDR content, is considered by many critical for the wider adoption of HDR displays. Depending on the operational cost, including content storage, network bandwidth, and processing time, content providers have two different approaches to support both formats.

For a system or distribution channel with abundant storage, but limited network bandwidth, most system providers prefer to use a unicast solution: namely, store both the SDR and HDR bit streams in the server side, and transmit either the SDR or the HDR bit stream, depending on the targeted display.

For a system with limited storage, or when the storage cost is high, the system providers prefer to provide a bitstream which can be viewed on a legacy SDR display, but also allows users with an HDR display to reconstruct the HDR content given some additional information. This design is referred to as a backward compatible codec. A legacy receiver can decode the base layer (BL), and the viewers can watch the SDR-grade video. With additional metadata and/or an enhancement layer (EL), the viewers can reconstruct and watch the HDR-grade video. The additional amount of information is often much smaller than the base layer. In this section, we will discuss dual-layer and single-layer backward-compatible HDR video codecs.

The backward compatible HDR codec can be categorized as either a dual-layer system or single-layer system. Each layer is encoded using a legacy video codec, such as AVC (H.264) or HEVC (H.265). Single-layer systems include a single coded bitstream and rely on metadata that define a parametric model to reconstruct the HDR signal. Dual-layer systems include a coded base layer, metadata for reconstructing an HDR estimate signal, and an additional layer of coded residual data which can be added to the HDR signal reconstructed by the base layer, thus, typically providing a more accurate representation of the HDR signal, but at the expense of more bandwidth.

4.1 Dual-Layer Coding

Motivated by the work of Ward and Simmons, in 2006, Mantiuk et al. [10] introduced an MPEG-backwards-compatible coding scheme for HDR video based on layered predictive coding. An example encoder of their scheme is shown in Fig. 4. As depicted in Fig. 4, the encoder receives an HDR video sequence. An HDR to SDR transformation block (e.g., a tone-mapper) applies any of the known HDR to SDR transformation techniques to generate an SDR version, representing the backward-compatible video to be rendered on an SDR display. The SDR version

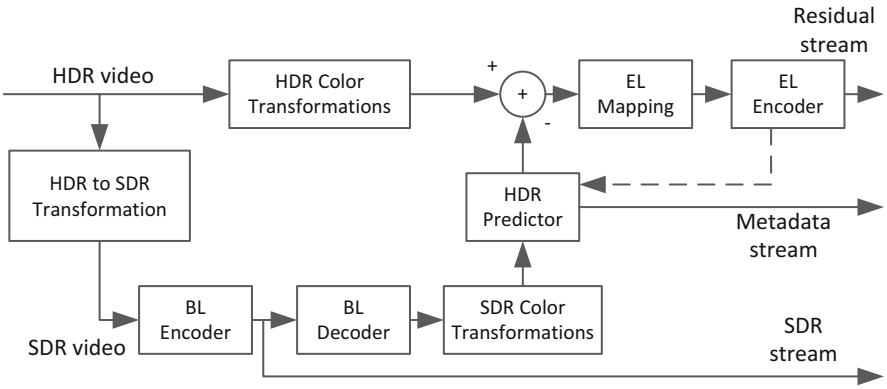


Fig. 4 Block diagram of a dual-layer HDR encoder

is coded with a legacy MPEG encoder. A decoded version, after optional color transformations to match the color format of the HDR video, is passed to a predictor, which estimates the input HDR signal based on the SDR input. The output of the predictor is subtracted from the original HDR input to generate a residual, which is further compressed by another encoder. The SDR stream, the residual stream, and information about the predictor (e.g., metadata) are all multiplexed to form an HDR-coded bitstream. The term "metadata," as used here, relates to any auxiliary information that is transmitted as part of a coded bitstream and assists a decoder to render a decoded image. Such metadata may include, but are not limited to, color space or gamut information, reference display parameters, and other auxiliary signal parameters that will be described later on in this chapter.

In a decoder (Fig. 5), a legacy decoder simply decodes the base layer to be displayed on an SDR display. An HDR decoder, applies the predictor function received from the encoder to the baseline SDR input to generate a predicted HDR signal, which is added to the decoded residual signal to reconstructs the output HDR signal to be displayed on an HDR display. Additional color transformations may be needed to match the color transformations in the encoder or the capabilities of the display pipeline.

To improve coding compression of the residual information, Mantiuk et al. proposed using custom color spaces for both the coded SDR and the HDR streams. For SDR, the input RGB data is converted to a luma-chroma space where, chroma is encoded using a format similar to logLuv encoding and luma is encoded by a special transformation using linear and power function segments [23]. For the HDR signal, after translating YCbCr or RGB data to XYZ, XYZ-float data are converted to a luma-chroma space where: chroma is encoded the same way as SDR chroma, and luma is encoded using a perceptual uniform luminance encoding.

For the prediction function, Mantiuk et al. used a method that combines "binning" with pixel averaging. For example, for SDR images with an 8-bit bit depth there will be 256 bins, where each bin represents all the HDR pixels

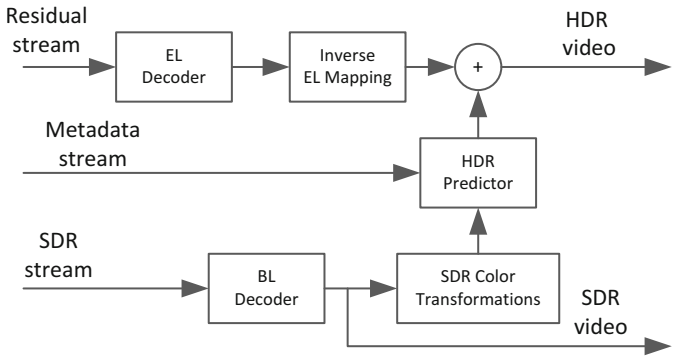


Fig. 5 Block diagram of dual-layer decoder

mapped to the specific (SDR) bin value. Thus, given N pixels in a frame, bin $\Omega_l = \{i = 1, \ldots, N : I_{sdr}(i) = l\}$, where $l = 0, 1, \ldots, 255$, and $I_{sdr}(i)$ denotes the tone-mapped SDR luma value for the i-th input (HDR) pixel ($I_{hdr}(i)$). Then, given an input SDR pixel value l, the corresponding predicted HDR pixel value $(\widehat{I}_{hdr}(l))$ is generated using the arithmetic mean of all HDR pixels that are mapped to the SDR value l in bin Ω_l, or

$$\widehat{I}_{hdr}(l) = \frac{1}{|\Omega_l|} \sum\nolimits_{i \in \Omega_l} I_{hdr}(i), \tag{6}$$

where $|\Omega_l|$ denotes the cardinality of the l-th bin.

Looking at the block diagram in Fig. 4, one can identify three key components that may affect dual-layered coding: (a) the proper color representation of the signals, (b) the choice of the HDR to SDR transformation, and (c) the choice of a prediction function. As discussed earlier, input HDR content may now be represented using both gamma- and PQ-quantization. Regarding the HDR to SDR transformation block, in many cases, this block may be omitted, since the SDR video may be provided separately. For example, the SDR video may represent the output of manual or semi-automatic color grading by a professional colorist who performed color grading according to a director's intent on a reference SDR display. Regarding the HDR predictor, there are multiple alternatives to Eq. (6) and will be discussed later.

In many applications, the residual signal may not be in a format suitable for encoding by a legacy encoder. For example, it may be in floating point, or it is possible that its dynamic range exceeds the dynamic range of the EL Encoder. Then, as shown in Fig. 4, the residual video signal may also be further processed by an enhancement layer mapping function (EL Mapping), such as a linear or non-linear quantizer. If that is the case, as shown in Fig. 5, a decoder needs to apply to the decoded residual an inverse mapping (Inverse EL Mapping).

In some encoder designs, information about the compressed EL bitstream can be fed back to the HDR prediction module as in-loop processing, or it can be accessed independently, as an open-loop process. In-loop designs can generate the reconstructed HDR signal by using inter-layer prediction, that is, by combining the inverse mapped SDR and the residual. This approach can enable temporal domain prediction to further reduce bit rates; however, the in-loop designs need to modify legacy encoders and makes the encoder design more complex. The open-loop approach is a much simpler design, but it loses compression efficiency owing to the lack of a temporal, inter-layer, prediction path.

Besides the temporal-prediction domain option, the EL Mapping block may also support re-scaling and/or sub-sampling in the spatial domain, so that the residual can be down-sampled to a lower resolution to reduce the bit rate. In this case, the EL inverse mapping block in the decoder should support performing the corresponding inverse re-scaling to up-sample the received EL signal to the same resolution as the BL signal. Rescaling may reduce the EL stream bit-rate requirements; however it may also introduce additional artifacts, such as blurred edges and texture. Applying spatial re-scaling is a design tradeoff that depends on the system's bandwidth and picture quality requirements.

One of the key components of any dual-layer system is the design of the HDR predictor. Some of the proposed designs are briefly reviewed next.

4.1.1 Piecewise Linear Model Representation

In [10], Mantiuk et al. proposed sending to the decoder a simple one-dimension look-up Table (1D-LUT) representing the one-to-one SDR to HDR mapping based on Eq. (6); however, the overhead to transmit an 1D-LUT for each frame or even a small collection of frames as metadata is big. Furthermore, Eq. (6) does not guarantee that the SDR to HDR mapping will satisfy certain important properties to reduce coding artifacts. One such property is that the inverse mapping function should be monotonically non-decreasing. This property is important for images with areas with smooth gradient. It has been observed that if the SDR-to-HDR mapping function has some ranges with decreasing slope at its first derivative, then it often creates "hole" artifacts for those smooth areas. To address these problems, in [11], the authors proposed communicating the prediction function to a decoder using a piecewise linear model.

Let $A^L()$ denote the HDR-to-SDR transformation for the luma component (e.g., a global tone-mapping function) and let $B^L()$ denote the inverse luma mapping (e.g., SDR-to-HDR), also referred to as the HDR prediction function. To construct the piecewise linear model for the HDR to SDR mapping, one needs to define: (a) the pivot or end points $\{s_l, s_0 < s_1 < \ldots < s_L\}$ separating/connecting two nearby pieces (or segments) in the piecewise function, and (b) the parameters of the linear model for each piece (e.g., for $y = m_l{*}x + b_l$, m_l and b_l). As the number of possible segments increases, the optimal solution problem becomes intractable very quickly. To simplify the solution, a fixed interval length in logarithm domain is

used to eliminate the need to compute the pivot points. The problem can be further simplified by only computing the slope $\{m_l\}$ in each piece. For the SDR to HDR inverse mapping, the pivot points can remain the same as $\{s_l\}$, and the slope will be the inverse value, as $\{1/m_l\}$, in each piece. The entire system can be formulated as a mean-square-error (MSE) optimization problem to reduce the end-to-end distortion between the original HDR picture and the reconstructed HDR picture:

$$\arg\min_{\{m_L\}} \left\| B^L\left(A^L\left(v_i^L\right)\right) - v_i^L \right\|^2, \tag{7}$$

where v_i^L denotes the intensity of the i-th HDR pixel (e.g., $I_{hdr}(i)$). Assuming a local linearity of the mapping function, the problem can be further simplified and solved with a closed form solution.

4.1.2 Multivariate Multiple Regression Predictor

In order to preserve the film-director's look or intent, content providers may use a colorist to manually color grade an HDR (or SDR) version of a movie based on an existing SDR (or HDR) version. In this case, the HDR to SDR mapping is often a complex function that cannot be represented by existing tone-mapping curves. Furthermore, this complex HDR to SDR process imposes great difficult for the inverse mapping. A single-channel (e.g., luma) HDR predictor cannot capture any chroma-related transformations. For example, chroma saturation and hues may be different in the SDR and HDR versions.

To address these issues, Su et al. [12] proposed separate schemes for predicting luma and chroma. For the luminance component, the HDR predictor uses a piecewise higher order polynomial which allows for more accurate mapping, but with fewer segments than the linear model. With the same overhead, a higher order polynomial can produce better accuracy and a smaller residual. The small residual often leads to smaller energy to encode in the enhancement layer, which yields a lower bit rate requirement for the coded residual stream.

Although a piecewise higher order polynomial can provide better inverse mapping accuracy, the required computation to find the optimal solution is very expensive. As in the piecewise linear model, the parameters in higher order polynomial include the pivot point selection and the polynomial coefficients. By stating the solution as an MSE minimization problem, one can take advantage of the matrix structures and pre-compute many parameters offline as look-up Tables. A faster algorithm (by a factor of 60) to achieve the same optimal solution as full search was proposed in [13].

For the chroma components (say, *Cb* and *Cr* in YCbCr), a cross-color channel inverse mapping, $B^C()$, is adopted to cope with the color space differences, saturation and hue adjustments, and bit depth requirement [14]. The cross-color channel inverse mapping takes the three color channels ($s_i{}^L$, $s_i{}^{C0}$, $s_i{}^{C1}$) for each SDR pixel, and converts them to HDR chroma values $\left(\widehat{v}_i^{C0}, \widehat{v}_i^{C1}\right)$ with parameter set

$\left\{m_{\alpha\beta\gamma}^{C0}, m_{\alpha\beta\gamma}^{C1}\right\}$ using a multivariate multiple regression (MMR) prediction model, which in its most general form can be expressed as

$$\begin{aligned}\widehat{v}_i^{C0} &= \sum_{\alpha}\sum_{\beta}\sum_{\gamma} m_{\alpha\beta\gamma}^{C0}\cdot\left(s_i^{L}\right)^{\alpha}\cdot\left(s_i^{C0}\right)^{\beta}\cdot\left(s_i^{C1}\right)^{\gamma},\\ \widehat{v}_i^{C1} &= \sum_{\alpha}\sum_{\beta}\sum_{\gamma} m_{\alpha\beta\gamma}^{C1}\cdot\left(s_i^{L}\right)^{\alpha}\cdot\left(s_i^{C0}\right)^{\beta}\cdot\left(s_i^{C1}\right)^{\gamma}.\end{aligned} \tag{8}$$

For example, using a second-order MMR predictor, Eq. (8) for color component C0 may be expressed as

$$\begin{aligned}\widehat{v}_i^{C0} = {} & m_0 + m_1 s_i^{L} + m_2 s_i^{C0} + m_3 s_i^{C1}\\ & + m_4 s_i^{L} s_i^{C0} + m_5 s_i^{L} s_i^{C1} + m_6 s_i^{C0} s_i^{C1} + m_7 s_i^{L} s_i^{C0} s_i^{C1}\\ & + m_8\left(s_i^{L}\right)^2 + m_9\left(s_i^{C0}\right)^2 + m_{10}\left(s_i^{C1}\right)^2\\ & + m_{11}\left(s_i^{L} s_i^{C0}\right)^2 + m_{12}\left(s_i^{L} s_i^{C1}\right)^2 + m_{13}\left(s_i^{C0} s_i^{C1}\right)^2 + m_{14}\left(s_i^{L} s_i^{C0} s_i^{C1}\right)^2.\end{aligned} \tag{9}$$

The optimal solution for the MMR prediction coefficients $\left(\left\{m_{\alpha\beta\gamma}^{C0}, m_{\alpha\beta\gamma}^{C1}\right\}\right)$ can be obtained via multivariate multiple regression which minimizes the mean-squared-error between the original HDR chroma and the reconstructed HDR chroma pixel values [14]. MMR-based prediction is used in the Dolby Vision® HDR format from Dolby Laboratories.

4.1.3 MPEG Color Gamut Scalability

In contrast to using a prediction polynomial for luma and/or chroma inverse mapping, one could also deploy a 3D-LUT method, which is commonly used in color science and display management system as a quantization process. This 3D-LUT method is standardized in MPEG as color gamut scalability [15].

In general, the 3D-LUT solution will partition the 3D space into multiple cubes. For each cube (x, y, z), one can have eight corresponding vertices with values denoted as $N(x, y, z)$. Each cube covers a range of code words in three dimensions. For any given input code word, $(s_i{}^{L}, s_i{}^{C0}, s_i{}^{C1})$, for pixel i, one can find which cube, (x, y, z), contains this triplet. Then, one finds out the eight corners (vertices) of this cube (for both chroma values) and performs interpolation to get the HDR values. In tri-linear interpolation, illustrated in Fig. 6, the interpolated value for one particular color channel can be computed as

$$\widehat{v}_i^{C0} = \sum_{\alpha=0}^{1}\sum_{\beta=0}^{1}\sum_{\gamma=0}^{1} w_{\alpha\beta\gamma} N^{C0}(x+\alpha, y+\beta, z+\gamma), \tag{10}$$

where $w_{\alpha\beta\gamma}$ are weighting factors depended on the distances to the eight vertices.

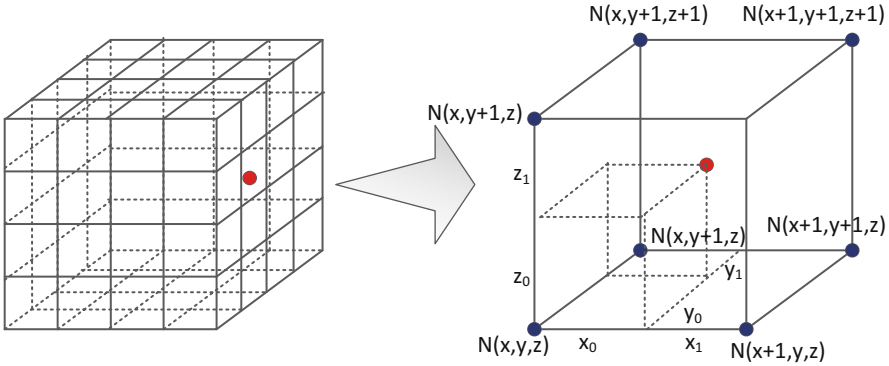


Fig. 6 3D LUT for Color-gamut scalability

The cube can be partitioned evenly in each dimension so one may have a regular cube size. However, the major problem for uniform partition requires a larger overhead to achieve required color precision. Consider a $17 \times 17 \times 17$ 3D LUT, then we need to store $17^3 = 4913$ node values for each color channel. Another approach is to adopt an octree-based structure where each parent cube contains eight smaller cubes via a non-uniform quantization process. The advantage of a tree based partition is to explore the color sensitivity diversity: give more precision for interpolation for more sensitive color regions, and assign less prevision for less sensitive color regions. The overhead for such a cube representation can be smaller. On the other hand, color artifacts along the cube boundaries should be carefully handled as they often represent a discrete value selection and might cause sudden color changes on a flat area.

4.1.4 System-Level Design Issues in Dual-Layer Systems

Dual-layer systems, in general, demand more system resources and higher data management flow for the entire pipeline. From a processing and computational point of view, at the encoder side, the encoding process will be longer, owing to two encoder instances. The overall processing time includes the BL encoding time, BL inverse mapping time, EL mapping time, and EL compression time. The decoder also needs to double its decoding time to handle both layers. It often requires more memory to handle EL information.

From the transmission's point of view, a dedicated multiplexing design is needed to transmit the BL stream, the EL stream, and metadata. To decode one HDR frame, the bitstreams need to be synchronized so the composing can be done based on the current frame. Thus, system level transport at the video elementary stream level is needed to ensure correct synchronization and decoding.

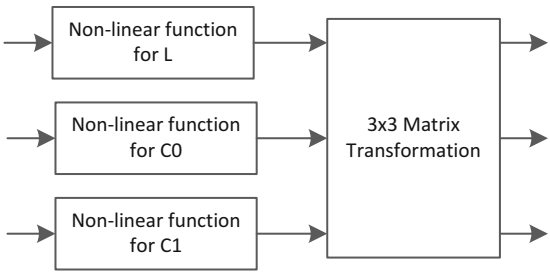


Fig. 7 Parametric model of HDR to SDR transformation

4.2 Single-Layer Methods

In a single-layer system there is only a base layer and metadata to help transform the base layer SDR signal into an HDR signal. The base layer bitstream can be directly decoded and shown on legacy SDR display. To construct the HDR signal, metadata is used to convert the SDR signal to the HDR signal. A single layer requires less processing and computation, and less bandwidth. On the other hand, to recreate a high quality HDR signal, the correlation between the SDR and the HDR signals should be able to be expressed using a well-defined parametric model. This approach, in general, limits the ability to match the director's intent in the reconstructed HDR signal.

As shown in Fig. 7, in a typical color grading process, it is observed that during the generation of the SDR color-graded version from the HDR original, the dynamic range is adjusted by applying: (a) a non-linear tone-mapping curve to each color channel (say, L, $C0$, and $C1$) to adjust the dynamic range; and (b) a 3×3 color rotation matrix to all three channels to adjust the hue and saturation. If one could express these operations in a simple parametric model, then, in a decoder, given the SDR signal, one could recreate the HDR content by doing the reverse, that is, by applying: (a) an inverse 3×3 transformation, followed by (b) inverse non-linear functions in each of the color channels.

To model this mapping, in [16] Su et al. proposed a non-linear matrix mapping model. The most common non-linear function in color grading is the slope, offset, and power (SOP) model [17] (also referred to as the lift, gain, and gamma (LGG) model), where each pixel is adjusted according to gain, offset, and power parameters, as

$$\widehat{v_i} = (S \cdot s_i + O)^P, \tag{11}$$

where s_i denotes an input SDR value, S denotes the slope (or gain), O denotes the offset (or lift), and P denotes the power (or gamma). The rotation matrix is a simple 3×3 matrix. The order of applying the non-linear function and the 3×3 matrix can be switched depending on the prediction accuracy. The SOP parameters and the elements of the 3×3 matrix may be recorded by the color-grading process or they could be estimated by the encoder. They can be passed to the decoder as metadata.

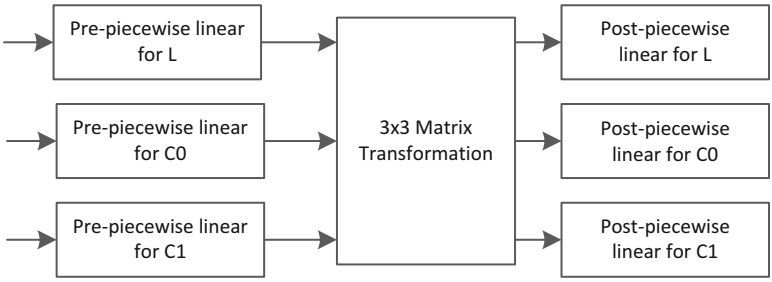


Fig. 8 HEVC model for color remapping information (CRI)

A recent version (v.4 or version 12/2016) of the HEVC (or H.265) standard [18] allows encoders to communicate more complex parametric models as color remapping information (CRI), by concatenating the encoder and decoder non-linear models. As shown in Fig. 8, the mapping process consists of three major stages: the first stage includes a piecewise-linear function (pre-piecewise linear) for each individual color channel to model the non-linear function; the second stage contains a 3×3 matrix; and the last stage includes another piecewise linear function (post-piecewise linear) to model the non-linear function.

The proposed system by Technicolor for single-layer HDR transmission uses similar principles and is discussed in detail in [19]. The design requires two LUTs (one for luma and one for chroma) and two dynamic scaling parameters (a and b). To accommodate multiple broadcasting scenarios, the authors propose that the two LUTs may be communicated either explicitly (table-based mode) or using a simplified set of parameters (parameter-based mode). A table-based mode may provide better quality, but at the expense of more bandwidth. The parameter-based mode assumes a fixed default LUT which can be adjusted by a piece-wise linear table of at most six points. In table-based mode, the two tables are explicitly coded using CRI data.

4.2.1 Philips HDR Codec

In 2015, in response to a call for evidence for HDR and wide-color-gamut coding, Philips [20] submitted a parameter-based, single layer, HDR plus SDR solution, where parameters related to the reconstruction of both the SDR and the HDR signals are embedded into the bitstream. Under the Philips model, each SDR RGB color plane (SDR_i, i = R, G, or B) is expressed as $SDR_i = \omega * HDR_i$, where the scaler ω is determined via a tone-mapping process of *MaxRGBY*; an input signal generated by a weighted combination of the luma (Y) and RGB values of the HDR input. The *MaxRGBY* signal is first translated into a perceptual-uniform signal via the inverse of a Philips-defined EOTF. The perceptual-uniform signal is mapped back to a linear signal via a multi-step process which includes (a) a black/white

level adaptation step, (b) a tone-mapping step using a non-linear curve expressed through parameters defining a "Shadow Gain control," a "Highlight Gain control," and a "Mid-Tones adjustment," and (c) a perceptual to linear SDR mapping step, which uses the Philips-defined EOTF. A receiver, given the received SDR signal and metadata defining ω, applies the inverse steps to generate an approximation of the HDR signal depending on the characteristics of the target display.

As will be discussed later, together with the Dolby Vision dual-layer HDR format, the Philips HDR format is one of the optional HDR formats in the Blu-Ray UHD specification. In 2016, Technicolor and Philips combined their formats into a single unified proposal.

5 Non-Backward-Compatible HDR Coding

Backward-compatible methods do not necessarily guarantee commercial success and they also face multiple challenges. First, a backward-compatible codec typically uses an advanced inter-layer prediction mechanism that demands a higher computational complexity. Second, the details in the higher- and lower-intensity regions are often asymmetric between the SDR and HDR versions, causing undesired clipping in those regions. Furthermore, they typically require a higher bit rate.

Recently, there has been an increasing interest in optimizing video quality for target HDR displays using the existing compression architectures, such as 8- or 10-bit legacy H.26x or VPx encoders, without the restriction of backward compatibility. In this case, the target application, say streaming using an over-the-top set-top box, is focused strictly on HDR playback. Codecs developed for such applications typically require less computational complexity, lower bit rate, and mitigate the clipping issues caused by the typical backward compatible codecs.

5.1 Multi-Layer Non-Backward-Compatible Systems

Given an HDR signal, multi-layer methods typically use multiple lower-bit depth encoders to provide an effective higher bit depth required to process HDR content. A base layer and one or more enhancement layers are used as a mechanism for coding and distributing the source video signals. A preprocessing step generally involves a signal splitting method to generate from the source signal multiple lower bit depth (8- or 10-bit) layers. These layers are then independently encoded to produce standard-compliant bitstreams so that they can be decoded by the existing decoding architectures which are in most cases hardware-based. In a decoder, the decoded layers are then combined to form the output HDR signal that is optimized for viewing on a HDR display. Note that the decoded SDR signal is never viewed.

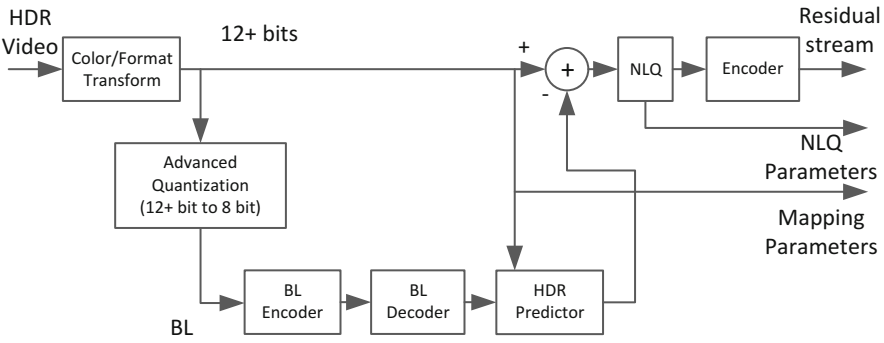


Fig. 9 Dual-layer, non-backward compatible, 8-bit Codec Architecture

5.1.1 Dolby Non-Backward-Compatible 8-Bit Dual-Layer Codec

In a technique developed by Su. et al. [21], an advanced quantization method is used to generate a non-backward-compatible BL and EL signals which may consist of residual values and quantization and mapping parameters that are obtained using a signal prediction mechanism. The proposed adaptive dynamic range adaptation techniques consider effects such as fade-in and fade-outs for an improved coding performance. Figure 9 shows a block diagram of the baseline profile architecture of this codec. The baseline profile restricts all video processing in the base and enhancement coding layers in the YCbCr 4:2:0 color space.

As shown in Fig. 9, an input 4:4:4 RGB (Rec. 709 or Rec.2020), 12+ bit, HDR sequence is first converted to 4:2:0 YCbCr color space. Then, advanced quantization is applied to generate an 8-bit BL sequence in the 4:2:0 YCbCr space. Unlike the backward-compatible case, the BL in not intended to be viewed on SDR displays. Rather the BL signal is optimized in such a way that it contains necessary information to minimize the overall bit requirement for HDR video data carried using multiple layers for the purpose of displaying it on HDR displays. Due to the absence of external color corrections in the BL signal, the clipping levels in the BL and EL are fully controlled by the codec itself. The Advanced quantization block supports many linear and non-linear mapping methods, such as linear quantization, linear stretching, curve-based or non-uniform quantization. Quantization can be done for each individual color channel or jointly, at the frame level or at the scene level.

As an example, a scene-adaptive linear stretching quantization method uses HDR values from each scene to generate corresponding BL values using a simple, invertible, linear mapping. Let v_k^i be the k-th pixel value of the i-th scene of an HDR sequence. Let $\left(v_{min}^i, v_{max}^i\right)$ denote the minimum and maximum pixel values in the i-th scene. Let $\left(s_{min}^i, s_{max}^i\right)$ be the min and max values of 8-bit YCbCr Rec.709 pixel values. Then, the scene adaptive linear stretching method generates base layer s_k^i as:

$$s_k^i = round\left(\frac{\left(s_{max}^i - s_{min}^i\right)}{\left(v_{max}^i - v_{min}^i\right)} \cdot \left(v_k^i - v_{min}^i\right) + s_{min}^i\right), \tag{12}$$

where $0 \leq v_{min}^i \leq v_{max}^i \leq (2^n - 1)$, for n-bit HDR signal.

The BL video sequence is encoded with a standard H.26x encoder using a compliant (such as 4:2:0 YUV) image container. The encoded BL is subsequently decoded to produce a 4:2:0 reconstructed BL to account for the approximations introduced by (possibly) the lossy BL encoder. As in the backward design (Fig. 4), an HDR predictor (e.g., using the inverse of Eq. (12)) may be used to approximate the input HDR signal and generate a residual which will be further coded as an enhancement layer. Since the dynamic range of the residuals may exceed the 8-bit dynamic range of the EL encoder, a non-linear quantizer (NLQ) is subsequently applied to generate 8-bit residuals. Mapping and NLQ parameters are transmitted as a part of metadata categories supported by the legacy video compression standard. For example, H.26x uses the SEI syntax to transmit metadata. For decoding, a similar decoder with the one shown in Fig. 5 may be used, except, that there is no need to display the decoded SDR signal.

5.1.2 HEVC Range Extension Proposals

In an effort to standardize extended range coding (e.g., more than 10 bits and additional color formats) in accordance with the existing HEVC standard, many proposals had been considered by the standardization bodies [22]. Range extensions support up to 16-bits per sample. Given the limitations of using 10-bit coding to encode HDR content, a number of alternative dual-layer systems have been proposed.

Some proposed methods use signal splitting at the encoder and recombination at the decoder [23]. The signal is split in additive layers that are encoded using HEVC 8/10-bit coders. In another method [24], the samples are split into the most significant bits (MSBs) and the least significant bits (LSBs) to obtain two layers. Some methods use overlapped bits in the layers [25–27]. The split signals are typically either packed in 4:4:4, low-bit-depth pictures, or two layers, BL and EL.

Figure 10 [23] shows a dual layer MSB/LSB splitting architecture proposed by Qualcomm. In this design, the input 16-bit picture is split into MSB and LSB layers by separating the most and the least significant bits of each sample to form two layers. In this way, the existing infrastructure in the distribution pipeline can be used for sending HDR signals.

In another proposal [28, 29], a preprocessing step is used to split the input HDR signal (P_{HDR}^i)into two limited dynamic range (LDR) signals using a non-linear mapping function (f), a modulation picture (P_{mod}^i),and a residual picture $\left(P_{LDR}^i\right)$. The modulation picture consists of a low frequency monochromatic version of the input signal, whereas the LDR residual picture represents the remaining relatively-high frequency portion of the signal. The function f is conceptually similar to an

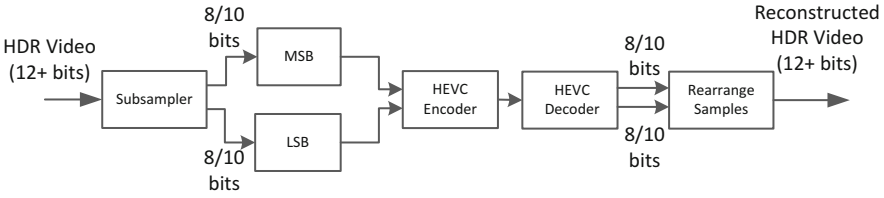


Fig. 10 Dual Layer MSB/LSB splitting architecture

OETF described earlier.

$$P^i_{HDR} = P^i_{mod} * f^{-1}\left(P^i_{LDR}\right). \tag{13}$$

The LDR picture is designed either to maintain backward compatibility or to improve the compression performance. In the latter (non-backward compatible) case, the LDR picture is built in the perceptual color space with the intention of optimizing the chroma quantization for a superior picture quality. Next, the split signals undergo a series of color space and signal format conversions before encoding them to be distributed to the receiver side. At the receiver, the decoder reconstructs the HDR signal by following the inverse operations. It involves inverse transforms and recombination of the layer signals. Arguably, this method has also many similarities with the proposed techniques in coding HDR still-pictures.

HEVC Range extensions (REext) are now part of the second (or later) edition of the HEVC specification [18]. Various encoding profiles of HEVC REext provide support for up to 12 or 16 bit signals. The high bit depth support is enabled using an extended precision mode that allows representing the transformed coefficients using 16-bit values [30] and by using a specific set of binarization [31] and entropy coding methods [32]. An overview of the HEVC REext standard is described in [22].

5.2 Single-Layer Solutions Using Signal Reshaping

In the backward-compatible case, dual-layer systems typically demand more computational and management resources. Therefore, several single layer solutions have been proposed. Generally, the base layer signal is accompanied with metadata to help reconstruction of HDR signal at the decoder. The process of deriving this metadata based on the HDR signal and the base layer is called "reshaping." There have been a number of approaches to design a reshaper which can effectively provide the mapping function for base layer to HDR conversion. Single layer solutions require a re-quantization of the HDR signal to form the base (or the only) layer of the signal that can be encoded and distributed via a standard pipeline.

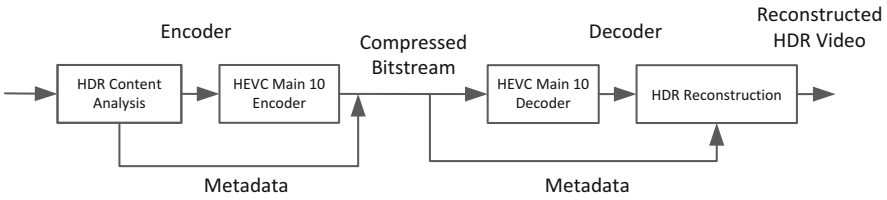


Fig. 11 System diagram of HEVC-based coding using non-backward compatible adaptive reshaping

5.2.1 MPEG Proposals for Reshaping Methods

There has been a considerable interest in the MPEG community to improve the HEVC Main 10 compression performance for HDR/WCG[5] video [33]. The HDR Exploratory Test Model (ETM) is a platform designed for coding Y′CbCr 10-bit 4:2:0 signals [34]. Figure 11 shows a typical system diagram of a non-backward compatible HDR codec using the HDR-10 framework [20, 35, 36].

One consideration for designing a good "reshaper" for a single layer codec is the usage of the PQ EOTF [8] and the practical limitations of HDR displays. PQ is designed to support luminance values of 0 to 10,000 nits, yet a majority of commercially available HDR displays do not exceed 1000 nits. Also, it is important to consider the director's (or the content producer's) creative intent (or "look"). Therefore, during coding, a better utilization of code words can be performed if the display range is known. In a recent study by Lu et al. [36], the impact of baseband quantization on the HDR coding efficiency was analyzed. The study states that matching code words to the target display range improves the coding efficiency. The term "baseband quantization" is defined as the range reduction step that uses a linear or non-linear mapping to convert a higher bit-depth signal to a lower (e.g., 8 or 10) bit signal that is encoded using the legacy encoder. The goal here is to quantify the mapping between the strength of the baseband quantizer and the coding efficiency measured in terms of peak-signal-to-noise-ratio (PSNR). The authors propose a method to estimate the error in reconstructed residues to be used later in a joint analysis of baseband and codec quantizers. The analysis shows that the coding efficiency is mainly lowered by the baseband quantization, and is less affected by the codec quantization [37].

There are many ways to design a reshaping function. Designing linear, piecewise linear, and non-linear, power functions are a few examples among those proposed by various study groups [20, 35, 36]. More advanced proposals apply adaptive codeword re-distribution and signal re-quantization of the three color components, which ultimately changes the bit-rate allocation. For example, in [38] the input

[5]WCG stands for wide color gamut, referring to any color gamut larger than the color gamut supported by the original analog television systems and CRTs. For example, Rec. BT. 2020 [51] defines a WCG container for ultra-high-definition TVs.

HDR signal is partitioned into various non-overlapping luma intensity bands. Each band is processed according to its perceptual significance (referred to as "band importance"), based on the HVS model. A reshaping curve that is non-linear in nature is constructed using this band importance to achieve a better compression efficiency.

Another major consideration in designing a reshaping function is that typically HDR/WCG signals are represented using a much larger color volume than SDR signals. A set of reshaping functions can be designed for each color space such as Y′CbCr, ICtCp [39], Y′CoCg, etc., in which each color channel can be reshaped on its own based on the input HDR signal. One of the main problems with the Y′CbCr color space is non-constant luminance (NCL), in which the mapping between the PQ luminance and Y' luma component from Y′CbCr is not linear. To counter this, a luma adjustment method is proposed in [40] by Ström et al. in Ericsson. This method uses a set of premises: (a) Y', the color-transformed luma value, can be changed independently at each pixel, and (b) Y' increases monotonically with the original (linear space) Y_0. Therefore, Y' is matched with the desired Y_0, the desired luma value. To speed up the implementation for practical purposes one may apply a 3D LUT that maps Cb, Cr and Y_0 values of a pixel to the desired Y' value. More details of the method, with a summary of experiments and performance comparison with the MPEG Call for Evidence (CfE) [33], are described in [40].

5.2.2 Encoder Optimization for PQ-Coded HDR Signals

Legacy encoders, such as HEVC, are highly tuned for compressing gamma-coded SDR signals. As described in Sect. 3, HDR contents are often encoded using the PQ EOTF. Encoding PQ-coded content by directly using gamma-based encoders, such as AVC or HEVC, may result into many undesirable artifacts. To address these issues, several authors have proposed alternative methods to control quantization within the HEVC codec.

A recent study by Lu et al. [41] shows the variance of the input signal to be an important statistic for encoder optimization, since it determines the bit rate allocation and quantization level for each pixel-block in a typical legacy encoder. For example, in a typical H.26x encoder rate control, a variance-based block quantization parameter (QP) model is applied in which, for lower block variance, QP is set to a smaller value. For gamma signal, typically darker areas have less codewords, leading to less variance and smaller QP values, whereas, for brighter areas, there are more codewords, hence higher QP values. However, for a PQ signal, the brighter areas are typically assigned less codewords than the darker areas, which demonstrates opposite behavior from the gamma domain signal. Therefore, applying the gamma-based model directly to PQ-domain signal leads to assigning higher QP values for highlight areas leading to artifacts, while the darker regions are assigned with smaller QP values, thus wasting bits.

A group within the ITU/ISO Joint Collaborative Team on Video Coding (JCT-VC) [42] has been working on the study and standardization of the use of

AVC/HEVC coders for HDR/WCG contents with PQ transfer characteristics. This effort primarily provides a set of recommended guidelines on processing of consumer distribution HDR/WCG video. In [42], two different models: the simple reference model and the enhanced reference model, are described for the pre-encoding and encoding processes. The simple reference model corresponds to the reference configuration used in the MPEG CfE on HDR and WCG [33], while the enhanced reference model corresponds to a new reference configuration that was developed in MPEG following the CfE.

To improve the coder performance, Liu et al. [41] proposed a method to adaptively select block QP for PQ-coded HDR signal based on luminance levels. To further improve the rate-distortion (RD) curve, block signal properties such as edge, texture, contour map or model fitting can be used along with luminance and region of interest (ROI). Another method for assigning QP values for a pixel block based on the luminance range is jointly proposed by Ericsson, Sharp Laboratories and Apple in [43]. It makes use of a fixed LUT for luma channel and a negative QP offset for chroma channels to make the encoder adaptive to the PQ content. In [44], as a pre-processing step to the above compression methods, Norkin proposed a fast down-sampling method to speed up the conversion of 4:4:4 RGB HDR video to the Y′CbCr 4:2:0 non-constant luminance format.

5.2.3 Perceptual Quality-Based Quantization Models

Encoding HDR signals often requires understanding of the perceived HDR image quality. To assess the quality of HDR images with respect to their quantization levels, many model-based approaches are studied. In [45], HDR-VDP-2.2, a calibrated method for objective quality prediction of high-dynamic range and standard images is proposed. HDR-VQM [46] is an alternative proposal for measuring an objective quality of HDR video.

A recent visual study suggests that there exists a relatively strong correlation between the required bit-depth for representing an HDR signal and the standard deviation in code values [47]. Based on this study, a dynamic signal quantization approach called “content-aware quantization” (CAQ) was developed to exploit signal noise and texture in images for reducing the effective bit-depth of the HDR signal. CAQ predicts the required quantization per intensity bin for an HDR image based on a noise/texture estimation model.

As shown in Fig. 12, this model uses a Gaussian high-pass filter to selectively enhance high-frequency content (such as noise/texture) followed by a spatial blurring filter. This model is consistent with HVS models such as the ones described in [48, 49]. To calculate minimum allowed quantization level per pixel, a calibration LUT is applied. A comparative image-based experimental study of CAQ and HDR-VDP is reported in [47].

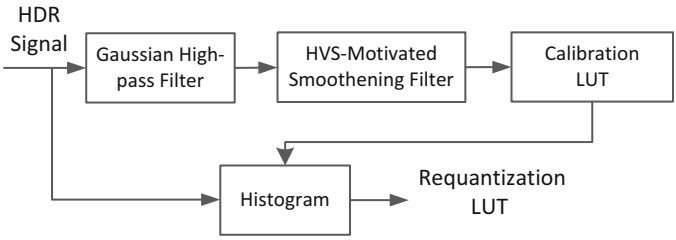


Fig. 12 Content-Aware Quantization (CAQ) for Single Layer Reshaping

5.3 The Ultra HD Blu-Ray Disc Format

As described in [50], Ultra HD Blu-ray™ can support video resolutions at up to 3840 × 2160 pixels (4K/UHD), at up to 60 frames per second. It can also support three different HDR formats: a mandatory Blu-Ray HDR format (also referred to as HDR10 or BDVM[6] HDR), and two optional formats, the Dolby Vision dual-layer format, and the Philips format.

The mandatory BDVM HDR format is a single-layer format that uses 10-bit, 4:2:0, YCbCr signals encoded using the SMPTE 2084 EOTF (PQ), in a BT. 2020 container [51]. The video is encoded using the Main10 Profile of the H.265/HEVC coding standard.

The Dolby Vision stream is composed of a BDVM HDR base layer and a Dolby Vision enhancement layer, with embedded Dolby metadata, which allows the reconstruction of 12-bit HDR video data. Reconstruction of a Dolby Vision HDR stream using the base layer and the enhancement layer follows the decoding scheme discussed in Fig. 5.

As described earlier, the Philips HDR format [20] includes a BDVM HDR video stream and Philips HDR SEI messaging which allows the conversion of the BDVM video to a format suitable for the target display.

6 Conclusions

The field of image and video coding for HDR signals has seen a tremendous growth in the last few years. HDR displays are now commercially available and HDR content is available for streaming. Video coding standards, like HEVC, do support video images with more than 8 bits per pixel, per color component; however, these standards were optimized for 8-bit, gamma-coded, YCbCr video signals, which do not represent the majority of new content, typically coded using either the PQ or

[6]BDVM stands for Blu-Ray Disc Movie.

HLG EOTFs. For most of the schemes we discussed in this chapter, the core SDR video encoder is treated as a "black box" that can be replaced by any video codec of choice (say, 8-bit AVC, 10-bit HEVC, and the like). For HDR coding, the codec's functionality may be enhanced using either preprocessing (say, PQ-coding and/or reshaping), or additional layers of information. We do not expect this to last for long. The "gamma" curve is already challenged, and new color formats, like ICtCp are challenging YCbCr domination as well. As the members of the MPEG committee start work on the next generation codec (to be completed sometime in 2020), we expect efficient HDR coding to be an integral part of any of the proposed coding tools, and not a simple after-thought. The future of HDR video coding is bright indeed.

Appendix: List of Abbreviations

AVC	Advanced Video coding
BDVM HDR	Blu-ray Disc Movie HDR
BL	Base Layer
CAQ	Content Adaptive Quantization
CfE	Call for Evidence
CRI	Color Remapping Information
CRT	Cathode Ray Tube
EL	Enhancement Layer
EOTF	Electro-Optical Transfer Function
HDR	High Dynamic Range
HDR ETM HDR	Exploratory Test Model
HDTV	High Definition Television
HEVC	High Efficiency Video Coding
HLG	Hybrid Log-Gamma
HVS	Human Visual System
ITU	International Telecommunication Union
JPEG	Joint Photographic Experts Group
LDR	Lower Dynamic Range
LSB	Least Significant Bit
LUT	Look-up Table
MMR	Multivariate Multiple Regression
MPEG	Moving Picture Experts Group
MSB	Most Significant Bit
MSE	Mean-Squared Error
NLQ	Non-Linear Quantizer
OETF	Opto-Electrical Transfer Function
OOTF	Opto-Optical Transfer Function
PQ	Perceptual Quantizer
PSNR	Peak Signal-to-Noise Ratio

QP	Quantization Parameter
RD	Rate-Distortion
ROI	Region of Interest
SDI	Serial Digital Interface
SDR	Standard Dynamic Range
SEI	Supplementary Enhancement Information
SMPTE	Society of Motion Picture and Television Engineers
TIFF	Tagged Image File Format
UHD	Ultra-high-definition
VDP	Visual Difference Predictor
VQM	Video Quality Measure
WCG	Wide Color Gamut

References

1. G. Ward and M. Simmons, "JPEG-HDR: A Backwards-Compatible, High Dynamic Range Extension to JPEG," ACM SIGGRAPH 2006.
2. A. Artusi et al. "JPEG XT: A compression standard for HDR and WCG images," *IEEE Signal Processing Magazine,* pp. 118-124, March 2016.
3. T. Richter, T. Bruylants, P. Schelkens, and T. Ebrahimi, "The JPEG XT Suite of standards: Status and Future Plans," SPIE Optical Engineering+ Applications, International Society for Optics and Photonics, Sept. 2015.
4. Report ITU-R BT. 2390-0, "High dynamic range television for production and international programme exchange," ITU, 2016.
5. W. Gish and S. Miller, "Unambiguous video pipeline description motivated by HDR." In Proc. IEEE Intern. Conf. on Image Processing (ICIP 2016), pp. 909-912. IEEE, 2016.
6. P.G.J. Barten, "Contrast sensitivity of the human eye and its effects on image quality," SPIE Optical Engineering Press: Bellingham, WA, 1999.
7. S. Miller et al., "Perceptual Signal Coding for More Efficient Usage of Bit Codes," *SMPTE Motion Imaging Journal,* vol. 122:(4), pp. 52-59, May-June 2013.
8. Rec. ITU-R BT. 2100, "Image parameter values for high dynamic range television for use in production and international programme exchange," ITU, July 2016.
9. Rec. ITU-R BT. 1866, "Reference electro-optical transfer function for flat panel displays used in HDTV studio production," ITU, 03/2011.
10. R. Mantiuk, A. Efremov, K. Myszkowski, and H.-P. Seidel, "Backward Compatible High Dynamic Range MPEG Video Compression," ACM Trans. on Graphics 25(3):713-723, July 2006.
11. Z. Mai, H. Mansour, R. Mantiuk, P. Nasiopoulos, R. K. Ward and W. Heidrich, "Optimizing a Tone Curve for Backward-Compatible High Dynamic Range Image/Video Compression," IEEE Trans. on Image Processing, Vol. 20, No. 6, pp. 1558 – 1571, June 2011.
12. G-M. Su, R. Atkins, and Q. Chen, "Backward-Compatible Coding for Ultra High Definition Video Signals with Enhanced Dynamic Range," US 9,549,207, January 17, 2017.
13. Q. Chen, G-M. Su, and P. Yin, "Near Constant-Time Optimal Piecewise LDR to HDR Inverse Tone Mapping," IS&T/SPIE Electronic Imaging, 2015.
14. G-M. Su, S. Qu, H. Koepfer, Y. Yuan, and S. Hulyalkar, "Multiple Color Channel Multiple Regression Predictor," US 8,811,490 B2, 2014.
15. P. Bordes, P. Andrivon, X. Li, Y. Ye, and Y. He, "Overview of Color Gamut Scalability," IEEE Trans. on Circuits and Systems for Video Technology, March 2016.

16. G-M. Su, S. Qu, W. Gish, H. Koepfer, Y. Yuan, and S. Hulyalkar, “Image Prediction based on Primary Color Grading Model,” US 8,731,287 B2, 2014.
17. “ASC Color Decision List (ASC CDL) Transfer Functions and Interchange Syntax,” ASC Technology Committee Digital Intermediate Subcommittee, 2008
18. ITU Rec. H.265, “High efficiency video coding,“ Series H: Audiovisual and Multimedia Systems, Infrastructure of audiovisual services – Coding of Moving Video, ITU, Dec 2016.
19. S. Lasserre, E. François, F. Le Léannec, and D. Touzé, “Single-layer HDR video coding with SDR backward compatibility,” SPIE Optical Engineering+ Applications (pp. 997108-997108), September, 2016.
20. R. Goris, R. Brondijk, R. van der Vleuten, “Philips response to CfE for HDR and WCG,” m36266, ISO/IEC JTC1/SC29/WG11, Warsaw, Poland, July 2015.
21. G-M. Su, S. Qu, S. Hulyalkar, T. Chen, W. Gish, and H. Koepfer, “Layered Decomposition in Hierarchical VDR Coding,” US 9,497,456 B2, November 15, 2016.
22. D. Flynn, D. Marpe, M. Naccari, T. Nguyen, C. Rosewarne, K. Sharman, J., and J. Xu “Overview of the range extensions for the HEVC standard: Tools, profiles, and performance,” IEEE Trans. on Circuits and Systems for Video Technology, vol. 26, no. 1, pp 4-19, January, 2016.
23. F. Dufaux, P. Le Callet, R. Mantiuk, and M. Mrak, eds. “High Dynamic Range Video: From Acquisition, to Display and Applications, “Academic Press, 2016.
24. E. Francois, C. Gisquet, G. Laroche, P. Onno, “AHG18: On 16-bits Support for Range Extensions, Document,” JCTVC-N0142, 14th JCT-VC Meeting Vienna, Austria, Jul-Aug. 2013.
25. W. S. Kim, W. Pu, J. Chen, Y. K. Wang, J. Sole, M. Karczewicz, “AHG 5 and 18: High Bit-Depth Coding Using Auxiliary Picture, Document,” JCTVC-O0090, 15th JCT-VC Meeting, Geneva, Switzerland, Oct.-Nov. 2013.
26. A. Aminlou, K. Ugar, “On 16 Bit coding,” Document JCTVC-P0162, 16th JCT-VC Meeting, San Jose, CA, Jan. 2014.
27. C. Auyeung, J. Xu, “AHG 5 and 18, Coding of High Bit-Depth Source with Lower Bit-Depth Encoders and a Continuity Mapping,” Document JCTVC-P0173, 16th JCT-VC Meeting, San Jose, CA, Jan. 2014.
28. S. Lasserre, F. Le Leannec, P. Lopez, Y. Olivier, D. Touze, E. Francois, “High Dynamic Range Video Coding,” JCTVC-P0159 (m32076), 16th JCT-VC Meeting, San Jose, CA, Jan. 2014.
29. F. Le Leannec, S. Lasserre, E. Francois, D. Touze, P. Andrivon, P. Bordes, Y. Olivier, “Modulation Channel Information SEI Message,” Document JCTVC-R0139 (m33776), 18th JCT-VC Meeting, Sapporo, Japan, Jun.-Jul. 2014.
30. K. Sharman, N. Saunders, and J. Gamei, “AHG5 and 18:Internal Precision for High Bit Depths,” document JCTVC-N0188, 14th Meeting, JCT-VC, Vienna, Austria, Jul. 2013.
31. M. Karczewicz and R. Joshi, “AHG18: Limiting the Worst-Case Length for Coeff_Abs_Level_Remaining Syntax Element to 32 Bits,” document JCTVC-Q0131, 17th Meeting, JCT-VC, Valencia, Spain, Apr. 2014.
32. K. Sharman, N. Saunders, and J. Gamei, “AHG5 and AHG18: Entropy Coding Throughput for High Bit Depths,” document JCTVC-O0046, 15th Meeting, JCT-VC, Geneva, Switzerland, Oct. 2013.
33. A. Luthra, E. Francois, W. Husak, “Call for Evidence (CfE) for HDR and WCG Video Coding”, MPEG2014/N15083, 110th MPEG Meeting, Geneva, 2015.
34. K. Minoo, T. Lu, P. Yin, L. Kerofsky, D. Rusanovskyy, E. Francois, “Description of the Exploratory Test Model (ETM) for HDR/WCG extension of HEVC”, JCT-VC Doc. W0092, San Diego, CA, Feb. 2016.
35. L. Kerofsky, Y. Ye, and Y. He. “Recent developments from MPEG in HDR video compression,” IEEE Intern. Conf. on Image Processing (ICIP), pp. 879-883. IEEE, 2016.
36. T. Lu, F. Pu, P. Yin, Y. He, L. Kerofsky, Y. Ye, Z. Gu, D. Baylon, “Compression Efficiency Improvement over HEVC Main 10 Profile for HDR and WCG Content,” Proc. of the IEEE Data Compression Conference (DCC), Snowbird, March 2016.

37. C. Wong, G-M. Su, M. Wu, "Joint Baseband Signal Quantization and Transform Coding for High Dynamic Range Video," IEEE Signal Processing Letters, 2016.
38. T. Lu, F. Pu, P. Yin, J. Pytlarz, T. Chen, and W. Husak. "Adaptive reshaper for high dynamic range and wide color gamut video compression," SPIE Optical Engineering+ Applications, pp. 99710B-99710B, International Society for Optics and Photonics, 2016.
39. T. Lu, F. Pu, P. Yin, T. Chen, W. Husak, J. Pytlarz, R. Atkins, J. Fröhlich, G-M. Su, "ITP Colour Space and its Compression Performance for High Dynamic Range and Wide Colour Gamut Video Distribution," ZTE Communications, Feb. 2016.
40. J. Ström, J. Samuelsson, and K. Dovstam, "Luma Adjustment for High Dynamic Range Video," Proc. of the IEEE Data Compression Conference (DCC), Snowbird, March 2016.
41. T. Lu, P. Yin, T. Chen, and G-M. Su, "Rate Control Adaptation for High-Dynamic Range Images," U.S. Patent Application Publication US 2016/0134870, 2016.
42. J. Samuelsson et al., "Conversion and coding practices for HDR/WCG YCbCr 4:2:0 video with PQ transfer characteristics," Draft new Supplement 15 to the H-Series of Recommendations, JCTVC-Z1017, 26-th meeting, Geneva, CH, Jan. 2017.
43. J. Ström, K. Andersson, M. Pettersson, P. Hermansson, J. Samuelsson, A. Segall, J. Zhao, S-H. Kim, K. Misra, A. M. Tourapis, Y. Su, and D. Singer, "High Quality HDR Video Compression using HEVC Main 10 Profile," in Proc. of the IEEE Picture Coding Symposium (PCS), Nuremberg, 2016.
44. A. Norkin, "Fast algorithm for HDR video pre-processing," in Proc. of the IEEE Picture Coding Symposium (PCS), Nuremberg, 2016.
45. R. Mantiuk, K. J. Kim, A. G. Rempel, and W. Heidrich. "HDR-VDP-2: a calibrated visual metric for visibility and quality predictions in all luminance conditions," ACM Trans. on Graphics (TOG), vol. 30, no. 4, p. 40. ACM, 2011.
46. M. Narwaria, M. P. Da Silva, and P. Le Callet. "HDR-VQM: An objective quality measure for high dynamic range video," Signal Processing: Image Communication, Vol. 35, pp. 46-60, 2015.
47. J. Froehlich, G-M. Su, S. Daly, A. Schilling, and B. Eberhardt. "Content aware quantization: Requantization of high dynamic range baseband signals based on visual masking by noise and texture," IEEE International Conf. on Image Processing (ICIP), pp. 884-888. IEEE, 2016.
48. S. Daly, "A visual model for optimizing the design of image processing algorithms," Proc. Intern. Conf. on Image Processing, (ICIP-94), vol. 2, pp. 16-20, 1994.
49. A. Lukin, "Improved visible differences predictor using a complex cortex transform," International Conf. on Computer Graphics and Vision, 2009.
50. Blu-Ray Disc Read-only Format, "Audio Visual Application Format Specifications for BD-ROM Version 3.1," White Paper, August 2016, Blu-Ray Disc Association.
51. Rec. ITU-R BT. 2020-1, "Parameter values for ultra-high definition television systems for production and international programme exchange," ITU, June 2014.

Signal Processing for Control

William S. Levine

Abstract Signal processing and control are closely related. In fact, many controllers can be viewed as a special kind of signal processor that converts an exogenous input signal and a feedback signal into a control signal. Because the controller exists inside of a feedback loop, it is subject to constraints and limitations that do not apply to other signal processors. A well known example is that a stable controller in series with a stable plant can, because of the feedback, result in an unstable closed-loop system. Further constraints arise because the control signal drives a physical actuator that has limited range. The complexity of the signal processing in a control system is often quite low, as is illustrated by the Proportional + Integral + Derivative (PID) controller. Model predictive control is described as an exemplar of controllers with very demanding signal processing. ABS brakes are used to illustrate the possibilities for improved controller capability created by digital signal processing. Finally, suggestions for further reading are included.

1 Introduction

There is a close relationship between signal processing and control. For example, a large amount of classical feedback control theory was developed by people working for Bell Laboratories who were trying to solve problems with the amplifiers used for signal processing in the telephone system. This emphasizes that feedback, which is a central concept in control, is also a very important technique in signal processing.

Conversely, the Kalman filter, which is now a critical component in many signal processing systems, was discovered by people working on control systems. In fact, the state space approach to the analysis and design of linear systems grew out of research and development related to control problems that arose in the American space program.

W. S. Levine (✉)
Department of ECE, University of Maryland, College Park, MD, USA
e-mail: wsl@ece.umd.edu

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_6

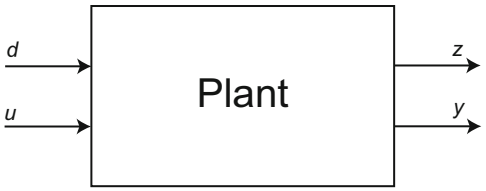


Fig. 1 A generic system to be controlled, called the plant. d is a vector of exogenous inputs; u is a vector of control inputs; y is a vector of measured outputs—available for feedback; and z is a vector of outputs that are not available for feedback

2 Brief Introduction to Control

The starting point for control is a system with inputs and outputs as illustrated schematically in Fig. 1. It is helpful to divide the inputs into those that are available for control and those that come from some source over which we have no control. They are conventionally referred to as control inputs and exogenous inputs. Similarly, it is convenient to divide the outputs into those that we care about, but are not available for feedback, and those that are measured and available for feedback. A good example of this is a helicopter. Rotorcraft have four control inputs, collective pitch, variable pitch, tail rotor pitch, and thrust. The wind is an important exogenous input. Wind gusts will push the aircraft around and an important role of the controller is to mitigate this.

Modern rotorcraft include a stability and control augmentation system (SCAS). This is a controller that, as its name suggests, makes it easier for the pilot to fly the aircraft. The main interest is in the position and velocity of the aircraft. In three dimensions, there are three such positions and three velocities. Without the SCAS, the pilot would control these six signals indirectly by using the four control inputs to directly control the orientation of the aircraft—these are commonly described as the roll, pitch, and yaw angles—and the rate of change of this orientation—the three angular velocities. Thus, there is a six component vector of angular orientations and angular velocities that are used by the SCAS. These are the feedback signals. There is also a six component vector of positions and velocities that are of interest but that are not used by the SCAS. While there are ways to measure these signals in a rotorcraft, in many situations there are signals of interest that can not be measured in real time.

The SCAS-controlled system is again a system to be controlled. There are two different control systems that one might employ for the combined aircraft and SCAS. One is a human pilot. The other is an autopilot. In both cases, one can identify inputs for control as well as exogenous inputs. After all, the wind still pushes the aircraft around. In both cases, there are some outputs that are available for feedback and others that are important but not fed back.

It should be clear from the example that once one has a system to be controlled there is some choice over what output signals are to be used by the controller directly. That is, one can, to some extent, choose which signals are to be fed back. One possible choice is to feed back no signals. The resulting controller is referred to as an open-loop controller. Open loop control has received relatively little attention in the literature. However, it is an important option in cases where any measurements of the output signals would be impossible or very inaccurate. A good example of this until very recently was the control of washing machines. Because it was impossible to measure how clean the dishes or clothing inside the machine were, the controller simply washed for a fixed period of time. Now some such machines measure the turbidity of the waste water as an indirect measurement of how clean the wash is and use this as feedback to determine when to stop washing.

There is also a possibility of feed forward control. The idea here is to measure some of the exogenous inputs and use this information to control the system. For example, one could measure some aspects, say the pH, of the input to a sewage treatment system. This input is normally uncontrollable, i.e., exogenous. Knowing the input pH obviously facilitates controlling the output pH. Note that a feed-forward controller is different from an open-loop controller in that the former is based on a measurement and the latter is not.

The fact remains that most control systems utilize feedback. This is because feedback controls have several significant advantages over open-loop controls. First, the designer and user of the controlled system does not need to know nearly as much about the system as he or she would need for an open-loop control. Second, the controlled system is much less sensitive to variability of the plant and other disturbances.

This is well illustrated by what is probably the oldest, most common, and simplest feedback control system in the world—the float valve level control in the toilet tank. Imagine controlling the level of water in the tank by means of an open-loop control. The user, in order to refill the tank after a flush, would have to open the valve to allow water to flow into the tank, wait just the right amount of time for the tank to fill, and then shut off the flow of water. Looking into the tank would be cheating because that would introduce feedback—the user would, in effect, measure the water level. Many errors, such as turning off the flow too late, opening the input valve too much, a large stone in the tank, or too small a prior flush, would cause the tank to overflow. Automation would not solve this.

Contrast the feedback solution. Once the tank has flushed, the water level is too low. This lowers the float and opens the input valve. Water flows into the tank, thereby raising the water level and the float, until the float gets high enough to close the input valve. None of the open-loop disasters can occur. The closed-loop system corrects for variations in the size and shape of the tank and in the initial water level. It will even maintain the desired water level in the presence of a modest leak in the tank. Furthermore, the closed-loop system does not require any action by the user. It responds automatically to the water level.

However, feedback, if improperly applied, can introduce problems. It is well known that the feedback interconnection of a perfectly stable plant with a perfectly

stable controller can produce an unstable closed-loop system. It is not so well known, but true, that feedback can increase the sensitivity of the closed-loop system to disturbances. In order to explain these potential problems it is first necessary to introduce some fundamentals of feedback controller design.

The first step in designing a feedback control system is to obtain as much information as possible about the plant, i.e., the system to be controlled. There are three very different situations that can result from this study of the plant. First, in many cases very little is known about the plant. This is not uncommon, especially in the process industry where the plant is often nonlinear, time varying, very complicated, and where models based on first principles are imprecise and inaccurate. Good examples of this are the plants that convert crude oil into a variety of products ranging from gasoline to plastics, those that make paper, those that process metals from smelting to rolling, those that manufacture semiconductors, sewage treatment plants, and electric power plants.

Second, in some applications a good non parametric model of the plant is known. This was the situation in the telephone company in the 1920s and 1930s. Good frequency responses were known for the linear amplifiers used in transmission lines. Today, one would say that accurate Bode plots were known. These plots are a precise and detailed mathematical model of the plant and are sufficient for a good feedback controller to be designed. They are non parametric because the description consists of experimental data alone.

Third, it is often possible to obtain a detailed and precise mathematical model of the plant. For example, good mathematical descriptions of many space satellites can be derived by means of Newtonian mechanics. Simple measurements then complete the mathematical description. This is the easiest situation to describe and analyze so it is discussed in much more detail here. Nonetheless, the constraints and limitations described below apply to all feedback control systems.

In the general case, where the plant is nonlinear, this description takes the form of a state-space ordinary differential equation.

$$\dot{x}(t) = f(x(t), u(t), d(t)) \tag{1}$$

$$y(t) = g(x(t), u(t), d(t)) \tag{2}$$

$$z(t) = h(x(t), u(t), d(t)) \tag{3}$$

In this description, $x(t)$ is the n-dimensional state vector, $u(t)$ is the m-dimensional input vector (the control input), $d(t)$ is the exogenous input, $y(t)$ is the p-dimensional measured output (available for feedback), and $z(t)$ is the vector of outputs that are not available for feedback. The dimensions of the $d(t)$ and $z(t)$ vectors will not be needed in this article.

In the linear time-invariant (LTI) case, this becomes

$$\dot{x}(t) = Ax(t) + Bu(t) + Ed(t) \tag{4}$$

$$y(t) = Cx(t) + Du(t) + Fd(t) \tag{5}$$

$$z(t) = Hx(t) + Gu(t) + Rd(t) \tag{6}$$

Another description is often used in the LTI case because of its many useful features. It is equivalent to the description above and obtained by simply taking Laplace transforms of Eqs. (4)–(6) and then doing some algebraic manipulation to eliminate $X(s)$ from the equations.

$$\begin{bmatrix} Y(s) \\ Z(s) \end{bmatrix} = \begin{bmatrix} G_{11}(s) & G_{12}(s) \\ G_{21}(s) & G_{22}(s) \end{bmatrix} \begin{bmatrix} U(s) \\ D(s) \end{bmatrix} \tag{7}$$

The capital letters denote Laplace transforms of the corresponding lower case letters and s denotes the complex variable argument of the Laplace transform. Note that setting $s = j\omega$ makes this description identical to the non parametric description of case 2 on the previous page when the plant is asymptotically stable and extends the frequency domain characterization (Bode plot) of the plant to the unstable case as well.

The issues of closed-loop stability and sensitivity can be easily demonstrated with a simplified version of Eq. (7) above. All signals will be of interest but it will not be necessary to display this explicitly so ignore $Z(s)$ and let all other signals be scalars. For simplicity, let $G_{12}(s) = 1$. Let $G_{11}(s) \triangleq G_p(s)$ and let $G_c(s)$ denote the dynamical part of the controller. Finally, suppose that the objective of the controller is to make $Y(s) \approx R(s)$ as closely as possible, where $R(s)$ is some exogenous input to the controller. It is generally impossible to make $Y(s) = R(s)\ \forall s$. The resulting closed-loop system will be, in block diagram form (Fig. 2).

Simple algebra then leads to

$$Y(s) = \frac{1}{1 + G_p(s)G_c(s)} D(s) + \frac{G_p(s)G_c(s)}{1 + G_p(s)G_c(s)} R(s) \tag{8}$$

$$U(s) = \frac{-G_c(s)}{1 + G_p(s)G_c(s)} D(s) + \frac{G_c(s)}{1 + G_p(s)G_c(s)} R(s) \tag{9}$$

It is conventional in control theory to define the two transfer functions appearing in Eq. (8) as follows. The transfer function in Eq. (9) is also defined below.

$$T(s) = \frac{G_p(s)G_c(s)}{1 + G_p(s)G_c(s)} \tag{10}$$

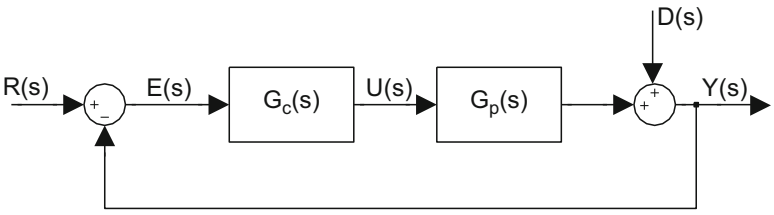


Fig. 2 Simple feedback connection

$$S(s) = \frac{1}{1 + G_p(s)G_c(s)} \tag{11}$$

$$V(s) = \frac{G_c(s)}{1 + G_p(s)G_c(s)} \tag{12}$$

Now that a complete description of a simple feedback control situation is available, it is straightforward to demonstrate some of the unique constraints on $G_c(s)$, the signal processor that is used to make the closed-loop system track $R(s)$.

2.1 Stability

A simple example demonstrating that the feedback interconnection of asymptotically stable and thoroughly well-behaved systems can result in an unstable closed-loop system is shown below. This example also illustrates that the usual culprit is too high an open-loop gain. Let

$$G_p(s) = \frac{3}{s^2 + 4s + 3} \tag{13}$$

$$G_c(s) = \frac{k}{s + 2} \tag{14}$$

The closed-loop system, $T(s)$ is, after some elementary calculations,

$$T(s) = \frac{3k}{(s + 2)(s^2 + 4s + 3) + 3k} \tag{15}$$

The denominator of $T(s)$ is also the denominator of $S(s)$ and $V(s)$ and it has roots with positive real parts for all $k > 20$. Thus, the closed-loop system is unstable for all $k > 20$.

2.2 Sensitivity to Disturbance

Notice that

$$T(s) + S(s) = 1 \quad \forall s \tag{16}$$

Since $S(s)$ is the response to a disturbance input (for example, wind gusts acting on an aircraft) and $T(s)$ is the response to the input that $Y(s)$ should match as closely as possible (for example, the pilot's commands), this appears to be an ideal situation. An excellent control design would make $T(s) \approx 1, \ \forall s$ and this would

have the effect of reducing the response to the disturbance to nearly zero. There is, unfortunately, a problem hidden in Eq. (16). For any real system, $G_p(s)$ goes to zero as $|s| \to \infty$. This, together with the fact that too large a gain for $G_c(s)$ will generally cause instability, implies that $T(s) \to 0$ as $|s| \to \infty$ and this implies that $S(s) \to 1$ as $|s| \to \infty$. Hence, there will be values of s at which the output of the closed-loop system will consist entirely of the disturbance signal. Remember that letting $s = j\omega$ in any of the transfer functions above gives the frequency response of the system. Thus, the closed-loop system is a perfect reproducer of high-frequency disturbance signals.

2.3 Sensitivity Conservation

Theorem (Bode Sensitivity Integral) *Suppose that $G_p(s)G_c(s)$ is rational, has no right half plane poles, and has at least 2 more poles than zeros. Suppose also that the corresponding closed-loop system $T(s)$ is stable. Then the sensitivity $S(s)$ satisfies*

$$\int_0^\infty log|S(j\omega)|d\omega = 0 \tag{17}$$

This is a simplified version of a theorem that can be found in [14]. The *log* in Eq. (17) is the natural logarithm.

The implications of this theorem and Eq. (16) are: if you make the closed-loop system follow the input signal $R(s)$ closely over some frequency range, as is the goal of most control systems, then inevitably there will be a frequency range where the closed-loop system actually amplifies disturbance signals. One might hope that the frequency range in which the sensitivity must be greater than one in order to make up for the region of low sensitivity could be infinite. This could make the magnitude of the increased sensitivity infinitesimally small. A further result [14] proves that this is not the case. A frequency range where the sensitivity is low implies a frequency range where it is high. Thus, a feedback control system that closely tracks exogenous signals, $R(j\omega)$ for some range of values of frequency ω (typically $0 \leq \omega < \omega_b$) will inevitably amplify any disturbance signals, $D(j\omega)$, over some other frequency range.

This discussion has been based on the assumption that both the plant and the controller are continuous-time systems. This is because most real plants are continuous-time systems. Thus, however one implements the controller, and it is certainly true that most controllers are now digitally implemented, the controller must act as a continuous-time controller. It is therefore subject to the limitations described above.

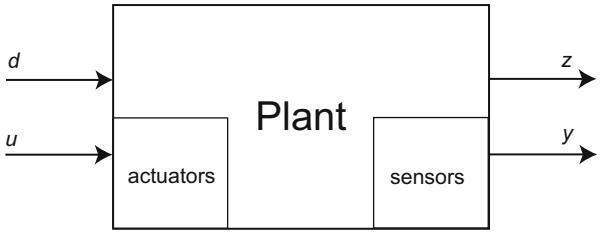


Fig. 3 The plant with sensors and actuators indicated

3 Signal Processing in Control

An understanding of the role of signal processing in control begins with a refinement of the abstract system model in Fig. 1 to that shown in Fig. 3.

Although control theorists usually include the sensors and actuators as part of the plant, it is important to understand their presence and their role. In almost all modern control systems excepting the float valve, the sensors convert the physical signal of interest into an electronic version and the actuator takes an electronic signal as its input and converts it into a physical signal. Both sensing and actuation involve specialized signal processing. A common issue for sensors is that there is very little energy in the signal so amplification and other buffering is needed. Most actuation requires much greater power levels than are used in signal processing so some form of power amplification is needed. Generally, the power used in signal processing for control is a small fraction of the power consumed by the closed-loop system. For this reason, control designers rarely concern themselves with the power needed by the controller.

Most modern control systems are implemented digitally. The part of a digital control system between the sensor output and the actuator input is precisely a digital signal processor (DSP). The fact that it is a control system impacts the DSP in at least three ways. First, it is essential that a control signal be produced on time every time. This is a hard real-time constraint. Second, the fact that the control signal eventually is input into an actuator also imposes some constraints. Actuators saturate. That is, there are limits to their output values that physically cannot be exceeded. For example, the rudder on a ship and the control surfaces on a wing can only turn so many degrees. A pump has a maximum amount it can move. A motor has a limit on the torque it can produce. Control systems must either be designed to avoid saturating the actuator or to avoid the performance degradation that can result from saturation. A mathematical model of saturation is given in Fig. 4.

Another nonlinearity that is difficult, if not impossible, to avoid is the so-called dead zone illustrated in Fig. 4. This arises, for example, in the control of movement because very small motor torques are unable to overcome stiction. Although there are many other common nonlinearities that appear in control systems, we will only mention one more, the quantization nonlinearity that appears whenever a continuous plant is digitally controlled.

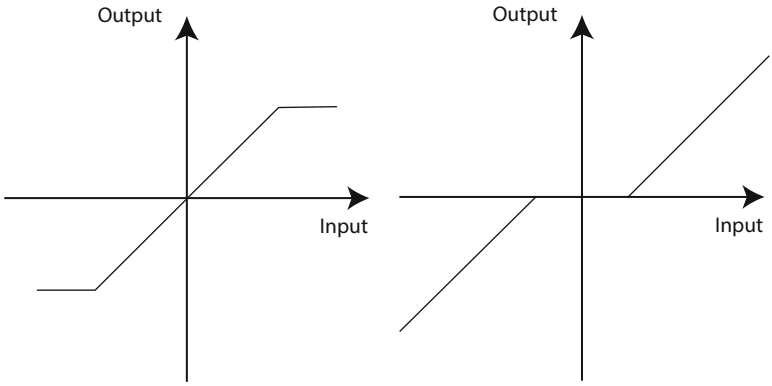


Fig. 4 A saturation nonlinearity is shown at the left and a dead zone nonlinearity at the right

Thirdly, the fact that the DSP is actually inside a feedback loop imposes the constraints discussed at the end of the previous section, too much gain causes closed-loop instability and too precise tracking can cause amplified sensitivity to disturbances.

The main impact of the real time constraint on the DSP is that it must be designed to guarantee that it produces a control signal at every sampling instant. The main impact of the nonlinearities is from the saturation. A control signal that is too large will be truncated. This can actually cause the closed-loop system to become unstable. More commonly, it will cause the performance of the closed-loop system to degrade. The most famous example of this is known as integrator windup as discussed in the following subsection. The dead zone limits the accuracy of the control. The impact of the sensitivity and stability constraints is straightforward. The design of DSPs for control must account for these effects as well as the obvious direct effect of the controller on the plant.

There is one more general issue associated with the digital implementation of controllers. It is well known that the Nyquist rate upper bounds the interval between samples when a continuous-time signal is discretized in time without loss of information. For purposes of control it is also possible to sample too fast. The intuitive explanation for this is that if the samples are too close together the plant output changes very little from sample to sample. A feedback controller bases its actions on these changes. If the changes are small enough, noise dominates the signal and the controller performs poorly.

3.1 Simple Cases

A generic feedback control situation is illustrated in Fig. 5. Fundamentally, the controller in the figure is a signal processor, taking the signals d_c and y as inputs and producing the signal u as output. The objective of the controller is to make

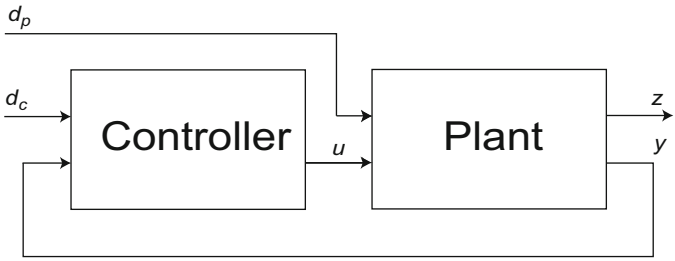


Fig. 5 The plant and controller. Notice that the exogenous input of Fig. 5 has been split into two parts with d_c denoting the exogenous input to the controller and d_p that to the plant

$y(t) = d_c(t)\ \forall t \geq 0$. This is actually impossible because all physical systems have some form of inertia. Thus, control systems are only required to come close to this objective. There are a number of precise definitions of the term "close." In the simplest cases, the requirements/specifications are that the closed-loop system be asymptotically stable and that $|y(t) - d_c(t)| < \epsilon\ \forall t \geq T \geq 0$, where T is some specified "settling time" and ϵ is some allowable error. In more demanding applications there will be specified limits on the duration and size of the transient response [21]. An interesting additional specification is often applied when the plant is approximately linear and time-invariant. In this case, an easy application of the final value theorem for Laplace transforms proves that the error in the steady-state response of the closed-loop system to a step input (i.e., a signal that is 0 up to $t = 0$ at which time it jumps to a constant value α) goes to zero as $t \to \infty$ provided the open-loop system (controller in series with the plant with no feedback) has a pole at the origin. Note that this leads to the requirement/specification that the open-loop system have a pole at the origin. This is achievable whereas, because of inaccuracies in sensors and actuators, zero steady-state error is not.

In many cases the signal processing required for control is straightforward and undemanding. For example, the most common controller in the world, used in applications from the toilet tank to aircraft, is the Proportional + Integral + Derivative (PID) Controller. A simple academic version of the PID Controller is illustrated in Fig. 6. Notice that this simple PID Controller is a signal processing system with two inputs, the signal to be tracked d_c and the actual measured plant output y and one output, the control signal u. This PID controller has three parameters that must be chosen so as to achieve satisfactory performance. The simplest version has k_d and k_i equal to zero. This is a proportional controller. The float valve in the toilet tank is an example. For most plants, although not the toilet tank, choosing k_p too large will result in the closed-loop system being unstable. That is, $y(t)$ will either oscillate or grow until saturation occurs somewhere in the system. It is also usually true that, until instability occurs, increasing k_p will decrease the steady-state error. This captures the basic trade off in control design. Higher control gains tend to improve performance but also tend to make the closed-loop system unstable.

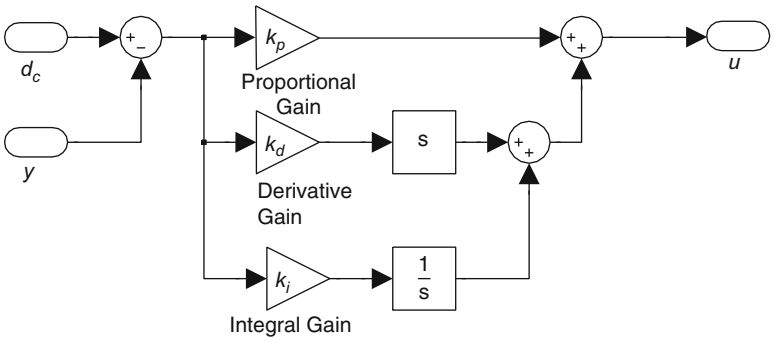


Fig. 6 A simple academic PID controller

Changing k_i to some non zero value usually improves the steady-state performance of the closed-loop system. To see this, first break the feedback loop by removing the connection carrying y in Fig. 5 and set $k_d = 0$. Then, taking Laplace transforms, gives

$$U(s) = (k_p + k_i/s)Y(s) \tag{18}$$

Assuming that the plant is linear and denoting its transfer function by $G_p(s)$ gives the open-loop transfer function,

$$\frac{Y(s)}{D_c(s)} = (\frac{(k_p s + k_i}{s})G_p(s) \tag{19}$$

This open-loop system will have at least one pole at the origin unless $G_p(s)$ has a zero there. As outlined in the previous section, this implies in theory that the steady-state error in response to a step input should be zero. In reality, dead zone nonlinearity and both sensor and actuator nonlinearity will result in nonzero steady-state errors.

It is relatively simple to adjust the gains k_p and k_i so as to achieve good performance of the closed-loop system. There are a number of companies that supply ready-made PID controllers. Once these controllers are connected to the plant, a technician or engineer "tunes" the gains to get good performance. There are several tuning rules in the literature, the most famous of which are due to Ziegler and Nichols. It should be noted that the Ziegler-Nichols tuning rules are now known to be too aggressive; better tuning rules are available [1].

Choosing a good value of k_d is relatively difficult although the tuning rules do include values for it. Industry surveys have shown that a significant percentage of PID controllers have incorrect values for k_d. When it is properly chosen, the D term improves the transient response.

There are three common improvements to the simple PID controller of Fig. 6. First, it is very unwise to include a differentiator in a physical system. Differentiation

amplifies high frequency noise, which is always present. Thus, the $k_d s$ term in the PID controller should be replaced by $\frac{k_d s}{k_d k_n s+1}$. This simply adds a low pass filter in series with the differentiator and another parameter, k_n to mitigate the problems with high frequency noise. Second, the most common input signal $d_c(t)$ is a step. Differentiating such a signal would introduce a signal into the system that is an impulse (infinite at $t = 0$ and 0 everywhere else). Instead, the signal input to the D-term is chosen to be $y(t)$, not $d_c(t) - y(t)$. Notice that this change does away with the impulse and only changes the controller output at $t = t_0$ whenever $d_c(t)$ is a step at t_0.

Lastly, when the PID controller saturates the actuator ($u(t)$ becomes too large for the actuator it feeds) a phenomenon known as integral windup often occurs. This causes the transient response of the closed-loop system to be much larger and to last much longer than would be predicted by a linear analysis. There are several ways to mitigate the effects of integral windup [1].

Nowadays the simple PID controller is almost always implemented digitally. The discretization in time and quantization of the signals has very little effect on the performance of the closed-loop system in the vast majority of cases. This is because the plant is usually very slow in comparison to the speeds of electronic signal processors. The place where digital implementation has its largest impact is on the possibility for improving the controller. The most exciting of these is to make the PID controller either adaptive or self-tuning. Both automate the tuning of the controller. The difference is that the self-tuning controller operates in two distinct modes, tuning and controlling. Often, an operator selects the mode, tuning when there is reason to believe the controller is not tuned properly and controlling when it is correctly tuned. The adaptive controller always operates in its adaptive mode, simultaneously trying to achieve good control and tuning the parameters. There is a large literature on both self tuning and adaptive control; many different techniques have been proposed. There are also about a dozen turnkey self-tuning PID controllers on the market [1].

There are many plants, even LTI single-input single-output (SISO) ones, for which the PID Controller is ineffective or inappropriate. One example is an LTI plant with a pair of lightly damped complex conjugate poles. Nonetheless, for many such plants a more complicated controller can be easily implemented in a DSP. There are many advantages to doing this. Most are obvious but one requires a small amount of background.

Most real control systems require a "safety net." That is, provisions must be made to deal with failures, faults, and other problems that may arise. For example, the float valve in the toilet tank will eventually fail to close fully. This is an inevitable consequence of valve wear and/or small amounts of dirt in the water. An overflow tube in the tank guarantees that this is a soft failure. If the water level gets too high, the excess water spills into the overflow tube. From there it goes into the toilet and eventually into the sewage line. The valve failure does not result in a flood in the house.

These safety nets are extremely important in the real world, as should be obvious from this simple example. It is easy to implement them within the controller DSP.

This is now commonly done. In fact, because many control applications are easily implemented in a DSP, and because such processors are so cheap, it is now true that there is often a good deal of unused capacity in the DSPs used for control. An important area of controls research is to find ways to use this extra capacity to improve the controller performance. One success story is described in the following section.

3.2 Demanding Cases

The signal processing necessary for control can be very demanding. One such control technique that is extensively used is commonly known as Model Predictive Control (MPC) [3] although it has a number of other names such as Receding Horizon Control, Dynamic Matrix Control, and numerous variants. The application of MPC is limited to systems with "slow" dynamics because the computations involved require too much time to be accomplished in the sampling interval for "fast" systems. The simplest version of MPC occurs when there is no exogenous input and the objective of the controller is to drive the system state to zero. This is the case described below.

The starting point for MPC is a known plant which may have multiple inputs and multiple outputs (MIMO) and a precise mathematical measure of the performance of the closed-loop system. It is most convenient to use a state-space description of the plant

$$\dot{x}(t) = f(x(t), u(t)) \tag{20}$$

$$y(t) = g(x(t), u(t)) \tag{21}$$

Note that we have assumed that the system is time-invariant and that the noise can be adequately handled indirectly by designing a robust controller (a controller that is relatively insensitive to disturbances) rather than by means of a more elaborate controller whose design accounts explicitly for the noise. It is assumed that the state $x(t)$ is an n-vector, the control $u(t)$ is an m-vector, and the output $y(t)$ is a p-vector.

The performance measure is generically

$$J(u_{[0,\infty)}) = \int_0^\infty l(x(t), u(t))dt \tag{22}$$

The notation $u_{[0,\infty)}$ denotes the entire signal $\{u(t) : 0 \leq t < \infty\}$.

The control objective is to design and build a feedback controller that will minimize the performance measure. It is possible to solve this problem analytically in a few special cases. When an analytic solution is not possible one can resort to the approximate solution known as MPC. First, discretize both the plant and the performance measure with respect to time. Second, further approximate the

performance measure by a finite sum. Lastly, temporarily simplify the problem by abandoning the search for a feedback control and ask only for an open-loop control. The open-loop control will depend on the initial state so assume that $x(0) = x_0$ and this is known.

The approximate problem is then to find the sequence $[u(0)\ u(1) \cdots u(N-1]$ that solves the constrained nonlinear programming problem given below

$$\min_{[u(0)\ u(1) \cdots u(N)]} \sum_{i=0}^{i=N} l(x(i), u(i)) + l_N(x(N+1)) \tag{23}$$

subject to

$$x(i+1) = f_d(x(i), u(i)) \quad i = 0, 1, \cdots, N \tag{24}$$

and

$$x(0) = x_0 \tag{25}$$

Notice that the solution to this problem is a discrete-time control signal,

$$[u^o(0)\ u^o(1) \cdots u^o(N)] \tag{26}$$

(where the superscript "o" denotes optimal) that depends only on knowledge of x_0.

This will be converted into a closed-loop (and an MPC) controller in two steps. The first step is to create an idealized MPC controller that is easy to understand but impossible to build. The second step is to modify this infeasible controller by a practical, implementable MPC controller.

The idealized MPC controller assumes that $y(i) = x(i),\ \forall i$. Then, at $i = 0$, $x(0) = x_0$ is known. Solve the nonlinear programming problem instantaneously. Apply the control $u^o(0)$ on the time interval $0 \le t < \delta$, where δ is the discretization interval. Next, at time $t = \delta$, equivalently $i = 1$, obtain the new value of x, i.e., $x(1) = x_1$. Again, instantaneously solve the nonlinear programming problem, exactly as before except using x_1 as the initial condition. Again, apply only the first step of the newly computed optimal control (denote it by $u^o(1)$) on the interval $\delta \le t < 2\delta$.

The idea is to compute, at each time instant, the open-loop optimal control for the full time horizon of $N + 1$ time steps but only implement that control for the first step. Continue to repeat this forever.

Of course, the full state is not usually available for feedback (i.e., $y(i) \neq x(i)$) and it is impossible to solve a nonlinear programming problem in zero time. The solution to both of these problems is to use an estimate of the state. Let an optimal (in some sense) estimate of $x(k+1)$ given all the data up to time k be denoted by $\hat{x}(i+1|i)$. For example, assuming noiseless and full state feedback, $y(k) = x(k)\ \forall k$, and the dynamics of the system are given by

$$x(i+1) = f_d(x(i), u(i)) \quad i = 0, 1, \cdots, N \tag{27}$$

then

$$\hat{x}(k+1|k) = f_d(x(k), u(k)) \tag{28}$$

The implementable version of MPC simply replaces x_i in the nonlinear programming problem at time $t = i\delta$ by $\hat{x}(i|i-1)$ and solves for $[u^o(i)\ u^o(i+1) \cdots u^o(N+i)]$. This means that the computation of the next control value can start at time $t = i\delta$ and can take up to the time $(i+1)\delta$. It can take a long time to solve a complicated nonlinear programming problem. Because of this the application of MPC to real problems has been limited to relatively slow systems, i.e., systems for which δ is large enough to insure that the computations will be completed before the next value of the control signal is needed. As a result, there is a great deal of research at present on ways to speed up the computations involved in MPC.

The time required to complete the control computations becomes even longer if it is necessary to account explicitly for noise. Consider the following relatively simple version of MPC in which the plant is linear and time-invariant except for the saturation of the actuators. Furthermore, the plant has a white Gaussian noise input and the output signal contains additive white Gaussian noise as well. The plant is then modeled by

$$x(i+1) = Ax(i) + Bu(i) + D\xi(i) \tag{29}$$

$$y(i) = Cx(i) + \nu(i) \tag{30}$$

The two noise signals are zero mean white Gaussian noises with covariance matrices $E(\xi(i)\xi^T(i)) = \Xi\ \forall i$ and $E(\nu(i)\nu^T(i)) = I\ \forall i$ where $E(\cdot)$ denotes expectation. The two noise signals are independent of each other.

The performance measure is

$$J(u_{(0,\infty]}) = E(\frac{1}{2}\sum_{i=0}^{i=\infty}(y^T(i)Qy(i) + u^T(i)Ru(i)) \tag{31}$$

In the equation above, Q and R are symmetric real matrices of appropriate dimensions. To avoid technical difficulties R is taken to be positive definite (i.e., $u'Ru > 0$ for all $u \neq 0$) and Q positive semidefinite (i.e., $y'Qy \geq 0$ for all y). In the absence of saturation, i.e., if the linear model is accurate for all inputs, then the solution to the stochastic control problem of minimizing Eq. (31) subject to Eqs. (29) and (30) is the Linear Quadratic Gaussian (LQG) regulator [15]. It separates into two independent components. One component is the optimal feedback control $u^o(i) = F^o x(i)$, where the superscript "o" denotes optimal. This control uses the actual state vector which is, of course, unavailable. The other component of the optimal control is a Kalman filter which produces the optimal estimate of $x(i)$ given the available data at time i. Denoting the output of the filter by $\hat{x}(i|i)$, the control signal becomes

$$u^o(i) = F^o\hat{x}(i|i) \tag{32}$$

This is known as the certainty equivalent control because the optimal state estimate is used in place of the actual state. For this special case, all of the parameters can be computed in advance. The only computations needed in real time are the one step ahead Kalman filter/predictor and the matrix multiplication in Eq. (32).

If actuator saturation is important, as it is in many applications, then the optimal control is no longer linear and it is necessary to use MPC. As before, approximate the performance measure by replacing ∞ by some finite N. An exact feedback solution to this new problem would be extremely complicated. Because of the finite time horizon, it is no longer true that the optimal control is time invariant. Because of the saturation, it is no longer true that the closed-loop system is linear nor is it true that the state estimation and control are separate and independent problems. Even though this separation is not true, it is common to design the MPC controller by using a Kalman filter to estimate the state vector. Denote this estimate by $\hat{x}(i|i)$ and the one step ahead prediction by $\hat{x}(i|i-1)$. The finite time LQ problem with dynamics given by Eqs. (29) and (30) and performance measure

$$J(u_{(0,N]}) = E(\frac{1}{2}\sum_{i=0}^{i=N}(y^T(i)Qy(i)+u^T(i)Ru(i))+y^T(N+1)Qy(N+1)) \tag{33}$$

is then solved open loop with initial condition $x(0) = \hat{x}(i|i-1)$ and with the constraint

$$u(i) = \begin{cases} u_{max}, & \text{if} \quad u(i) \geq u_{max} \\ u(i), & \text{if} \quad |u(i)| < u_{max} \\ -u_{max}, & \text{if} \quad u(i) \leq -u_{max} \end{cases} \tag{34}$$

This is a convex programming problem which can be quickly and reliably solved. As is usual in MPC, only the first term of the computed control sequence is actually implemented. The substitution of the one step prediction for the filtered estimate is done so as to provide time for the computations.

There is much more to MPC than has been discussed here. An introduction to the very large literature on the subject is given in Sect. 5. One issue that is too important to omit is that of stability. The rationale behind MPC is an heuristic notion that the performance of such a controller is likely to be very good. While good performance implicitly requires stability, it certainly does not guarantee it. Thus, it is comforting to know that stability is theoretically guaranteed under reasonably weak assumptions for a broad range of MPC controllers [20].

3.3 Exemplary Case

ABS brakes are a particularly vivid example of the challenges and benefits associated with the use of DSP in control. The basic problem is simply stated. Control the automobile's brakes so as to minimize the distance it takes to stop. The theoretical solution is also simple. Because the coefficient of sliding friction is smaller than the coefficient of rolling friction, the vehicle will stop in a shorter distance if the braking force is the largest it can be without locking the wheels. All this is well known. The difficulty is the following. The smallest braking force that locks the wheels depends on how slippery the road surface is. For example, a very small braking force will lock the wheels if they are rolling on glare ice. A much larger force can be applied without locking the wheels when they are rolling on dry pavement. Thus, the key practical problem for ABS brakes is how to determine how much force can be applied without locking the wheels. In fact, there is as yet no known way to measure this directly. It must be estimated. The control, based on this estimate, must perform at least as well as an unassisted human driver under every possible circumstance.

The key practical idea of ABS brakes is to frequently change the amount of braking force to probe for the best possible value. Probing works roughly as follows. Apply a braking force and determine whether the wheels have locked. If they have, reduce the braking force to a level low enough that the wheels start to rotate again. If they have not, increase the braking force. This is repeated at a reasonably high frequency.

Probing cannot stop and must be repeated frequently because the slipperiness of the road surface can change rapidly. It is important to keep the braking force close to its optimal value all the time. Balancing the dual objectives of quickly detecting changes in the slipperiness of the surface and keeping the braking force close to its optimal value is one of the keys to successful ABS brakes. Note that this tradeoff is typical of adaptive control problems where the controller has to serve the dual objectives of providing good control and providing the information needed to improve the controller.

The details of the algorithm for combining probing and control are complicated [2]. The point here is that the complicated algorithm is a life saving application of digital signal processing for control. Two other points are important. In order for ABS brakes to be possible, it was first necessary to develop a hydraulic braking system that could be electronically modulated. That is, the improved controller depends on a new actuator. Second, there is very little control theory available to assist in the design of the controller upon which ABS braking depends.

There are three reasons why control theory is of little help in designing ABS brakes. First, the interactions between tires and the road surface are very complex. The dynamic distortion of the tire under loading plays a fundamental role in traction (see [8] for a dramatic example). While there has been research on appropriate mathematical models for the tire/road interaction, realistic models are too complex for control design and simpler models that might be useful for controller design are not realistic enough.

Second, the control theory for systems that have large changes in dynamics as a result of small changes in the control—exactly what happens when a tire loses traction—is only now being developed. Such systems are one form of hybrid system. The control of hybrid systems is a very active current area of research in control theory.

Third, a crucial component of ABS brakes is the electronically controllable hydraulic brake cylinder. This device is obviously needed in order to implement the controller. Control theory has very little to say about the physical devices needed to implement a controller.

4 Conclusions

Nowadays most control systems are implemented digitally. This is motivated primarily by cost and convenience. It is abetted by continuing advances in sensing and actuation. More and more physical signals can be converted to electrical signals with high accuracy and low noise. More and more physical actuators convert electrical inputs into physical forces, torques, etc. Sampling and digitization of the sensor signals is also convenient, very accurate if necessary, and inexpensive. Hence, the trend is very strongly towards digital controllers.

The combination of cheap sensing and digital controllers has created exciting opportunities for improved control and automation. Adding capability to a digitally implemented controller is now often cheap and easy. Nonlinearity, switching, computation, and logical decision making can all be used to improve the performance of the controller. The question of what capabilities to add is wide open.

5 Further Reading

There are many good undergraduate textbooks dealing with the basics of control theory. Two very popular ones are by Dorf and Bishop [4] and by Franklin et al. [6]. A standard undergraduate textbook for the digital aspects of control is [5]. A very modern and useful upper level undergraduate or beginning graduate textbook is by Goodwin et al. [7]. An excellent source for information about PID control is [1].

Graduate programs in control include an introductory course in linear system theory that is very similar to the ones for signal processing. The book by Kailath [10] although old is very complete and thorough. Rugh [19] is more control oriented and newer. The definitive graduate textbook on nonlinear control is [11].

For more advanced and specialized topics in control, The Control Handbook [13] is an excellent source. It was specifically designed for the purpose of providing a starting point for further study. It also contains good introductory articles about undergraduate topics and a variety of applications.

There is a very large literature on MPC. Besides the previously cited [3], there are books by Maciejowski [16] and Kwon and Han [12] as well as several others. There are also many survey papers. The one by Rawlings [18] is easily available and recommended. Another very useful and often cited survey is by Qin and Badgwell [17]. This is a particularly good source for information about companies that provide MPC controllers and other important practical issues,

Lastly, the Handbook of Networked and Embedded Control Systems [9] provides introductions to most of the issues of importance in both networked and digitally controlled systems.

References

1. Astrom, K.J., Hagglund, T.: PID Controllers: Theory, Design, and Tuning (2nd edtion), International Society for Measurement and Control, Seattle, (1995)
2. Bosch: Driving Safety Systems (2nd edition), SAE International, Warrenton, (1999)
3. Camacho, E.F., Bordons, C.: Model Predictive Control, 2nd Edition, Springer, London, (2004)
4. Dorf, R.C., Bishop, R. H.: Modern Control Systems, (11th edition), Pearson Prentice-Hall, Upper Saddle River, (2008)
5. Franklin, G.F., Powell, J.D., Workman, M.: Digital Control of Dynamic Systems (3rd edition), Addison-Wesley, Menlo Park (1998)
6. Franklin, G.F., Powell, J.D., Emami-Naeini, A.: Feedback Control of Dynamic Systems (5th edition), Prentice-Hall, Upper Saddle River, (2005)
7. Goodwin, G.C., Graebe, S.F., Salgado, M.E.: Control System Design, Prentice-Hall, Upper Saddle River, (2001)
8. Hallum, C.: The magic of the drag tire. SAE Paper 942484, Presented at SAE MSEC (1994)
9. Hristu-Varsakelis, D., Levine, W.S.: Handbook of Networked and Embedded Control Systems, Birkhauser, Boston, (2005)
10. Kailath, T. Linear Systems, Prentice-Hall, Englewood Cliffs, (1980)
11. Khalil, H.K. Nonlinear Systems (3rd edition), Prentice-Hall, Upper Saddle River, (2001)
12. Kwon, W. H.,Han, S.: Receding Horizon Control: Model Predictive Control for State Models. Springer, London (2005)
13. Levine, W.S. (Editor): The Control Handbook (2nd edition), CRC Press, Boca Raton (2011)
14. Looze, D. P., Freudenberg, J. S.: Tradeoffs and limitations in feedback systems. The Control Handbook, pp 537–550, CRC Press, Boca Raton(1995)
15. Lublin, L., Athans, M.: Linear quadratic regulator control. The Control Handbook, pp 635–650, CRC Press, Boca Raton (1995)
16. Maciejowski, J. M.: Predictive control with constraints. Prentice Hall, Englewood Cliffs (2002)
17. Qin, S. J., Badgwell, T. A.: A survey of model predictive control technology. Control Engineering Practice,11, pp 733–764 (2003)
18. Rawlings, J. B.: Tutorial overview of model predictive control. IEEE Control Systems Magazine, 20(3) pp 38–52, (2000)
19. Rugh, W.J.: Linear System Theory (2nd edition), Prentice-Hall, Upper Saddle River, (1996)
20. Scokaert, P. O. M, Mayne, D. Q., Rawlings, J. B.: Suboptimal model predictive control (feasibility implies stability). IEEE Transactions on Automatic Control, 44(3) pp 648–654 (1999)
21. Yang, J. S., Levine, W. S.: Specification of control systems. The Control Handbook, pp 158–169, CRC Press, Boca Raton (1995)

MPEG Reconfigurable Video Coding

Marco Mattavelli, Jorn W. Janneck, and Mickaël Raulet

Abstract The current monolithic and lengthy scheme behind the standardization and the design of new video coding standards is becoming inappropriate to satisfy the dynamism and changing needs of the video coding community. Such a scheme and specification formalism do not enable designers to exploit the clear commonalities between the different codecs, neither at the level of the specification nor at the level of the implementation. Such a problem is one of the main reasons for the typical long time interval elapsing between the time a new idea is validated until it is implemented in consumer products as part of a worldwide standard. The analysis of this problem originated a new standard initiative within the ISO/IEC MPEG committee, called Reconfigurable Video Coding (RVC). The main idea is to develop a video coding standard that overcomes many shortcomings of the current standardization and specification process by updating and progressively incrementing a modular library of components. As the name implies, flexibility and reconfigurability are new attractive features of the RVC standard. The RVC framework is based on the usage of a new actor/dataflow oriented language called CAL for the specification of the standard library and the instantiation of the RVC decoder model. CAL dataflow models expose the intrinsic concurrency of the algorithms by employing the notions of actor programming and dataflow. This chapter gives an overview of the concepts and technologies building the standard RVC framework and the non standard tools supporting the RVC model from the instantiation and simulation of the CAL model to the software and/or hardware code synthesis.

M. Mattavelli (✉)
SCI-STI-MM, EPFL, Lausanne, Switzerland
e-mail: marco.mattavelli@epfl.ch

J. W. Janneck
Department of Computer Science, Lund University, Lund, Sweden
e-mail: jwj@cs.lth.se

M. Raulet
ATEME, Rennes, France
e-mail: m.raulet@ateme.com

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_7

1 Introduction

A large number of successful MPEG (Moving Picture Expert Group) video coding standards has been developed since the first MPEG-1 standard in 1988 [20]. The standardization efforts in the field, besides having as first objective to guarantee the interoperability of compression systems, have also aimed at providing appropriate forms of specifications for wide and easy deployment. While video standards are becoming increasingly complex, and they take ever longer to be produced, this makes it difficult for standards bodies to produce timely specifications that address the need to the market at any given point in time. The structure of past standards has been one of a monolithic specification together with a fixed set of *profiles* that subset the functionality and capabilities of the complete standard. Similar comments apply to the reference code, which in more recent standards has become normative itself. Video devices are typically supporting a single profile of a specific standard, or a small set of profiles. They have therefore only very limited adaptivity to the video content, or to environmental factors (bandwidth availability, quality requirements).

Within the ISO/IEC MPEG committee, Reconfigurable Video Coding (RVC) [6, 38] standard is intended to address the two following issues: make standards faster to produce, and permit video devices based on those standards to exhibit more flexibility with respect to the coding technology used for the video content. The key idea is to standardize a library of video coding components, instead of an entire video decoder. The standard can then evolve flexibly by incrementally extending that library, and video devices can configure themselves to support a variety of coding algorithms by composing encoders and decoders from that library of predefined coding modules. Recently the concepts of standardizing modular components have been also extended with success to the coding of 3-D graphic objects, achieving the same objectives initially identified within the video coding field [7, 8]. For this reason the standard framework is now also referred to as Reconfigurable Media Framework (RMC) to acknowledge the inclusion of others media than video.

This chapter gives an overview of the concepts and technologies building the standard RVC framework shown in Fig. 1 and can be complemented by the chapter of the handbook: “Dataflow modeling for reconfigurable signal processing systems” [12].

2 Requirements and Rationale of the MPEG RVC Framework

Started in 2004, the MPEG Reconfigurable Video Coding (RVC) framework [6] is a new ISO standard (Fig. 1) aiming at providing an alternative form of video codec specifications by standardizing a library of modular dataflow components instead of monolithic sequential algorithms. RVC provides the new form of specification by

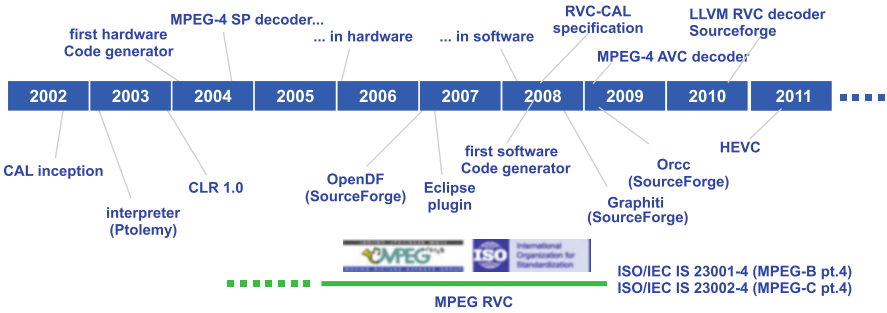


Fig. 1 CAL and RVC standard timeline

defining two standard elements: a dataflow language with which video decoders can be described (ISO/IEC23001-4 or MPEG-B pt. 4 [34]) and a library of video coding tools employed in MPEG standards (ISO/IEC23002-4 or MPEG-C pt. 4 [35]). The new concept is to be able to specify a decoder of an existing standard or a completely new configuration that may better satisfy application-specific constraints by selecting standard components from a library of standard coding algorithms. Such possibility also requires extended methodologies and new tools for describing the new bitstream syntaxes and the instantiation of the parsers of such new codecs. These extensions has been recently finalized and are currently available in new amendments of the standard [37]. An additional possibility of RVC is also to be able to enable at runtime the dynamic reconfiguration of codecs at terminal side. Such option also requires normative extensions of the system layer for the transport of the new configurations and the associated signaling and is currently under study by the MPEG committee, in particular addressing the possibility of instantiating simplified decoder configurations providing low-power performance or configurations exposing different levels of parallelism.

The essential concepts of the RVC framework (Fig. 2) are the following:

- RVC-CAL [23], a dataflow language describing the Functional Unit (FU) behavior. The language defines the behavior of dataflow components called actors (or FUs in MPEG), which is a modular component that encapsulates its own state such that an actor can neither read nor modify the state of any other actor. The only interaction between actors is via messages (known in CAL as tokens) which flow from an output of one actor to an input of another. The behavior of an actor is defined in terms of a set of atomic actions. The execution inside an actor is purely sequential: at any point in time, only one action can be active inside an actor. An action can consume (read) tokens, modify the internal state of the actor, produce tokens, and interact with the underlying platform on which the actor is running.

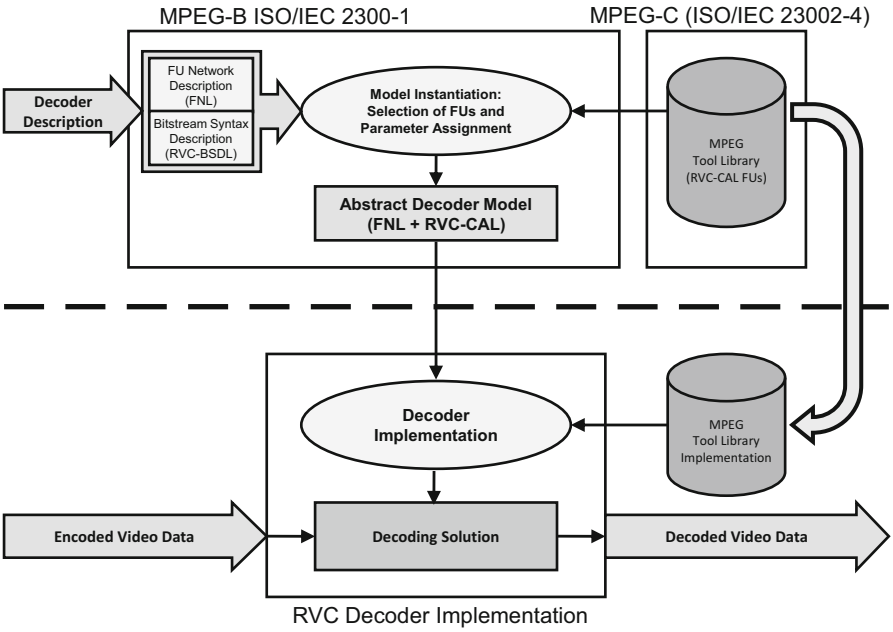


Fig. 2 RVC standard

- FNL (Functional unit Network Language), a language describing the video codec configurations. FNL is an XML dialect that lists the FUs composing the codec, the parameterization of these FUs and the connections between the FUs. FNL allows hierarchical constructions: an FU can be defined as a composition of other FUs and described by another FND (FU Network Description).
- BSDL (Bitstream Syntax Description Language), a language describing the structure of the input bitstream. BSDL is a XML dialect that lists the sequence of the syntax elements with possible conditioning on the presence of the elements, according to the value of previously decoded elements. BSDL is further explained in Sect. 4.4.
- A library of video coding tools, also called Functional Units (FU) covering all MPEG standards (the "MPEG Toolbox"). This library is specified and provided using RVC-CAL (a subset of the original CAL language that is standardized by MPEG) as specification language for each FU.
- An "Abstract Decoder Model" (ADM) constituting a codec configuration (described using FNL) instantiating FUs of the MPEG Toolbox. Figure 2 depicts the process of instantiating an "Abstract Decoder Model" in RVC.
- Tools simulating and validating the behavior of the ADM (Open DataFlow environment [56]).
- Tools automatically generating software and hardware descriptions of the ADM.

3 Rationale for Changing the Traditional Specification Paradigm Based on Sequential Model of Computation

As briefly introduced in the previous section, one of the more radical innovations introduced by the RVC standard is the adoption of a non-traditional model of computation and a new specification language. The main reasons for this change in building specifications are discussed here in more depth [9].

For most of the history of silicon-based computing, the steady scaling of silicon technology has led to ever faster sequential computing machines, with higher clock rates and more sophisticated internal architectures exploiting the improvements in silicon manufacturing. Backwards compatible processor designs ensured that software remained portable to new machines, which in turn implied that legacy software automatically benefited from any progress in the way processors were built, and so have complex algorithms, such as video codecs, and the associated reference SW descriptions, using generic, sequential programming languages.

In recent years, however, this has ceased to be the case. In spite of continued scaling of silicon technology, individual sequential processors are not becoming faster any more, but slightly slower while reducing power dissipation [3]. Consequently, rather than building more sophisticated and complex single processors, manufacturers have used the space gained from scaling the technology by building more processors onto a single chip, making multi-core machines and heterogeneous systems a nearly ubiquitous commodity in a wide (and increasing) range of computing applications. As a result, the performance gains of modern computing machines are primarily due to an increase in the available parallelism (Fig. 3).

These developments pose qualitatively novel challenges to the portability of specifications, applications and ultimately the software that is used to implement them, as well as to software engineering and implementation methodology in general: while sequential software used to automatically execute faster on a faster processor, an increase in performance of an application on a new platform that provides more parallelism is predicated on the ability to effectively exploit that parallelism, i.e. to parallelize the application and thus match it to the respective computing substrate.

Traditionally, applications described in the style of mostly sequential algorithms have taken advantage of multiple execution units using threads and processes, thereby explicitly structuring an application into a (usually small) set of concurrently executing sequential activities that interact with each other using shared memory or other means of communication (e.g. messages, pipes, semaphores) often provided either by the operating system or some middleware. However, this parallel programming approach has some significant drawbacks [48]. First, it poses considerable engineering challenges—large collections of communicating threads are difficult to test since errors often arise due to the timing of activities in ways that cannot be detected or reproduced easily, and the languages, environments, and tools usually provide little or no support for managing the complexities of highly parallel execution. Second, a thread-based approach scales poorly across

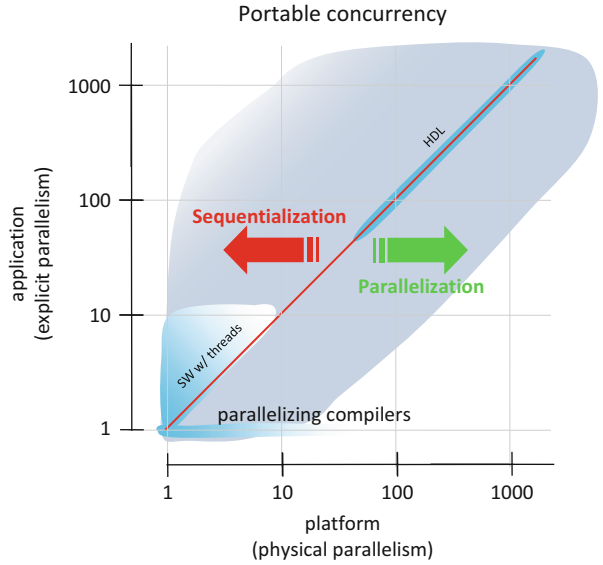


Fig. 3 Representation of a unified SW-HW design space, showing how leaving current sequential Von Neumann based approaches, portable parallelism and SW/HW unified programming/design can be achieved in a much larger design space by developing efficient sequentialization and parallelization techniques. Currently only design spaces labeled in the picture as "HDL" "SW w/threads" and "parallelising compilers" are covered. Dataflow approach will allow to cover a much larger space (gray area)

platforms with different degrees of parallelism if the number of execution units is significantly different from the number of threads. Too few execution units mean that several threads need to be dynamically scheduled onto each of them, incurring scheduling overhead. If the number of processors exceeds the number of threads, the additional processors remain unused. The result is that threaded applications either need to be overengineered to using as many threads as possible, with the attendant consequences for engineering cost and performance on less parallel hardware, or they will underutilize highly parallel platforms. Either way, the requirement to construct an application with a particular degree of parallelism in mind is a severe obstacle to the portability of threaded software.

In an effort to implement sequential or threaded applications on platforms that provide higher degrees of parallelism than the application itself, parallelizing compilers have been used with some success [53]. However, the effectiveness of automatic parallelization depends highly on the application and the details of the algorithm description, and it does not scale well for larger programs.

For algorithm specifications and corresponding software to scale to future parallel computing platforms as seamlessly as possible, it is necessary to describe algorithms in a way that:

1. exposes as much parallelism of the application as practical,
2. provides simple and natural abstractions that help manage the high degree of parallelism and permits principled composition of and interaction between modules,
3. makes minimal assumptions about the physical architecture of the computing machine it is implemented on,
4. is efficiently implementable on a wide range of computing substrates, including traditional sequential processors, shared-memory multicores, manycore processor arrays, and programmable logic devices, as well as combinations thereof.

This is not a trivial proposition, since it implies among other things that the current body of software will not by itself be implementable efficiently on future computers but will have to be rewritten if it is supposed to take advantage of the parallel performance of these machines. In fact, the requirements above suggest a programming style and a tool support that formulates applications in as parallel a way as possible, so that implementation frequently involves *sequentializing* [22] as well as parallelizing [53] applications. This is effectively the antithesis of the current approach to mapping software onto parallel platforms, which tends to begin with sequential code, and parallelization, either manually or automatically, is the process of adapting the sequential algorithm and code to a parallel implementation target. Looking at the above criteria, shared-memory threads for instance fulfill the first requirement, but essentially fail on the other three. By comparison, hardware description languages such as VHDL and Verilog fulfill the first two criteria, but as they fail on the third point by assuming a particular model of (clocked) time and their implementability is essentially limited to hardware and hardware-like programmable logic (FPGAs). Needless to say CAL dataflow programming is a good candidate to be able to satisfy the above requirements and for such reasons has been selected and adopted by the MPEG RVC standards.

3.1 Limits of Previous Monolithic Specifications

MPEG has produced several generations of video coding standards such as MPEG-1, MPEG-2, MPEG-4 Video, AVC (Advanced Video Coding) and SVC (Scalable Video Coding) its scalable profile. The last generation of video coding standards, published in 2013, is called High Efficiency Video Coding (HEVC), yielding more than a factor 2 gain versus the previous generation standard performance (AVC) particularly for Ultra High Definition resolution video content. While at the beginning MPEG-1 and MPEG-2 were only specified by textual descriptions, with the increasing complexity of algorithms, starting with the MPEG-4 set of standards, C or C++ specifications, called also reference software, have became the formal specification of the standards. However, the past monolithic specification of such standards (usually in the form of C/C++ programs) lacks flexibility and does not allow to use the combination of coding algorithms from different standards

enabling to achieve specific design or performance trade-offs and thus fill, case by case, the requirements of specific applications. Indeed, not all coding tools defined in a *profile@level* of a specific standard are required in all application scenarios. For a given application, codecs are either not exploited at their full potential or require unnecessarily complex implementations. However, a decoder conformant to a standard has to support all of them and may results in non-efficient implementations.

Moreover, such descriptions composed of non-optimized non-modular software packages have started to show many limits. If we consider that they are in practice the starting point of any implementation, system designers have to rewrite these software packages not only to try to optimize performances, but also to transform these descriptions into appropriate forms adapted to the current system design methodologies. Such monolithic specifications hide the inherent parallelism and the dataflow structure of the video coding algorithms, features that are necessary to be exploited for efficient implementations. In the meanwhile the evolution of video coding technologies, leads to solutions that are increasingly complex to be designed and present significant overlap between successive versions of the standards.

Why C etc. Fail? The control over low-level details, which is considered a merit of C language, typically tends to over-specify programs. Not only the algorithms themselves are specified, but also how inherently parallel computations are sequenced, how and when inputs and outputs are passed between the algorithms and, at a higher level, how computations are mapped to threads, processors and application specific hardware. In general, it is not possible to recover the original knowledge about the intrinsic properties of the algorithms by means of analysis of the software program and the opportunities for restructuring transformations on imperative sequential code are very limited compared to the parallelization potential available on multi-core platforms [5]. These in conjunction with the previously discussed motivations, are the main reasons for which C has been replaced by CAL in MPEG RVC [43].

3.2 Reconfigurable Video Coding Specification Requirements

Scalable Parallelism In parallel programming, the number of things that are happening at the same time can scale in two ways: It can increase with the size of the problem or with the size of the program. Scaling a regular algorithm over larger amounts of data is a relatively well-understood problem, while building programs such that their parts execute concurrently without much interference is one of the key problems in scaling the von Neumann model. The explicit concurrency of the actor model provides a straightforward parallel composition mechanism that tends to lead to more parallelism as applications grow in size, and scheduling techniques permit scaling concurrent descriptions onto platforms with varying degrees of parallelism.

Modularity and Reuse The ability to create new abstractions by building reusable entities is a key element in every programming language. For instance, object-oriented programming has made huge contributions to the construction of von Neumann programs, and the strong encapsulation of actors along with their hierarchical composability offers an analog for parallel programs.

Concurrency In contrast to procedural programming languages, where control flow is made explicit, the actor model emphasizes explicit specification of concurrency. Rallying around the pivotal and unifying von Neumann abstraction has resulted in a long and very successful collaboration between processor architects, compiler writers, and programmers. Yet, for many highly concurrent programs, portability has remained an elusive goal, often due to their sensitivity to timing. The untimedness and asynchrony of stream-based programming offers a solution to this problem. The portability of stream-based programs is underlined by the fact that programs of considerable complexity and size can be compiled to competitive hardware [42] as well as software [66], which suggests that stream-based programming might even be a solution to the old problem of flexibly co-synthesizing different mixes of hardware/software implementations from a single source.

Encapsulation The success of a stream programming model will in part depend on its ability to configure dynamically and to virtualize, i.e. to map to collections of computing resources too small for the entire program at once. Moving parts of a program on and off a resource requires encapsulation, i.e. a clear distinction between those pieces that belong to the parts to be moved and those that do not. The transactional execution of actors generates points of *quiescence*, the moments between transactions, when the actor is in a defined and known state that can be safely transferred across computing resources.

4 Description of the Standard or Normative Components of the Framework

The fundamental element of the RVC framework, in the normative part, is the Decoder Description (Fig. 2) that includes two types of data:

The Bitstream Syntax Description (BSD), which describes the structure of the bitstream. The BSD is written in RVC-BSDL. It is used to generate the appropriate parser to decode the corresponding input encoded data [33, 64].

The FU Network Description (FND), which describes the connections between the coding tools (i.e. FUs). It also contains the values of the parameters used for the instantiation of the different FUs composing the decoder [21, 42, 66]. The FND is written in the so called FU Network Language (FNL). The syntax parser (built from the BSD), together with the network of FUs (built from the FND), form a CAL model called the Abstract Decoder Model (ADM), which is the normative behavioral model of the decoder.

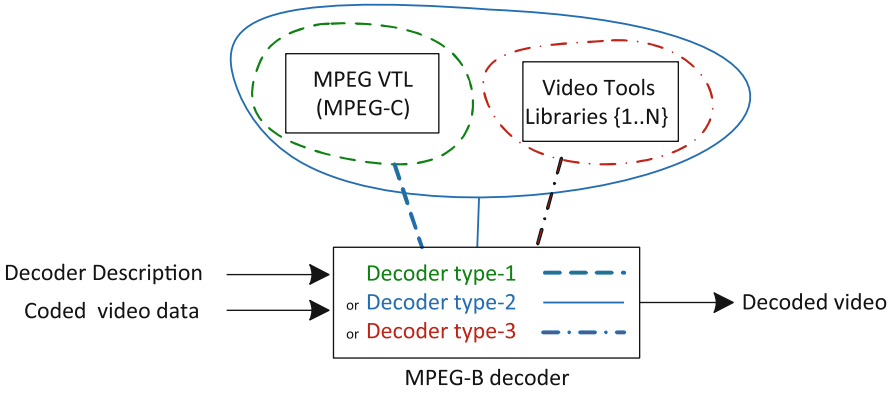


Fig. 4 The conceptual view of RVC

4.1 The Toolbox Library

An interesting feature of the RVC standard that distinguishes it from traditional decoders-rigidly-specified video coding standards is that, a description of the decoder can be associated to the encoded data in various ways according to each application scenario. Figure 4 illustrates this conceptual view of RVC [50, 51]. All the three types of decoders are within the RVC framework and constructed using the MPEG-B standardized languages. Hence, they all conform to the MPEG-B standard. A `Type-1` decoder is constructed using the FUs within the MPEG Video Tool Library (VTL) only. Hence, this type of decoder conforms to both the MPEG-B and MPEG-C standards. A `Type-2` decoder is constructed using FUs from the MPEG VTL as well as one or more proprietary libraries (VTL 1-n). This type of decoder conforms to the MPEG-B standard only. Finally, a `Type-3` decoder is constructed using one or more proprietary VTL (VTL 1-n), without using the MPEG VTL. This type of decoder also conforms to the MPEG-B standard only. An RVC decoder (i.e. conformant to MPEG-B) is composed of coding tools described in VTLs according to the decoder description. The MPEG VTL is described by MPEG-C. Traditional programming paradigms (monolithic code) are not appropriate for supporting such types of modular framework. A new dataflow-based programming model is thus specified and introduced by MPEG RVC as specification formalism.

The MPEG VTL is normatively specified using RVC-CAL. An appropriate level of granularity for the components of the standard library is important, to enable an effective possibility of reconfigurations, for codecs, and an efficient reuse of components in codecs implementations. If the library is composed of too coarse modules, such modules will be too large/coarse to allow their usage in different and interesting codec configurations, whereas, if the library component granularity level is too fine, the number of modules in the library will result to be too large for an efficient and practical reconfiguration process at the codec implementation

side, and may obscure the desired high-level description and modeling features of the RVC codec specifications. Most of the efforts behind the standardization of the MPEG VTL were devoted to study the best granularity trade-off level of the VTL components. However, it must be noticed that the choice of the best trade-off in terms of high-level description and module re-usability, does not really affect the potential parallelism of the algorithm that can be exploited in multi-core and FPGA implementations.

4.2 The CAL Actor Language

CAL [23] is a domain-specific language that provides useful abstractions for dataflow programming with actors. For more information on dataflow methods, the reader may refer to Part IV (Design Methods), which contains several chapters that go into detail on various kinds of dataflow techniques for design and implementation of signal processing systems. CAL has been used in a wide variety of applications and has been compiled to hardware and software implementations, and work on mixed HW/SW implementations is under way. The next section provides a brief introduction to some key elements of the language.

4.2.1 Basic Constructs

The basic structure of a CAL actor is shown in the `Add` actor (Fig. 5), which has two input ports `A` and `B`, and one output port `Out`, all of type `T`. `T` may be of type `int`, or `uint` for respectively integers and unsigned integers, of type `bool` for booleans, or of type `float` for floating-point integers. Moreover CAL designers may assign a number of bits to the specific integer type depending on the variable numeric size. The actor contains one *action* that consumes one token on each input ports, and produces one token on the output port. An action may *fire* if the availability of tokens on the input ports matches the *port patterns*, which in this example corresponds to one token on both ports `A` and `B`.

```
actor Add() T A, T B ⇒ T Out :
  action [a], [b] ⇒ [sum]
  do
     sum := a + b;
  end
end
```

Fig. 5 Basic structure of a CAL actor

```
actor Select () S, A, B ⇒ Output:

    action S: [sel], A: [v] ⇒ [v]
    guard sel end

    action S: [sel], B: [v] ⇒ [v]
    guard not sel end
end
```

Fig. 6 Guard structure in a CAL actor

```
actor PingPongMerge () Input1, Input2 ⇒ Output:

  A: action Input1: [x] ⇒ [x] end
  B: action Input2: [x] ⇒ [x] end

  schedule fsm s1:
    s1 (A) --> s2;
    s2 (B) --> s1;
  end
end
```

Fig. 7 FSM structure in a CAL actor

An actor may have any number of actions. The untyped `Select` actor (Fig. 6) reads and forwards a token from either port `A` or `B`, depending on the evaluation of guard conditions. Note that each of the actions has empty bodies.

4.2.2 Priorities and State Machines

An action may be labeled and it is possible to constrain the legal firing sequence by expressions over labels. In the `PingPongMerge` actor, reported in Fig. 7, a finite state machine *schedule* is used to force the action sequence to alternate between the two actions `A` and `B`. The schedule statement introduces two states `s1` and `s2`.

The `Route` actor, in Fig. 8, forwards the token on the input port `A` to one of the three output ports. Upon instantiation it takes two parameters, the functions `P` and `Q`, which are used as predicates in the guard conditions. The selection of which action to fire is in this example not only determined by the availability of tokens and the guards conditions, by also depends on the `priority` statement.

4.2.3 CAL Subset Language for RVC

For an in-depth description of the language, the reader is referred to the language report [23], for the specific subset specified and standardized by ISO in the Annex C of [34]. This subset only deals with fully typed actors and some restrictions on

```
actor Route (P, Q) A ⇒ X, Y, Z:

    toX: action [v] ⇒ X: [v]
         guard P(v) end
    toY: action [v] ⇒ Y: [v]
         guard Q(v) end
    toZ: action [v] ⇒ Z: [v] end

    priority
        toX > toY > toZ;
    end
end
```

Fig. 8 Priority structure in a CAL actor

```
actor Select () T1 S, T2 A, T3 B ⇒ T3 Output:

    action S: [sel], A: [v] ⇒ [v]
    guard sel end

    action S: [sel], B: [v] ⇒ [v]
    guard not sel end
end
```

Fig. 9 Guard structure in a RVC-CAL actor

```
actor PingPongMerge () T Input1, T Input2 ⇒ T Output:

  A: action Input1: [x] ⇒ [x] end
  B: action Input2: [x] ⇒ [x] end

  schedule fsm s1:
    s1 (A) --> s2;
    s2 (B) --> s1;
  end
end
```

Fig. 10 FSM structure in a RVC-CAL actor

the CAL language constructs from [23] to have efficient hardware and software code generations without changing the expressivity of the algorithm. For instance, Figs. 6, 7 and 8 are not RVC-CAL compliant and must be changed as Figs. 9, 10 and 11 where *T1, T2, T* are the types and only typed parameters can be passed to the actors not functions as *P, Q*.

A large selection of example actors is available at the OpenDF repository [56], among them can also be found the MPEG-4 decoder discussed below. Many other actors written in RVC-CAL are available as reference SW of the standard MPEG RVC tool repository (ISO/IEC 23002-4). Currently beside the MPEG-4 SP, MPEG-A Part 10 AVC is available as Constrained Baseline Profile, Progressive High Profile

```
actor Route () T A ⇒ T X, T Y, T Z:
    funtion P(T v_in)--> T:
    \\ body of the function P
      P(v_in)
    end
    funtion Q(T v_in)--> T:
    \\ body of the function P
      Q(v_in)
    end

    toX: action [v] ⇒ X: [v]
         guard P(v) end
    toY: action [v] ⇒ Y: [v]
         guard Q(v) end
    toZ: action [v] ⇒ Z: [v] end

    priority
        toX > toY > toZ;
    end
end
```

Fig. 11 Priority structure in a RVC-CAL actor

and their scalable profile version, as well as the last generation of HEVC decoder including different versions of actors that make possible to implement decoders with different levels of parallelisms including multiple parser decoder versions.

4.2.4 Non-standard Process Language Extension to CAL

The constructs discussed above make CAL a very versatile and general language for expressing actors that are processing streaming data, permitting the construction of a wide range of such actors, from highly regular and static ones, to actors with very data-dependent behavior, to actors sensitive to the timing of token arrivals, and even nondeterministic actors. However, in practice, many kernels are fairly simple and often do not require the generality provided by the language. In fact, in some cases, the requirement to describe the computation of an kernel as a set of actions whose selection is governed by token availability, guards, and state can lead to overly complex programs.

Consider the actor in Fig. 12, `SumN`, which reads a number from one of its input ports, and then reads that many tokens from its other input port, adds them, and produces the result. CAL as standardized by RVC requires that the computation be structured into three actions, whose selection is determined by a state machine as

```
actor SumN() X, N ⇒ Sum :
  n; sum;

  start:
    action N:[nbr] ⇒
    do
      sum := 0;
      n := nbr;
    end

  add:
    action X:[x] ⇒
    guard n > 0
    do
      sum := sum + x;
      n := n - 1;
    end

  done:
    action ⇒ Sum:[sum]
    guard n <= 0 end

  schedule s0:
    s0 (start) --> s1;
    s1 (add) --> s1;
    s1 (done) --> s0;
  end
end
```

Fig. 12 SumN actor

well as the guards of its last two actions, making an otherwise simple procedure rather difficult to read and understand, and also error-prone to write.[1]

Actors such as this have motivated the search for other ways of writing stream processing kernels. The TŸCHO framework (cf. Sect. 5.6) includes an alternative syntax for stream processing kernels inspired by the process language used by Kahn in [44]. In it, a kernel is described as a *process*, i.e. a sequential program, typically an infinite loop, which explicitly reads tokens from input ports (using the syntax *Port* `->` *Variable*) and writes the value of an expression to output ports (with *Port* `<-` *Expression*).

Figure 13 shows the description of SumN in Fig. 12 as such a process. One interesting aspect of the Tÿcho implementation of this process language is the fact that it compiles to the same intermediate representation as the original CAL actor descriptions. This implies that actor and process descriptions can be mixed freely

[1] A common mistake in a situation such as this is to omit the seemingly redundant inverted guard of the third action and replace it with a priority between the second and the third action, resulting in a rather difficult-to-find a subtle error that manifests only in some circumstances.

```
process SumN() X, N ⇒ Sum :
  n; sum; x;

  repeat
    N —> n;
    sum := 0;
    while n > 0 do
      X —> x;
      sum := sum + x;
      n := n - 1;
    end
    Sum <— sum;
  end
end
```

Fig. 13 SumN actor in Fig. 12 expressed as a process

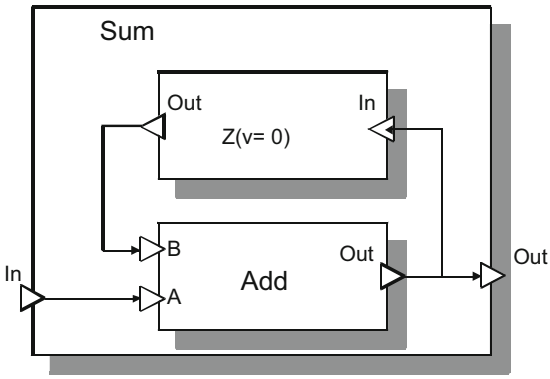


Fig. 14 A simple CAL network

in the same dataflow program, and also that processes are amenable to the same analyses, optimizations, and code generation techniques as actors. See [15, 16] for more details on this topic. The described language extension provide a very natural way of specifying source and sink actors of a network (for instance parsers and displays for video codecs) which can then be synthesized to efficient SW or HW implementations.

4.3 FU Network Language for Codec Configurations

A set of CAL actors are instantiated and connected to form a CAL application, i.e. a CAL network. Figure 14 shows a simple CAL network Sum, which consists of the previously defined RVC- CAL Add actor and the delay actor shown in Fig. 15.

The source/language that defined the network Sum is found in Fig. 16. Please, note that the network itself has input and output ports and that the instantiated entities may be either actors or other networks, which allow for a hierarchical design.

```
actor Z (v) T In ⇒ T Out:

  A: action ⇒ [v] end
  B: action [x] ⇒ [x] end

  schedule fsm s0:
    s0 (A) —> s1;
    s1 (B) —> s1;
  end
end
```

Fig. 15 RVC-CAL delay actor

```
network Sum () In ⇒ Out:

entities
  add = Add();
  z = Z(v=0);

structure
  In —> add.A;
  z.Out —> add.B;
  add.Out —> z.In;
  add.Out —— > Out;
end
```

Fig. 16 Textual representation of the Sum network

Formerly, networks have been traditionally described in a textual language, which can be automatically converted to FNL and vice versa—the XML dialect standardized by ISO in Annex B of [34]. XML (Extensible Markup Language) is a flexible way to create common information formats. XML is a formal recommendation from the World Wide Web Consortium (W3C). XML is not a programming language, it is rather a set of rules that allow you to represent data in a structured manner. Since the rules are standard, the XML documents can be automatically generated and processed. Its use can be gauged from its name itself:

- Markup: Is a collection of Tags
- XML Tags: Identify the content of data
- Extensible: User-defined tags

The XML representation of the `Sum` network is found in Fig. 17. A graphical editing framework called Graphiti editor [32] is available to create, edit, save and display a network. The XML and textual format for the network description are supported by such an editor.

```
<?xml version="1.0" encoding="UTF-8"?>
<XDF name="Sum">
   <Port kind="Input" name="In"/>
   <Port kind="Output" name="Out"/>
   <Instance id="add"/>
   <Instance id="z">
      <Class name="Z"/>
      <Parameter name="v">
         <Expr kind="Literal"
               literal-kind="Integer" value="0"/>
      </Parameter>
   </Instance>
   <Connection dst="add" dst-port="A"
         src="" src-port="In"/>
   <Connection dst="add" dst-port="B"
         src="z" src-port="Out"/>
   <Connection dst="z" dst-port="In"
         src="add" src-port="Out"/>
   <Connection dst="" dst-port="Out"
         src="add" src-port="Out"/>
</XDF>
```

Fig. 17 XML representation of the Sum network

4.4 Bitstream Syntax Specification Language BSDL

MPEG-B Part 5 is an ISO/IEC international standard that specifies BSDL [33] (Bitstream Syntax Description Language), an XML dialect describing generic bitstream syntaxes. In the field of video coding, the bitstream description in BSDL of MPEG-4 AVC [69] bitstreams represents all the possible structures of the bitstream which conforms to MPEG-4 AVC. A Binary Syntax Description (BSD) is one unique instance of the BSDL description. It represents a single MPEG-4 AVC encoded bitstream: it is no longer a BSDL schema but a XML file showing the data of the bitstream. Figure 18 shows a BSD associated to its corresponding BSDL schema.

An encoded video bitstream is described as a sequence of binary elements of syntax of different lengths: some elements contain a single bit, while others contain many bits. The Bitstream Schema (in BSDL) indicates the length of these binary elements in a human- and machine-readable format (hexadecimal, integers, strings...). For example, hexadecimal values are used for start codes as shown in Fig. 18. The XML formalism allows organizing the description of the bitstream in a hierarchical structure. The Bitstream Schema (in BSDL) can be specified at different levels of granularity. It can be fully customized to the application requirements [67]. BSDL was originally conceived and designed to enable adaptation of scalable multimedia contents in a format-independent manner [68]. In the RVC framework, BSDL is used to fully describe video bitstreams. Thus, BSDL schemas must specify all the elements of syntax, i.e. at a low level of granularity. Before the use of BSDL in RVC, the existing BSDL descriptions described scalable contents at a high level of granularity. Figure 18 is an example BSDL description for video in MPEG-4 AVC format.

```
<NALUnit>
  <startCode>00000001</startCode>
  <forbidden0bit>0</forbidden0bit>
  <nalReference>3</nalReference>
  <nalUnitType>20</nalUnitType>
  <payload>5 100</payload>
</NALUnit>
<NALUnit>
<startCode>00000001</startCode>
<!-- and so on... -->
</NALUnit>
```

```
<element name="NALUnit"
  bs2:ifNext="00000001">
<xsd:sequence>
  <xsd:element name="startCode"
               type="avc:hex4" fixed="00000001"/>
  <xsd:element name="nalUnit"
               type="avc:NALUnitType"/>
  <xsd:element ref="payload"/>
</xsd:sequence>
<!-- Type of NALUnitType -->
<xsd:complexType name="NALUnitType">
  <xsd:sequence>
    <xsd:element name="forbidden_zero_bit"
                 type="bs1:b1" fixed="0"/>
    <xsd:element name="nal_ref_idc" type="bs1:b2"/>
    <xsd:element name="nal_unit_type" type="bs1:b5"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="payload" type="bs1:byteRange"/>
```

Fig. 18 A Bitstream Syntax Description (BSD) fragment of an MPEG-4 AVC bitstream and its corresponding BS schema fragment codec in RVC-BSDL

In the RVC framework, BSDL has been chosen because:

- it is stable and already defined by an international standard;
- the XML-based syntax interacts well with the XML-based representation of the configuration of RVC decoders;
- the parser may be easily generated from the BSDL schema by using standard tools (e.g. XSLT);
- the XML-based syntax integrates well with the XML infrastructure of the existing tools.

4.5 Instantiation of the ADM

In the RVC framework, the decoding platform acquires the Decoder Description that fully specifies the architecture of the decoder and the structure of the incoming bitstream. So as to instantiate the corresponding decoder implementation, the platform uses a library of building blocks specified by MPEG-C. Conceptually, such a library is a user defined proprietary implementation of the MPEG RVC standard library, providing the same I/O behavior. Such a library can be expressly developed to explicitly expose an additional level of concurrency and parallelism appropriate for implementing a new decoder configuration on user specific multi-core target platforms. The dataflow form of the standard RVC specification, with the associated Model of Computation, guarantee that any reconfiguration of the user defined proprietary library, developed at whatever lower level of granularity, provides an implementation that is consistent with the (abstract) RVC decoder model that is originally specified using the standard library. Figures 2 and 4 show how a decoding solution is built from, not only the standard specification of the codecs in RVC-CAL by using the normative VTL, and this already provides an explicit, concurrent and parallel model, but also from any non-normative "multi-core-friendly" proprietary Video Tool Libraries, that increases if necessary the level of explicit concurrency and parallelism for specific target platforms. Thus, the standard RVC specification, which is already an explicit model for concurrent systems, can be further improved or specialized by proprietary libraries that can be used in the instantiation phase of an RVC codec implementation.

4.6 Case Study of New and Existing Codec Configurations

4.6.1 Commonalities

All existing MPEG codecs are based on the same structure, the hybrid decoding structure including a parser that extracts values for texture reconstruction and motion compensation [19]. Therefore, MPEG-4 SP and MPEG-4 AVC are hybrid decoders. Figure 19 shows the main functional blocks composing an hybrid decoder structure.

As said earlier, an RVC decoder is described as a block diagram with FNL [34], an XML dialect that describes the structural network of interconnected actors from the Standard MPEG Toolbox. The only 2 case studies performed so far by MPEG RVC experts [42, 66] are the RVC-CAL specifications of MPEG-4 Simple Profile decoder and MPEG-4 AVC decoder [27].

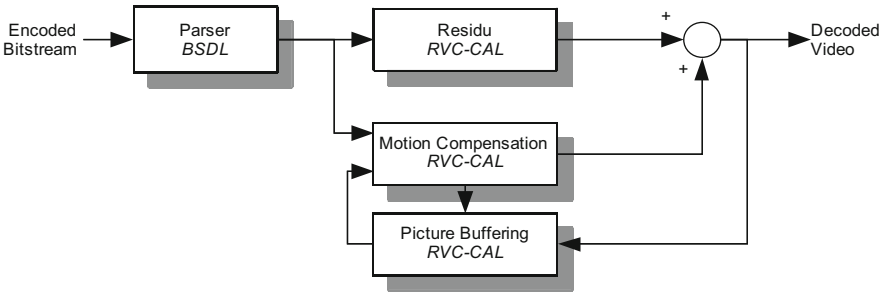


Fig. 19 Hybrid decoder structure

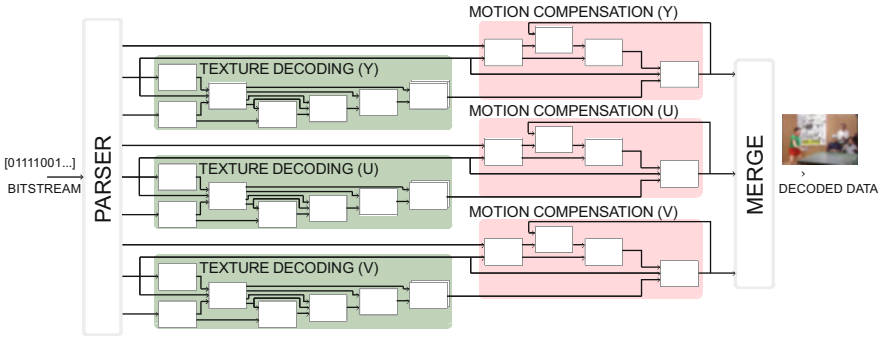


Fig. 20 MPEG-4 Simple Profile decoder description

4.6.2 MPEG-4 Simple Profile (SP) Decoder

Figure 20 shows the network representation of the macroblock-based MPEG-4 Simple Profile decoder description. The parser is a hierarchical network of actors (each of them is described in a separate FNL file). All other blocks are atomic actors programmed in RVC-CAL. Figure 20 presents the structure of the MPEG-4 Simple Profile ADM as described within RVC. Essentially it is composed of four main parts: the parser, a luminance component (Y) processing path, and two chrominance component (U, V) processing paths. Each of the paths is composed by its texture decoding engine as well as its motion compensation engine (both are hierarchical RVC-CAL Functional Units).

The MPEG-4 Simple Profile abstract decoder model that essentially results to be a dataflow program (Fig. 20, Table 3), is composed of 27 atomic FUs (or actors in dataflow programming) and 9 sub-networks (actor/network composition); atomic actors can be instantiated several times, for instance there are 42 actor instantiations in this dataflow program. Figure 25 shows a top-level view of the decoder. The main functional blocks include the bitstream parser, the reconstruction block, the 2D inverse cosine transform, the frame buffer and the motion compensation module. These functional units are themselves hierarchical compositions of actor networks.

4.6.3 MPEG-4 AVC Decoder

MPEG-4 Advanced Video Coding (AVC), or also know as H.264 [69], is a state-of-the-art video compression standard. Compared to previous coding standards, it is able to deliver higher video quality for a given compression ratio, and 30% better compression ratio compared to MPEG-4 SP for the same video quality. Because of its complexity, many applications including Blu-ray, iPod video, HDTV broadcasts, and various computer applications use variations of MPEG-4 AVC codec (also called *profiles*). A popular uses of MPEG-4 AVC is the encoding of high definition video contents. Due to high resolutions processing required, HD video is the application that requires the highest performance for decoding. Common formats used for HD include 720p (1280×720) and 1080p (1920×1080) resolutions, with frame rates between 24 and 60 frames per second.

The decoder introduced in this section corresponds to the *Constrained Baseline Profile* (*CBP*). This profile is primarily fitted to lowest-cost applications and corresponds to a subset of features that are in common between the *Baseline*, *Main*, and *High Profiles*.

The description of this decoder expresses the maximum of parallelism and mimics the MPEG4 SP. This description is composed of different hierarchical level. Figure 21 shows a view of the highest hierarchy of the MPEG-4 AVC decoder—note that for readability, one input represents a group of input for similar information on each actor. The main functional block includes a parser, one *luma* and *two chroma* decoders.

The parser analyses the syntax of the bitstream with a given formal grammar. This grammar, written by hand, will later be given to the parser by a BSDL [64] description. As the execution of a parser strongly depends on the context of the bitstream, the parser incorporates a Finite State Machine so that it can sequentially extract the information from the bitstream. This information passes through an

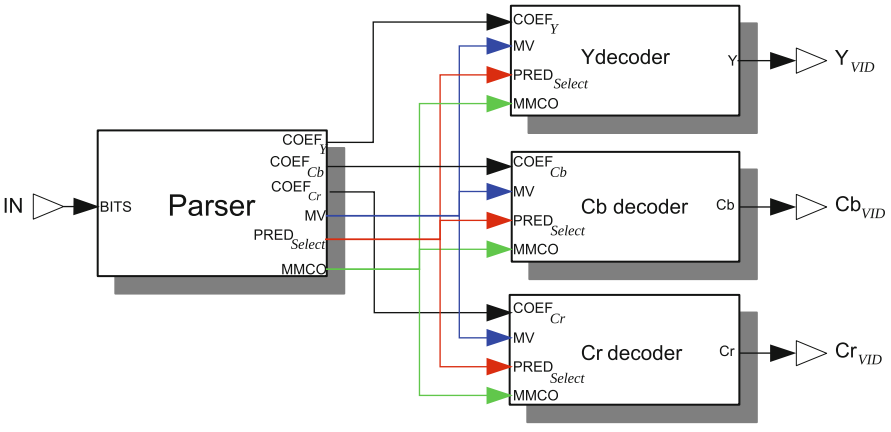


Fig. 21 Top view of MPEG-4 Advanced Video Coding decoder description

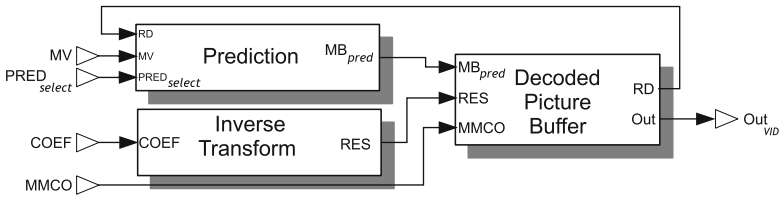


Fig. 22 Structure of decoding actors

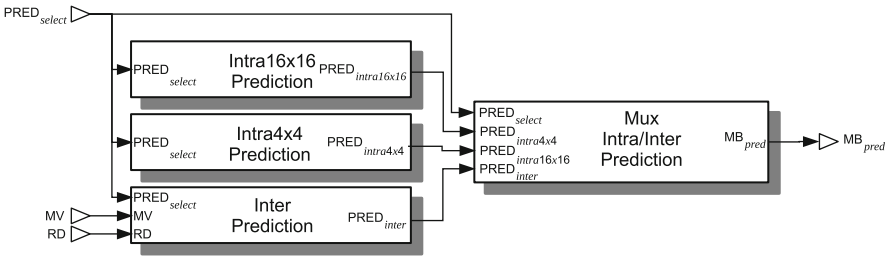


Fig. 23 Structure of prediction actor

entropy decoder and is then encapsulated in several kinds of tokens (residual coefficients, motion vectors...). These tokens are finally sent to the selected input port of the *luma/chroma* decoding actor.

Because decoding a *luma/chroma* component does not need to share information with the other *luma/chroma* component, we choose to encapsulate each *luma/chroma* decoding in a single actor. This means that each decoding actor can run independently and at the same time in a separate thread. The entire decoding component actor has the same structure.

Luma/chroma decoding actors (Fig. 22) decode a picture and store the decoded picture for later use in inter-prediction process. Each component owns the memory needed to store pictures, encapsulates into the *Decoded Picture Buffer* (*DPB*) actor. The *DPB* actor also contains the *Deblocking Filter* and is a buffering solution to regulate and reorganize the resulting video flow according to the *Memory Management Control Operations* (*MMCO*) input.

The *Decoded Picture Buffer* creates each frame by adding prediction data, provided by the actor *prediction*, and residual data, provided by the actor *Inverse Transform*. The *Prediction* actor (Fig. 23) encompasses *inter/intra prediction* modes and a multiplexer that sends prediction results to the output port. The $PRED_{select}$ input port has the role to stoke the right actors contingent on a prediction mode. The target of this structure is to offer a quasi-static work of the global actor and, by adding or removing prediction modes, to easily switch between configurations of the decoder. For instance, adding *B inter-prediction mode* into this structure switches the decoder into the *main profile* configuration.

5 Tools and Integrated Environments Supporting Development, Analysis and Synthesis of Implementations

Although some years have already passed since the first components of RVC have been developed, there is still the room of extending the RVC framework and for improving the performance and functionality of the non-normative tools and integrated environments supporting simulation, analysis and direct implementation synthesis. Indeed, besides the goal of providing a unified and high level specification formalism, an innovative objective of RVC is to narrow the gap between the algorithmic specification and the generation of the corresponding implementations. Such gap not only constitutes a serious impediment for the efficient development of implementations, but the augmented complexity of the new generation of video codecs, and the increasing heterogeneity of processing platforms, that may include many-core, multi-core and GPUs, make the gap wider. The fact that an RVC specification does not imply a specific processing architecture (the single processor), but abstracts from it, and results to be portable on any combination of architectures, is a very attractive feature that opens the path to the usage of different tools and integrated design flows. All of them attempt to ease the development cycles by implementing:

- Assisted writing of the dataflow program: by the support of fully integrated development environments including design exploration capabilities.
- Systematic validation of the dataflow program: by verification of integrated simulators.
- Develop and optimize once, but run everywhere: by generating hardware and/or software implementations that can be executed on a large panel of platforms by means of transcompilation using the appropriate back-ends.

This section briefly describes some of the numerous tools appeared and still under development to improve performance and functionality, that support the different stages of design flows of an RVC data-flow specification. More examples and tutorials for the installation and usage of some of the tools and integrated environments described below are available in a separate technical report which constitute a non-normative part of the RVC standard [36].

5.1 OpenDF Framework

CAL is supported by a portable interpreter infrastructure that can simulate a hierarchical network of actors. This interpreter was first used in the Moses [54] project. Moses features a graphical network editor, and allows the user to monitor actors execution (actor state and token values). The project being no longer maintained, it has been superseded by an Eclipse environment composed of two tools/plugins—the Open Dataflow environment for CAL editing (OpenDF [56] for short) and the Graphiti editor for graphically editing the network.

One interesting and very attracting implementation methodology of MPEG RVC decoder descriptions is the direct synthesis of the standard specification. OpenDF is also a compilation framework. It provides a source of relevant application of realistic sizes and complexity and also enables meaningful experiments and advances in dataflow programming. More details on the software and hardware code generators can be found in [41, 70]. Today there exists a backend for generation of HDL (VHDL/Verilog) [41, 42]. A second backend targeting ARM11 and embedded C is under development [57] as part of the EU project ACTORS [2]. It is also possible to simulate CAL models in the Ptolemy II [59] environment.

5.2 *Orcc Framework*

Works made on action synthesis and actor synthesis [66, 70] led to the creation of a compiler framework called Open RVC CAL Compiler (Orcc) [55]. This framework is designed to support multiple language front-ends, each of which translates actors written in RVC-CAL and FNL network into an Intermediate Representation (IR), and to support multiple language back-ends, each of which translates the Intermediate Representation into the supported languages. IR provides a dataflow representation that can be easily transformed in low level languages. Currently the only maintained back-end is a C language backend (Fig. 24).

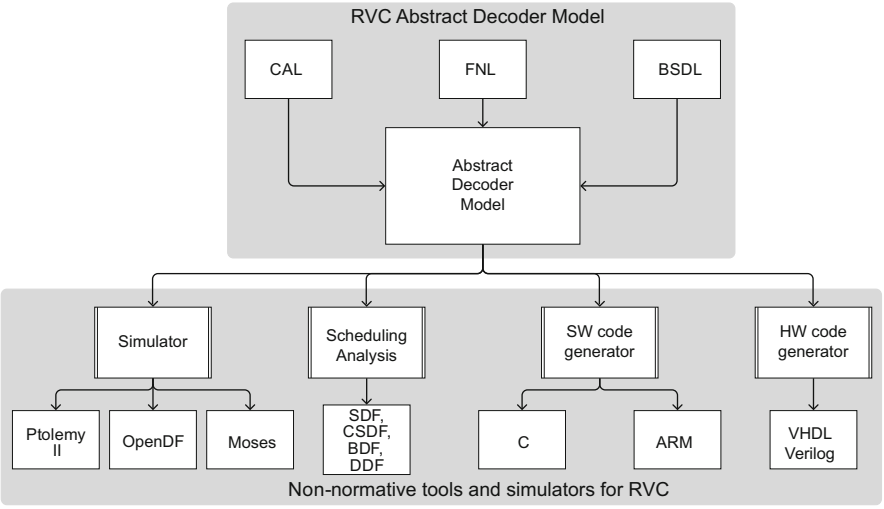


Fig. 24 OpenDF: tools

Table 1 Hardware synthesis results for a proprietary implementation of a MPEG-4 Simple Profile decoder

	Size slices, BRAM	Speed (kMB/s)	Code size (kSLOC)	Dev. time MM
CAL	3872, 22	290	4	3
VHDL	4637, 26	180	15	12
Improv. factor	1.2	1.6	3.75	4

The numbers are compared with a reference hand written design in VHDL
kMB/s kilo macroblocks per second, *kSLOC* kilo source lines of code

5.3 CAL2HDL Synthesis

Some of the authors have performed an implementation study [41], in which the RVC MPEG-4 Simple Profile decoder specified in CAL according to the MPEG RVC formalism has been implemented on an FPGA using a CAL-to-RTL code generator called Cal2HDL. The objective of the design was to support 30 frames of 1080p in the YUV420 format per second, which amounts to a production of 93.3 MB of video output per second. The given target clock rate of 120 MHz implies 1.29 cycles of processing per output sample on average.

The results of the implementation study were encouraging in that the code generated from the MPEG RVC CAL specification did not only outperform the handwritten reference in VHDL, both in terms of throughput and silicon area, but also allowed for a significantly reduced development effort. Table 1 shows the comparison between CAL specification and the VHDL reference implemented over a Xilinx Virtex 2 pro FPGA running at 100 MHz.

It should be emphasized that this counter-intuitive result cannot be attributed to the sophistication of the synthesis tool. On the contrary the tool does not perform a number of potential optimizations, such as for instance optimizations involving more than one actor. Instead, the good results appear to be yield by the implementation and development process itself. The implementation approach was based generating a proprietary implementation of the standard MPEG RVC toolbox composed of FUs of lower level of granularity. Thus the implementation methodology was to substitute the FU of the standard abstract decoder model of the MPEG-4 SP with an equivalent implementation, in terms of behavior. Essentially standard toolbox FUs were substituted with networks of FU described as actors of lower granularity (Fig. 25) [28–30, 46].

The initial design cycle of the proprietary RVC library resulted in an implementation that was not only inferior to the VHDL reference, but one that also failed to meet the throughput and area constraints. Subsequent iterations explored several other points in the design space until arriving at a solution that satisfied the constraints. At least for the considered implementation study, the benefit of short design cycles

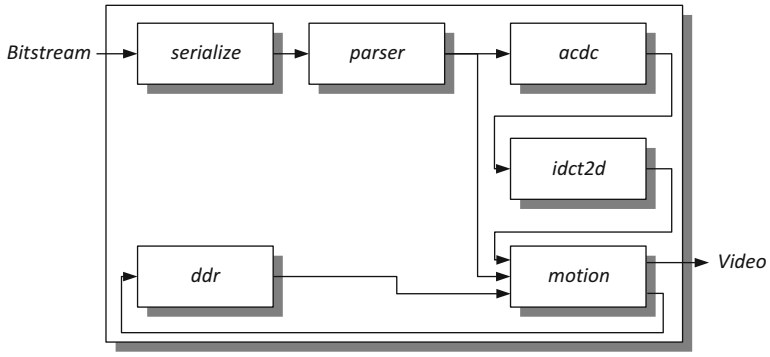


Fig. 25 Top-level dataflow graph of the proprietary implementation of the RVC MPEG-4 decoder

seem to outweigh the inefficiencies that resulted from high-level synthesis and the reduced control over implementation details.

In particular, the asynchrony of the programming model and its realization in hardware allowed for convenient experiments with design ideas. Local changes, involving only one or a few actors, do not break the rest of the system in spite of a significantly modified temporal behavior. In contrast, any design methodology that relies on precise specification of timing—such as RTL, where designers specify behavior cycle-by-cycle—would have resulted in changes that propagate through the design.

Table 1 shows the quality of result produced by the RTL synthesis engine of the MPEG-4 Simple Profile video decoder. Note that the code generated from the high-level dataflow RVC description and proprietary implementation of the MPEG toolbox actually outperforms the hand-written VHDL design in terms of both throughput and silicon area for a FPGA implementation.

5.4 CAL2C Synthesis

Another synthesis tool called Cal2C [66, 70] currently available at [55] validates another implementation methodology of the MPEG-4 Simple Profile dataflow program provided by the RVC standard (Fig. 20). The SW code generator presented in details in [66] uses process network model of computation [44] to implement the CAL dataflow model. The compiler creates a multi-thread program from the given dataflow model, where each actor is translated into a thread and the connectivity between actors is implemented via software FIFOs. Although the generation provides correct SW implementations, inherent context switches occur during execution, due to the concurrent execution of threads, which may lead to inefficient SW execution if the granularity of actor is too fine.

Table 2 MPEG-4 Simple Profile decoder speed and SLOC

MPEG4 SP decoder	Speed (kMB/s)	Clock speed (GHz)	Code size (kSLOC)
CAL simulator	0.015	2.5	3.4
Cal2C	8	2.5	10.4
Cal2HDL	290	0.12	4

Table 3 Code size and number of files automatically generated for MPEG-4 Simple Profile decoder

MPEG-4 SP decoder	CAL	C actors	C scheduler
Number of files	27	61	1
Code size (kSLOC)	2.9	19	2

Major problems with multi-threaded programs are discussed in [48]. A more appropriate solution that avoids thread management are presented in [49, 58]. Instead of suspending and resuming threads based on the blocking read semantic of process network [45], actors are, instead, managed by a user-level scheduler that select the sequence of actor firing. The scheduler checks, before executing an actor, if it can fire, depending on the availability of tokens on inputs and the availability of rooms on outputs. If the actor can fire, it is executed (these two steps refers to the *enabling function* and the *invoking function* of [58]). If the actor cannot fire, the scheduler simply tests the next actor to fire (sorted following an appropriate given strategy) and so on. This code generator based on this concept [70] is available at [55]. Such a compiler presents a scheduler that has the two following characteristics: (1) actor firings are checked at run-time (the dataflow model is not scheduled statically), (2) the scheduler executes actors following a round-robin strategy (actors are sorted a priori).

In the case of the standard RVC MPEG-4 SP dataflow model such a generated mono-thread implementation is about four times faster than the one obtainable by [66]. Table 2 shows that synthesized C-software is faster than the simulated CAL dataflow program (80 frames/s instead of 0.15 frames/s), and twice the real-time decoding for a QCIF format (25 frames/s). However it remains slower than the automatically synthesized hardware description by Cal2HDL [41].

As described above, the MPEG-4 Simple Profile dataflow program is composed of 61 actor instantiations in the flattened dataflow program. The flattened network becomes a C file that currently contains a round robin scheduler for the actor scheduling and FIFOs connections between actors. Each actor becomes a C file containing all its action/processing with its overall action scheduling/control. Its number of SLOC is shown in Table 3. All of the generated files are successfully compiled by gcc. For instance, the "ParserHeader" actor inside the "Parser" network is the most complex actor with multiple actions. The translated C-file (with actions and state variables) includes 2062 SLOC for both actions and action scheduling. The original CAL file contains 962 lines of codes as a comparison.

A comparison of the CAL description (Table 4) shows that the MPEG-4 AVC CAL decoder is twice more complex in RVC-CAL than the MPEG-4 Simple Profile CAL description. Some parts of the model have already been redesign in order to improve pipelining and parallelism between actors. A simulation of the MPEG-4

Table 4 Code size and number of files automatically generated for MPEG-4 AVC decoder

MPEG-4 AVC decoder	CAL	C actors	C scheduler
Number of files	43	83	1
Code size (kSLOC)	5.8	44	0.9

AVC CAL model on a *Intel Core 2 Duo* @ 2.5 GHz is more than 2.5 slower than the RVC MPEG-4 Simple Profile description.

Comparing to the MPEG-4 Simple Profile CAL model, the MPEG-4 AVC decoder has been modeled to use more CAL possibility (for instance processing of several tokens in one firing) while staying fully RVC conformant. Thanks to this increasing complexity, MPEG-4 AVC CAL model is the most reliable way to test the accordance and the efficiency of the current RVC tools. The current SW code generation of MPEG-4 AVC is promising since it can achieve up to 53 fps and can be further partitioned over more processors for the instantiation of parallel implementations.

5.5 Integrated Design Flows Including Design Exploration and Full SW/HW Synthesis Capabilities

Orcc is also available as an Eclipse-based Integrated Development Environment (IDE) integrated with several other tools providing design exploration capabilities and extended synthesis functionality for SW and HW component, including the synthesis support of the SW/HW interconnections for some heterogeneous platforms. The environment is composed of two editors dedicated to handle both actor programming and network designs. A graphical editor enables the building of the actor network using visual programming graphical primitives. The editor also supports hierarchical representations, assigning whole subnetworks to graph nodes, and enabling hierarchical navigation. When the dataflow network is built, a full-blown RVC-CAL editor with advanced features, syntax coloring, content assist and code validation, supports the development of the actors. The development environment is able to parse the actors and build the intermediate representation on-the-fly, in a incremental fashion, allowing fast simulation and compilation. In addition to the editors functionality, Orcc provides a complete Java-based simulator which enable the test and validation of the dataflow program without taking in consideration low-level details relative to the target platform, but focusing only the correctness of the algorithm specification. The simulator does not simply interpret the intermediate representation of networks and actors, but it also performs all interactions required to perform a full functional validation, such as displaying text, images or videos to the screen. Orcc includes back-ends that generates C/C++ programs supporting many and multi-core processor platforms. The Orcc compilation framework is also completed by several other tools for performance analysis, design space exploration and HW generation and optimization constituting

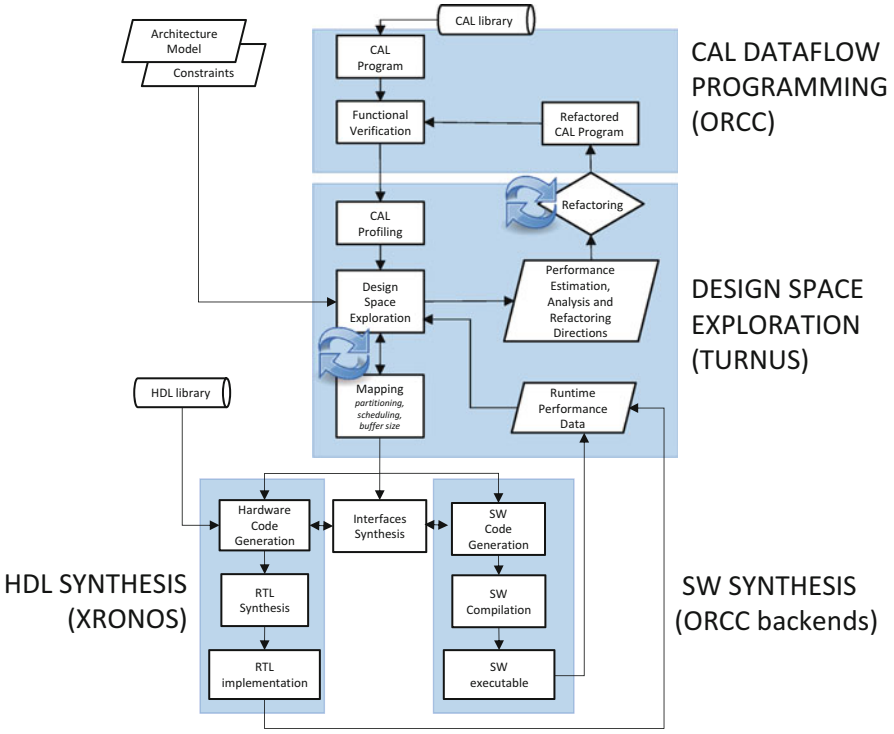


Fig. 26 Illustration of the design flow supporting, development, design exploration and synthesis of system implementations on heterogeneous platforms of RVC specifications, including the supporting tools and the associated dependencies

a complete system design environment for heterogeneous systems. A graphic representation of the system design flow is provided in Fig. 26. In the picture the functionality of the design flow are labelled with their dependencies and mapped into the corresponding tool environment. Whereas Orcc provides dataflow program development functionalities and simulation capabilities (top section of the design flow) and SW generation (right bottom part of the flow) Turnus provides a design space environment integrated as Plug-in of the Orcc Eclipse environment and Xronos an HDL synthesis tool (left bottom part of the design flow). Both Turnus and Xronos of are available as open source tools at [60, 61].

5.5.1 Turnus Design Exploration Framework

The first step of the design space exploration provided by TURNUS is a functional high-level and platform-independent profiled simulation [14, 62]. During this stage, an analysis of the design under study can be performed leading to the identification of the processing structure and associated complexity. This initial analysis

is useful for finding complexity bottlenecks and identify potential parallelism. Two approaches to profiling are supported by TURNUS. An abstract profiling of operators is provided by adding profiling information on top of the Orcc simulator: for each executed action both (a) the computational load and (b) the data-transfers and storage load are evaluated, thus the computational load is measured in terms of executed operators and control statements (i.e. comparison, logical, arithmetic and data movement instructions). The data-transfers and storage load are evaluated in terms of state variables utilization, input/output port utilization, buffers utilization and tokens production/consumption. A second profiling approach is based on extracting the causation trace of a run of the simulation corresponding to a given input data vector. Then the causation trace is annotated by adding the profiling information corresponding to each action execution and data token exchange obtained by a single execution on a specific platform. By analyzing the annotated causation trace is then possible to efficiently explore the design space in terms of looking for close-to-optimal partitioning configurations, buffer dimension specifications and scheduling strategies. More details of the methodologies supported by TURNUS framework for jointly exploring the partitioning, buffer dimensioning and scheduling configurations can be found in [17, 52, 71, 72]. In these work it is shown how important is a joint exploration of the design space for maximizing the performance of the RVC HEVC decoder. Close-to-optimal results are systematically obtained by the exploration tools supported by TURNUS for several different implementation configurations on many-core platforms.

5.5.2 Xronos System Design Synthesis Framework

Xronos although based on the XLIM backend of the Orcc compiler is a complete new framework for generating RTL descriptions from RVC-CAL dataflow programs. Xronos is based on two tools: the Orcc compiler used as front-end and the OpenForge synthesizer which constitute an integral part of Xronos. Orcc parses the RVC-CAL actors and generates an intermediate representation, then the IR is serialized to an actor object that contains all the information originally present in the RVC-CAL file. Then a set of interfaces can generate different LIM objects which are transformed by the following set of transformation:

- **Read/Store Once**: the number of load and stores is minimized, so that a read and store operation can be done at best only once in a block of sequential instructions.
- **Function Inliner**: all functions are automatically inlined.
- **SSA**: a single static assignment is provided to each variable.
- **3AC**: each operation is transformed into a 4-tuple of (operator, operand1, operand2, result).
- **Cast Adder**: the necessary casting is provided to each operation.
- **Repeat Pattern**: the transformation supporting the CAL repeat statement is provided

- **Input/Output port Statement Finder**: the creation of the dataflow representation binding input and outputs of loop and branches statements.

Finally the final design object is generated by allocating the necessary memory and creating a slave LIM task visiting all actions of the actor, and by generating the master task, the scheduler of the actors, and all the actions firing rules and the actors finite state machine if actors have any. Relying on the Orcc intermediate representation and associated compiler, it is also possible to generate C code, thus it is possible to simulate and debug the RVC-CAL dataflow program by saving all tokens that are consumed and produced by each actor. Thus, Xronos for each synthesized actor generates a RTL testbench that takes as inputs the token traces, and if a difference is found on a synthesized actor output the framework stops the behavioural RTL simulation and indicates to the designer where an error has occurred. More details and functionality of the synthesis framework can be found in [1, 4, 18, 63, 65].

5.6 *The* TÝCHO *Framework*

A more recent tool infrastructure supporting CAL and RVC-CAL is the TÝCHO framework [16].[2] Its distinguishing characteristic is that it is built on *actor machines* [39, 40], an abstract machine model for representing and manipulating actors which serves as the internal representation for actors in TÝCHO. As a consequence, TÝCHO can support different input formats (currently CAL, RVC-CAL and the process extension discussed in Sect. 4.2.4), which can be freely mixed and matched within a dataflow program. Optimizations, transformations, and code generation operate exclusively on the internal representation and thus work equally regardless of the particular input language.

Among the optimizations relevant to software synthesis TÝCHO supports a family of *reductions*, which transform non-deterministic actor machines into deterministic and sequential ones by scheduling the logical steps required to execute a single actor at compile time, which can be seen as a first step toward code generation. It also includes *composition*, the integration of several (usually connected) actors into a single actor, often involving compile-time scheduling of the concurrent activities among them based on their data dependencies. Composition is fully general and makes no assumptions regarding the nature of the composed actors, although it produces best results when the data dependencies between them are very regular, in the limit leading to a fully static schedule of the composed actors. TÝCHO's composition represents a generalization of previous efforts at *actor merging* or static scheduling (e.g. in [10–13, 24–26, 31, 47]), which only apply to a limited class of dataflow actors.

[2] http://tycho.cs.lth.se.

6 Conclusion

This chapter describes the essential components of the ISO/IEC MPEG Reconfigurable Video Coding framework based on the dataflow concept. The RVC MPEG tool library, that covers in modular form video compression and 3-D graphics compression algorithms from the different MPEG coding standards, shows that dataflow programming is an appropriate way to build complex heterogeneous systems from high level system specifications. The MPEG RVC framework is also supported by simulators, software and hardware code synthesis tools and full integrated frameworks including full systems synthesis and design exploration capabilities. CAL dataflow models used by the MPEG RVC standard result also particularly efficient for specifying many classes of signal processing systems in a very synthetic form compared to classical imperative languages. Moreover, CAL model libraries can be developed in the form of libraries of proprietary implementations of standard RVC components to describe architectural features of the desired implementation platform, thus enabling the RVC implementer/designer to work at level of abstraction comparable to the one of the RVC video coding algorithms. Hardware and software code generators then provide the low level system implementation of the actors and associated network of actors for different and possibly heterogeneous target implementation platforms including multi-core and many-core processors and programmable hardware (FPGA).

References

1. A. Ab Rahman, A. Prihozhy, M. Mattavelli: Pipeline Synthesis and Optimization of FPGA-based Video Processing Applications with CAL, Eurasip Journal on Image and Video Processing, 2011, 2011:19, http://jivp.eurasipjournals.com/content/2011/1/19.
2. Actors FP7 project: http://www.actors-project.eu
3. V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. 2000. Clock rate versus IPC: the end of the road for conventional microarchitectures. SIGARCH Comput. Archit. News 28, 2 (May 2000), 248–259.
4. E. Bezati, R. Thavot, G. Roquier, M. Mattavelli: High-Level Data Flow Design of Signal Processing Systems for reconfigurable and multi-core heterogeneous platforms, Journal of Real Time Image Processing, 2012.
5. S.S. Bhattacharyya, G. Brebner, J.W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet: OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems. SIGARCH Comput. Archit. News **36**(5), 29–35 (2008). https://doi.org/10.1145/1556444.1556449
6. S.S. Bhattacharyya, J. Eker, J.W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet: Overview of the MPEG reconfigurable video coding framework. Journal of Signal Processing Systems (2011). https://doi.org/10.1007/s11265-009-0399-3
7. C. Tulvan and M. Preda: 3D Graphics Coding in a Reconfigurable Environment. Image Communications (2013). https://doi.org/10.1016/j.image.2013.08.010

8. J.S. Euee, M. Mattavelli, M. Preda, M. Raulet and H. Sun: Overview of the MPEG reconfigurable video coding framework. Image Communications (2013). https://doi.org/10.1016/j.image.2013.08.008
9. B. Bhattacharya and S.S. Bhattacharyya, "Parameterized Dataflow Modeling for DSP Systems," *IEEE Transactions on Signal Processing*, vol. 49, pp. 2408–2421, 2001.
10. G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on signal processing*, vol. 44, no. 2, pp. 397–408, 1996.
11. J. Boutellier, C. Lucarz, S. Lafond, V.M. Gomez, and M. Mattavelli, "Quasi-static scheduling of CAL actor networks for reconfigurable video coding," *Journal of Signal Processing Systems*, pp. 1–12, 2009.
12. J. Boutellier, J. Ersfolk, J. Lilius, M. Mattavelli, G. Roquier, and O. Silven, "Actor merging for dataflow process networks," *IEEE Transactions on Signal Processing*, vol. 63, no. 10, pp. 2496–2508, 2015.
13. J. Boutellier, O. Silvén, and M. Raulet: "Automatic synthesis of TTA processor networks from RVC-CAL dataflow programs.," *Signal Processing Systems (SiPS), 2011 IEEE Workshop on, pp.25–30, 4–7 Oct. 2011.* https://doi.org/10.1109/SiPS.2011.6088944
14. S. Casale Brunet, E. Bezati, R. Thavot, G. Roquier, M. Mattavelli, J. W. Janneck, Methods to Explore Design Space for MPEG RVC Codec Specifications, in Signal Processing Image Communication, Special Issue on Reconfigurable Media Coding, 2013.
15. G. Cedersjö and J. W. Janneck. Processes and actors: Translating Kahn processes to dataflow with firing, in 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), pp. 21–30, IEEE, 2016.
16. G. Cedersjö, Efficient Software Implementation of Stream Programs, dissertation, LU-CS-DISS 2017–3, Department of Computer Science, Lund University, 2017
17. Simon Casale Brunet, Analysis and optimization of dynamic dataflow programs, Thèse École polytechnique fédérale de Lausanne EPFL, no. 6663 (2015). http://infoscience.epfl.ch/record/208775
18. Endri Bezati, High-level synthesis of dataflow programs for heterogeneous platforms: design flow tools and design space exploration, Thèse École polytechnique fédérale de Lausanne EPFL, no. 6653 (2015). http://infoscience.epfl.ch/record/207992
19. Y. Chen and L. Chen. Video Compression. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*. Springer, second edition, 2012.
20. L. Chiariglione Editor: The MPEG Representation of Digital Media. Springer Ed. 2011. http://dx.doi.org/10.1007/978-1-4419-6184-6_12
21. D. Ding, L. Yu, C. Lucarz, and M. Mattavelli: Video decoder reconfigurations and AVS extensions in the new MPEG reconfigurable video coding framework. In: Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on, pp. 164–169 (2008). https://doi.org/10.1109/SIPS.2008.4671756
22. S. A. Edwards. 2003, *Tutorial: Compiling concurrent languages for sequential processors*, ACM Trans. Des. Autom. Electron. Syst. 8, 2 (April 2003), 141–187.
23. J. Eker and J.W. Janneck: CAL Language Report Specification of the CAL Actor Language. Tech. Rep. UCB/ERL M03/48, EECS Department, University of California, Berkeley (2003)
24. J. Ersfolk, G. Roquier, J. Lilius, M. Mattavelli Scheduling of dynamic data flow programs based on state space analysis, 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2012, pp. 1661–1664. https://doi.org/10.1109/ICASSP.2012.6288215.
25. J. Ersfolk, G. Roquier, F. Jokhio, J. Lilius, M. Mattavelli, Scheduling of dynamic data flow programs with model checking,, 2011 IEEE Workshop on Signal Processing Systems (SiPS), 2011, pp. 37–42. https://doi.org/10.1109/SiPS.2011.6088946.
26. O. Esko, P. Jääskeläinen, P. Huerta, C. S. de La Lama, J. Takala, and J. I. Martinez : "Customized exposed datapath soft-core design flow with compiler support," IEEE conference on Field Programmable Logic and Applications (FPL), 2010, pp. 217–222.

27. J. Gorin, M. Raulet, Y.L. Cheng, H.Y. Lin, N. Siret, K. Sugimoto, and G. Lee: An RVC dataflow description of the AVC Constrained Baseline Profile decoder. In: IEEE International Conference on Image Processing, Special Session on Reconfigurable Video Coding. Cairo, Egypt (2009)
28. J. Gorin, M. Wipliez, J. Piat, M. Raulet, and F. Preteux. A portable Video Tools Library for MPEG Reconfigurable Video Coding using LLVM representation. In *Design and Architectures for Signal and Image Processing (DASIP 2010)*, pages 281–286, 2008.
29. J. Gorin, M. Wipliez, F. Preteux, and M. Raulet. LLVM-based and scalable MPEG-RVC decoder. *Journal of Real-Time Image Processing*, pages 1–12.
30. J. Gorin, M. Wipliez, M. Raulet, and F. Preteux. An LLVM-based decoder for MPEG Reconfigurable Video Coding. In *IEEE Workshop on Signal Processing Systems (SiPS 2010), Washington, D.C., USA*, pages 281–286, 2008.
31. R. Gu, J.W. Janneck, S.S. Bhattacharyya, M. Raulet, M. Wipliez, and W. Plishker, "Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 11, 2009.
32. Graphiti Editor sourceforge: http://graphiti-editor.sf.net
33. International Standard ISO/IEC FDIS 23001-5: MPEG systems technologies - Part 5: Bitstream Syntax Description Language (BSDL)
34. ISO/IEC International Standard 23001-4: MPEG systems technologies – Part 4: Codec Configuration Representation (2011)
35. ISO/IEC International Standard 23002-4: MPEG video technologies – Part 4: Video tool library (2010)
36. ISO/IEC International Standard 23002-6: MPEG systems technologies – Part 6: Tools for reconfigurable media coding implementations (2017)
37. ISO/IEC International Standard 23001-4: MPEG systems technologies – Part 4: Codec Configuration Representation (2017)
38. E.S. Jang, J. Ohm, and M. Mattavelli: Whitepaper on Reconfigurable Video Coding (RVC). In: ISO/IEC JTC1/SC29/WG11 document N9586. Antalya, Turkey (2008). http://www.chiariglione.org/mpeg/technologies/mpb-rvc/index.h%tm
39. J. W. Janneck: Actor machines - a machine model for dataflow actors and its applications, Department of Computer Science, Lund University, Tech. Rep. LTH 96-2011, LU-CS-TR 201–247, (2011).
40. J. W. Janneck: A Machine Model for Dataflow Actors and its Applications 45th Annual Asilomar Conference on Signals, Systems, and Computers November 6–9, 2011.
41. J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, and M. Raulet: Synthesizing hardware from dataflow programs. Journal of Signal Processing Systems (2011). http://dx.doi.org/10.1007/s11265-009-0397-5
42. J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, and M. Raulet: Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. In: Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on, pp. 287–292 (2008). https://doi.org/10.1109/SIPS.2008.4671777
43. J. W. Janneck, M. Mattavelli, M. Raulet, and M. Wipliez: Reconfigurable video coding: a stream programming approach to the specification of new video coding standards. in *MMSys '10: Proceedings of the first annual ACM SIGMM conference on Multimedia systems*. USA: ACM, 2010, pp. 223–234.
44. G. Kahn: The semantics of a simple language for parallel programming. In: J.L. Rosenfeld (ed.) Information processing, pp. 471–475. North Holland, Amsterdam, Stockholm, Sweden (1974)
45. G. Kahn, MacQueen, D.B.: Coroutines and networks of parallel processes. In: IFIP Congress, pp. 993–998 (1977)

46. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.
47. E.A. Lee and D.G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.
48. E.A. Lee: The problem with threads. IEEE Computer Society **39**(5), 33–42 (2006). http://doi.ieeecomputersociety.org/10.1109/MC.2006.180
49. E.A. Lee and T.M. Parks: Dataflow Process Networks. Proceedings of the IEEE **83**(5), 773–801 (1995)
50. C. Lucarz, I. Amer, and M. Mattavelli: Reconfigurable Video Coding: Concepts and Technologies. In: IEEE International Conference on Image Processing, Special Session on Reconfigurable Video Coding. Cairo, Egypt (2009)
51. M. Mattavelli, I. Amer, and M. Raulet, "The Reconfigurable Video Coding Standard [Standards in a Nutshell]," *Signal Processing Magazine, IEEE*, vol. 27, no. 3, pp. 159–167, May 2010.
52. Malgorzata Maria Michalska, Systematic Design Space Exploration of Dynamic Dataflow Programs for Multi-core Platforms, Thèse École polytechnique fédérale de Lausanne EPFL, no. 7607 (2017). http://infoscience.epfl.ch/record/226334
53. S. P. Midkiff, *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers 2012
54. Moses project: http://www.tik.ee.ethz.ch/moses/
55. The Open RVC CAL Compiler project sourceforge: http://orcc.sf.net
56. The OpenDF dataflow project sourceforge: http://opendf.sf.net
57. C. von Platen and J. Eker: Efficient realization of a cal video decoder on a mobile terminal (position paper). In: Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on, pp. 176–181 (2008). https://doi.org/10.1109/SIPS.2008.4671758
58. W. Plishker, N. Sane, M. Kiemb, K. Anand, and S.S. Bhattacharyya: Functional DIF for Rapid Prototyping. In: Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping - Volume 00, pp. 17–23. IEEE Computer Society (2008)
59. Ptolemy II: http://ptolemy.eecs.berkely.edu
60. TURNUS: http://github.com/turnus
61. XRONOS: http://github.com/orcc/xronos
62. J. Janneck, I.D. Miller, and D.B. Parlour: Profiling dataflow programs. in: Proceedings of the IEEE International Conference on Multimedia and Expo, 2008, pp. 1065–1068.
63. S. Casale-Brunet, M. Mattavelli, A. Elguindy, E. Bezati, R. Thavot, G. Roquier, and J. Janneck: Methods to explore design space for MPEG RMC codec specifications. In: Journal of Signal Processing Image Communication, Elsevier, (2013).
64. M. Raulet, J. Piat, C. Lucarz, and M. Mattavelli: Validation of bitstream syntax and synthesis of parsers in the MPEG Reconfigurable Video Coding framework. In: Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on, pp. 293–298 (2008). https://doi.org/10.1109/SIPS.2008.4671778
65. G. Roquier, E. Bezati and M. Mattavelli: Hardware and Software Synthesis of Heterogeneous Systems from Dataflow Programs, Journal of Electrical and Computer Engineering, special issue on "ESL Design Methodology", vol. 2012, Article ID 484962, 11 pages, 2012. doi: 10.1155/2012/484962.
66. G. Roquier, M. Wipliez, M. Raulet, J.W. Janneck, I.D. Miller, and D.B. Parlour: Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study. In: Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on, pp. 281–286 (2008). https://doi.org/10.1109/SIPS.2008.4671776
67. J. Thomas-Kerr, J.W. Janneck, M. Mattavelli, I. Burnett, and C. Ritz: Reconfigurable media coding: Self-Describing multimedia bitstreams. In: Signal Processing Systems, 2007 IEEE Workshop on, pp. 319–324 (2007). https://doi.org/10.1109/SIPS.2007.4387565

68. J.A. Thomas-Kerr, I. Burnett, C. Ritz, S. Devillers, D.D. Schrijver, and R. Walle: Is that a fish in your ear? a universal metalanguage for multimedia. Multimedia, IEEE **14**(2), 72–77 (2007). https://doi.org/10.1109/MMUL.2007.38
69. T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra: Overview of the H.264/AVC video coding standard. Circuits and Systems for Video Technology, IEEE Transactions on **13**(7), 560–576 (2003). https://doi.org/10.1109/TCSVT.2003.815165
70. M. Wipliez, G. Roquier, and J.F. Nezan: Software code generation for the RVC-CAL language. Journal of Signal Processing Systems (2011). http://dx.doi.org/10.1007/s11265-009-0390-z
71. M. Michalska, N. Zufferey, E. Bezati, M. Mattavelli: "High-precision performance estimation of dynamic dataflow programs," IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, IEEE MCSoC, Lyon, France, September 21–23 2016.
72. M. Michalska, N. Zufferey, E. Bezati, M. Mattavelli: "Design space exploration problem formulation for dataflow programs on heterogeneous architectures," IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, IEEE MCSoC, Lyon, France, September 21–23 2016.

Signal Processing for Wireless Transceivers

Markku Renfors, Markku Juntti, and Mikko Valkama

Abstract The data rates as well as quality of service (QoS) requirements for rich user experience in wireless communication services are continuously growing. While consuming a major portion of the energy needed by wireless devices, the wireless transceivers have a key role in guaranteeing the needed data rates with high bandwidth efficiency. The cost of wireless devices also heavily depends on the transmitter and receiver technologies. In this chapter, we concentrate on the problem of transmitting information sequences efficiently through a wireless channel and performing reception such that it can be implemented with state of the art signal processing tools. The operations of the wireless devices can be divided to RF and baseband (BB) processing. Our emphasis is to cover the BB part, including the coding, modulation, and waveform generation functions, which are mostly using the tools and techniques from digital signal processing. But we also look at the overall transceiver from the RF system point of view, covering issues like frequency translations and channelization filtering, as well as emerging techniques for mitigating the inevitable imperfections of the analog RF circuitry through advanced digital signal processing methods.

1 Introduction and System Overview

The data rates as well as quality of service (QoS) requirements for rich user experience in wireless communication services are continuously growing. More and more devices will be connected to the global ubiquitous information network. According to Cisco's prediction, the volume of mobile data traffic will expand

M. Renfors (✉) · M. Valkama
Tampere University of Technology, Faculty of Computing and Electrical Engineering, Tampere, Finland
e-mail: markku.renfors@tut.fi; mikko.e.valkama@tut.fi

M. Juntti
University of Oulu, Centre for Wireless Communications, Oulu, Finland
e-mail: markku.juntti@oulu.fi

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_8

seven times over the next 4 years, reaching nearly 12 billion mobile devices and generating 49 exabytes of mobile traffic by 2021 [39]. The diversity of the devices and services will increase. While the demand of high data rates to provide multimedia services, like video transmission, is increasing, the demand of low rate sensor information to enable location and context awareness of the services is also increasing. While the 4th generation (4G) LTE network, supporting mainly mobile broadband communications, has been widely deployed, the on-going 5th generation (5G) wireless cellular system development aims to create a multi-service network supporting a wide range of services with different requirements regarding data rate, latency, and reliability. These services include enhanced mobile broadband (eMBB) targeting at Gbps peak data rates, massive machine-type communications (mMTC) closely related to the Internet-of-things (IoT) concept, and ultra reliable low-latency communications (URLLC) needed, e.g., in the contexts of smart traffic, remote control of vehicles and industrial processes, and so-called tactile communications [150].

To enable the cost, energy and bandwidth efficient realization of the vision, the transceiver and technology need to make major leaps. One of the key concerns is the overall power and energy consumption of the devices and the whole network infrastructure. The energy efficiency is major issue from battery and device operation perspective, but also relates to the sustainable development when the complete system is concerned. Therefore, in addition to more conventional target of *bandwidth efficiency* and increasing the data rates, also the *power and energy efficiency* of the evolving wireless systems is of major concern. The goal of this chapter is to introduce the key aspects of the baseband (BB) and radio frequency (RF) signal processing chains of wireless transmitters and receivers. Our emphasis is on cellular type systems, but many of the principles can be applied in various short range, wireless local area networks and other wireless applications.

The higher layers of the communication protocol stack of the Open System Interconnect (OSI) model have conventionally been designed separate from the physical layer. However, the current wireless systems are introducing more and more crosslayer design and optimization. As an example, the evolving cellular Third Generation (3G) Long Term Evolution (LTE) systems use so called channel aware user scheduling and radio resource management (RRM) techniques. The applied methodology capitalizes on signal processing tools and uses to some extent similar approach as the physical layer signal processing. However, we do not cover those either, but they are definitely important currently evolving fields of research and development. Signal processing tools are applied in wireless devices also in multimedia and application processing, data compression, etc. However, we do not cover those aspects, but concentrate on the connectivity related problems on the physical layer.

The typical transmitter (TX) and receiver (RX) functionalities are summarized in Fig. 1. Starting with the first block in the TX chain, information is coded using forward error control (FEC) coding with interleaving. The purpose of this is to protect the information from errors. Data modulation transforms the information bit sequence into a complex multi-level symbol sequence with reduced sample rate and

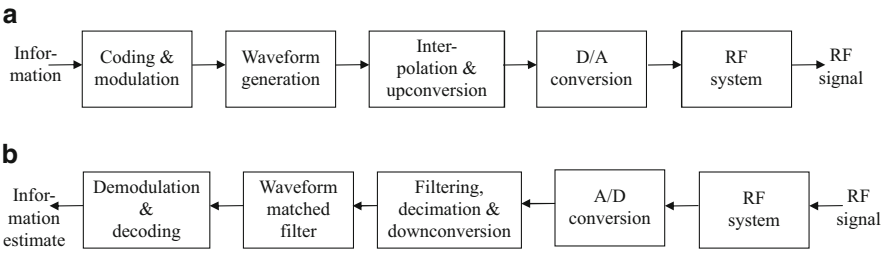


Fig. 1 Simplified wireless transceiver processing chain: (**a**) transmitter, (**b**) receiver

bandwidth. The waveform generation block creates discrete-time baseband signal with specific spectral and time-domain characteristics suitable for transmission in the used frequency band and radio propagation environment. The fundamental classes of waveforms include linear and FSK-type single-carrier transmission, multicarrier transmission, as well as spread-spectrum techniques. Multiplexing and multiple-access functionalities are also closely related with waveform generation. Finally, the generated waveform is upconverted to the used RF channel and amplified to desired transmission power level. Depending on the used transmitter architecture, the upconversion can be done in multiple steps, using intermediate frequency (IF) processing stages along the way. Also, the upconversion process may be carried out at least partially in the DSP domain. In general, digital-to-analog (D/A) converter, which acts as the interface between digital and analog front-ends, is gradually moving towards the antenna. The receiver side processing in Fig. 1b performs the opposite operations to recover the original information sequence with as little errors as possible while keeping the processing latency and energy consumption feasible.

This chapter is organized as follows. Section 2 introduces the concepts for coding, interleaving and modulation as well as their receiver counterparts. Because receiver processing in general and equalization in particular is the more demanding task, the emphasis is on that side of the problem. One of the main capacity boosters at the physical layer is the use of multiple antennas both/either in a transmitter and/or in a receiver or so called multiple-input multiple-output (MIMO) communications; it is considered as a key example in the receiver processing. Section 3 focuses on the waveform generation and its inverse operations and it has special emphasis on multicarrier techniques which have been adopted in most of the recent and emerging broadband wireless system standards. Also the timely topic of spectrum agility, facilitating effective fragmented spectrum use, is addressed. The generation of the actual transmitted signal, using both digital signal processing and analog RF processing, is treated in Sect. 4. Because RF parts are usually the most expensive and power hungry components of a wireless device, it often makes sense to use BB processing to compensate for RF non-idealities; this is also a major topic in that section. Finally, conclusions and some further topics are discussed in Sect. 5

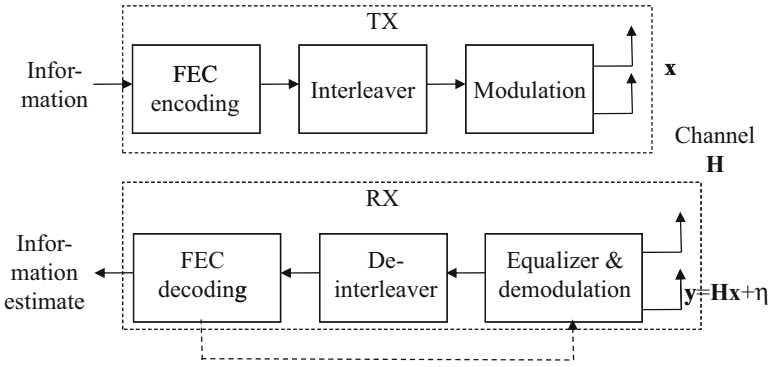


Fig. 2 Symbol rate system for coding, modulation, demodulation, equalization and decoding

2 Equalization and MIMO Processing

This section focuses on the demodulation and decoding block of Fig. 1, which belongs to the most computation-intensive parts of the receiver baseband processing. We also consider the channel equalization as part of this problem. The model is simplified such that all our processing is performed on symbol rate, while the subsequent blocks of Fig. 1 perform all the higher sampling rate operations needed in radio transmission and reception. The simplified system model is depicted in Fig. 2. In other words, we focus on coding and modulation in the transmitter side and their counterpart operations in the receive end. In addition, the channel impulse response needs to be estimated, and that is considered as well.

2.1 System Model

We consider transmission of a binary information stream or data packet via *bit interleaved coded modulation* (BICM). The information sequence is first FEC encoded by some appropriate coding method, like block, convolutional or concatenated coding [22, 126, 148]. Parallel concatenated convolutional (PCC) or so called turbo codes [24] are among the most commonly applied codes currently. They have been adopted to 3G and LTE cellular systems, amongst others. Other popular codes include low-density parity check (LDPC) codes [61]. As shown in Fig. 2, the coded information is interleaved and modulated. The purpose of interleaving is to protect the data from bursty errors due to fading of the wireless channel. It re-organizes the order in which encoded bits are transmitted so that the consequent bits are uncorrelated. This maintains the error correction capability of the code [22, 66, 126]. Several interleaver designs exist, but we do not discuss that further. We assume any interleaving with sufficient length compared to the channel coherence time.

Multiple-input-multiple-output radio channel, i.e., multiple transmit and receive antennas [27, 66, 165] is considered. The MIMO technology can be used to boost both/either the performance (error rate) and/or data rate of a single link as well as the whole system by applying multiuser MIMO processing. We assume that the channel is frequency-flat so that no inter-symbol interference (ISI) is generated. This can be achieved, e.g., by orthogonal frequency division multiplexing (OFDM), which is commonly used in current wireless systems like in the downlink 3GPP Long Term Evolution (LTE) and its Advanced version (LTE-A) [45], wireless local loops (WLAN) 802.11a/g/n, and Worldwide Interoperability for Microwave Access (WiMAX). If ISI is generated, an equalizer is needed as is discussed later in this chapter. The channelization and different multiplexing schemes are covered in more detail in Sect. 3. Perfect time and frequency synchronization is assumed.

A MIMO transmission system with N TX and M RX antennas, where $N \leq M$, is considered. This assumption is used to guarantee unique detectability of the data. We assume a linear quadrature amplitude modulation (QAM). The received signal can be described with the equation

$$\mathbf{y} = \mathbf{HPx} + \boldsymbol{\eta}, \tag{1}$$

where $\mathbf{x} \in \Omega^N$ is the vector of transmitted data symbols, $\Omega \subset \mathbb{C}$ is a discrete set of modulation symbols, $\boldsymbol{\eta} \in \mathbb{C}^M$ is a vector containing identically distributed circularly symmetric complex Gaussian noise samples with variance σ^2, $\mathbf{H} \in \mathbb{C}^{M \times N}$ is the channel matrix containing complex Gaussian fading coefficients, and $\mathbf{P} \in \mathbb{C}^{N \times N}$ is the pre-coding matrix. In other words, the element at the mth row and nth column of $\mathbf{H}$ is the complex channel coefficient between TX antenna n and RX antenna m. The pre-coding matrix can be used for beamforming to improve the system performance in case some degree of channel knowledge is available at the transmitter. That can be achieved by some feedback mechanism or assuming reciprocal reverse channel, which may be the case in time-division duplex (TDD) systems, for example.

The modulated symbols, i.e., the entries of $\mathbf{x}$ are drawn from a complex QAM constellation Ω with size $|\Omega| = 2^Q$, where Q is the number of encoded bits per symbol. For example, the 16-QAM constellation would be $\Omega = \{(\pm 3 \pm j3), (\pm 3 \pm j), (\pm 1 \pm j3), (\pm 1 \pm j)\}$, where $j^2 = -1$. The modulation mapping from consequent encoded and interleaved bits is typically performed by Gray mapping [126, Sect. 4.3]. We denote the bijective mapping function by ψ such that the binary encoded bit vector $\mathbf{b}_n \in \{-1, +1\}^Q$ is mapped to symbol $x_n = \psi(\mathbf{b})$ or $\mathbf{x} = \psi(\mathbf{b})$, where $\mathbf{b} = [\mathbf{b}_1^{\mathrm{T}}, \mathbf{b}_2^{\mathrm{T}}, \ldots, \mathbf{b}_N^{\mathrm{T}}]^{\mathrm{T}} \in \{-1, +1\}^{QN}$. The coded bit sequence $\mathbf{b}$ has been obtained from the original information bit sequence via FEC encoding, whose operation depends on the applied coding scheme.

The model presented herein is a MIMO system in a frequency-flat channel with no ISI. However, the mathematical formulation can be relatively straightforwardly generalized to cover also multipath propagation and ISI. The receiver principles and the solutions proposed below are also applicable to a large extent for such a model. The equalizer principles developed for ISI channels have been a source of inspiration also for the MIMO problem and from mathematical perspective they are equivalent to a large extent.

The model above covers several MIMO configurations. It can incorporate *space-time coding* or transmit *diversity* schemes, which usually aim at increasing the diversity gain or robustness to fading [27, 66, 165]. They can similarly include *spatial multiplexing* (SM), wherein the key target is to increase the data rate of the transmission. From receiver signal processing perspective, which is the key topic of this chapter and best aligned on the scope of this handbook, the SM is conceptually the simplest yet very challenging. Therefore, we focus on that in most of the discussion.

SM can apply different so called *layering* solutions. A layer refers to a coded data *stream* which can be multiplexed to transmit antennas using different schemes. In *horizontal* layering, each stream is transmitted from different antenna, which makes the spatial separation of the streams somewhat more straightforward. *Vertical* layering multiplexes each stream to all transmit antennas, which enables achieving spatial diversity amongst encoded bits, but complicates the receiver processing.

In the forthcoming discussion on the receiver design in Sects. 2.2–2.4, we assume for the simplicity of notation that $\mathbf{P} = \mathbf{I}_N$ (where $\mathbf{I}_N$ is a $N \times N$ identity matrix), i.e., no pre-coding without loss of generality. If pre-coding is applied, we just need to replace $\mathbf{H}$ by $\mathbf{HP}$ in the discussion below.

2.2 Optimum Detector and Decoding

The ultimate target of the receiver processing is to reproduce the true transmitted information bit sequence at the FEC decoder output. This is of course usually not perfectly possible, because of the random noise, fading, interference and other sources of distortion in the radio channel and in the communication equipment. Therefore, a pragmatic optimum receiver would minimize the probability of decoding errors given the received observation $\mathbf{y}$ in (1). Such an approach would lead to jointly optimum decoding, demodulation and equalization, which is practically too complex to be realized [109]. This is the reason, why practical receivers are partitioned as shown in Figs. 1b and 2. Therein the equalizer and demodulator process the received signal $\mathbf{y}$ to provide an estimate of the coded bit sequence $\mathbf{b}$ in a form applicable for the FEC decoder, which then provides the final estimate of the information bit sequence.

If there were no FEC coding, the optimum detector would simply make a hard decision by finding the most likely transmitted data symbol vector $\mathbf{x}$ given the observed received signal $\mathbf{y}$, or $\hat{\mathbf{x}}_{\mathrm{MAP}} = \arg\min_{x \in \Omega^N} p(\mathbf{x}|\mathbf{y})$, where $p(x|y)$ denotes the conditional probability density (or mass) function (PDF) (depending on the context). We also assume herein that the channel matrix $\mathbf{H}$ is perfectly known. In the receiver context $p(x|y)$ is usually called as the *a posteriori* probability (APP), and the optimum detector is the maximum APP (MAP) receiver, which minimizes the average probability of symbol sequence decision error; the same principle has also been called maximum likelihood sequence estimation (MLSE) in the ISI channel context [126]. By Bayes rule $p(\mathbf{x}|\mathbf{y}) = p(\mathbf{x}, \mathbf{y})/p(\mathbf{y}) = p(\mathbf{y}, \mathbf{x})p(\mathbf{x})/p(\mathbf{y})$. Thus, if

there is no *a priori* information or all the possible modulation symbols are equally likely, the maximization in the MAP sequence detector reduces to the maximum likelihood (ML) sequence detector $\hat{\mathbf{x}}_{\mathrm{ML}} = \arg\min_{x\in\Omega^N} p(\mathbf{y}|\mathbf{x})$. In the Gaussian channel with known channel realization, $p(\mathbf{y}|\mathbf{x})$ is the Gaussian PDF the ML detection reduces to finding the constellation points with the minimum Euclidean distance (ED) to the received signal vector $\mathbf{y}$, or

$$\hat{\mathbf{x}}_{\mathrm{ML}} = \arg\min_{x\in\Omega^N} ||\mathbf{y} - \mathbf{H}\mathbf{x}||^2. \tag{2}$$

The FEC decoding is assumed to be a soft-input soft-output (SfISfO) decoder [148], which is the practically pervasive choice in current wireless devices. This means that the decoder needs probability information about the coded bits to be able to calculate the corresponding most likely information bit sequence. This is usually represented as by log-likelihood ratio (LLR) value of the kth element of $\mathbf{b}$ as

$$\begin{aligned} L_D(b_k|\mathbf{y}) &= \ln\frac{\Pr(b_k = 1|\mathbf{y})}{\Pr(b_k = 0|\mathbf{y})} \\ &= \ln(p(\mathbf{y}|b_k = 1)) - \ln(p(\mathbf{y}|b_k = 0)). \end{aligned} \tag{3}$$

If the interleaver is sufficiently long, the consequent bits become (approximately) independent of each other. In that case, the logarithm of the APP above become by the Bayes rule [77, 90]

$$L_D(b_k|\mathbf{y}) = L_A(b_k) + \ln\frac{\sum_{\mathbf{b}\in \mathrm{L}_{k,+1}} \exp(\Lambda(\mathbf{b}, \mathbf{b}_{[k]}, \mathbf{l}_{A,[k]}|\mathbf{y}, \mathbf{H}))}{\sum_{\mathbf{b}\in \mathrm{L}_{k,-1}} \exp(\Lambda(\mathbf{b}, \mathbf{b}_{[k]}, \mathbf{l}_{A,[k]}|\mathbf{y}, \mathbf{H}))}, \tag{4}$$

where

$$L_A(b_k) = \ln\frac{\Pr(b_k = 1)}{\Pr(b_k = 0)}, \tag{5}$$

is *a priori* information or LLR,

$$(\Lambda(\mathbf{b}, \mathbf{b}_{[k]}, \mathbf{l}_{A,[k]}|\mathbf{y}, \mathbf{H})) = -\frac{1}{2\sigma^2}||\mathbf{y} - \mathbf{H}\mathbf{x}||^2 + \frac{1}{2}\mathbf{b}_{[k]}^T\mathbf{l}_{A,[k]}, \tag{6}$$

$\mathbf{b}_{[k]} \in \{-1, +1\}^{QN-1}$ consists of all the elements of $\mathbf{b}$ excluding the kth one, $\mathbf{l}_{A,[k]}$ is a vector of L_A for all bits in $\mathbf{b}_{[k]}$, and $\mathrm{L}_{k,\beta} = \{\mathbf{b} \in \{-1, +1\}^{QN}|\mathbf{b}_k = \beta\}$. The expression in (6) follows from the fact that $(\mathbf{y}|\mathbf{b}, \mathbf{H})$ in (1) is Gaussian. Therefore, the LLR is related to the Euclidean distance metric.

The above expression is in general complex to evaluate, because the number of elements in the summation (4) is exponential in the number of spatial channels (or the number of TX antennas N) and the number of bits per symbol Q. This also

implies a polynomial complexity in terms of the size of the modulation alphabet. In other words, the search of the maximum APP performed by the MAP receiver is exponentially complex. Therefore, approximations are usually needed, and those will be discussed in more detail below in Sect. 2.3. Equivalent problem has been classically considered in the context of equalizers for ISI channels [59, 126]. The idea in those is to limit the search space, while still achieving reasonably good performance.

In practical receivers, also the LLR *computation* is usually approximated in addition to reducing the search space. A typical approximation is to use a small look-up table and the Jacobian logarithm

$$\text{jacln}(a_1, a_2) := \ln(e^{a_1} + e^{a_2}) = \max(a_1, a_2) + \ln(1 + e^{-|a_1 - a_2|}). \tag{7}$$

The Jacobian logarithm in (7) can be computed without the logarithm or exponential functions by storing $r(|a_1 - a_2|)$ in a look-up table, where $r(\cdot)$ is a refinement of the approximation $\max(a_1, a_2)$ [77].

2.3 Suboptimal Equalization

The suboptimal detector or equalizer principles are similar to those applied earlier in ISI channels [126] or in multiuser detection to mitigate multiple-access interference (MAI) [83, 177]. Among the simplest approaches is to process the received signal (1) linearly, i.e., apply linear equalizer. It can be represented as multiplying $\mathbf{y}$ by an equalizer represented as a matrix $\mathbf{W}$ so that the equalizer output is

$$\mathbf{y}_{EQ} = \mathbf{W}\mathbf{y} = \mathbf{W}\mathbf{H}\mathbf{x} + \mathbf{W}\boldsymbol{\eta}. \tag{8}$$

The simplest choice for the equalizer would be the complex conjugate transpose of the channel realization, i.e., $\mathbf{W} = \mathbf{H}^{\mathrm{H}}$, where $(\cdot)^{\mathrm{H}}$ denotes the complex conjugate transpose. This corresponds to the channel matched filter (MF) maximizing the signal-to-noise ratio (SNR) of each of the spatial channels with no consideration on the spatial multiplexing interference (SMI) often present in MIMO systems; in spread spectrum or code-division multiple access (CDMA), this would be called the rake receiver or conventional MF detector. The equalizer perfectly removing all the SMI is the zero-forcing (ZF) one or $\mathbf{W} = (\mathbf{H}^{\mathrm{H}}\mathbf{H})^{-1}\mathbf{H}^{\mathrm{H}}$, which is the pseudo-inverse of the channel realization yielding the linear least squares estimate of the transmitted symbol vector $\mathbf{x}$. It completely removes all the SMI, but it has the commonly known drawback of noise enhancement. In other words, it can be seen as maximizing signal-to-interference ratio (SIR) with no consideration on the noise; in the CDMA context this is often called as decorrelator. Finally, the linear minimum mean square error (LMMSE) equalizer

$$\mathbf{W} = \mathbf{B}(\mathbf{H}^{\mathrm{H}}\mathbf{H} + \sigma^2\mathbf{I}_M)^{-1}\mathbf{H}^{\mathrm{H}} \tag{9}$$

makes a controlled compromise by jointly minimizing the impact of both noise and SMI or ISI. For the Wiener filter or the actual LMMSE equalizer $\mathbf{B} = \mathbf{I}$, but its output is in general biased, because its expected output is a scaled version of $\mathbf{x}$, not $\mathbf{x}$ itself. The bias can be removed by the choice $\mathbf{B} = \mathrm{diag}[\mathrm{diag}((\mathbf{H}^{\mathrm{H}}\mathbf{H} + \sigma^2\mathbf{I}_M)^{-1}\mathbf{H}^{\mathrm{H}})^{-1}]$. In that case, the mth diagonal element of $\mathbf{B}$ becomes [40] $B_{m,m} = (\rho_m + 1)/\rho_m$, where the signal-to-interference-plus-noise ratio (SINR) per stream is

$$\rho_m = \frac{1}{\sigma^2[(\mathbf{H}^{\mathrm{H}}\mathbf{H} + \sigma^2\mathbf{I}_M)^{-1}]_{m,m}} - 1. \tag{10}$$

This scaled version of the LMMSE equalizer maximizes the SINR with some penalty in mean square error (MSE) [73, 165].

Calculating the soft output for the FEC decoder from the linear equalizer output requires some further attention. Because linear processing maintains sufficient statistics, the optimum MAP detection would remain equally complex as above. However, there are reasonably good simplified approximations of the LLR for BICM. One efficient method has been presented in [40]. It reduces complexity and latency with only a minor impact on performance. Instead of calculating the Euclidean distance between the LMMSE equalizer output and all the possible transmitted symbols, Gray labeling of the signal points is exploited therein. The LLR bit-metric $\hat{L}(b^\xi|\mathbf{y}_{EQ}, \mathbf{W})$ for bit b^ξ (where ξ is an integer) can be approximated as $\rho_k \Xi(b^\xi, \mathbf{y}_{EQ})$, where

$$\Xi(b^\xi, \mathbf{y}_{EQ}) = \min_{\tilde{x}_k \in \mathrm{X}^0_{k,\xi}} |y_{EQ,k} - \tilde{x}_k|^2 - \min_{\tilde{x}_k \in \mathrm{X}^1_{k,\xi}} |y_{EQ,k} - \tilde{x}_k|^2, \tag{11}$$

where $k = \lfloor \xi/Q \rfloor + 1$, $\mathrm{X} = \{x_k : b^\xi = i\}$ is the subset of hypersymbols $\{x\}$ for which the ξth bit of label b is i. $\Xi(b^\xi, \mathbf{y}_{EQ})$ can be simplified by considering $y_{EQ,k}$ in only one quadrature dimension given by ξ [40].

Decision-feedback equalization (DFE) is a classic alternative to linear processing to improve the performance both under ISI or MAI. One version is based on successive interference cancellation (SIC) and linear MMSE equalization. It was proposed in the early MIMO communication proposals known as Bell Labs layered space-time (BLAST) scheme [182]. It is best applicable for horizontally layered spatial multiplexing, because then the layers align on physical channels transmitted from a transmit antenna. The received layers are ordered with respect to their SNR or received power level. The strongest signal is detected and decoded first so that the SMI it suffers from the weaker ones is suppressed by a linear equalizer, which is typically based on MMSE or maximum SINR (9) criterion. The interference it causes to the other streams is estimated based on the decoded data and subtracted from them. Then the second strongest signal is similarly detected, decoded and canceled from the remaining signals and so on. This also is called successive nulling and interference cancellation. The decoding requires deinterleaving, which imposes latency to the processing.

The weight matrix is calculated with the LMMSE rule as in (9). The layer for detection is chosen according to the post-detection SINR and the corresponding nulling vector is chosen from the weight matrix $\mathbf{W}$ [182]. All the weight matrices in an OFDM symbol are calculated and the layer to be detected is chosen according to the average over all the subcarriers. After the first iteration, the canceled symbol expectation is used to update the weight matrix. The weight matrix for the second layer to be canceled is calculated as

$$\mathbf{W} = (E\{x\}E\{x\}^{*}\mathbf{h}_k\mathbf{h}_k^{\mathrm{H}} + \mathbf{H}_k(\mathbf{I} - (E\{x\}E\{x\}^{*})\mathbf{H}_k^{\mathrm{H}} + \sigma^2\mathbf{I}_M))^{-1}\mathbf{h}_k^{\mathrm{H}}, \tag{12}$$

where $\mathbf{h}_k$ is the kth vector from matrix $\mathbf{H}$, k is the layer to be detected, $\mathbf{H}_k$ is matrix $\mathbf{H}$ with the vectors from previously detected layers removed and $E\{x\}$ is the symbol expectation.

The detected layer is decoded and symbol expectations from the soft decoder outputs can be calculated as [167]

$$E\{x\} = (\frac{1}{2})^{Q}\sum_{x_l\in\Omega} x_l \prod_{i=1}^{Q}(1 + b_{i,l}\tanh(L_A(b_i)/2)), \tag{13}$$

where $L_A(b_i)$ are the LLRs of coded bits corresponding to x and $b_{i,l}$ are bits corresponding to constellation point x_l. The expectation calculation in (13) can be simplified to the form

$$E\{x\}_{\mathrm{re}} = \mathrm{sgn}(L_A(b_i))S|\tanh(L_A(b_{i+2}))|. \tag{14}$$

The constellation point S is chosen from {1,3,5,7} depending on the signs of $L_A(b_{i+1})$ and $L_A(b_{i+2})$.

In addition to the linear and decision-feedback based equalization, there are also several other suboptimal equalizers, e.g., based on various tree-search approaches. One of the most popular ones is the concept of sphere detector (SD). Another closely related one is a selective spanning with fast enumeration (SSFE) [98]. In the case of transmission with no FEC coding, a SD calculates the ML solution by taking into account only the lattice points that are inside a sphere of a given radius [46, 58]. The SDs take into account only the constellation points that are inside a sphere of a given radius, or

$$||\mathbf{y} - \mathbf{H}\mathbf{x}||^2 \leq C_0. \tag{15}$$

After QR decomposition (QRD) of the channel matrix $\mathbf{H}$ in (15), it can be rewritten as

$$||\mathbf{y}' - \mathbf{R}\mathbf{x}||^2 \leq C_0', \tag{16}$$

where $C_0' = C_0 - ||(\mathbf{Q}')^{\mathrm{H}}\mathbf{y}||^2$, $\mathbf{y}' = \mathbf{Q}^{\mathrm{H}}\mathbf{y}$, $\mathbf{R} \in \mathbb{C}^{N\times N}$ is an upper triangular matrix with positive diagonal elements, $\mathbf{Q} \in \mathbb{C}^{M\times N}$ and $\mathbf{Q}' \in \mathbb{C}^{M\times(M-N)}$ are orthogonal matrices.

The squared partial Euclidean distance (PED) of $\mathbf{x}_i^N$, i.e., the square of the distance between the partial candidate symbol vector and the partial received vector, can be calculated as

$$d(\mathbf{x}_i^N) = \sum_{j=i}^{N} \left| y_j' - \sum_{l=j}^{N} r_{j,l} x_l \right|^2, \tag{17}$$

where $i = N \ldots, 1$ and $\mathbf{x}_i^N$ denotes the last $N - i + 1$ components of vector $\mathbf{x}$ [46].

In the presence of FEC coding, the SD must be modified to provide an appropriate soft output to approximate the MAP detector. A list sphere detector (LSD) [77] is capable of doing that by providing a list L of candidates and their APP or LLR values of the coded bits in $\mathbf{b}$ to the FEC decoder. There are different strategies to perform the search of the potential candidates. Most of them have been originally proposed for the conventional sphere detector and then subsequently generalized for the LSD version. The breadth-first tree search based K-best LSD algorithm [67, 148, 183] is a variant of the well known M algorithm [9, 81]. It keeps the K nodes which have the smallest accumulated Euclidean distances at each level. If the PED is larger than the squared sphere radius C_0, the corresponding node will not be expanded. We assume no sphere constraint or $C_0 = \infty$, but set the value for K instead, as is common with the K-best algorithms. The depth-first [154] and metric-first [119] sphere detectors have a closer to optimal search strategy and achieve a lower bit error rate than the breadth-first detector. However, the K-best LSD has received significant attention, because it can be easily pipelined and parallelized and provides a fixed detection rate. The breadth-first K-best LSD can also be more easily implemented and provide the high and constant detection rates required in the LTE.

In the discussion above, we have assumed mostly one-pass type receiver processing. In other words, equalization/detection and channel estimation are performed first. The detector soft output is then forwarded to the FEC decoder where the final data decisions are made. However, the performance can be enhanced by iterative information processing based on so called turbo principle [1, 2, 69], originating from the concept of parallel (or serial) concatenated convolutional codes often known as turbo codes [24, 25, 148]. This means that the feedback from FEC decoder to the equalizer as shown in Fig. 2 is applied. Therein, the decoder output extrinsic LLR value is used as *a priori* LLR value in the second equalization iteration [188]. This typically improves the performance at the cost of increased latency and complexity [90]. Because the decoder is also usually iterative, the arrangement results in multiple iterations, i.e., local iterations within the (turbo type) decoder and global iterations between the equalizer and decoder. The useful number of iterations is usually determined by computer simulations or semianalytical study of the iteration performance.

2.4 Channel Estimation

The discussion above assumes that the channel realization or the matrix $\mathbf{H}$ is perfectly known, which is the basic assumption in coherent receivers. Therefore, channel estimation needs to be performed. This is usually based on transmitting reference or pilot symbols known by the receiver [34]. By removing their impact, the received signal reduces to the unknown channel realization and additive Gaussian noise. Classical or Bayesian estimation framework [86, 147] can be then applied to estimate the channel realization. The channel time and frequency selectivity and other propagation phenomena need to be appropriately modeled to create a realistic channel model and corresponding estimation framework [123]. If orthogonal frequency-division multiplexing (OFDM) [70] is assumed, the frequency-selectivity of the channel can be handled very efficiently. This is a benefit from the equalizer complexity perspective.

It should be noted here that the assumption of no pre-coding makes channel estimation different to the case with pre-coding. Pre-coding optimization is typically based on the channel state, and in that sense to the channel estimate. Therefore, there are two options to deal with this case. The channel estimate is usually based on pilot or reference signals, which may either be similarly precoded as the data symbols or not precoded.

The system model for the channel estimation for an OFDM based MIMO transmission system is defined below. The received signal vector $\mathbf{y}(n)$ on the m_Rth receive antenna at discrete time index n after the discrete Fourier transform (DFT) can be described as

$$\underline{\mathbf{y}}_{m_R}(n) = \mathbf{X}(n)\mathbf{F}\mathbf{h}_{m_R}(n) + \mathbf{w}_{m_R}(n), \tag{18}$$

where $\mathbf{X} = [\mathbf{X}_1, \ldots, \mathbf{X}_N] \in \mathbb{C}^{P \times PN}$ is the transmitted signal over P subcarriers, $\mathbf{w}_{m_R} \in \mathbb{C}^{P \times 1}$ contains identically distributed complex white Gaussian noise, $\mathbf{F}$ is a $NP \times NL$ matrix from the DFT matrix with $[\mathbf{F}]_{u,s} = \frac{1}{\sqrt{P}} e^{-j2\pi us/P}$, $u = 0, \ldots, P-1$, $s = 0, \ldots, L-1$, L is the length of the channel impulse response and $\mathbf{h}_{m_R}$ is the time domain channel vector. $\mathbf{X}_{m_T} \in \mathbb{C}^{P \times P}$ is a diagonal matrix with entries from a complex quadrature amplitude modulation (QAM) constellation Ω and $|\Omega| = 2^Q$, where Q is the number of bits per symbol and $m_T = 1, \ldots, N$ and $m_R = 1, \ldots, M$.

The reference signal or pilot symbol positions in 3GPP Long Term Evolution (LTE) resource blocks are illustrated in Fig. 3. [62]. A downlink slot consist of 7 OFDM symbols and reference signals are transmitted in the first, second and fifth OFDM symbols of every slot. The reference signal positions for each antenna port are indicated in the figure, while nothing is transmitted on the other antenna ports when a reference signal is transmitted on one antenna port. The pilot overhead, in terms of the portion of data symbols in time or frequency used for training, is in the 2×2 MIMO roughly 9.5% and in the 4×4 MIMO about 14%. With 8×8 MIMO the pilot overhead could be close to 30% [15].

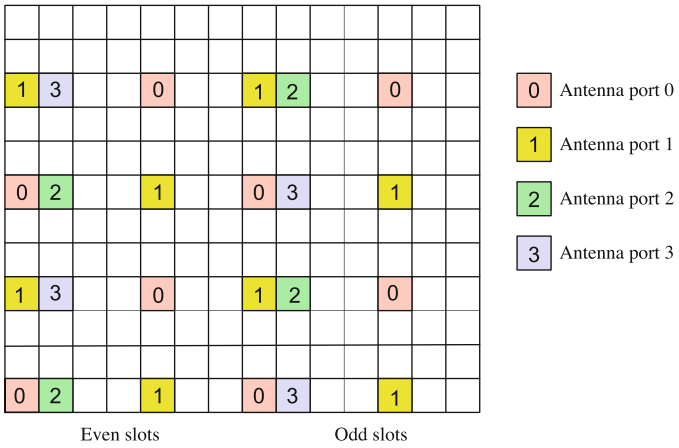


Fig. 3 Pilot symbol spacing in LTE standard for 4×4 MIMO channel [91]. The figure shows two resource blocks, each consisting of seven QAM symbols (horizontal dimension) in 12 subcarriers (vertical dimension)

The least-squares (LS) channel estimator based on training symbols is probably the simplest one to calculate the channel estimates from pilot symbols. The received symbol vector is often transformed into frequency domain before the LS channel estimation. The result of the LS estimator, on the other hand, is in time domain in the formulation below and it has to be transformed into frequency domain for the detector. The LS estimate of the channel can be calculated as

$$\hat{\mathbf{h}}_{m_R}(n) = (\mathbf{F}^H \mathbf{X}^H(n)\mathbf{X}(n)\mathbf{F})^{-1}\mathbf{F}^H \mathbf{X}^H(n)\underline{\mathbf{y}}_{m_R}(n), \tag{19}$$

where $\mathbf{X}$ contains the pilot symbols, which are known by the receiver. Because of that, the matrix inverse can be pre-computed and stored in a memory. Usually orthogonal (in time or frequency) training sequences or a diagonal matrix $\mathbf{X}$ are used such that there is no SMI in the channel estimate. The performance of the LS estimator can be improved by applying the Bayesian philosophy, i.e., by using the channel statistics to optimize the channel estimation filtering in frequency, spatial or temporal domain [110].

The reference signals or pilot symbols used in channel estimation are placed in the OFDM time-frequency grid at certain intervals. The interval may not be sufficiently short when the user velocity is high and the channel is fast fading. Furthermore, the pilot overhead increases with the number of MIMO streams. It becomes problematic already in the 4×4 antenna system and is significant (almost 30%) with an 8×8 system [15]. Decision directed (DD) channel estimation can be used to improve the performance or to reduce the pilot overhead. This can also be based on the same principle as the pilot based LS estimate (19), such that matrix $\mathbf{X}$ now includes the data decisions. However, this increases the complexity, because the matrix inverse must be computed now in real-time [189]. Typically this is realized

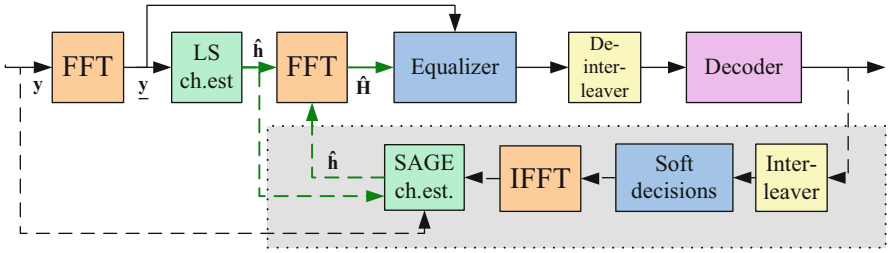


Fig. 4 Decision-directed channel estimation in MIMO receiver [91]

in the form of iterative receivers. The principle therein is similar to the one in Sect. 2.3 with the iterative detection–decoding, while now we have in general three blocks for the global iterations, namely, detection–decoding–channel estimation. This framework has been analyzed in detail, e.g., in [79, 94, 186, 188]. Several approaches are based on expectation-maximization (EM) algorithm [48, 108] or space-alternating generalized EM (SAGE) algorithm [56]. A the resulting receiver structure is illustrated in Fig. 4.

2.5 *Implementations*

The MIMO detection and channel estimation algorithms have found practical deployment in cellular and Wi-Fi WLAN standards, for example. Therefore, several works on practical receiver implementations and transceiver designs have been made. The computationally most demanding part of the filter matrix computation is the matrix inverse or some equivalent operation such as QR decomposition calculation. Designs for the MIMO detector context can be found, e.g., in [16, 32, 184]. In the sphere detector and other similar tree search algorithms, the search indexing and sorting are usually the most complex functionalities [31, 117].

Recent implementations include [17, 90, 117, 118, 155–157, 162]. The recent work by Suikkanen [156, 157] illustrates the trade-off between the receiver energy efficiency and useful data rate or *goodput*, which is defined as the minimum of the *detection rate* enabled by the receiver hardware and useful *throughput* of the communications system [90]. The latter depends on the error rate performance and the nominal data rate such that the value gives the error free or reliable transmission rate, practically achieved via hybrid automatic repeat request (HARQ) protocol with price of introduced latency. The throughput analysis assumed 4G cellular system or LTE-A standard system assumptions. The detection rate and receiver power consumption results were based on 28 nm CMOS technology based receiver baseband designs and the real time detection requirements of 4G cellular systems. High performance sphere detectors become necessary to achieve highest reliable throughput, but their energy efficiency in terms of processing energy per transmitted bit is often not as good as that of the simple linear detectors, which suffer data rate penalty.

3 Multicarrier Waveforms

Referring to Fig. 1, this section addresses the *waveform generation* function on the transmitter side, as well as the corresponding block on the receiver side.

3.1 Waveform Processing in OFDM Systems

The *coding and modulation block* produces a sequence of typically QAM modulated symbols, and the purpose of the *waveform generation* block is to produce a digital sample sequence which corresponds to the discrete-time baseband version of the final RF signal to be transmitted. Likewise, on the receiver side the waveform processing block receives the corresponding digital sample sequence, but affected by additive noise and interferences as well as various distortion effects, and produces a sample sequence corresponding to the QAM modulated symbol sequence at the *coding and modulation* block output.

In today's wireless communication system, various waveforms are utilized including linear single carrier modulation, i.e., QAM-type symbol sequence with Nyquist pulse shaping, Gaussian minimum shift keying (GMSK), and various types of spread-spectrum techniques, including direct sequence (DS) spread-spectrum with code-division multiple access (CDMA) [22, 165]. However, we focus here on the celebrated multicarrier transmission technique called orthogonal frequency-division multiplexing (OFDM) [26, 45, 95, 121, 127, 164, 180], which is the basis for most of the recent broadband wireless systems, including 802.11 WLAN family, DVB-T terrestrial TV broadcasting standards, WiMAX, 3GPP-LTE and LTE-Advanced.

3.1.1 OFDM Principle

A fundamental issue in wireless communications with increasing data rates is the complexity of the channel equalization. Channel equalization is needed in practically all wireless communication systems for compensating the effects of the multipath propagation channel, which appears as frequency dependency (frequency-selectivity) of the channel response experienced by the transmitted waveform. More importantly, this effect introduces dispersion to the symbol pulses which appears as inter-symbol interference (ISI), and eventually as errors in detecting the transmitted symbol values [22]. Traditional time-domain techniques for channel equalization, based on adaptive filtering or maximum likelihood sequence detection, would have prohibitive complexity at the signal bandwidths adopted in many of the recent communication standards.

As illustrated in Fig. 5, OFDM solves the problem by splitting the high-rate symbol sequence into a high number (N) of lower-rate sequences which are transmitted

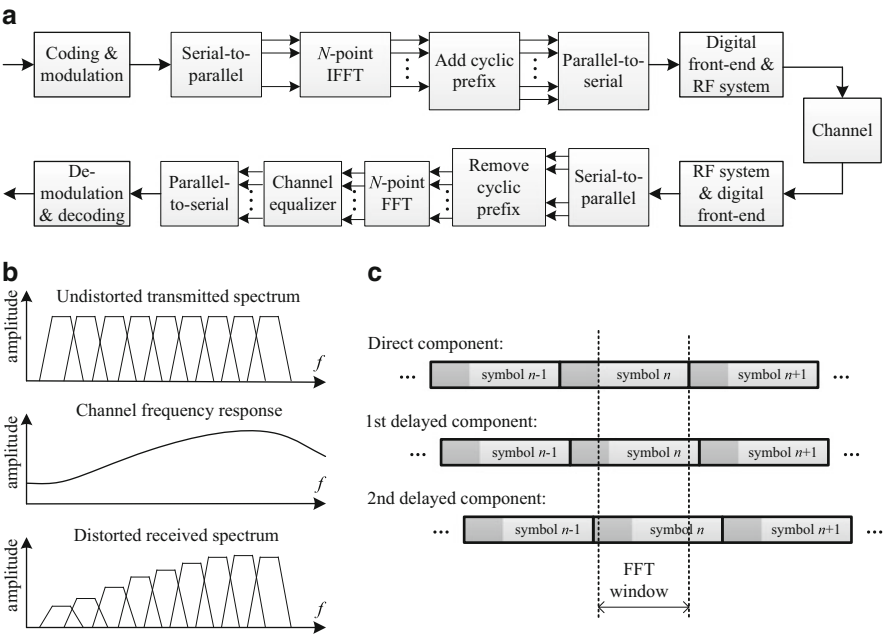


Fig. 5 (**a**) Basic OFDM transmission chain. (**b**) Effect of channel frequency selectivity. (**c**) Effect of multipath delays not exceeding the channel delay spread in CP-OFDM

in parallel, over a spectrally compact multiplex of orthogonal subchannels. Due to the increased symbol interval in the subchannels, the effects of channel dispersion are reduced, and the channel frequency response within each subchannel is, at most, mildly frequency selective. Furthermore, a cyclic prefix (CP) is commonly inserted in front of each OFDM symbol. The idea of CP is that it will absorb the variations in the delays of different multipath components of the channel, preventing ISI if the length of the CP is at least equal to the maximum delay spread of the channel. In this case, the effect of the channel can be modeled as a cyclic convolution. Consequently, the channel effect can be precisely modeled as flat fading at subcarrier level, and can be compensated by a single complex multiplication for each data symbol modulated to a subcarrier [45, 127].

In existing specifications, the FFT size of OFDM systems ranges from 64 in IEEE 802.11a/g WLAN to 32k in DVB-T2 [175]. The subcarrier spacings range, correspondingly, from 325 kHz to 279 Hz. As an important example, 3GPP-LTE uses 15 kHz subcarrier spacing and up to 20 MHz bandwidth, the maximum FFT-size being 2048 [45].

The practical implementation of OFDM utilizes inverse fast Fourier transform (IFFT) for multiplexing each block of parallel data symbols. Correspondingly, FFT is used for demultiplexing the block of complex sample values corresponding to the data symbols. Orthogonality of the subchannels follows directly from the properties

of discrete Fourier transform (DFT). In the channel, each data symbols appears as a square-windowed sinusoid, the frequency of which is determined by the subcarrier index and amplitude and phase are determined by the transmitted complex symbol value. Using continuous-time model, the transmitter and receiver OFDM waveform processing can be formulated as follows.

An OFDM symbol with IFFT size of N and duration of T_s is given by

$$x(t) = \sum_{k=0}^{N-1} X(k) e^{j2\pi f_k t}, \quad t \in [0,\ T_s] \tag{20}$$

where $X(k)$, $k = 0, \ldots,\ N-1$, are complex data symbols, typically from a QAM alphabet,

$$f_k = f_0 + k \cdot \Delta f \tag{21}$$

are the subcarrier frequencies and

$$\Delta f = \frac{1}{T_s} \tag{22}$$

is the frequency separation between subcarriers. With this choice, the subcarriers are orthogonal, i.e.,

$$\frac{1}{T_s} \int_0^{T_s} e^{j2\pi f_l t} e^{-j2\pi f_k t} dt = \delta_{kl} = \begin{cases} 1, & k = l \\ 0, & \text{otherwise} \end{cases} \tag{23}$$

Therefore in the absence of noise and other imperfections, the kth symbol is demodulated as

$$\frac{1}{T_s} \int_0^{T_s} x(t) e^{-j2\pi f_k t} dt = \frac{1}{T_s} \int_0^{T_s} \sum_{l=0}^{N-1} X(l) e^{j2\pi f_l t} e^{-j2\pi f_k t} dt = X(k). \tag{24}$$

In practical systems, guard-bands are introduced in the OFDM signal spectrum by modulating zero-valued symbols to the subcarriers close to the band edges. The requirements of the digital/analog anti-imaging filter, needed at the digital-to-analog interface, depend essentially on the width of the guard-band. Similarly, the guard-band width affects also the specifications of the channelization filtering on the receiver side.

The signal path of an OFDM transmission link, as illustrated in Fig. 5a, includes on the transmitter side the IFFT for a block of data symbols and copying a number of IFFT output samples in front of the produced OFDM symbol as a cyclic prefix, along with the needed buffering and serial-parallel and parallel-serial operations. On the receiver side, the core functions include extracting a block of N ISI-free samples

from the baseband sample sequence, FFT, and 1-tap subcarrier-wise equalizers. Additionally, a channel estimation function, usually based on known subcarrier symbols (scattered pilots and/or preambles) is needed, as described in Sect. 2.4. Also time and frequency synchronization functionalities are necessary in OFDM, as in any communication link [127].

3.1.2 Synchronization, Adaptive Modulation and Coding, and Multiple Access

The coarse time synchronization, i.e., determination of the optimum FFT window location, is commonly based on the correlation introduced to the signal by the cyclic prefixes. Residual timing offsets can be estimated using the pilot sequences and compensated by adjusting the channel equalizer coefficients accordingly. Various techniques are available in the literature for estimating the coarse frequency offsets, due to imprecise local oscillators in the transmission link. Fine frequency estimation can again be carried out using the pilots [45, 127].

Due to the narrow spacing of subcarriers (e.g., 1 kHz in DVB-T and 15 kHz in 3GPP-LTE), OFDM systems are quite sensitive to carrier frequency offset, the target values being at the order of $\pm 1\%$ of the subcarrier spacing, or less. This makes OFDM systems rather sensitive to fast-fading channels, and even to phase noise of the local oscillators. In general, these effects introduce inter-carrier interference (ICI).

Since OFDM is meant to be used with frequency/time-selective channels, some of the subcarrier symbols are bound to experience severe attenuation in the transmission channel, and the corresponding information bits would be lost in symbol-wise detection. In general, the channel gain for each subcarrier symbol depends on the instantaneous channel frequency response during the transmission. On the other hand, the whole OFDM multiplex has usually wide bandwidth compared to the channel coherence bandwidth, i.e., the channel appears as heavily frequency selective. While some of the subcarrier symbols are lost, a majority of them is received with good quality. Using FEC, the average bit-error rate (BER) or frame error rate (FER) achieves a targeted low value, in spite of some of the symbols being lost. Thus FEC is an essential element on OFDM systems, helping to exploit the inherent frequency diversity of the wideband transmission channel, and sometimes the scheme is referred to as coded OFDM (COFDM) [95].

The different subcarrier symbols in OFDM are transmitted independently of each other, through orthogonal subchannels. Then it is obvious that a single OFDM symbol is able to carry multiple users' data, using so-called orthogonal frequency division multiple access (OFDMA) [45]. In the downlink direction (from base-station, BS, to mobile stations, MS) this is quite straightforward. In the uplink direction, a BS receives a multiplex of subcarriers composed of subcarriers originating from different transmitters. In order to maintain orthogonality, so-called quasi-synchronous operation must be established. This means that the MS's must be precisely synchronized in frequency (say $\pm 1\%$ of subcarrier spacing), and different

mobiles' OFDM symbols, as seen at the BS receiver, must be time-aligned in such a way that the cyclic prefix is able to absorb both the channel delay spread and relative timing offsets between different MS's, as illustrated in Fig. 5c. Additionally, effective power control is needed to avoid excessive differences in the power levels of the received signals, thus avoiding serious problems due to RF impairments.

The practical OFDMA schemes are dynamic in the sense that variable data rates can be supported for each user. To achieve this, the BS must send side information to each MS about the set of subcarrier symbols allocated to each user, both for uplink and downlink. To keep the amount of side information reasonable, the allocation is commonly done using a resource block as the basic unit. For example in 3GPP-LTE, the resource block consists of 12 subcarriers and 7 consecutive symbols (this for the most commonly used transmission mode; there are also others) [45].

The basic form of OFDM systems uses the same modulation scheme (e.g., QPSK, 16QAM, or 64QAM) and code rate for all subcarriers and all OFDM symbols. The specifications are usually flexible, and allow the configuration of the system for different tradeoffs between data rate and robustness through the choice of modulation level and code rate. In broadcast systems, this is the scheme that has to be followed as it is not possible to tailor the transmission parameters separately for different users. However, in two-way communication, like cellular mobile systems and wireless local area networks (WLANs), it is possible to provide feedback information to the transmitter end about the channel quality and characteristics. If the transmitter has knowledge of the signal-to-interference-plus-noise (SINR) of each subcarrier, then the water-filling principle can be used for determining the optimal modulation level for each subcarrier. In OFDMA, the feedback information can also be used for allocating resource blocks optimally for the users based on the instantaneous channel response and quality (including various interferences) experienced by each user at each specific frequency slot. Furthermore, the modulation level and code rate can be tuned independently for each user to optimize the usage of transmission resources. This scheme is generally known as adaptive modulation and coding (AMC) [45].

3.2 Enhanced Multicarrier Waveforms

OFDM solves in an elegant and robust way the fundamental channel equalization problem in wideband wireless communications, and it provides efficient means for channel aware scheduling of the transmission resources in an optimal way to different users. Due to the flat-fading channel characteristics at subcarrier level, CP-OFDM is also an excellent basis for different multi-antenna (MIMO) techniques which are able to enhance the performance at link and system levels [45]. However, OFDM has also a number of limitations, which have motivated research on various enhancements as well as on alternative waveforms.

3.2.1 Peak-to-Average Power Ratio Issues and SC-FDMA

OFDM, and multicarrier waveforms in general, have the problem of high crest factor or peak-to-average power ratio (PAPR). This means that the peak envelope value of the modulated waveform is much higher than the RMS value, which introduces great challenges to the transmitter power amplifier implementation because high linearity is needed in order to avoid serious distortion effects [127]. Why the PAPR becomes high can be easily seen when we consider the OFDM signal as a sum of sinusoids with amplitudes and phases determined by the modulating symbol values. In the worst case, the amplitudes add up at some point within the OFDM symbol interval, and the PAPR is proportional to the number of active subcarriers. However, the probability of such a worst-case situation is in practice very small, and the PAPR characteristics of a waveform are better characterized by the complementary cumulative distribution function (see Fig. 7 for an example). Various techniques for reducing the PAPR of OFDM-modulated signals can be found from the literature [82, 127]. This problem is common with CDMA waveforms, and also various generic methods for reducing PAPR have also been developed, e.g., based on envelope peak clipping with smooth widowing [168].

Mainly due to the critical PAPR problem in hand-held devices, the single-carrier waveform has re-appeared in the OFDM context, in the form of so-called single-carrier frequency division multiple access (SC-FDMA) [45, 120, 164] . As shown, in Fig. 6, using DFT transform as precoding, a SC-FDMA block can be included in an OFDMA transmission frame while maintaining all the flexibility in allocation the resources to each user. The cascade of DFT and IFFT transforms (also referred to as DFT-spread-OFDM[1]) in the transmitter side effectively provides frequency shift of the single carrier symbol block to the frequency slot corresponding to the allocated subcarriers, as well as time-domain interpolation and rudimentary pulse shaping for the symbol pulses. With this model in mind, it is clear that accumulation of high PAPR does not take place in this process. However, while the pulse shaping

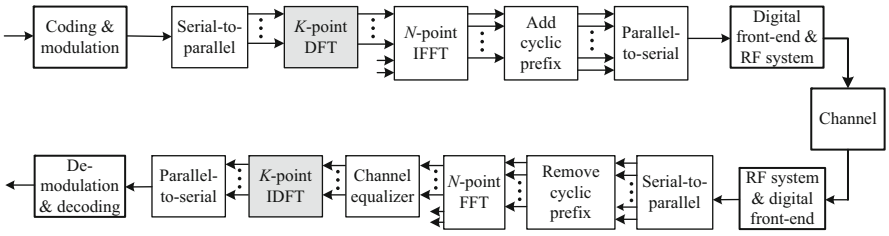


Fig. 6 SC-FDMA transmission link

[1]The terminology reflects the fact that the transform length in the core OFDM system is typically a power of two, whereas also other lengths need to be considered for the SC symbol block in order to reach sufficient flexibility.

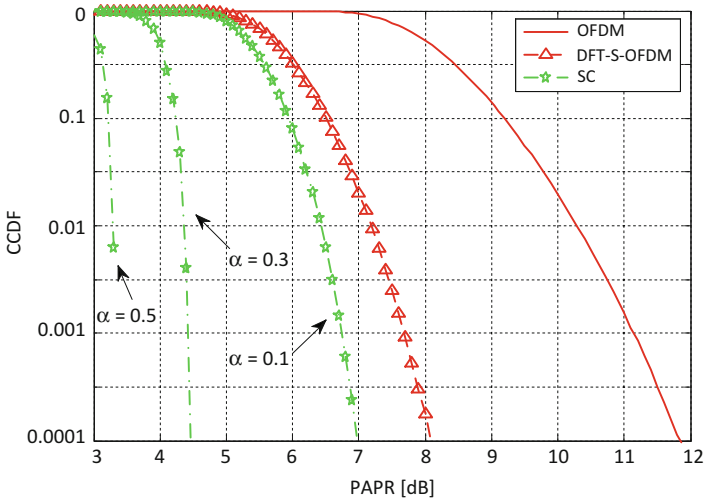


Fig. 7 Complementary cumulative distribution functions for the PAPR of OFDM, SC-FDMA, and single-carrier waveforms with different excess bandwidths. QPSK modulation, 160 subcarriers in OFDM and SC-FDMA. The roll-off parameter α controls the signal bandwidth as $(1 + \alpha)/T$, where T is the symbol interval in traditional SC-transmission

provided by the DFT-spread-OFDM processing satisfies the Nyquist criteria for zero ISI, the pulse shaping is sub-optimal and has small excess bandwidth. This leads to relatively high PAPR for SC-modulation, yet significantly smaller than in OFDM, as illustrated in Fig. 7. On the other hand, good spectral efficiency is achieved as different SC-FDMA blocks can be allocated next to each other without any guard-band in-between, as long as the conditions for quasi-synchronicity are maintained. Since the high PAPR of OFDM is mainly a problem on the mobile transmitter side, the SC-FDMA scheme is mainly considered for uplink transmission. An alternative implementation structure has been developed in [178], with additional flexibility for the DFT block size.

What was described above is the so-called contiguous subcarrier allocation case of SC-FDMA. Also a uniformly interleaved subcarrier allocation is possible, without any effects on the PAPR,[2] but has not been adopted in practice due to increased sensitivity to time selectivity, frequency offsets, and phase noise.

From the channel equalization point of view, the channel estimation and equalizer structure is the same as in the core OFDM system, except that scattered pilots cannot be utilized in SC-FDMA. From the SC-modulation point of view, the single-tap subcarrier equalizers correspond to a frequency-domain implementation of a linear equalizer [52, 145]. The MSE criterion is preferred over zero-forcing solution to

[2]This follows from the fact that uniform subcarrier interleaving corresponds to pulse repetition in time domain.

reduce the noise enhancement effects. The linear equalizer can be complemented with a decision-feedback structure. The noise prediction based DFE principle is particularly suitable for this configuration [23, 199], and including the FEC decoding in the DFE feedback loop leads to an effective iterative receiver structure with significantly improved performance over the linear equalizer solution.

Since SC-FDMA is based on a core OFDM system, various multiantenna schemes can be combined with it, including space-time and space-frequency block coding and spatial multiplexing [45, 164].

3.2.2 Enhancing Spectral Containment of OFDM

OFDM systems maintain orthogonality between spectral components which are synchronized in time and frequency to satisfy the quasi-synchronicity conditions. However, the spectral containment of the OFDM waveform is far from ideal (see Fig. 8), and the attenuation of a basic OFDM receiver for non-synchronized spectral components (interferences, adjacent channels) is limited.

Spectrum agile waveform processing is needed in case of various co-existence scenarios, where the idea is to use effectively frequency slots between channels occupied by legacy radio communication systems, as illustrated in Fig. 9. This is one central theme in the cognitive radio context [7] but also considered in various other developments of broadband wireless communications under concepts like carrier aggregation [37] and broadband-narrowband coexistence [131]. A very flexible way of approaching these goals can be named as non-contiguous multicarrier modulation, as a generalization of non-contiguous OFDM [194]. Here the idea is that the spectrum of the transmitted waveform can be controlled by activating only those subcarriers which are available and have been allocated for transmission, and modulating zero-symbols on the others. The approach is the same as the basic idea of OFDMA, but now the target is to be able to tolerate asynchronous waveforms in the unused frequency slots. Using basic OFDM in this way, the spectrum leakage would necessitate considerable guardbands between the active subcarriers and occupied frequency channels, and would thus lead to low spectrum efficiency.

The on-going 5th generation (5G) wireless cellular system development under 3GPP aims to create a multi-service network supporting a wide range of services with different requirements regarding data rate, latency, and reliability. These services include enhanced mobile broadband (eMBB) targeting at Gbps peak data rates, massive machine-type communications (mMTC) closely related to the Internet-of-things (IoT) concept, and ultra reliable low-latency communications (URLLC) needed, e.g., in the contexts of smart traffic, distant control of vehicles and industrial processes, and so-called tactile communications [150]. The 5G Phase 1 physical layer development in 3GPP, the so-called 5G New Radio, is also based on the OFDM waveform, but certain spectrum enhancement schemes can be applied to improve the quality of multi-service operation [20, 63]. Generally, it would be very difficult to satisfy the requirements of all the mentioned services by an OFDM

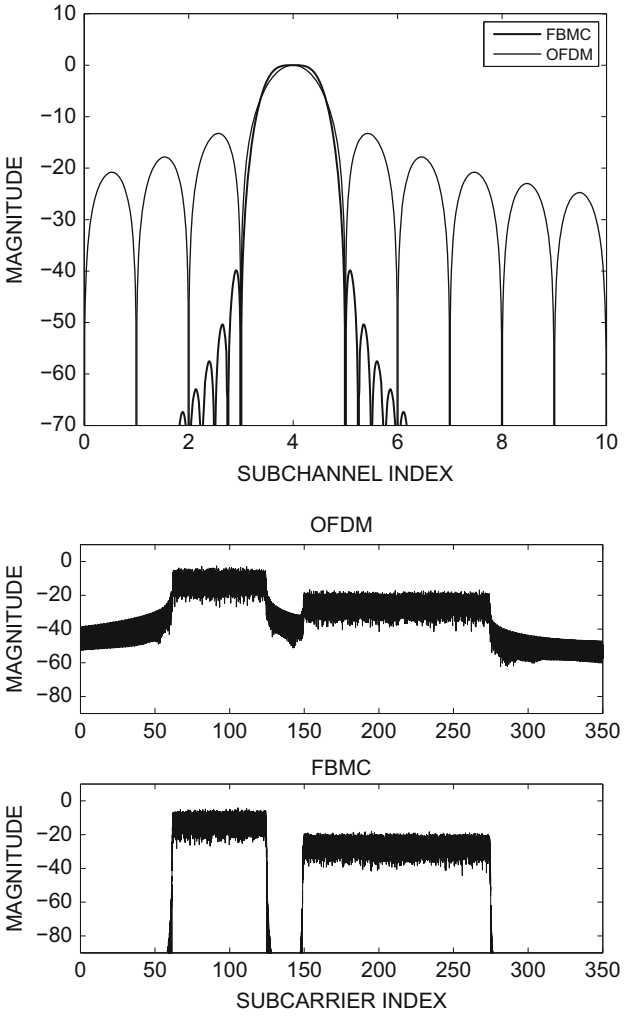


Fig. 8 OFDM and FBMC/OQAM spectra for individual subcarriers (top) and for the transmitted signal (bottom). Effects of nonlinearities are not included. The FBMC prototype filter design is from [179] with overlapping factor 4

system with fixed parametrization and, therefore, the concept of mixed numerology OFDM system has emerged. Here the idea is to utilize different subcarrier spacings and/or CP-lengths (guard periods) in different subbands of an OFDM carrier. However, this cannot be achieved without destroying the strict orthogonality of OFDM subcarriers. Then methods to reduce the OFDM spectral sidelobes are needed to be able to allocate groups of subcarriers with different numerologies in the same OFDM multiplex, with narrow guardband (few subcarriers) in-between, while keeping the interference leakage at an acceptable level.

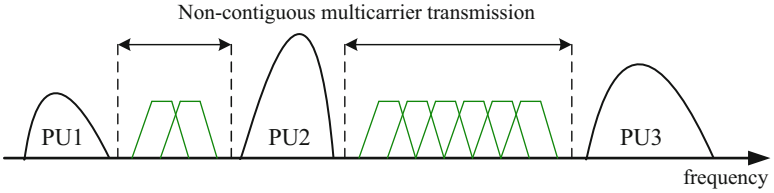


Fig. 9 Non-contiguous multicarrier transmission in spectrum gaps between primary users (PU's)

Another related aspect is that for sporadic low-rate multiuser uplink communication, the overhead to synchronize the devices for quasi-synchronous operation is significant. Then asynchronous operation mode, with relaxed time synchronization, would be preferred. Also in such scenarios, the strong sidelobes of basic OFDM is an issue. Notably, this aspect is relevant in OFDM based uplink, whereas the sidelobes issue is critical also in OFDMA downlink with mixed numerology.

Various techniques have been presented in the literature for reducing the spectral leakage in CP-OFDM-based systems. Two of these methods, time-domain windowing and OFDM subband filtering, are under consideration for 5G, and they will be discussed below in some more details. Other methods include subcarrier weighting [41], cancellation carrier methods [30, 103, 194], and pre-coding methods [36].

The general idea of time-domain windowing is to use a tapered time-domain window for OFDM symbols [18, 181], instead of rectangular windowing. Especially, raised cosine window in combination with extended CP has been widely considered. For effective spectrum leakage suppression, the CP has to be significantly extended to accommodate a higher roll-off of the RC-window (longer tapering interval), leading to reduced spectrum efficiency. Raised-cosine windowing can be used also on the receiver side for better rejection of interference leakage from the unused spectral slots [18, 116], with similar tradeoffs. In [103, 142], it is proposed to use the windowing in edge subcarriers only to improve spectrum efficiency. In the 5G New Radio context, time-domain windowing is referred to as windowed-overlap add (WOLA) [129, 195], and it is considered be applied in the transmitter, receiver, or both.

Another obvious alternative to control OFDM spectrum is the filtering of independently generated groups of subcarriers, typically by FIR filters, before combining them as the OFDM multiplex signal to be provided to the DAC and RF stages of the transmitter [6, 54, 97, 106, 146, 187]. On the receiver side, channelization filtering can be done separately for different groups of subcarriers to reduce leakage from adjacent asynchronously operated subcarrier groups, or groups with different numerologies. This general idea is referred to as filtered OFDM (F-OFDM). One related target in 5G is to reduce the spectral overhead due to guardbands between active transmission channels from 10% to about 1%. Together with increasing carrier bandwidth (e.g., 100 MHz instead of 20 MHz in LTE), this leads to high complexity of traditional FIR-type channelization filters. The general

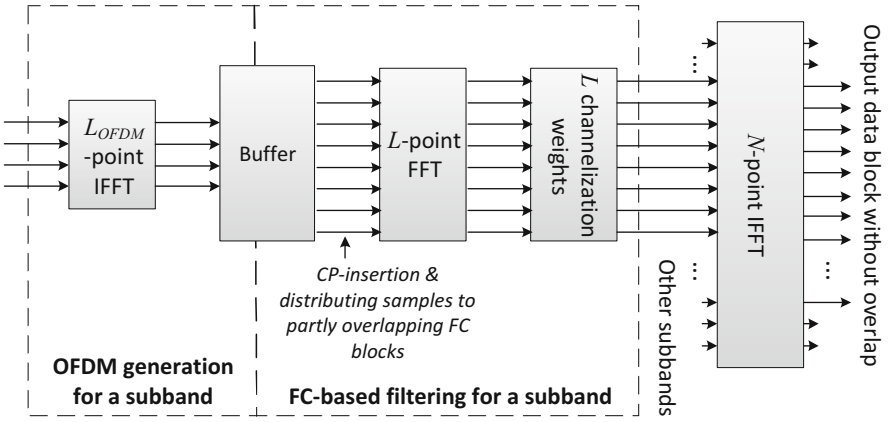


Fig. 10 Fast-convolution filtered OFDM transmitter structure

target of F-OFDM is to support flexible allocation of different numerologies in a single OFDM multiplex, in which case traditional digital filtering solutions would have high structural and computational complexity. This is especially the case on the base-station side, while mobile devices typically need to process only one subband, and basic time-domain filtering with reasonable complexity and sufficient flexibility is achievable [187].

An alternative approach to subband filtering by individual filters is to use uniform filter banks for combining filtered subbands on the transmitter side and for separating filtered subbands on the receiver side [97]. In case of regular subband structure, this would be a very effective approach, but it has limited flexibility for dynamic adaptation of the subband widths.

The third approach is to define the filtering in FFT-domain, using the fast-convolution approach [28, 122, 136]. Figure 10 illustrates this scheme for the transmitter side [187]. First CP-OFDM signals are generated individually for each subcarrier group which needs to be isolated by filtering. Then short FFTs are applied to partly overlapping blocks of the CP-OFDM signal. The filter is defined by FFT-domain weights, and the output signal is generated by long IFFTs. The output sample sequence is obtained by collecting non-overlapping samples from the IFFT output blocks. This model utilizes fast-convolution with overlap-save processing to implement linear convolution by the FFT-domain filtering process, which implements cyclic convolution by nature. With sufficiently long overlap, perfect linear convolution would be reached. However, by allowing tolerable amount of distortion, the overlap can be significantly reduced, resulting in remarkable reduction in the computational complexity. Typical values of the overlap are 25–50%. In case of multiple filtered subbands, the CP-OFDM generation, short FFT, and FFT-domain weights are specific to each subband, but the long IFFT is common to all. A narrow guardband (e.g., 1–6 subcarriers) is inserted between active subcarriers of different groups.

Tight filtering harms the orthogonality of subcarriers in all F-OFDM schemes, introducing inband interference especially to the subcarriers close to subband edges [54, 106]. Effective FFT-domain weight optimization scheme is presented in [187] for minimizing the inband interference under constraints on the out-of-band power leakage. This optimization methods takes into account both the filtering effect on OFDM subcarriers, as well as the cyclic distortion caused by the reduced overlap in fast-convolution processing.

3.2.3 Filterbank Multicarrier Waveforms

Another approach for spectrally agile waveforms and signal processing is filter bank based multicarrier modulation (FBMC) [35, 51, 53, 75, 125, 143, 153]. Here the idea is to use spectrally well-contained synthesis and analysis filter banks in the transmultiplexer configuration, instead of the IFFT and FFT, respectively. The most common approach is to use modulated uniform polyphase filter banks based on a prototype filter design, which determines the spectral containment characteristics of the system. Figure 8 shows an example of the resulting spectral characteristics, in comparison with basic OFDM without any additional measures for controlling the sidelobes. It can be seen that the FBMC is able to reduce the sidelobes to a level which depends in practice only on the spectral leakage (spectral regrowth) resulting from the transmitter power amplifier nonlinearities.

The two basic alternatives are filtered multitone modulation (FMT) [38, 174] and FBMC/OQAM (or OFDM/OQAM) [53, 153]. In typical FBMC/OQAM designs (like the example case of Fig. 8), each subchannel overlaps with the adjacent ones, but not with the more distant ones, and orthogonality of subcarriers is achieved by using offset-QAM modulation of subcarriers, in a specific fashion [153]. Due to the absence of cyclic prefix and reduced guard-bands in frequency domain, FBMC/OQAM reaches somewhat higher spectral efficiency than CP-OFDM [137]. However, its main benefits can be found in scenarios with asynchronous multiuser operation, mixed numerology, or dynamic and non-contiguous (i.e., fragmented) spectrum allocation [149, 196]. Its main drawbacks are due to the need to use offset (staggered) QAM modulation, leading to somewhat more complicated pilot structures for synchronization and channel estimation. OQAM signal structure causes also difficulties with certain multiantenna transmission schemes, especially with Alamouti space-time coding [133]. FBMC/OQAM has also higher computational complexity, which in terms of real multiplication rate, is three to five times that of OFDM with the same transform size [19, 63].

In FMT, the adjacent subchannels are isolated by designing them to have non-overlapping transition bands and, for each subcarrier, basic subcarrier modulation, like QAM with Nyquist pulse shaping, can be used. The principle of FMT is just frequency division multiplexing/multiple access. It relies on specific uniform multirate filter bank structures, typically based on IFFT/FFT transforms complemented by polyphase filtering structures. To reach high spectral efficiency, narrow transition bands should be used, leading to increased latency and high implementation complexity, also in comparison with FBMC/OQAM.

Both FBMC/OQAM and FMT systems can be designed to have similar number of subcarriers as an OFDM system, in which case the channel can usually be considered as flat-fading at subcarrier level, and one-tap complex subcarrier-wise channel equalizers are sufficient. However, there is also the possibility to increase the subcarrier spacing, e.g., in order to relax the ICI effects with high mobility, in which case multi-tap equalizers are needed [75]. A convenient approach for realizing multitap subcarrier equalizers is based on frequency sampling [80]. The special OQAM-type signal structure has to be taken into account when designing the pilot structures for channel estimation and synchronization [96], and it introduces also difficulties in adapting certain multiantenna schemes to the FBMC/OQAM context.

Fast-convolution based filterbank (FC-FB) schemes have been proposed also for flexible and effective implementation of FBMC/OQAM and FMT waveform processing. Actually, FC-FB can be seen as a generic waveform processing engine, facilitating simultaneous processing of different multicarrier and single-carrier waveforms [132, 135, 136, 152].

In recent years, also a family of multicarrier waveforms which apply CPs for blocks of multicarrier symbols has been introduced. These include generalized frequency division multiplexing (GFDM) [63, 111], Cyclic Block-Filtered MultiTone (CB-FMT) [65], and Circular Offset Quadrature Amplitude Modulation (COQAM) [99]. The CP-insertion works basically in the same way as with CP-OFDM, but since CP is applied for a block of P multicarrier symbols (i.e., PN high-rate samples), the CP-overhead can be greatly reduced for a given channel delay spread. GFDM uses QAM subcarrier modulation with filtered subcarrier signals spaced at $1/T_S$, leading to non-orthogonal subcarriers. Therefore, some form of ICI cancellation is required, at least for high-order modulations. CB-FMT is cyclic block-filtered variant of FMT, maintaining orthogonality of subcarriers. COQAM uses OQAM subcarrier modulation, as in FBMC/OQAM, also maintaining subcarrier orthogonality. In basic form, all these waveforms apply rectangular window over the block of multicarrier symbols, resulting in sinc-type spectra. Since the rectangular window length is increased in time, the sidelobes decay faster. Well-contained spectra have been demonstrated for these waveforms by applying sidelobe suppression methods introduced earlier for the OFDM case, in somewhat relaxed ways. Also effective realizations for these schemes are available, based FFT-domain filtering using cyclic convolution (i.e., FC without overlap).

In summary, FBMC and enhanced OFDM schemes are alternative approaches for developing flexible spectrum agile waveforms with improved spectral containment, which is particularly important in fragmented spectrum use, asynchronous multiuser operation, or mixed numerology cases.

4 Transceiver RF System Fundamentals and I/Q Signal Processing

This section looks at radio transceiver fundamentals from a broader perspective, by considering also the essentials of analog radio frequency (RF) functionalities in addition to digital front-end and digital baseband aspects described in the previous sections. Overall, understanding the RF world is one central aspect in radio communications since the energy of the true electromagnetic waves radiated and absorbed by the antennas, and thus the spectral contents of the underlying electrical signals, are indeed located at radio frequencies. Depending on the actual radio system and radio application, the used RF band is typically within the range of few tens or hundreds of MHz up to several GHz.

In this section, we'll go through the basics of transceiver signal processing from radio architecture perspective, with main focus on frequency translations and filtering tasks. The exact circuit-level treatments are out of our scope, and we focus on signal and RF-module level aspects only. One central tool in the presentation is the deployment of complex-valued I/Q signal and processing models, especially in the frequency translation and filtering tasks. In addition to RF front-end, the notion of complex-valued I/Q signals is central also in the digital front-end and baseband processing units as is evident from the presentation in the previous sections which all rely on complex-valued signals. Some classical literature in this field are, e.g., [44, 57, 60, 107, 109, 112]. Some sections in the following also build on the presentation of [170].

4.1 RF-System Fundamentals

The fundamental tasks of transmitter RF front-end are to upconvert the data-modulated communication waveform to the desired RF (carrier) frequency and produce the needed RF power to the transmit signal. How these are exactly organized and implemented in the full transmitter chain, depends on the chosen radio architecture. Independently of this, the transmitter performance is typically measured in terms of spectral purity or spectral mask which dictates how much energy the transmitter can leak outside its own frequency band. Such out of band emissions can stem, e.g., from transmit chain nonlinearities and/or insufficient filtering. Another important aspect is the in-band purity of the RF waveform which quantifies the waveform generation accuracy from the data modulation and transmission point of view. One typically deployed measure here is the error vector magnitude (EVM).

On the receiver side, the key tasks of the RF front-end are to amplify the weak received desired signal, downconvert the desired signal from RF down to lower frequencies, and to at least partially attenuate the undesired other radio signals picked up by the antenna. Again, the chosen radio architecture has a big influence on

how these tasks are implemented in the receiver chain. In general, one can perhaps claim that the implementation challenges on receiver side are typically even bigger than on the transmitter side. This is indeed because the antenna is picking up also many other radio signals, in addition to the desired one, which can also be several tens of dB's stronger than the desired one. Thus being able to demodulate and detect a weak desired signal in the presence of strong neighboring channels is indeed a complicated task. The receiver front-end performance is typically measured, e.g., in terms of sensitivity, linearity and spurious free dynamic range. In short, sensitivity measures the ability to detect very weak signals in noise-limited scenarios. Linearity and spurious-free dynamic range, in turn, measure the relative levels of spurious components stemming from the intermodulation of the strong neighboring channels and out-of-band blocking signals, falling on top of the desired signal band. Measures like input-intercept point (IIP, specifically IIP2 and IIP3 for second-order and third-order nonlinearities, respectively) are typically used to measure receiver linearity.

4.2 Complex I/Q Signal Processing Fundamentals

4.2.1 Basic Definitions and Connection to Bandpass Signals

All physical signals and waveforms, like voltage or current as a function of time, are by definition real-valued. However, when modeling, analyzing and processing bandpass signals whose spectral content is located around some center-frequency f_c, the use and notion of complex-valued signals turns out to be very useful. This has then direct applications in radio communications, like various complex modulation methods and more generally different frequency translations and filtering methods in transceiver analog and digital front-ends. This is where we have main emphasis on in this section. Furthermore, complex-valued signal and processing models are fundamental also in digital baseband processing, including e.g. modeling of radio channel impacts on the modulating data and the resulting equalization and detection processing in receiver baseband parts. Examples of this can be found from earlier sections. Useful general literature in this field are, e.g., [68, 107, 166, 170].

By definition, the time domain waveform $x(t)$ of a complex signal is complex-valued, i.e.

$$x(t) = x_{\mathrm{I}}(t) + jx_{\mathrm{Q}}(t) = \Re\left[x(t)\right] + j\Im\left[x(t)\right] \tag{25}$$

In practice, this is nothing more than a pair of two real-valued signals $x_{\mathrm{I}}(t)$ and $x_{\mathrm{Q}}(t)$ carrying the real and imaginary parts. Similarly, a complex linear system is defined as a system with complex-valued impulse response

$$h(t) = h_{\mathrm{I}}(t) + jh_{\mathrm{Q}}(t) = \Re\left[h(t)\right] + j\Im\left[h(t)\right] \tag{26}$$

One of the beautiful properties of complex-valued models is that in frequency domain, there are no symmetry constraints opposed to real-valued signals/systems which are always forced to have even-symmetric amplitude spectrum/response and odd-symmetric phase spectrum/response with respect to the zero frequency in two-sided spectral analysis. In the following presentation, we focus mostly on continuous-time waveform and system aspects, but similar concept carry on to discrete-time world as well. Some additional digital filter specific aspects are also addressed in Sect. 4.3.2.

One basic operation related to complex quantities is complex-conjugation. Now if the spectrum (Fourier transform) of $x(t)$ is denoted by $X(f)$, then the spectrum of complex-conjugated signal $x^*(t)$ is $X^*(-f)$. This implies that the amplitude spectra of $x(t)$ and $x^*(t)$ are mirror images of each other. Notice that physically, complex conjugation is nothing more than changing the sign of the Q branch signal. This simple result related to conjugation has an immediate consequence that if one considers only the real part of $x(t)$, i.e., $y(t) = \Re[x(t)] = (x(t) + x^*(t))/2$, its spectrum is $Y(f) = (X(f) + X^*(-f))/2$. Now if $X(f)$ and $X^*(-f)$ are not overlapping, $y(t) = \Re[x(t)]$ contains all the information about $x(t)$. Based on this, it directly follows that for any complex signal $x(t)$ such that $X(f)$ and $X^*(-f)$ are not overlapping, $y(t) = \Re[x(t)]$ contains all the information about $x(t)$.

The notion of complex signals has strong connection to bandpass signals. By definition, a general real-valued bandpass signal can be written as

$$\begin{aligned} v_{\mathrm{BP}}(t) &= A(t)\cos(2\pi f_{\mathrm{c}} t + \phi(t)) = v_{\mathrm{I}}(t)cos(2\pi f_{\mathrm{c}} t) - v_{\mathrm{Q}}(t)\sin(2\pi f_{\mathrm{c}} t) \\ &= \Re\left[v_{\mathrm{LP}}(t)e^{j2\pi f_{\mathrm{c}} t}\right] = \frac{v_{\mathrm{LP}}(t)e^{j2\pi f_{\mathrm{c}} t} + v_{\mathrm{LP}}^*(t)e^{-j2\pi f_{\mathrm{c}} t}}{2} \end{aligned} \tag{27}$$

where $v_{\mathrm{LP}}(t) = v_{\mathrm{I}}(t) + jv_{\mathrm{Q}}(t) = A(t)e^{j\phi(t)}$ is the corresponding lowpass or baseband equivalent signal, $v_{\mathrm{I}}(t)$ and $v_{\mathrm{Q}}(t)$ are the inphase (I) and quadrature (Q) components, and $A(t)$ and $\phi(t)$ denote envelope and phase functions. Principal spectral characteristics are illustrated in Fig. 11. Thus in the general case, the baseband equivalent of a real-valued bandpass signal is complex-valued. Intuitively, the complex-valued baseband equivalent describes the oscillating physical bandpass signal with a time-varying phasor (complex number at any given time) such that the length of the phasor corresponds to physical envelope and the phase to the physical phase characteristics.

Two basic operations related to processing of complex signals are (1) complex multiplication and (2) complex convolution (filtering). In the general case, by simply following the complex arithmetic, these can be written as

$$\begin{aligned} x(t) \times y(t) &= \left(x_{\mathrm{I}}(t) + jx_{\mathrm{Q}}(t)\right) \times \left(y_{\mathrm{I}}(t) + jy_{\mathrm{Q}}(t)\right) \\ &= x_{\mathrm{I}}(t) \times y_{\mathrm{I}}(t) - x_{\mathrm{Q}}(t) \times y_{\mathrm{Q}}(t) + j\left(x_{\mathrm{I}}(t) \times y_{\mathrm{Q}}(t) + x_{\mathrm{Q}}(t) \times y_{\mathrm{I}}(t)\right) \end{aligned} \tag{28}$$

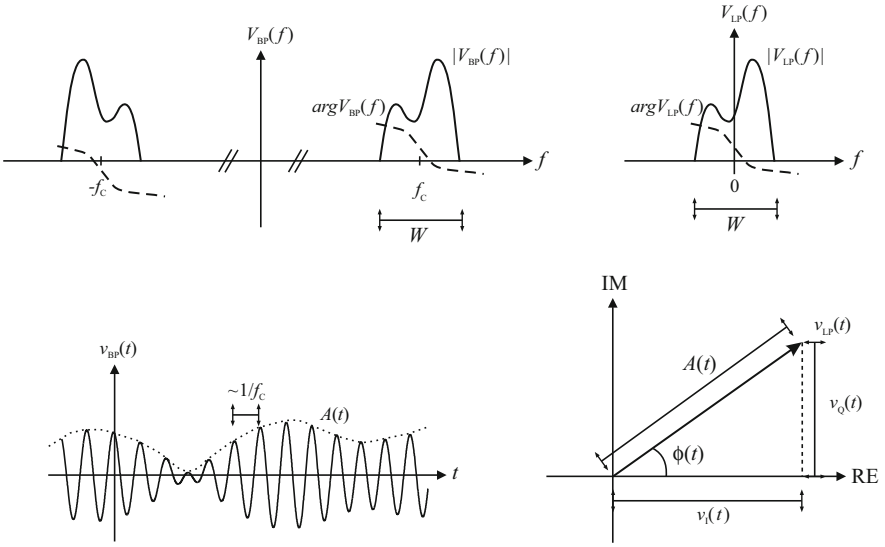


Fig. 11 Illustration of bandpass signal structure in time- and frequency domains. Left half shows a principal bandpass signal spectrum and the corresponding time-domain waveform. Right half, in turn, shows the corresponding lowpass equivalent signal spectrum and the corresponding time-domain complex signal as a time-varying phasor in complex plane

$$\begin{aligned} x(t) * h(t) &= \left(x_{\mathrm{I}}(t) + j x_{\mathrm{Q}}(t)\right) * \left(h_{\mathrm{I}}(t) + j h_{\mathrm{Q}}(t)\right) \\ &= x_{\mathrm{I}}(t) * h_{\mathrm{I}}(t) - x_{\mathrm{Q}}(t) * h_{\mathrm{Q}}(t) + j\left(x_{\mathrm{I}}(t) * h_{\mathrm{Q}}(t) + x_{\mathrm{Q}}(t) * h_{\mathrm{I}}(t)\right) \end{aligned} \tag{29}$$

Thus in general, four real multiplications (plus two additions) and four real convolutions (plus two additions) are needed, respectively, in the physical implementations. This is illustrated in Fig. 12 for general complex convolution. Obvious simplifications occur if either of the components (input signal or filter impulse response) is real valued.

4.2.2 Analytic Signals and Hilbert Transforms

Hilbert transformer [68] is generally defined as an allpass linear filter which shifts the phase of its input signal by 90°. In the continuous-time case, the (non-causal) impulse and frequency responses can be formulated as

$$h_{\mathrm{HT}}(t) = \frac{1}{\pi t} \tag{30}$$

$$H_{\mathrm{HT}}(t) = \begin{cases} -j, & f > 0 \\ +j, & f < 0 \end{cases} \tag{31}$$

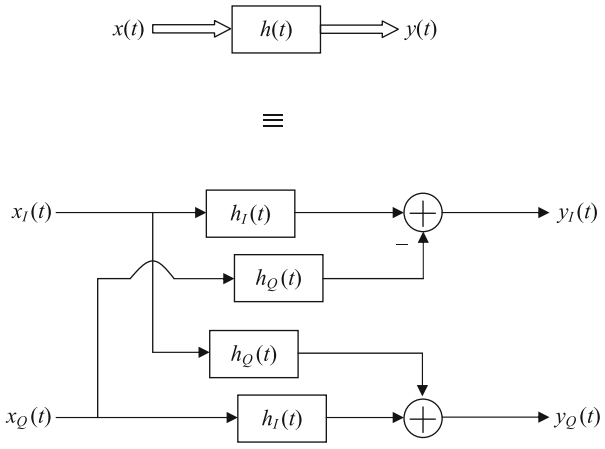


Fig. 12 Illustration of complex filtering (complex convolution) in terms of complex signals (upper) and parallel real signals (lower)

Similar concepts carry on also to discrete-time filters [122].

In practice, the above behavior can be well approximated over any finite bandwidth. One fascinating property related to Hilbert filters/transformers is that they can be used to construct signals with only positive or negative frequency content. These kind of signals are generally termed analytic signals and they are always complex-valued. The simplest example is to take a cosine wave $A\cos(\omega_0 t)$ whose Hilbert transform is $A\sin(\omega_0 t)$. Then these together when interpreted as I and Q components of a complex signal result in $A\cos(\omega_0 t) + jA\sin(\omega_0 t) = Ae^{j\omega_0 t}$ whose spectrum has an impulse at ω_0 (but not at $-\omega_0$). The elimination of the negative (or positive) frequencies can more generally be formulated as follows. Starting from an arbitrary signal $x(t)$ we form a complex signal $x(t) + jx_{\mathrm{HT}}(t)$ where $x_{\mathrm{HT}}(t)$ denotes the Hilbert transform of $x(t)$. This is illustrated in Fig. 13. In practice a proper delay is needed in the upper branch to facilitate the delay of a practical HT. Then the spectrum of the complex signal is $X(f) + jX_{\mathrm{HT}}(f) = X(f)[1 + jH_{\mathrm{HT}}(f)]$ where $1 + jH_{\mathrm{HT}}(f) = 0$ for $f < 0$. Based on this, it can easily be shown that the I and Q (real and imaginary parts) of any analytic signal are always related through Hilbert transform.

4.3 Frequency Translations and Filtering

4.3.1 Frequency Translations for Signals

One key operation in radio signal processing is the shifting of a signal spectrum from one center-frequency to another. Conversions between baseband and bandpass representations and I/Q modulation and demodulation (synchronous detection) are

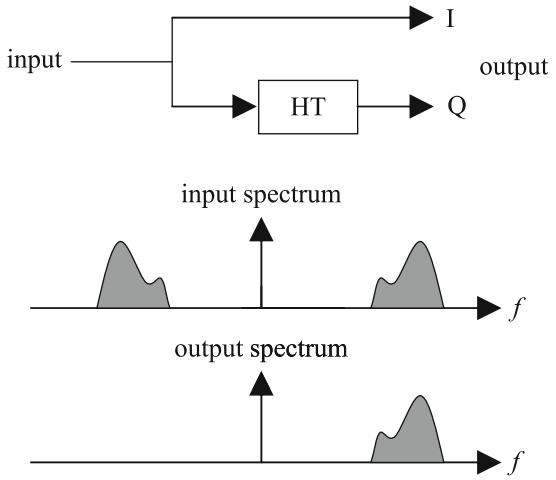


Fig. 13 Illustration of creating analytic signal using a Hilbert transformer

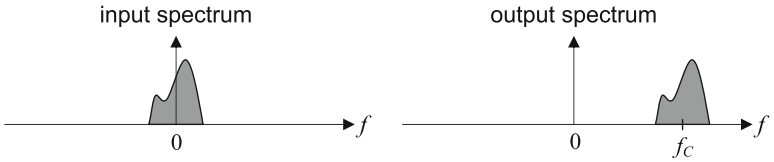


Fig. 14 An example of pure frequency translation using complex mixing

special cases of this. The basis of all the frequency translations lies in multiplying a signal with a complex exponential, generally referred to as complex or I/Q mixing. This will indeed cause a pure frequency shift, i.e.,

$$y(t) = x(t)e^{j\omega_{\mathrm{LO}}t} \quad \Leftrightarrow \quad Y(f) = X(f - f_{\mathrm{LO}}) \tag{32}$$

where $\Leftrightarrow$ denotes transforming between time and frequency domain. This forms the basis, e.g., for all the linear modulations, and more generally for all frequency translations. This is illustrated in frequency domain in Fig. 14 in the case where the input signal is at baseband.

In general, since

$$\begin{aligned} x(t)e^{j\omega_{\mathrm{LO}}t} &= x_{\mathrm{I}}(t)\cos(\omega_{\mathrm{LO}}t) - x_{\mathrm{Q}}(t)\sin(\omega_{\mathrm{LO}}t) \\ &\quad + j\left(x_{\mathrm{Q}}(t)\cos(\omega_{\mathrm{LO}}t) + x_{\mathrm{I}}(t)\sin(\omega_{\mathrm{LO}}t)\right), \end{aligned} \tag{33}$$

four real mixers and two adders are needed to implement a full complex mixer (full complex multiplication). This illustrated in Fig. 15. Notice again that in the special case of real-valued input signal, only two mixers are needed.

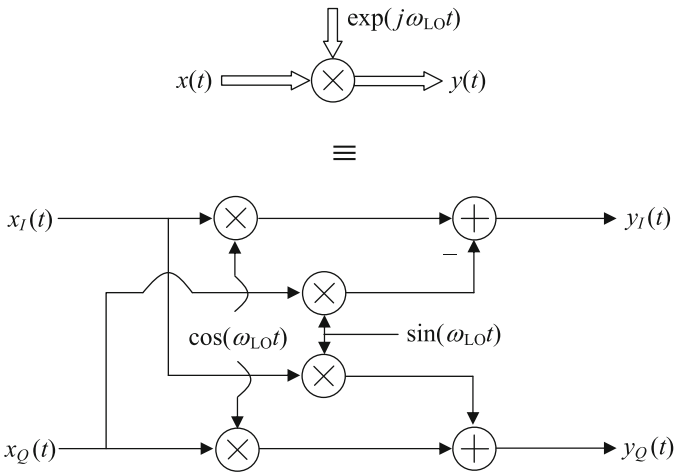


Fig. 15 Illustration of complex mixing (complex signal multiplication) in terms of complex signals (upper) and parallel real signals (lower)

Real mixing is obviously a special case of the previous complex one and results in two frequency translations:

$$\begin{aligned} y(t) &= x(t)\cos(\omega_{\mathrm{LO}}t) \\ &= x(t)\frac{1}{2}\left(e^{j\omega_{\mathrm{LO}}} + e^{-j\omega_{\mathrm{LO}}t}\right) \Leftrightarrow Y(f) = \frac{1}{2}X(f - f_{\mathrm{LO}}) + \frac{1}{2}X(f + f_{\mathrm{LO}}) \end{aligned} \tag{34}$$

Here, the original spectrum appears twice in the mixer output, the two replicas being separated by $2f_{\mathrm{LO}}$ in frequency. In receivers, this results in the so called image signal or mirror-frequency problem since the signals from both $f_{\mathrm{c}} + f_{\mathrm{LO}}$ and $f_{\mathrm{c}} - f_{\mathrm{LO}}$ will appear at f_{c} after a real mixing stage. Thus if real mixing is used in the receiver, the image signal or mirror-frequency band needs to be attenuated before the actual mixer stage. This is the case, e.g., in the classical superheterodyne receiver. Similar effects have to be taken into consideration also in transmitters, meaning that the unwanted spectral replica produced by real mixing needs to be attenuated.

Linear I/Q modulation methods are basically just a special case of complex mixing. Given a complex message signal $x(t) = x_{\mathrm{I}}(t) + jx_{\mathrm{Q}}(t)$, it is first complex-modulated as $x(t)e^{j\omega_{\mathrm{c}}t}$, after which only the real part is actually transmitted. This can be written as

$$\begin{aligned} y(t) &= \Re\left[x(t)e^{j\omega_{\mathrm{C}}t}\right] = x_{\mathrm{I}}(t)\cos(\omega_{\mathrm{c}}t) - x_{\mathrm{Q}}(t)\sin(\omega_{\mathrm{c}}t) \\ &= \frac{1}{2}x(t)e^{j\omega_{\mathrm{C}}t} + \frac{1}{2}x^{*}(t)e^{-j\omega_{\mathrm{C}}t} \end{aligned} \tag{35}$$

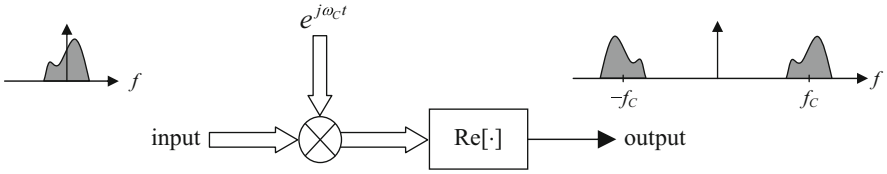


Fig. 16 Principal structure of I/Q modulation using complex signal notations

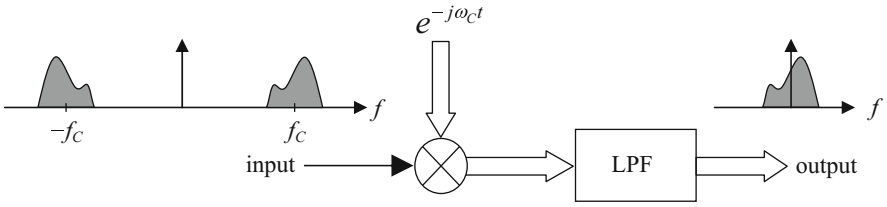


Fig. 17 Principal structure of I/Q demodulation using complex signal notations

While physical implementations build on the middle expression where $x_{\mathrm{I}}(t)$ and $x_{\mathrm{Q}}(t)$ are modulated onto two orthogonal (cosine and sine) carriers, the complex models are very handy e.g. from spectral analysis point of view. Notice that both terms or spectral components (at $+f_{\mathrm{c}}$ and $-f_{\mathrm{c}}$) contain all the original information (i.e., $x(t)$). This overall process, also termed lowpass-to-bandpass transformation, is pictured at conceptual level in Fig. 16.

On the receiver side, the goal in the demodulation phase is to recover the original message $x(t)$ from the carrier-modulated signal $y(t)$. Based on the previous discussion, it's easy to understand that either of the signal components at $+f_{\mathrm{c}}$ or $-f_{\mathrm{c}}$ can be used for that purpose, while the other one should be rejected. Since

$$y(t)e^{-j\omega_{\mathrm{c}}t} = \left(\frac{1}{2}x(t)e^{j\omega_{\mathrm{c}}t} + \frac{1}{2}x^*(t)e^{-j\omega_{\mathrm{c}}t}\right)e^{-j\omega_{\mathrm{c}}t} = \frac{1}{2}x(t) + \frac{1}{2}x^*(t)e^{-j2\omega_{\mathrm{c}}t} \tag{36}$$

the message $x(t)$ can be fully recovered by simply lowpass filtering the complex receiver mixer output. Practical implementation builds again on parallel real downconversion with cosine and sine followed by lowpass filtering in both branches. Formal block-diagram for the I/Q demodulator in terms of complex signals is presented in Fig. 17.

4.3.2 Frequency Translations for Linear Systems and Filters

The idea of frequency translations can be applied not only to signals but linear systems or filters as well [122]. Good example is bandpass filter design through proper modulation of lowpass prototype filter. In other words, assuming a digital

filter with impulse response $h(n)$, modulated filter coefficients are of the form $h(n)e^{j\omega_0 n}$, $h(n)\cos(\omega_0 n)$, and/or $h(n)sin(\omega_0 n)$ which have frequency-shifted or modulated frequency responses compared to $h(n)$. In general, such frequency translation principles apply to both analog and digital filters but our focus in the notations here is mostly on digital filters. Notice also that analytic bandpass filters of the form $h(n)e^{j\omega_0 n}$ has direct connection to Hilbert transforms.

When it comes to digital filters, very interesting and low-complexity transforms are obtained when the modulating sequence is either $e^{j\pi n} = \{\ldots, +1, -1, +1, -1, +1, -1, \ldots\}$ or $e^{j\frac{\pi}{2}n} = \{\ldots, +1, +j, -1, -j, +1, +j, -1, -j, \ldots\}$ which correspond to frequency translation by $f_s/2$ and $f_s/4$, respectively. Implementation-wise, these are close to trivial mappings (only sign changes and proper changes between I and Q branch sequences) which means very efficient implementation. This applies of course also to digital downconversion and demodulation as well which is one reason why $f_s/4$ is a popular choice for intermediate frequency (IF) in many advanced receivers. Notice also that in general, coefficient symmetry can be exploited in modulated filter implementation as long as the prototype filter $h(n)$ is symmetric.

One additional key property is obtained from the transfer function interpretation of modulated complex filters. For $H(z) = \sum_{n=0}^{N} h(n)z^{-n}$, we can write

$$\sum_{n=0}^{N} \left(h(n)e^{j\omega_0 n}\right) z^{-n} = \sum_{n=0}^{N} h(n) \left(z^{-1}e^{j\omega_0}\right)^n = H(z)|_{z^{-1} \leftarrow z^{-1}e^{j\omega_0}} \tag{37}$$

This means that the modulated filter can also be implemented by simply replacing the unit delays (z^{-1} elements) of the original filter with generalized elements $z^{-1}e^{j\omega_0}$. Thus implementing frequency translations is very straight-forward also for IIR type filters.

We illustrate the modulated FIR filter characteristics with a design example where analytic bandpass filter is obtained through complex modulation. Target is to have passband at $0.6\pi \ldots 0.8\pi$ and the filter length is 50. Equiripple (Remez) design is used, and the lowpass prototype is an ordinary LPF with passband $-0.1\pi \ldots 0.1\pi$. Then complex modulation with $e^{j0.7\pi n}$ is deployed. The results are illustrated in Fig. 18.

After learning that we can generally build complex (analytic) bandpass filters, it's also easy to devise an alternative strategy, other than the classical scheme with complex down-conversion and lowpass filtering, for I/Q demodulation. This is illustrated in Fig. 19, and uses the idea of filtering the signal first with complex bandpass filter after which complex downconversion takes place. Notice that in this scheme the complex bandpass filter creates already complex output signal and thus a true complex mixer is required (4 muls and 2 adds). This structure has, however, some benefits e.g. from analysis point of view, and it is also very suitable for digital I/Q demodulation combined with decimation/down-sampling since the complex filter output is free from negative frequencies.

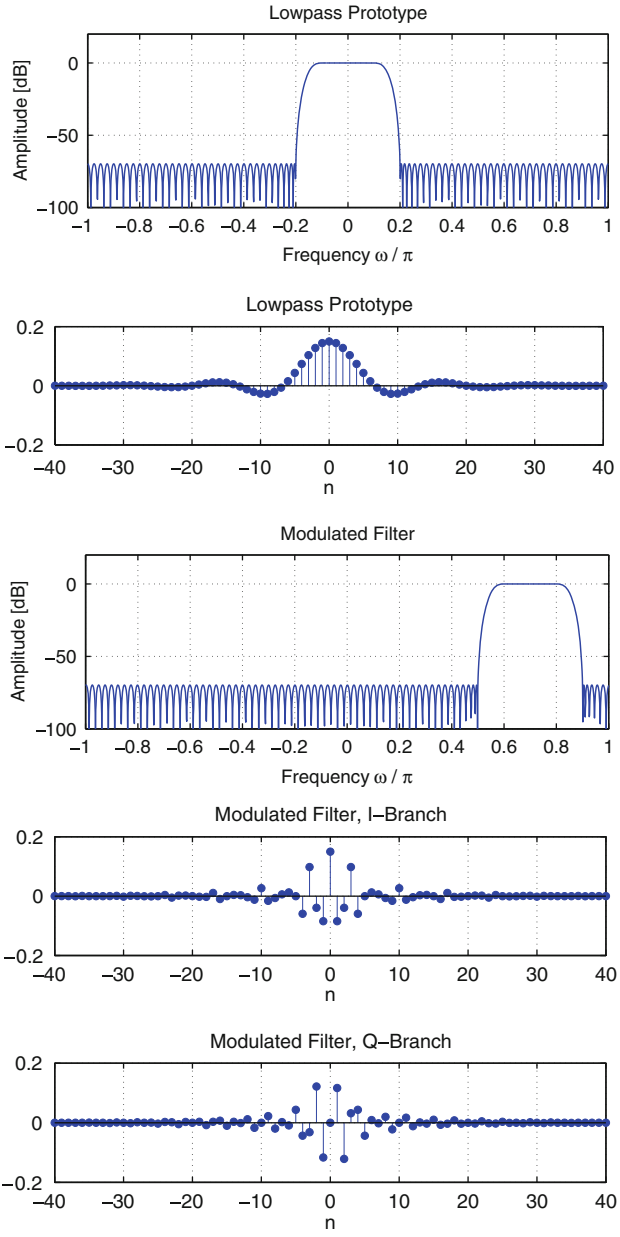


Fig. 18 An illustration of analytic bandpass filter generation through complex modulation of a lowpass prototype

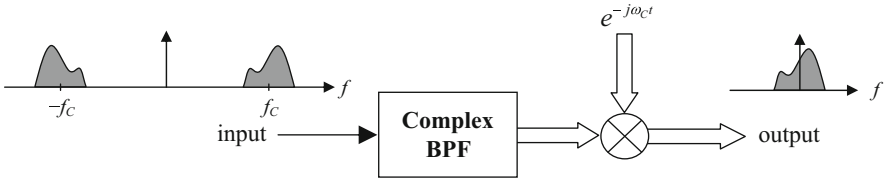


Fig. 19 An alternative structure for I/Q demodulation using complex bandpass filtering and complex downconversion

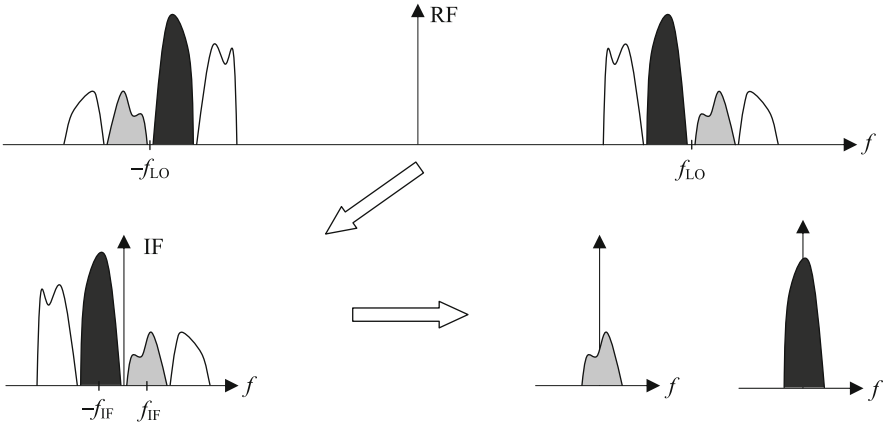


Fig. 20 A principal spectral illustration of two-carrier low-IF receiver principle using wideband complex I/Q downconversion

Additional good example of applying complex signal processing tools in radio transceivers is, e.g., a dual-carrier or dual-channel receiver in which the RF front-end implements wideband I/Q downconversion of the received signal such that the two interesting carriers are located at positive and negative (small) intermediate frequencies (IFs) after the analog front-end. The signal is then sampled and the two carriers are demodulated in parallel in the digital front-end to baseband for equalization and detection purposes. This is conceptually illustrated in Fig. 20. Now there are two possibilities how to implement the carrier separation and demodulation in the digital front-end: (1) complex digital bandpass filters centered at positive and negative IFs, respectively, followed by complex digital downconversions or (2) complex digital downconversions from positive and negative IFs to baseband (in parallel) and real digital lowpass filtering for both signals. In practice, this is also accompanied with sample rate adaptation (decimation).

4.4 Radio Architecture Basics

In general the term radio architecture refers to the communication circuit and module level arrangements in radio devices, and especially to how the elementary tasks like frequency translations, filtering and amplification are organized and sequenced in the radio chain. For presentation purposes we focus here on the receiver side, while many of the principles and observations are valid also on the transmitter side. There are also many transmitter-specific architectures, like polar transmitter and other envelope/phase oriented structures, which focus specifically on limiting the peak-to-average power ratio (PAPR) at the power amplifier input or improving the PA power efficiency.

Theoretically, on the receiver side, the desired frequency channel could be selected from the received radio frequency (RF) signal using a tunable and highly-selective bandpass filter. This is, however, not feasible in practice since the used RF bands are commonly in the GHz range while the interesting or desired signal is typically very narrowband compared to the center-frequency. Therefore, the received signal is downconverted to lower frequencies, either intermediate frequency (IF) or directly to baseband, where selectivity filtering and other processing can be implemented in a more feasible manner. Below we review how such frequency translations and filtering are implemented in the most typical receiver structures, namely superheterodyne, direct-conversion and low-IF type receivers. Useful general literature is this field are, e.g., [44, 105, 112]. We also shortly touch the subsampling aspects [42, 176] where controlled aliasing, instead of explicit mixing, is used for frequency translation. As in the whole communications signal processing field, the concept of complex-valued or I/Q signals plays an essential role also here in designing and understanding different receiver principles.

4.4.1 Superheterodyne Receiver

The previously-described real mixing approach is deployed in the traditional superheterodyne receiver. A tunable local oscillator is used to select the channel of interest which is translated to a fixed intermediate frequency using real mixing. At the IF stage, a highly selective bandpass filter is used to separate the desired channel signal from the others. Tunability in the local oscillator facilitates the use of a fixed intermediate frequency, thus enabling efficient implementation of the IF channel selection filter. Special analog filter technologies, such as surface acoustic wave (SAW), can be deployed in the implementation. After this, the signal is traditionally quadrature downconverted to baseband, possibly through an additional IF stage, and the baseband signal is finally A/D converted. Another more advanced alternative is to sample and digitize the signal directly at IF and carry out the final I/Q demodulation using DSP. The overall structure with baseband A/D conversions is illustrated in Fig. 21.

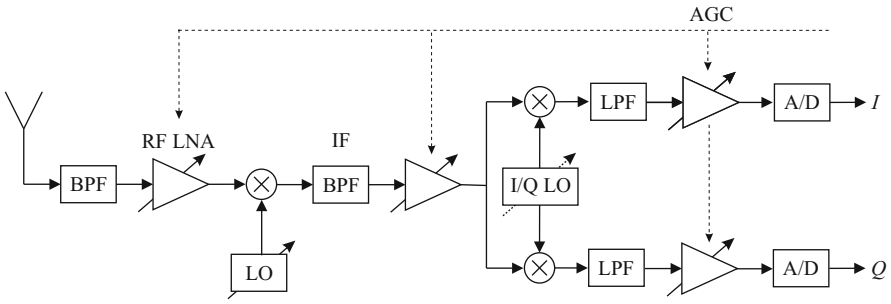


Fig. 21 Principal structure of classical superheterodyne radio receiver

As shortly discussed already earlier, a real mixer is equally sensitive to frequencies below and above the oscillator frequency. Thus for oscillator frequency f_{LO}, any input signal component at some frequency f_{c} will appear at both $f_{\mathrm{c}} - f_{\mathrm{LO}}$ and $f_{\mathrm{c}} + f_{\mathrm{LO}}$ at the mixer output. Thus in addition to the desired channel signal, also the so called image band signal will appear at the IF if not filtered away before the downconversion. For this purpose, superheterodyne receivers always use RF image rejection filtering. In general, the used LO frequencies can be either below ($f_{\mathrm{LO}} = f_{\mathrm{c}} - f_{\mathrm{IF}}$, lower side injection) or above ($f_{\mathrm{LO}} = f_{\mathrm{c}} + f_{\mathrm{IF}}$, upper side injection) the desired channel center-frequency. In any case, the frequency separation between the desired and image signals is always $2f_{\mathrm{LO}}$. Thus in practice the image band is located at the distance $2f_{\mathrm{IF}}$ either below or above the desired channel, depending on the side of LO injection. The basic superheterodyne principle can also be extended to double-IF or triple-IF scenario where the signal is brought to baseband through many consecutive IFs, and selectivity is implemented step by step.

From the receiver design point of view, a proper compromise is required in selecting or specifying the intermediate frequency. On one hand, a high enough IF should be used since the desired and image bands are separated by $2f_{\mathrm{IF}}$ and the image rejection filtering is performed at RF. On the other hand, a low enough IF is needed to make the implementation of the IF channel selectivity filtering as feasible as possible. As an example, intermediate frequencies around 71 MHz (first) and 13 MHz (second) are traditionally used in superheterodyne based GSM receivers, whereas IFs around 10 MHz are typical in broadcast FM receivers.

4.4.2 Direct-Conversion Receiver

Due to the high number of discrete components and high power consumption, the above superheterodyne architecture is, however, not the most appropriate choice for highly integrated transceiver implementations in mass-market devices. Furthermore, the use of fixed discrete components in the RF front-end limits the receiver flexibility. Thus, architectures with more simplified analog front-ends with less RF processing are in general desirable.

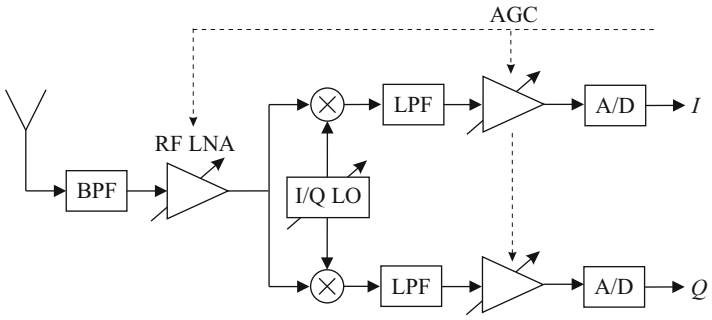


Fig. 22 Principal structure of direct-conversion radio receiver

A simple way to reduce the number of components in the receiver and alleviate the problem of receiver complexity is to avoid the use of intermediate frequency stage and use complex or quadrature downconversion of the desired channel signal from RF directly to baseband. Complete elimination of the IF stage results in highly simplified structure where most of the channel selectivity and amplification are implemented at baseband. In practice, depending on the performance of the A/D interface, the overall selectivity can be split properly between analog and digital filters. On one hand, since most of the signal processing tasks take place at low frequencies, the power consumption of the radio is minimized. On the other hand, very low noise operation is called for in all the remaining analog components since the amplification provided by the RF stage is only moderate. The basic block-diagram for RF I/Q downconversion based receivers is illustrated in Fig. 22.

In theory, the complex mixing approach corresponds to pure frequency translation and the image signal related problems present in real mixer are basically avoided. In practice, however, complex-valued processing always calls for two parallel signal branches (I and Q, e.g. two mixers and LO signals in case of real-valued input and complex mixer) whose characteristics are (unintentionally) likely to differ to some extent. This so-called I/Q imbalance problem has the net effect of reducing the image rejection capability to only 20 . . . 40 dB in practical analog I/Q front-ends, at least without digital calibration. In the pure direct-conversion radio, the image signal band is the desired signal itself (at negative center-frequency), and the I/Q imbalances cause self-image interference. Other practical implementation problems, stemming from direct RF-baseband downconversion, are LO leakage and DC offsets, or in general second order intermodulation (IM2), which create spurious signal energy and interference on top of the desired signal. We will discuss these aspects, together with other RF impairment issues, in more details in Sect. 4.6.

4.4.3 Low-IF Receiver

In the basic low-IF receiver, in order to reduce the effects of LO leakage and DC offsets, the desired signal is I/Q or quadrature downconverted to a low but non-zero IF. Thus the basic structure is similar to previous direct-conversion block-diagram but the complex I/Q signal after I/Q downconversion is located at low intermediate frequency. As an example, intermediate frequencies in the order of one or two channel bandwidths have been proposed and considered. Selectivity can be implemented with special complex analog bandpass filters, centered at low IF, or then with more wideband lowpass filter after which the final selectivity and downconversion from IF to baseband is carried out digitally after A/D interface. Notice that since the image signal in RF-IF downconversion comes now again from another channel/band with a (possibly) very high power level, the use of a non-zero IF reintroduces the image signal problem to big extent and the practical 20–40 dB image attenuation of analog I/Q downconversion can easily be insufficient.

In a "per-channel" downconverting low-IF receiver, the image signal originates from one of the nearby (adjacent) channels. Though the image problem is in this case partly alleviated by the system specifications, which usually limit the power difference of the nearby channels to 10 . . . 25 dB, the 20 . . . 40 dB attenuation provided by a practical analog front-end is clearly inadequate for most communication waveforms. In a multichannel scenario, which is especially interesting, e.g., on the base station side of cellular systems, several channels are downconverted as a whole and the image frequency band may carry a signal at the maximum allowed (blocking) signal level. Thus, for some of the channels, the image band signal can be up to 50 . . . 100 dB stronger than the desired signal, and the imbalanced analog front-end image attenuation is clearly insufficient. Obviously, to facilitate the use of these low-IF schemes in future high-performance highly-integrated receivers, novel digital techniques enhancing the analog front-end image rejection to an acceptable level are needed. Some example techniques are shortly cited in Sect. 4.6. Using the multichannel direct-conversion/low-IF scheme with demanding mobile communication system specifications is generally a very challenging idea. With a proper combination of advanced analog signal processing (like the complex analog Hilbert filtering type technique) and advanced DSP solutions, the required performance is still feasible.

4.4.4 RF/IF Subsampling Receiver

One interesting class of receivers builds on bandpass subsampling principle, in which the incoming radio (RF or IF) signal is deliberately sampled below the classical Nyquist rule. Stemming from the bandlimited nature of the radio signals, aliasing in the sense of creating new frequencies or "images" of the original signal at lower center-frequencies can actually be allowed, as long as the original modulating or information bearing signal remains undistorted. This is called subsampling and essentially means that aliasing is used in a controlled manner to bring the signal closer to baseband without explicit mixer techniques.

Starting from a real-valued incoming bandpass signal, the subsampling radio can be building on either (1) real or (2) complex I/Q subsampling. In case of real subsampling, the signal is simply periodically sampled at a deliberate rate below the Nyquist rate and the output sequence is still a real bandpass signal but at a new lower center-frequency. Because of general bandpass radio waveform contains I and Q components, the resulting signal cannot be aliased directly to baseband but needs to be still in bandpass form. In case of complex I/Q subsampling, the idea is to sample the incoming real-valued bandpass signal in two parallel branches; one branch is directly the original input signal and the other branch is a 90° phase-shifted version which is obtained using a Hilbert transformer type filter discussed earlier in this Chapter. In such case, when the two parallel signals are viewed as a complex signal, the sampler input is free from negative frequencies and thus aliasing can be used more flexibly without the constraints of real subsampling. As an extreme example, if the input center-frequency is an integer multiple of the sampling rate, a direct bandpass-baseband conversion is obtained and the resulting two parallel sample streams are sampled I and Q components of the original baseband signal.

One of the biggest practical limitations in deploying bandpass sampling, especially at RF frequencies in the GHz range, is related to practical imperfections of the sampling circuits. Especially the impact of uncertainties in the sampling instants, called sampling jitter, is generally increased when the center frequency is increased [13]. This is because the instantaneous rate of change of the time domain waveform is directly proportional to the center frequency. Different SNR degradation rules are available in the literature to quantify the impact of sampling jitter in bandpass sampling, see e.g. [13].

There are also recent advances in the concept called charge-domain sampling and its applications in radio devices. Interested reader is referred to [76, 115].

4.5 Transceiver Digital Front-End

The waveform generation block of Fig. 1 produces a digital sample sequence which corresponds to the discrete-time baseband version of the final RF signal to be transmitted. The up-conversion of the baseband signal to the RF carrier frequency can be done solely by the analog RF module, following D/A conversion of the generated waveform. As discussed above, the up-conversion can be done in multiple steps. Likewise, the received signal at the wanted RF channel is bandpass filtered and down-converted to baseband, traditionally within the *RF system block*. Eventually, a digital sample sequence corresponding to the *coding and modulation* block output (but affected by additive noise and interferences as well as various distortion effects) is fed to the *demodulation and decoding block*.

4.5.1 Traditional vs. Software Defined Radio Models

In basic single-mode transceiver solutions, the *interpolation and upconversion* and *filtering, decimation and down-conversion* blocks of Fig. 1 maybe absent or minimal, and DAC and ADC are working at a sampling rate which is at or close to the minimum required for the specific waveform processing. However, in many applications, and wireless mobile communication terminals in particular, the device needs to implement multiple radio systems (e.g., GSM, WCDMA, 3GPP LTE, 802.11 WLAN, Bluetooth, GPS), and a multi-radio platform is needed. Even though most of the current implementations still use different radio circuits for different systems (see Fig. 23a), there is increasing interest for a highly configurable radio platform able to implement different wireless system standards. The concept of DSP-intensive software defined radio (SDR) has emerged from this need [74, 113, 166, 170]. In such DSP intensive solutions, the roles of *interpolation and upconversion* and *filtering, decimation and down-conversion* modules is pronounced and they are intended to take over various functionalities traditionally implemented by the *RF system* blocks. In addition to multi-standard transceivers, multichannel transceiver, utilizing common analog sections and DSP techniques for combining/separating different frequency channels, is another motivation for DSP intensive solutions, especially on the base-station side. The spectrum agile radio concept, discussed in Sect. 3.2.2, inevitably leads to the same direction.

In such solutions, the DAC and ADC sampling rates are typically much higher than the symbol rate, and multirate signal processing is used to implement channelization filtering and up- and down-conversion functions. In the extreme case (so-called direct digital synthesis transmitter and RF sampling receiver), the RF system blocks would include only amplification and rudimentary filtering operations. Even though the needed technologies are not mature enough for full SDR implementations of wireless consumer devices, the development is gradually moving in that direction.

In a SDR receiver, the digital front-end includes adjustable channelization filtering and sampling rate reduction, jointly implemented through digital multirate filtering. Depending on the radio architecture, this may be implemented as a lowpass decimation filter if the wanted frequency channel is down-converted to baseband using analog or digital mixing stages (see Fig. 23b). Alternatively, a bandpass decimation structure may be used, which utilizes the aliasing effects in sampling rate reduction for frequency translation purposes (see Fig. 23c) [166]. This approach usually allows to down-convert the wanted frequency channel close to baseband, after which a fine-tuning mixing operation is usually needed for compensating the frequency offsets due to the limited granularity of this principle, together with the compensation of frequency offsets of the local oscillators of the transmission link.

In a DSP intensive transmitter or receiver, the ADC/DAC sampling rate is often high compared to the channel bandwidth, and a very effective channelization filtering solution is needed in order not to increase the implementation complexity of the overall solution significantly. Luckily, in a well-design multirate filtering solution, the complexity is proportional to the low sampling rate (filter input

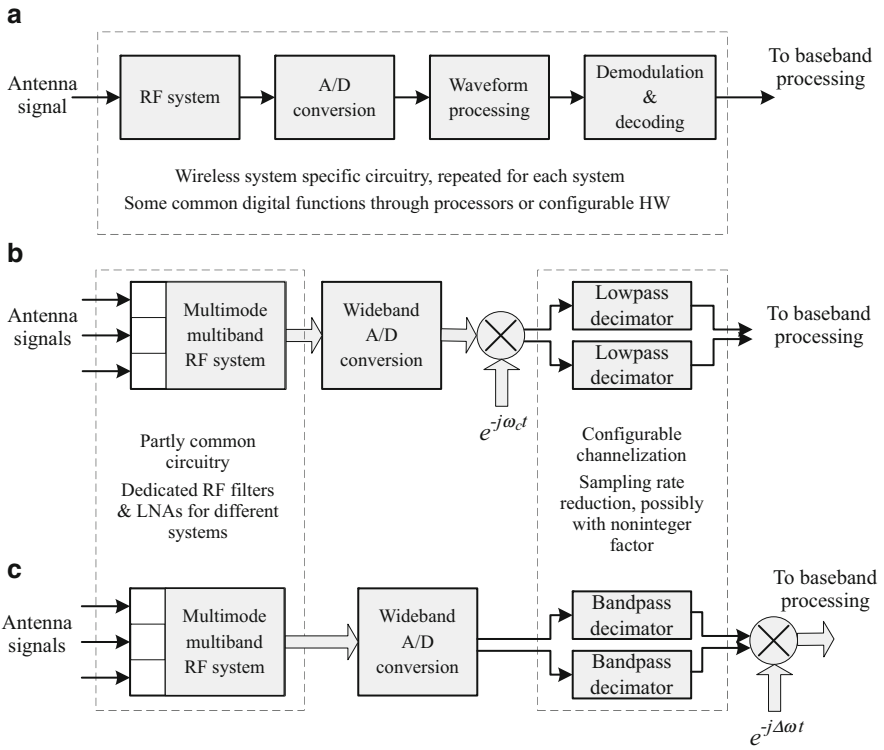


Fig. 23 Alternative multi-radio approaches. (**a**) Traditional receiver structure. (**b**) Configurable receiver based on digital I/Q mixing and baseband decimation filtering. (**c**) Configurable receiver based on bandpass decimation filtering and frequency offset compensation at low sample rate

sampling rate in transmitter and output sampling rate in receiver) [43]. Multi-stage interpolation/decimation structures are commonly considered as they are often most effective in terms of multiplication and addition rates, as well as coefficient and data memory requirements [134]. Typically the first stages of a decimator and last stages of an interpolator have relaxed frequency response requirements, and multiplication-free solutions are available, like the cascaded ingrator-comb (CIC) structures [78, 144]. Considering the bandpass decimator based receiver structure of Fig. 23c, one quite flexible and efficient approach is to use lowpass/bandpass/highpass FIR or IIR half-band filters in cascade [71]. Filter bank based channelizers provide computationally effective solutions for multichannel transmitters and receivers. [72].

A SDR is often expected to do the waveform processing for communication signals with a wide range of signal bandwidths and, therefore, the sampling rate conversion factor has to be adjustable. Furthermore, in different systems the sampling rates of modulation and demodulation blocks are seldom in a simple relation with each other. Yet it is often desirable to use a fixed ADC/DAC clock

frequency for different waveforms to simplify clock synthesizer implementation or to facilitate simultaneously operating multiradio solutions. If different types of signals are to be transmitted or received at the same time, adjusting the sampling clock is not a possible solution. Even though sampling rate conversion with simple fractional factors is possible with basic multirate signal processing methods, techniques for arbitrary sampling rate conversion are very useful in the SDR context. For time-synchronization purposes, fractional delay filters are also useful. Both of these functions can be implemented using polynomial interpolation based on the Farrow structure. [74, 109, 170]

In a SDR transmitter, the dual elements are needed. Digital interpolation filtering, in combination with I/Q mixing is used for increasing the sampling rate and frequency translation. Arbitrary sampling rate conversion may be needed also in this context.

The compensation of time and frequency synchronization offsets needs to be included in the receiver signal path, either as explicit functions as indicated above, or in waveform-specific way in combination with channel equalization, as discussed in Sect. 3.1 in the OFDM context. Additionally, waveform-specific time and frequency offset estimation functions are needed in the digital front-end, either explicitly or in a feedback loop configuration. [109]

4.6 RF Imperfections and DSP

The term RF imperfection refers to the circuit implementation nonidealities and the resulting signal distortion in the central building blocks, like amplifiers, mixers, oscillators and data converters, used in radio transceivers [57, 169, 173]. These aspects have become more and more important in the recent years, stemming from the development and utilization of more and more complex (and thus sensitive) communication waveforms like multicarrier signal structures with high-order subcarrier modulation as well as the carrier aggregation (CA) principle, in modern radio communications. Such wideband complex waveforms are much more sensitive to any signal distortion or interference, compared to earlier narrowband binary-modulated waveforms. The other reason for increased interest towards these issues is demands for transceiver flexibility which typically implies, e.g., less RF filtering and increased dynamic range on the RF modules especially on the receiver side. Also increasing miniaturization of the used electronics and underlying silicon processes, together with decreasing supply voltages and increasing center frequencies, all tend to make electronics more “dirty”.

Understanding and recognizing the above RF imperfection aspects are central in modern radio communications, both at circuit and system levels. Stemming from the increasing digital number crunching power of digital circuits, one interesting R&D field in radio communications is then to develop digital signal processing (DSP) methods and algorithms, perhaps specifically tailored for certain modulation and/or radio architecture, to suppress or mitigate the impact of these RF imperfections.

Best known example of such methods is transmitter power amplifier linearization, through for example digital predistortion (DPD), which has been researched for several decades. But during the past 10 years or so, also many other RF impairments, like mirror-frequency interference due to I/Q imbalances, oscillator phase noise, receiver small signal component nonlinearities, A/D interface nonlinearities, and sampling circuit imperfections, have also been studied. This section shortly addresses these aspects, at very coarse or introductory level, and gives some directions in the recent literature where interested readers can find more information on this theme.

4.6.1 I/Q Imbalance and Mirror-Frequency Interference

Due to finite tolerances of practical analog electronics, there's always some imbalance or mismatch between the relative amplitudes and phases of the analog I and Q branches in transmitters and receivers. This is called I/Q mismatch. Commonly, mismatch levels around 1–5% in amplitude and 1–5° in phase are stated feasible or realistic. This has the impact of creating mirror-frequency distortion or interference to the signal. With the previous mismatch levels, the mirror-frequency attenuation is in the order of 40 . . . 25 dB. In the very basic single-channel direct-conversion radio, the mirror-frequencies are the mirror-image of the signal itself (baseband signal spectrum flipped), and thus the problem is not extremely challenging since the strength of the mirror-frequencies is in the same order as the actual signal frequencies. In case of OFDM, for example, the impact is to create cross-talk between the mirror-symmetric subcarrier pairs.

In case of more general I/Q downconversion based receiver, e.g. I/Q downconversion of a collection of frequency channels or subbands as a whole, the mirror-frequencies of an individual channel or subband are coming from a different channel or subband, and can thus potentially have much more severe effects due to possibly higher power level at the mirror band. An extreme example could be an overall I/Q downconversion of, e.g., whole GSM 1800 MHz uplink band in a base-station device, where in principle the total dynamic range of the overall signal could be in the order of 50–100 dB. In such cases, the image rejection requirements from individual channel perspective are in the same order, and thus impossible to achieve without digital calibration.

The available literature in this field, in terms of digital I/Q calibration and imbalance compensation, is already fairly massive. To get an overview of different digital compensation and calibration methods, both data-aided and non-data-aided, and different radio architecture aspects, the reader is referred to [11, 12, 55, 160, 161, 171, 201].

4.6.2 Transmitter Nonlinearities

When emphasizing power-efficient operation, the power amplifier is always operating in a nonlinear region. This has the impact of creating intermodulation at the PA output. These intermodulation components are basically falling both on top of the ideal waveform bandwidth (inband effect, degrades EVM) as well as next to the ideal waveform bandwidth which is typically called spectral regrowth. Such spectral regrowth can potentially interfere with either other signals of the same radio system or even signals of other radio systems (or both), and is thus typically controlled in the radio system specifications through different emission masks, particularly in the form of adjacent channel leakage ratio (ACLR). Furthermore, out-of-band emissions beyond the ACLR region are also regulated through, e.g., the general spurious emission limits. Particularly in cases with non-contiguous transmit spectrum, it is many times the ACLR and spurious emission limitations, instead of EVM, that form the most severe emission limits thus also then limiting the available or usable transmit power.

Simple way to reduce the intermodulation is to backoff the amplifier input closer to the linear region. This, however, also directly reduces the efficiency and typically also the output power. In order have good balance between output power, efficiency and linearity, digital predistortion techniques can be deployed in which the digital transmit data is pre-processed such that when going through the nonlinear PA, the intermodulation levels are still within the target limits. Alternative method for PA linearization is, e.g. feedforward linearization in which the intermodulation of the core PA is explicitly estimated and subtracted properly from the final transmitter output.

The literature in this field is even more massive than in the previous sub-section, but some seminal works are, e.g., [10, 14, 49, 84, 85, 89, 93, 101, 114, 197, 198]. More recent works specifically developed and tailored to linearizing very wideband transmitters and/or transmitters with non-contiguous transmit spectrum are, e.g., [3–5, 21, 29, 33, 64, 92, 100, 102, 104, 128, 138, 139, 190–193].

4.6.3 Receiver and ADC Nonlinearities

Even though the typical signal levels on the receiver side are much smaller than on the transmitter side, also many receiver components are nonlinear. This applies, e.g., to low noise amplifier (LNA), mixers and also to A/D interface. The most challenging conditions are the cases when the desired signal is weak (close to sensitivity level) while the neighboring channels, or more far away blocking signals, are several tens of decibels stronger. Then depending on the receiver linearity, the neighboring channels and/or blocking signals create intermodulation on top of the weak desired signal. For the RF components, measures like input intercept point (IIP) are typically used to quantify this phenomenon. IIP2 and IIP3 measure second-order and third-order intermodulation behavior, respectively. It is also somewhat radio architecture specific whether the second-order or third-order intermodulation

is the critical interference source. In plain direct-conversion receiver, the second-order effects are typically dominating while in IF-receivers it can be third-order intermodulation.

An interesting research direction is to devise receiver linearization signal processing. Such approach has not been studied very extensively but some seminal works are available, see e.g., [50, 87, 88, 140, 151, 172]. They can be broadly categorized to interference cancellation methods where intermodulation is suppressed explicitly from the weak desired signal band, either using analog or digital signal processing, and hybrid receiver or module calibration methods where e.g. the mixer bias conditions are tuned to optimize IP2 or IP3 using a feedback from downconverted signal.

In addition to actual RF components, also the A/D interface is inherently nonlinear creating spurious components. In radio signal context, especially with wideband multichannel A/D conversion, these spurious components result in intermodulation between the signal bands. A/D interface linearization, especially through offline calibration with e.g. lookup tables, has been also studied fairly widely, but recently also some online signal processing innovations for challenging radio applications have been reported [8].

4.6.4 Oscillator Phase Noise

Phase noise refers to random fluctuations of the instantaneous phase or frequency of the oscillator signals used in radio devices e.g. for frequency translations. Simple behavior modeling reveals that such phase noise appears as additional phase modulation in the time-domain waveform, or when viewed from complex baseband equivalent signal perspective, in multiplicative form as a complex exponential multiplier with the phase jitter in the exponent. This has the principal effect of broadening the signal spectrum.

From an individual waveform point of view, such additional time-domain phase modulation or spectral broadening depends heavily of the used communication waveform. For single-carrier signals, this is directly additional phase jitter in the constellation while in the multicarrier/OFDM case, the spectral broadening of individual subcarriers causes intercarrier interference (ICI) between the neighboring subcarriers.

In a wider scale, the spectral broadening causes the energy of an individual radio signal to leak on top of the neighboring channels. Again due to possibly different power levels of different signals or subbands, this can be potentially much bigger interference source, compared to above single-waveform impact, and typically dictates the oscillator design—especially from large frequency offsets perspective.

In the recent years, the issue of phase noise estimation and digital suppression has also started to raise some interest. Some seminal works in this field, mostly focusing to ICI estimation and suppression with OFDM signals, are e.g. [47, 124, 130, 158, 163, 185, 200].

4.6.5 Sampling Jitter

Sampling jitter refers to the instantaneous timing uncertainties in the sampling process and sample instants. This has typically big effect when the signal that is sampled has high rate of change, which is the case in IF and especially RF sampling, or high instantaneous envelope dynamics. With bandpass signals, the impact of timing jitter is basically similar to phase noise, meaning that it is seen as additional random phase modulation in the sampled sequence. How the power of the interference or distortion due to jitter is distributed in the frequency domain, depends heavily on the correlation properties of the jitter process itself. Some elementary receiver system calculations typically assume white jitter and thereon white jitter noise, but if the jitter process has more correlation between consecutive sample instants, the induced noise has also more structure. In the literature, some works exists where this phenomenon is utilized, the reader is directed e.g. to [141, 159] and the references therein.

5 Concluding Remarks

This chapter has focused on the algorithms for baseband processing and digital front end of wireless communication systems. The field is rapidly developing and the timely topics of R&D activities include technologies for flexible and effective spectrum use, supporting a wide range of services including mobile broadband with highly increasing data rate and speed of mobility (e.g., high-speed trains), massive machine-type communications and Internet-of-things, as well ultra reliable and low-latency communications. The cellular mobile network is evolving towards a multi-service network for all these services, while the development of dedicated networks for specific services is on-going in parallel. Meanwhile, the used carrier frequencies are extending towards mm-wave frequency bands (30–100 GHz) and the carrier bandwidths are growing to several hundreds of MHz and beyond.

On the other hand, the practical implementation of the algorithms, derived from communication theoretic viewpoint, requires another round of optimization exploring the tradoffs between algorithmic simplifications and implementation related cost criteria (complexity, energy consumption, etc.). This optimization depends greatly on the target hardware architecture, which could be based on dedicated VLSI, processors, or FPGAs.

References

1. IEEE Journal on Selected Areas in Communications, Special issue on the turbo principle: From theory to practise I, May (2001)
2. IEEE Journal on Selected Areas in Communications, Special issue on the turbo principle: From theory to practise II, Sep (2001)
3. Abdelaziz, M., Anttila, L., Kiayani, A., Valkama, M.: Decorrelation-based concurrent digital predistortion with a single feedback path. IEEE Transactions on Microwave Theory and Techniques **PP**(99), 1–14 (2017). https://doi.org/10.1109/TMTT.2017.2706688
4. Abdelaziz, M., Anttila, L., Tarver, C., Li, K., Cavallaro, J.R., Valkama, M.: Low-complexity subband digital predistortion for spurious emission suppression in noncontiguous spectrum access. IEEE Transactions on Microwave Theory and Techniques **64**(11), 3501–3517 (2016). https://doi.org/10.1109/TMTT.2016.2602208
5. Abdelaziz, M., Fu, Z., Anttila, L., Wyglinski, A.M., Valkama, M.: Digital predistortion for mitigating spurious emissions in spectrally agile radios. IEEE Communications Magazine **54**(3), 60–69 (2016). https://doi.org/10.1109/MCOM.2016.7432149
6. Abdoli, J., Jia, M., Ma, J.: Filtered OFDM: A new waveform for future wireless systems. In: IEEE Int. Workshop on Signal Processing Advances in Wireless Communications (SPAWC), pp. 66–70 (2015). https://doi.org/10.1109/SPAWC.2015.7227001
7. Akyildiz, I., Lee, W., Vuran, M., Mohanty, S.: Next generation/dynamic spectrum access/cognitive radio wireless networks: A survey. Computer Networks Journal, Elsevier **50**, 2127–2159 (2006)
8. Allen, M., Marttila, J., Valkama, M.: Digital post-processing for reducing A/D converter nonlinear distortion in wideband radio receivers. In: Signals, Systems and Computers, 2009 Conference Record of the Forty-Third Asilomar Conference on, pp. 1111 –1114 (2009)
9. Anderson, J., Mohan, S.: Source and channel coding: An algorithmic approach. IEEE Trans. Commun. **32**(2), 169–176 (1984)
10. Anttila, L., Händel, P., Valkama, M.: Joint mitigation of power amplifier and I/Q modulator impairments in broadband direct-conversion transmitters. IEEE Transactions on Microwave Theory and Techniques **58**(4), 730–739 (2010)
11. Anttila, L., Valkama, M., Renfors, M.: Circularity-based I/Q imbalance compensation in wideband direct-conversion receivers. IEEE Trans. Veh. Technol. **57**(4), 2099 –2113 (2008)
12. Anttila, L., Zou, Y., Valkama, M.: Digital compensation and calibration of I/Q gain and phase imbalances, chap. 16. Cambridge University Press, Cambridge, UK (2011)
13. Arkesteijn, V., Klumperink, E., Nauta, B.: Jitter requirements of the sampling clock in software radio receivers. IEEE Trans. Circuits Syst. II **53**(2), 90 – 94 (2006)
14. Aschbacher, E.: Digital predistortion of microwave power amplifiers. Ph.D. thesis, Technishe Universitat Wien (2004)
15. Auer, G.: Bandwidth efficient 3D pilot design for MIMO-OFDM. In: Proc. European Wireless Conf. Lucca, Italy (2010)
16. Auras, D., Leupers, R., Ascheid, G.: Efficient VLSI architecture for matrix inversion in soft-input soft-output MMSE MIMO detectors. In: Proc. IEEE Int. Symp. Circuits and Systems, pp. 1018–1021. Melbourne, Australia (2014)
17. Auras, D., Leupers, R., Ascheid, G.: A novel reduced-complexity soft-input soft-output MMSE MIMO detector: Algorithm and efficient VLSI architecture. In: Proc. IEEE Int. Conf. Commun., pp. 4722–4728. Sydney, Australia (2014)
18. Bala, E., Li, J., Yang, R.: Shaping spectral leakage: A novel low-complexity transceiver architecture for cognitive radio. IEEE Vehicular Technology Magazine **8**(3), 38–46 (2013). https://doi.org/10.1109/MVT.2013.2269178
19. Baltar, L., Schaich, F., Renfors, M., Nossek, J.: Computational complexity analysis of advanced physical layers based on multicarrier modulation. In: Proc. Future Network & Mobile Summit, pp. 1–8. Warsaw, Poland (2011)

20. Banelli, P., Buzzi, S., Colavolpe, G., Modenini, A., Rusek, F., Ugolini, A.: Modulation Formats and Waveforms for 5G Networks: Who Will Be the Heir of OFDM?: An overview of alternative modulation schemes for improved spectral efficiency. IEEE Signal Processing Mag. **31**(6), 80–93 (2014). https://doi.org/10.1109/MSP.2014.2337391
21. Bassam, S., Ghannouchi, F., Helaoui, M.: 2-D Digital Predistortion (2-D-DPD) architecture for concurrent dual-band transmitters. IEEE Transactions on Microwave Theory and Techniques **59**, 2547–2553 (Oct. 2011)
22. Benedetto, S., Biglieri, E.: Principles of Digital Transmission; With Wireless Applications. Kluwer Academic Publishers, New York (1999)
23. Benvenuto, N., Tomasin, S.: On the comparison between OFDM and single carrier modulation with a DFE using a frequency-domain feedforward filter. IEEE Trans. Commun. **50**(6), 947–955 (2002)
24. Berrou, C., Glavieux, A.: Near optimum error correcting coding and decoding: Turbo codes. IEEE Trans. Commun. **44**(10), 1261–1271 (1996)
25. Berrou, C., Glavieux, A., Thitimajshima, P.: Near Shannon limit error correcting coding and decoding: Turbo codes. In: Proc. IEEE Int. Conf. Commun., vol. 2, pp. 1064–1070. Geneva, Switzerland (1993)
26. Bingham, J.: Multicarrier modulation for data transmission: An idea whose time has come. IEEE Communications Magazine **28**(5), 5–14 (1990)
27. Boelcskei, H., Gesbert, D., Papadias, C.B., van der Veen, A.J.: Space-Time Wireless Systems: From Array Processing to MIMO Communications. Cambridge University Press, Cambridge, UK (2006)
28. Borgerding, M.: Turning overlap-save into a multiband mixing, downsampling filter bank. IEEE Signal Processing Mag. pp. 158–162 (2006)
29. Braithwaite, R.: Closed-loop digital predistortion (DPD) using an observation path with limited bandwidth. IEEE Transactions on Microwave Theory and Techniques **63, no. 2**, 726–736 (Feb. 2015)
30. Brandes, S., Cosovic, I., Schnell, M.: Sidelobe suppression in OFDM systems by insertion of cancellation carriers. In: Proc. IEEE Veh. Technol. Conf. Fall, pp. 152–156. Los Angeles, CA, USA (2005)
31. Burg, A., Borgmann, M., Wenk, M., Zellweger, M., Fichtner, W., Bölcskei, H.: VLSI implementation of MIMO detection using the sphere decoding algorithm. IEEE J. Solid-State Circuits **40**(7), 1566–1577 (2005)
32. Burg, A., Haene, S., Perels, D., Luethi, P., Felber, N., Fichtner, W.: Algorithm and VLSI architecture for linear MMSE detection in MIMO–OFDM systems. In: Proc. IEEE Int. Symp. Circuits and Systems. Kos, Greece (2006)
33. Cabarkapa, M., Neskovic, N., Budimir, D.: A Generalized 2-D linearity enhancement architecture for concurrent dual-band wireless transmitters. IEEE Transactions on Microwave Theory and Techniques **61**(12), 4579–4590 (2013). https://doi.org/10.1109/TMTT.2013.2287679
34. Cavers, J.K.: An analysis of pilot symbol assisted modulation for Rayleigh fading channels. IEEE Trans. Veh. Technol. **40**(4), 686–693 (1991)
35. Chang, R.: High-speed multichannel data transmission with bandlimited orthogonal signals. Bell Syst. Tech. J. **45**, 1775–1796 (1966)
36. Chen, H.M., Chen, W.C., Chung, C.D.: Spectrally precoded OFDM and OFDMA with cyclic prefix and unconstrained guard ratios. IEEE Trans. Wireless Commun. **10**(5), 1416 – 1427 (2011)
37. Chen, L., Chen, W., Zhang, X., Yang, D.: Analysis and simulation for spectrum aggregation in LTE-advanced system. In: Proc. IEEE Veh. Technol. Conf. Fall, pp. 1–6. Anchorage, AK, USA (2009)
38. Cherubini, G., Eleftheriou, E., Olcer, S.: Filtered multitone modulation for VDSL. In: Proc. IEEE Global Telecommun. Conf., pp. 1139–1144 (1999)
39. CISCO: Visual networking index (VNI) mobile white paper [online]. available at http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html (2017)

40. Collings, I., Butler, M., McKay, M.: Low complexity receiver design for MIMO bit-interleaved coded modulation. In: Proc. IEEE Int. Symp. Spread Spectrum Techniques and Applications, pp. 1993–1997. Sydney, Australia (2004)
41. Cosovic, I., Brandes, S., Schnell, M.: Subcarrier weighting: a method for sidelobe suppression in OFDM systems. IEEE Commun. Lett. **10**(6), 444–446 (2006)
42. Coulson, A., Vaughan, R., Poletti, M.: Frequency-shifting using bandpass sampling. IEEE Trans. Signal Processing **42**(6), 1556 –1559 (1994)
43. Crochiere, R., Rabiner, L.: Multirate Digital Signal Processing. Prentice-Hall, Englewood Cliffs, NJ, USA (1983)
44. Crols, J., Steyaert, M.: CMOS Wireless Transceiver Design. Kluwer, Dordrecht, The Netherlands (1997)
45. Dahlman, E., Parkvall, S., Sköld, J.: 4G LTE / LTE-Advanced for Mobile Broadband. Academic Press (2011)
46. Damen, M.O., Gamal, H.E., Caire, G.: On maximum–likelihood detection and the search for the closest lattice point. IEEE Trans. Inform. Theory **49**(10), 2389–2402 (2003)
47. Demir, A., Mehrotra, A., Roychowdhury, J.: Phase noise in oscillators: A unifying theory and numerical methods for characterization. Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on **47**(5), 655 –674 (2000)
48. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the EM algorithm. J. Royal Stat. Soc. **39**(1), 1–38 (1977)
49. Ding, L.: Digital predistortion of power amplifiers for wireless applications. Ph.D. thesis, School of Electrical and Computer Engineering, Georgia Institute of Technology (2004)
50. Dufrêne, K., Boos, Z., Weigel, R.: Digital adaptive IIP2 calibration scheme for CMOS downconversion mixers. IEEE J. Solid-State Circuits **43**(11), 2434–2445 (2008)
51. EMPHATIC: (2015). INFSO-ICT-211887 Project EMPHATIC Deliverables [Online]. Available at http://www.ict-emphatic.eu
52. Falconer, D., Ariyavisitakul, S.L., Benyamin-Seeyar, A., Eidson, B.: Frequency domain equalization for single-carrier broadband wireless systems. IEEE Commun. Mag. **40**(4), 58–66 (2002)
53. Farhang-Boroujeny, B., Kempter, R.: Multicarrier communication techniques for spectrum sensing and communication in cognitive radios. IEEE Commun. Mag. **46**(4), 80–85 (2008)
54. Faulkner, M.: The effect of filtering on the performance of OFDM systems. IEEE Trans. Veh. Technol. **49**(9), 1877–1884 (2000)
55. Faulkner, M., Mattsson, T., Yates, W.: Automatic adjustment of quadrature modulators. IEE Electron. Lett. **27**(3), 214 –216 (1991)
56. Fessler, J., Hero, A.: Space-alternating generalized expectation-maximization algorithm. IEEE Trans. Signal Processing **42**(10), 2664–2677 (1994)
57. Fettweis, G., Löhning, M., Petrovic, D., Windisch, M., Zillmann, P., Rave, W.: Dirty RF: A new paradigm. Int. J. Wireless Inform. Networks **14**, 138–148 (2007)
58. Fincke, U., Pohst, M.: Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. Math. Comput. **44**(5), 463–471 (1985)
59. Forney, G.D.: The Viterbi algorithm. Proc. IEEE **61**(3), 268–278 (1973)
60. Frerking, M.E.: Digital Signal Processing in Communication Systems. Chapman & Hall, New York, USA (1994)
61. Gallager, R.: Low-Density Parity-Check Codes. MIT Press, Cambridge, USA (1963)
62. 3rd Generation Partnership Project (3GPP); Technical Specification Group Radio Access Network: Evolved universal terrestrial radio access E-UTRA; physical channels and modulation TS 36.211 (version 8.5.0). Tech. rep. (2008)
63. Gerzaguet, R., Bartzoudis, N., Baltar, L.G., Berg, V., Doré, J.B., Kténas, D., Font-Bach, O., Mestre, X., Payaró, M., Färber, M., Roth, K.: The 5G candidate waveform race: A comparison of complexity and performance. EURASIP Journal on Wireless Communications and Networking **2017**(1), 13 (2017). https://doi.org/10.1186/s13638-016-0792-0

64. Gilabert, P.L., Montoro, G.: 3-D distributed memory polynomial behavioral model for concurrent dual-band envelope tracking power amplifier linearization. IEEE Transactions on Microwave Theory and Techniques **63**(2), 638–648 (2015). https://doi.org/10.1109/TMTT.2014.2387825
65. Girotto, M., Tonello, A.M.: Orthogonal design of cyclic block filtered multitone modulation. IEEE Transactions on Communications **64**(11), 4667–4679 (2016). https://doi.org/10.1109/TCOMM.2016.2606624
66. Goldsmith, A.: Wireless Communications. Cambridge University Press, New York, USA (2005)
67. Guo, Z., Nilsson, P.: Algorithm and implementation of the K-best sphere decoding for MIMO detection. IEEE J. Select. Areas Commun. **24**(3), 491–503 (2006)
68. Hahn, S.L.: Hilbert Transforms in Signal Processing. Artech House, MA, USA (1996)
69. Hanzo, L., Liew, T., Yeap, B.: Turbo Coding, Turbo Equalisation and Space-Time Coding for Transmission over Fading Channels. John Wiley & Sons, Chichester, UK (2002)
70. Hara, S., Prasad, R.: Design and performance of multicarrier CDMA system in frequency-selective Rayleigh fading channels. IEEE Trans. Veh. Technol. **48**(5), 1584–1595 (1999)
71. fred harris, Venosa, E., Chen, X., Renfors, M.: Cascade linear phase recursive half-band filters implement the most efficient digital down-converter. In: SDR'11 - Wireless Innovation Forum Conference on Communications Technologies and Software Defined Radio. Washington DC, USA (2011)
72. harris, f., McGwier, R., Egg, B.: A versatile multichannel filter bank with multiple channel bandwidths. In: Proc. IEEE Int. Conf. Cognitive Radio Oriented Wireless Networks and Communications, pp. 1 –5. Cannes, France (2010)
73. Haykin, S.: Adaptive Filter Theory, 3rd edn. Prentice Hall, Upper Saddle River, NJ, USA (1996)
74. Hentschel, T.: Sample rate conversion in software configurable radios. Artech House, Norwood, MA, USA (2002)
75. Hirosaki, B.: An orthogonally multiplexed QAM system using the discrete Fourier transform. IEEE Trans. Commun. **29**(7), pp. 982 – 989 (1981)
76. Ho, Y.C., Staszewski, R.B., Muhammad, K., Hung, C.M., Leipold, D., Maggio, K.: Charge-domain signal processing of direct RF sampling mixer with discrete-time filter in Bluetooth and GSM receivers. EURASIP J. Wireless Comm. and Netw. **2006**(3), 1–14 (2006)
77. Hochwald, B., ten Brink, S.: Achieving near-capacity on a multiple-antenna channel. IEEE Trans. Commun. **51**(3), 389–399 (2003)
78. Hogenauer, E.: An economical class of digital filters for decimation and interpolation. IEEE Trans. Acoust., Speech, Signal Processing **29**(2), 155 – 162 (1981)
79. Huang, Y., Ritcey, J.A.: Joint iterative channel estimation and decoding for bit-interleaved coded modulation over correlated fading channels. IEEE Trans. Wireless Commun. **4**(5), 2549–2558 (2005)
80. Ihalainen, T., Ikhlef, A., Louveaux, J., Renfors, M.: Channel equalization for multi-antenna FBMC/OQAM receivers. IEEE Trans. Veh. Technol. **60**(5), 2070–2085 (2011)
81. Jelinek, F., Anderson, J.: Instrumentable tree encoding of information sources. IEEE Trans. Inform. Theory **17**(1), 118–119 (1971)
82. Jiang, T., Wu, Y.: An overview: Peak-to-average power ratio reduction techniques for OFDM signals. IEEE Trans. Broadcast. **54**(2), 257–268 (2008)
83. Juntti, M., Glisic, S.: Advanced CDMA for wireless communications. In: S.G. Glisic, P.A. Leppänen (eds.) Wireless Communications: TDMA Versus CDMA, chap. 4, pp. 447–490. Kluwer (1997)
84. Katz, A.: Linearization: reducing distortion in power amplifiers. IEEE Microwave **2**(4), 37 –49 (2001)
85. Katz, A., Gray, R., Dorval, R.: Truly wideband linearization. IEEE Microwave Magazine **10**(7), 20–27 (2009)
86. Kay, S.M.: Fundamentals of Statistical Signal Processing: Estimation Theory. Prentice-Hall, Englewood Cliffs, NJ, USA (1993)

87. Keehr, E., Hajimiri, A.: Equalization of third-order intermodulation products in wideband direct conversion receivers. IEEE J. Solid-State Circuits **43**(12), 2853 –2867 (2008)
88. Keehr, E., Hajimiri, A.: Successive regeneration and adaptive cancellation of higher order intermodulation products in RF receivers. IEEE Trans. Microwave Theory Tech. **59**(5), 1379 –1396 (2011)
89. Kenington, P.B.: Linearized transmitters: An enabling technology for software defined radio. IEEE Communications Magazine **40**(2), 156–162 (2002)
90. Ketonen, J., Juntti, M., Cavallaro, J.: Performance-complexity comparison of receivers for a LTE MIMO-OFDM system. IEEE Trans. Signal Processing **58**(6), 3360–3372 (2010)
91. Ketonen, J., Juntti, M., Ylioinas, J.: Decision directed channel estimation for reducing pilot overhead in LTE-A. In: Proc. Annual Asilomar Conf. Signals, Syst., Comp., pp. 1503–1507. Pacific Grove, USA (2010)
92. Kim, J., Roblin, P., Chaillot, D., Xie, Z.: A generalized architecture for the frequency-selective digital predistortion linearization technique. IEEE Transactions on Microwave Theory and Techniques **61**, 596–605 (Jan. 2013)
93. Kim, W.J., Stapleton, S.P., Kim, J.H., Edelman, C.: Digital predistortion linearizes wireless power amplifiers. IEEE Microwave Magazine **6**(3), 54–61 (2005)
94. Komninakis, C., Wesel, R.D.: Joint iterative channel estimation and decoding in flat correlated Rayleigh fading. IEEE J. Select. Areas Commun. **19**(9), 1706 – 1717 (2001)
95. Le Floch, B., Alard, M., Berrou, C.: Coded orthogonal frequency division multiplex. Proc. IEEE **83**(6), 982–996 (1995)
96. Lélé, C., Javaudin, J.P., Legouable, R., Skrzypczak, A., Siohan, P.: Channel estimation methods for preamble-based OFDM/OQAM modulation. European Trans. Telecommun. **19**(7), 741–750 (2008)
97. Li, J., Bala, E., Yang, R.: Resource block filtered-OFDM for future spectrally agile and power efficient systems. Physical Communication **14**, 36–55 (2014). http://dx.doi.org/10.1016/j.phycom.2013.10.003
98. Li, M., Bougart, B., Lopez, E., Bourdoux, A.: Selective spanning with fast enumeration: A near maximum-likelihood MIMO detector designed for parallel programmable baseband architectures. In: Proc. IEEE Int. Conf. Commun., pp. 737 – 741. Beijing, China (2008)
99. Lin, H., Siohan, P.: Multi-carrier modulation analysis and WCP-COQAM proposal. EURASIP Journal on Advances in Signal Processing **2014**(1), 1–19 (2014). https://doi.org/10.1186/1687-6180-2014-79
100. Liu, J., Zhou, J., Chen, W., Zhou, B., Ghannouchi, F.: Low-complexity 2D behavioural model for concurrent dual-band power amplifiers. Electronics Letters **48**(11), 620–621 (2012). https://doi.org/10.1049/el.2012.1183
101. Liu, T., Boumaiza, S., Ghannouchi, F.: Augmented Hammerstein predistorter for linearization of broad-band wireless transmitters. IEEE Trans. Microwave Theory and Techniques **54**(4), 1340–1349 (2006)
102. Liu, Y., Yan, J., Asbeck, P.: Concurrent dual-band digital predistortion with a single feedback loop. IEEE Transactions on Microwave Theory and Techniques **63, no. 5**, 1556–1568 (May 2015)
103. Loulou, A., Renfors, M.: Enhanced OFDM for fragmented spectrum use in 5G systems. Trans. Emerging Tel. Tech. **26**(1), 31–45 (2015). https://doi.org/10.1002/ett.2898
104. Ma, Y., Yamao, Y.: Spectra-folding feedback architecture for concurrent dual-band power amplifier predistortion. IEEE Transactions on Microwave Theory and Techniques **63**(10), 3164–3174 (2015). https://doi.org/10.1109/TMTT.2015.2472011
105. Mak, P.I., U, S.P., Martins, R.: Transceiver architecture selection: Review, state-of-the-art survey and case study. IEEE Circuits Syst. Mag. **7**(2), 6 –25 (2007)
106. Maliatsos, K., Adamis, A., Kanatas, A.G.: Interference versus filtering distortion trade-offs in OFDM-based cognitive radios. Transactions on Emerging Telecommunications Technologies **24**(7-8), 692–708 (2013). https://doi.org/10.1002/ett.2727
107. Martin, K.: Complex signal processing is not complex. IEEE Trans. Circuits Syst. I **51**(9), 1823 – 1836 (2004)

108. McLachlan, G.J., Krishnan, T.: The EM Algorithm and Extensions. Wiley, New York, USA (1997)
109. Meyr, H., Moeneclaey, M., Fechtel, S.A.: Digital Communication Receivers: Synchronization, Channel Estimation and Signal Processing. John Wiley and Sons, New York, USA (1998)
110. Miao, H., Juntti, M.: Space-time channel estimation and performance analysis for wireless MIMO-OFDM systems with spatial correlation. IEEE Trans. Veh. Technol. **54**(6), 2003–2016 (2005)
111. Michailow, N., Matthé, M., Gaspar, I.S., Caldevilla, A.N., Mendes, L.L., Festag, A., Fettweis, G.: Generalized frequency division multiplexing for 5th generation cellular networks. IEEE Transactions on Communications **62**(9), 3045–3061 (2014). https://doi.org/10.1109/TCOMM.2014.2345566
112. Mirabbasi, S., Martin, K.: Classical and modern receiver architectures. IEEE Commun. Mag. **38**(11), 132 – 139 (2000)
113. Mitola, J.: The software radio architecture. IEEE Commun. Mag. **33**(5), 26 –38 (1995)
114. Morgan, D., et al.: A generalized memory polynomial model for digital predistortion of RF power amplifiers. IEEE Trans. Signal Processing **54**(10), 3852–3860 (2006)
115. Muhammad, K., Staszewski, R., Leipold, D.: Digital RF processing: Toward low-cost reconfigurable radios. Communications Magazine, IEEE **43**(8), 105 – 113 (2005)
116. Muschallik, C.: Improving an OFDM reception using an adaptive Nyquist windowing. In: 1996. Digest of Technical Papers., International Conference on Consumer Electronics, pp. 6– (1996). https://doi.org/10.1109/ICCE.1996.517186
117. Myllylä, M.: Detection algorithms and architectures for wireless spatial multiplexing in MIMO–OFDM systems. Ph.D. thesis, Acta Univ. Oul., C Technica 380, University of Oulu (2011)
118. Myllylä, M., Cavallaro, J.R., Juntti, M.: Architecture design and implementation of the metric first list sphere detector algorithm. IEEE Trans. VLSI Syst. **19**(5), 895–899 (2011)
119. Myllylä, M., Juntti, M., Cavallaro, J.: Architecture design and implementation of the increasing radius - List sphere detector algorithm. In: Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing, pp. 553–556. Taipei, Taiwan (2009)
120. Myung, H.G., Junsung, L., Goodman, D.J.: Single carrier FDMA for uplink wireless transmission. IEEE Veh. Technol. Mag. **1**(7), 30–38 (2006)
121. Nee, R.V., Prasad, R.: OFDM for Wireless Multimedia Communications. Arthec House, Boston (2000)
122. Oppenheim, A.V., Schafer, R.W.: Discrete-Time Signal Processing. Prentice-Hall, Englewood Cliffs, NJ, USA (1989)
123. Parsons, J.D.: The Mobile Radio Propagation Channel, second edn. John Wiley & Sons (2001)
124. Petrovic, D., Rave, W., Fettweis, G.: Effects of phase noise on OFDM systems with and without PLL: Characterization and compensation. IEEE Transactions on Communications **55**(8), 1607 –1616 (2007)
125. PHYDYAS: (2011). INFSO-ICT-211887 Project PHYDYAS Deliverables, [Online]. Available at http://www.ict-phydyas.org
126. Proakis, J.G.: Digital Communications, 4th edn. McGraw-Hill, New York (2000)
127. Pun, M.O., Morelli, M., Kuo, C.C.: Multi-Carrier Techniques for Broadband Wireless Communications. Imperial College Press (2007)
128. Qian, H., Yao, S., Huang, H., Yang, X., Feng, W.: Low complexity coefficient estimation for concurrent dual-band digital predistortion. IEEE Transactions on Microwave Theory and Techniques **63**(10), 3153–3163 (2015). https://doi.org/10.1109/TMTT.2015.2472002
129. Qualcomm: 5G Waveform & Multiple Access Techniques (2015). Online: www.qualcomm.com/media/documents/files/5g-waveform-multiple-access-techniques.pdf, last accessed 3 June 2016
130. Rabiei, P., Namgoong, W., Al-Dhahir, N.: A non-iterative technique for phase noise ICI mitigation in packet-based OFDM systems. IEEE Trans. Signal Processing **58**(11), 5945 –5950 (2010)

131. Renfors, M., Bader, F., Baltar, L., Ruyet, D.L., Roviras, D., Mege, P., Haardt, M., Stitz, T.H.: On the use of filter bank based multicarrier modulation for professional mobile radio. In: 2013 IEEE 77th Vehicular Technology Conference (VTC Spring), pp. 1–5 (2013). https://doi.org/10.1109/VTCSpring.2013.6692670
132. Renfors, M., et al.: Flexible and spectrally localized waveform processing for next generation wireless communications (2015). INFSO-ICT-211887 Project PHYDYAS, White Paper, [Online]. Available at http://www.ict-emphatic.eu/dissemination.html
133. Renfors, M., Ihalainen, T., Stitz, T.: A block-Alamouti scheme for filter bank based multicarrier transmission. In: European Wireless Conference, pp. 1031 –1037 (2010). https://doi.org/10.1109/EW.2010.5483517
134. Renfors, M., Saramäki, T.: Recursive Nth-band digital filters- Part II: Design of multistage decimators and interpolators. IEEE Trans. Circuits Syst. **34**(1), 40 – 51 (1987)
135. Renfors, M., Yli-Kaakinen, J.: Flexible fast-convolution implementation of single-carrier waveform processing. In: IEEE Int. Conf on Communications Workshops, ICCW 2015, pp. 1243–1248. London, UK (2015). https://doi.org/10.1109/ICCW.2015.7247352
136. Renfors, M., Yli-Kaakinen, J., Harris, F.: Analysis and design of efficient and flexible fast-convolution based multirate filter banks. IEEE Trans. Signal Processing **62**(15), 3768–3783 (2014)
137. Ringset, V., Rustad, H., Schaich, F., Vandermot, J., Najar, M.: Performance of a filterbank multicarrier (FBMC) physical layer in the WiMAX context. In: Proc. Future Network & Mobile Summit. Florence, Italy (2010)
138. Roblin, P., Myoung, S.K., Chaillot, D., Kim, Y.G., Fathimulla, A., Strahler, J., Bibyk, S.: Frequency-selective predistortion linearization of RF power amplifiers. IEEE Transactions on Microwave Theory and Techniques **56**, 65–76 (Jan. 2008)
139. Roblin, P., Quindroit, C., Naraharisetti, N., Gheitanchi, S., Fitton, M.: Concurrent linearization. IEEE Microwave Magazine pp. 75–91 (Nov. 2013)
140. Rodriguez, S., Rusu, A., Zheng, L.R., Ismail, M.: CMOS RF mixer with digitally enhanced IIP2. Electronics Letters **44**, 121–122 (2008)
141. Rutten, R., Breems, L., van Veldhoven, R.: Digital jitter-cancellation for narrowband signals. In: Proc. IEEE Int. Symp. Circuits and Systems, pp. 1444 –1447 (2008)
142. Sahin, A., Arslan, H.: Edge windowing for OFDM based systems. IEEE Commun. Lett. **15**(11), 1208–1211 (2011)
143. Saltzberg, B.: Performance of an efficient parallel data transmission system. IEEE Trans. Commun. Technol. **15**(6), 805–811 (1967)
144. Saramäki, T., Ritoniemi, T.: A modified comb filter structure for decimation. In: Proc. IEEE Int. Symp. Circuits and Systems, pp. 2353–2356. Hong-Kong (1997)
145. Sari, H., Karim, G., Jeanclaude, I.: Transmission techniques for digital terrestrial TV broadcasting. IEEE Commun. Mag. **33**(2), 100–109 (1995)
146. Schaich, F., Wild, T., Chen, Y.: Waveform contenders for 5G – Suitability for short packet and low latency transmissions. In: IEEE Vehicular Technology Conference (VTC Spring 2014), pp. 1–5 (2014)
147. Scharf, L.L.: Statistical Signal Processing: Detection, Estimation, and Time Series Analysis. Addison-Wesley, Reading, MA, USA (1991)
148. Schlegel, C., Pérez, L.: Trellis and Turbo Coding. Wiley IEEE Press Publication, Piscataway, USA (2004)
149. Shaat, M., Bader, F.: Computationally efficient power allocation algorithm in multicarrier-based cognitive radio networks: OFDM and FBMC systems. EURASIP J. Advances Signal Processing **2010**, 1–13 (2010)
150. Shafi, M., Molisch, A.F., Smith, P.J., Haustein, T., Zhu, P., Silva, P.D., Tufvesson, F., Benjebbour, A., Wunder, G.: 5G: A tutorial overview of standards, trials, challenges, deployment and practice. IEEE Journal on Selected Areas in Communications **PP**(99), 1–1 (2017). https://doi.org/10.1109/JSAC.2017.2692307

151. Shahed, A., Valkama, M., Renfors, M.: Adaptive compensation of nonlinear distortion in multicarrier direct-conversion receivers. In: IEEE Radio Wireless Conf., RAWCON'04, pp. 35–38. Atlanta, GA (2004)
152. Shao, K., Alhava, J., Yli-Kaakinen, J., Renfors, M.: Fast-convolution implementation of filter bank multicarrier waveform processing. In: IEEE Int. Symp. on Circuits and Systems (ISCAS 2015), pp. 978–981. Lisbon, Portugal (2015). https://doi.org/10.1109/ISCAS.2015.7168799
153. Siohan, P., Siclet, C., Lacaille, N.: Analysis and design of OFDM-OQAM systems based on filterbank theory. IEEE Trans. Signal Processing **50**(5), 1170–1183 (2002)
154. Studer, C., Burg, A., Bolcskei, H.: Soft-output sphere decoding: algorithms and VLSI implementation. IEEE J. Select. Areas Commun. **26**(2), 290–300 (2008)
155. Studer, C., Fateh, S., Seethaler, D.: ASIC implementation of soft-input soft-output MIMO detection using MMSE parallel interference cancellation. IEEE J. Solid-State Circuits **46**(7), 1754–1765 (2011)
156. Suikkanen, E.: Detection algorithms and ASIC designs for MIMO-OFDM downlink receivers. Ph.D. thesis, Acta Univ. Oul., C Technica 606, University of Oulu, Oulu, Finland (2017)
157. Suikkanen, E., Juntti, M.: ASIC implementation and performance comparison of adaptive detection for MIMO–OFDM system. In: Proc. Annual Asilomar Conf. Signals, Syst., Comp., pp. 1632–1636. Pacific Grove, USA (2015)
158. Syrjälä, V., Valkama, M.: Analysis and mitigation of phase noise and sampling jitter in OFDM radio receivers. Int. J. Microwave and Wireless Technologies **2**(4), 193–202 (2010)
159. Syrjälä, V., Valkama, M.: Sampling jitter cancellation in direct-sampling radio. In: Proc. IEEE Wireless Commun. and Networking Conf., pp. 1 –6 (2010)
160. Tandur, D., Moonen, M.: Joint adaptive compensation of transmitter and receiver IQ imbalance under carrier frequency offset in OFDM-based systems. IEEE Trans. Signal Processing **55**(11), 5246 –5252 (2007)
161. Tarighat, A., Bagheri, R., Sayed, A.: Compensation schemes and performance analysis of IQ imbalances in OFDM receivers. IEEE Trans. Signal Processing **53**(8), 3257 – 3268 (2005)
162. Tarver, C., Sun, Y., Amiri, K., Brogioli, M., Cavallaro, J.R.: Application-specific accelerators for communications. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
163. Tomba, L.: On the effect of Wiener phase noise in OFDM systems. IEEE Trans. Commun. **46**(5), 580 –583 (1998)
164. Toskala, A., Holma, H.: LTE for UMTS - OFDMA and SC-FDMA Based Radio Access. John Wiley and Sons, New York, USA (2009)
165. Tse, D., Viswanath, P.: Fundamentals of Wireless Communication. Cambridge University Press, Cambridge, UK (2005)
166. Tsui, J.: Digital Techniques for Wideband Receivers. Artech House, Norwood, MA, USA (1995)
167. Tüchler, M., Singer, A.C., Koetter, R.: Minimum mean squared error equalisation using a priori information. IEEE Trans. Signal Processing **50**(3), 673–683 (2002)
168. Väänänen, O., Vankka, J., Halonen, K.: Simple algorithm for peak windowing and its application in GSM, EDGE and WCDMA systems. IEE Proc. – Commun. **152**(3), 357–362 (2005)
169. Valkama, M.: RF impairment compensation for future radio systems. In: G. Hueber and R.B. Staszewski, Eds., Multi-Mode/Multi-Band RF Transceivers for Wireless Communications: Advanced Techniques, Architectures, and Trends. Wiley/IEEE Press, U.K. (2010)
170. Valkama, M., Pirskanen, J., Renfors, M.: Signal processing challenges for applying software radio principles in future wireless terminals: An overview. Int. Journal of Communication Systems, Wiley **15**, 741–769 (2002)
171. Valkama, M., Renfors, M., Koivunen, V.: Advanced methods for I/Q imbalance compensation in communication receivers. IEEE Trans. Signal Processing **49**(10), 2335 –2344 (2001)
172. Valkama, M., Shahed hagh ghadam, A., Anttila, L., Renfors, M.: Advanced digital signal processing techniques for compensation of nonlinear distortion in wideband multicarrier radio receivers. IEEE Trans. Microwave Theory and Techniques **54**(6), 2356–2366 (2006)

173. Valkama, M., Springer, A., Hueber, G.: Digital signal processing for reducing the effects of RF imperfections in radio devices – An overview. In: Proc. IEEE Int. Symp. Circuits and Systems, pp. 813 –816 (2010)
174. Vallet, R., Taieb, K.H.: Fraction spaced multi-carrier modulation. Wireless Pers. Commun., Kluwer **2**, 97–103 (1995)
175. Vangelista, L., Benvenuto, N., Tomasin, S., Nokes, C., Stott, J., Filippi, A., Vlot, M., Mignone, V., Morello, A.: Key technologies for next-generation terrestrial digital television standard DVB-T2. IEEE Commun. Mag. **47**(10), 146–153 (2009)
176. Vaughan, R., Scott, N., White, D.: The theory of bandpass sampling. IEEE Trans. Signal Processing **39**(9), 1973 –1984 (1991)
177. Verdú, S.: Multiuser Detection. Cambridge University Press, Cambridge, UK (1998)
178. Viholainen, A., Ihalainen, T., Rinne, M., Renfors, M.: Localized mode DFT-S-OFDMA implementation using frequency and time domain interpolation. EURASIP Journal on Advances in Signal Processing **2009**, 1–9 (2009). https://doi.org/10.1155/2009/750534
179. Viholainen, A., Ihalainen, T., Stitz, T.H., Renfors, M., Bellanger, M.: Prototype filter design for filter bank based multicarrier transmission. In: Proc. European Sign. Proc. Conf. Glasgow, Scotland (2009)
180. Weinsten, S.B., Ebert, P.M.: Data transmission by frequency division multiplexing using the discrete Fourier transform. IEEE Trans. Commun. Technol. **19**(5), 628–634 (1971)
181. Weiss, T.A., Hillenbrand, J., Krohn, A., Jondral, F.K.: Mutual interference in OFDM-based spectrum pooling systems. In: Proc. IEEE Veh. Technol. Conf. Spring, pp. 1872–1877. Dallas, TX, USA (2004)
182. Wolniansky, P.W., Foschini, G.J., Golden, G.D., Valenzuela, R.A.: V-BLAST: An architecture for realizing very high data rates over the rich-scattering wireless channel. In: International Symposium on Signals, Systems, and Electronics (ISSSE), pp. 295–300. Pisa, Italy (1998)
183. Wong, K., Tsui, C., Cheng, R.K., Mow, W.: A VLSI architecture of a K-best lattice decoding algorithm for MIMO channels. In: Proc. IEEE Int. Symp. Circuits and Systems, vol. 3, pp. 273–276. Scottsdale, AZ (2002)
184. Wu, M., Yin, B., Vosoughi, A., Studer, C., Cavallaro, J.R., Dick, C.: Approximate matrix inversion for high-throughput data detection in the large-scale MIMO uplink. In: Proc. IEEE Int. Symp. Circuits and Systems, pp. 2155–2158. Beijing, China (2013)
185. Wu, S., Bar-Ness, Y.: OFDM systems in the presence of phase noise: Consequences and solutions. IEEE Trans. Commun. **52**(11), 1988 – 1996 (2004)
186. Xie, Y., Georghiades, C.N., Li, Q.: A novel low complexity detector for MIMO system. In: Proc. Annual Asilomar Conf. Signals, Syst., Comp., vol. 1, pp. 208 – 212 (2004)
187. Yli-Kaakinen, J., Levanen, T., Valkonen, S., Pajukoski, K., Pirskanen, J., Renfors, M., Valkama, M.: Efficient fast-convolution based waveform processing for 5G physical layer. IEEE Journal on Selected Areas in Communications **35**, 1–18 (2017)
188. Ylioinas, J., Juntti, M.: Iterative joint detection, decoding, and channel estimation in turbo coded MIMO-OFDM. IEEE Trans. Veh. Technol. **58**(4), 1784–1796 (2009). https://doi.org/10.1109/TVT.2008.2005724
189. Ylioinas, J., Raghavendra, M.R., Juntti, M.: Avoiding matrix inversion in DD SAGE channel estimation in MIMO-OFDM with M-QAM. In: Proc. IEEE Veh. Technol. Conf., pp. 1–5. Anchorage, USA (2009)
190. Younes, M., Kwan, A., Rawat, M., Ghannouchi, F.M.: Linearization of concurrent tri-band transmitters using 3-D phase-aligned pruned volterra model. IEEE Transactions on Microwave Theory and Techniques **61**(12), 4569–4578 (2013). https://doi.org/10.1109/TMTT.2013.2287176
191. Yu, C., Allegue-Martinez, M., Guo, Y., Zhu, A.: Output-controllable partial inverse digital predistortion for RF power amplifiers. IEEE Transactions on Microwave Theory and Techniques **62**(11), 2499–2510 (2014). https://doi.org/10.1109/TMTT.2014.2360175
192. Yu, C., Guan, L., Zhu, E., Zhu, A.: Band-limited volterra series-based digital predistortion for wideband RF power amplifiers. IEEE Transactions on Microwave Theory and Techniques **60**(12), 4198–4208 (2012). https://doi.org/10.1109/TMTT.2012.2222658

193. Yu, C., Xia, J., Zhu, X., Zhu, A.: Single-model single-feedback digital predistortion for concurrent multi-band wireless transmitters. IEEE Transactions on Microwave Theory and Techniques **63**(7), 2211–2224 (2015). https://doi.org/10.1109/TMTT.2015.2429633
194. Yuan, Z., Wyglinski, A.: On sidelobe suppression for multicarrier-based transmission in dynamic spectrum access networks. IEEE Trans. Veh. Technol. **59**(4), 1998 – 2006 (2010)
195. Zayani, R., Medjahdi, Y., Shaiek, H., Roviras, D.: WOLA-OFDM: A potential candidate for asynchronous 5G. In: 2016 IEEE Globecom Workshops (GC Wkshps), pp. 1–5 (2016). https://doi.org/10.1109/GLOCOMW.2016.7849087
196. Zhang, H., LeRuyet, D., Roviras, D., Medjahdi, Y., Sun, H.: Spectral efficiency comparison of OFDM/FBMC for uplink cognitive radio networks. EURASIP J. Advances Signal Processing **2010**, 1–14 (2010)
197. Zhou, D., DeBrunner, V.E.: Novel adaptive nonlinear predistorters based on the direct learning algorithm. IEEE Trans. Signal Processing **55**(1), 120–133 (2007)
198. Zhou, G.T., et al.: On the baseband representation of a bandpass nonlinearity. IEEE Trans. Signal Processing **53**(8), 2953–2957 (2005)
199. Zhu, Y., Letaief, K.: Single carrier frequency domain equalization with time domain noise prediction for wideband wireless communications. IEEE Trans. Wireless Commun. **5**(12), 3548–3557 (2006)
200. Zou, Q., Tarighat, A., Sayed, A.: Compensation of phase noise in OFDM wireless systems. IEEE Trans. Signal Processing **55**(11), 5407 –5424 (2007)
201. Zou, Y., Valkama, M., Renfors, M.: Digital compensation of I/Q imbalance effects in space-time coded transmit diversity systems. IEEE Trans. Signal Processing **56**(6), 2496 –2508 (2008)

Signal Processing for Radio Astronomy

Alle-Jan van der Veen, Stefan J. Wijnholds, and Ahmad Mouri Sardarabadi

Abstract Radio astronomy is known for its very large telescope dishes but is currently making a transition towards the use of a large number of small antennas. For example, the Low Frequency Array, commissioned in 2010, uses about 50 stations each consisting of 96 low band antennas and 768 or 1536 high band antennas. The low-frequency receiving system for the future Square Kilometre Array is envisaged to initially consist of over 131,000 receiving elements and to be expanded later. These instruments pose interesting array signal processing challenges. To present some aspects, we start by describing how the measured correlation data is traditionally converted into an image, and translate this into an array signal processing framework. This paves the way to describe self-calibration and image reconstruction as estimation problems. Self-calibration of the instrument is required to handle instrumental effects such as the unknown, possibly direction dependent, response of the receiving elements, as well a unknown propagation conditions through the Earth's troposphere and ionosphere. Array signal processing techniques seem well suited to handle these challenges. Interestingly, image reconstruction, calibration and interference mitigation are often intertwined in radio astronomy, turning this into an area with very challenging signal processing problems.

A.-J. van der Veen (✉)
TU Delft, Faculty of EEMCS, Delft, The Netherlands
e-mail: a.j.vanderveen@tudelft.nl

S. J. Wijnholds
Netherlands Institute for Radio Astronomy (ASTRON), Dwingeloo, The Netherlands
e-mail: wijnholds@astron.nl

A. M. Sardarabadi
University of Groningen, Kapteyn Astronomical Institute, Groningen, The Netherlands
e-mail: ammsa@astro.rug.nl

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_9

1 Introduction

Astronomical instruments measure cosmic particles or electromagnetic waves impinging on the Earth. Astronomers use the data generated by these instruments to study physical phenomena outside the Earth's atmosphere. In recent years, astronomy has transformed into a multi-modal science in which observations at multiple wavelengths are combined. Figure 1 provides a nice example showing the lobed structure of the famous radio source Cygnus A as observed at 240 MHz with the Low Frequency Array (LOFAR) overlaid by an X-Ray image observed by the Chandra satellite, which shows a much more compact source.

Such images are only possible if the instruments used to observe different parts of the electromagnetic spectrum provide similar resolution. Since the resolution is determined by the ratio of observed wavelength and aperture diameter, the aperture of a radio telescope has to be 5 to 6 orders of magnitude larger than that of an optical telescope to provide the same resolution. This implies that the aperture of a radio telescope should have a diameter of several hundreds of kilometers. Most current and future radio telescopes therefore exploit interferometry to synthesize a large aperture from a number of relatively small receiving elements.

An interferometer measures the correlation of the signals received by two antennas spaced at a certain distance. After a number of successful experiments in the 1950s and 1960s, two arrays of 25-m dishes were built in the 1970s: the 3 km Westerbork Synthesis Radio Telescope (WSRT, 14 dishes) in Westerbork, The Netherlands and the 36 km Very Large Array (VLA, 27 movable dishes) in Socorro, New Mexico, USA. These telescopes use Earth rotation to obtain a sequence of

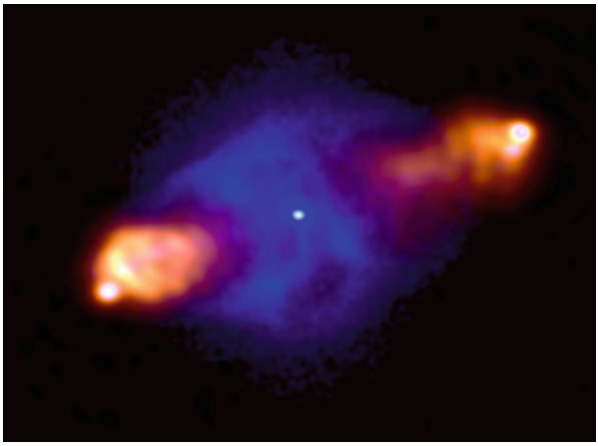


Fig. 1 Radio image of Cygnus A observed at 240 MHz with the Low Frequency Array (showing mostly the lobes left and right), overlaid over an X-Ray image of the same source observed by the Chandra satellite (the fainter central cloud) [65] (*Courtesy of Michael Wise and John McKean*)

correlations for varying antenna baselines, resulting in high-resolution images via *synthesis mapping*. A more extensive historical overview is presented in [52].

The radio astronomy community has recently commissioned a new generation of radio telescopes for low frequency observations, including the Murchison Widefield Array (MWA) [38, 53] in Western Australia and the Low Frequency Array (LOFAR) [24, 58] in Europe. These telescopes exploit phased array technology to form a large collecting area with ∼1000 to ∼50,000 receiving elements. The community is also making detailed plans for the Square Kilometre Array (SKA), a future radio telescope that should be one to two orders of magnitude more sensitive than any radio telescope built to date [18]. Even in its first phase of operation, the low-frequency receiving system of the SKA (SKA-low) is already envisaged to consist of over 131,000 receiving elements [17, 56].

The individual antennas in a phased array telescope have an extremely wide field-of-view, often the entire visible sky. This poses a number of signal processing challenges, because certain assumptions that work well for small fields-of-view (celestial sphere approximated by a plane, homogenous propagation conditions over the field-of-view), are no longer valid. Furthermore, the data volumes generated by these new instruments will be huge and will have to be reduced to manageable proportions by a real-time automated data processing pipeline. This combination of challenges led to a flurry of research activity in the area of array calibration, imaging and RFI mitigation, which are often intertwined in the astronomical data reduction.

The goal of calibration is to find the unknown instrumental, atmospheric and ionospheric disturbances. The imaging procedure should be able to apply appropriate corrections based on the outcome of the calibration process to produce a proper image of the sky. In this chapter, we review some of the array processing techniques that have been proposed for use in standard calibration and imaging pipelines, many of which are already being used in data reduction pipelines of instruments like LOFAR.

2 Notation

Matrices and vectors will be denoted by boldface upper-case and lower-case symbols, respectively. Entries of a matrix $\mathbf{A}$ are denoted by a_{ij}, and its columns by $\mathbf{a}_i$. Overbar $\overline{(\cdot)}$ denotes complex conjugation. The transpose operator is denoted by ^{T}, the complex conjugate (Hermitian) transpose by ^{H} and the Moore-Penrose pseudo-inverse by †. For matrices $\mathbf{A}$ of full column rank, i.e., $\mathbf{A}^H\mathbf{A}$ invertible, this is equal to the left inverse:

$$\mathbf{A}^\dagger = (\mathbf{A}^H\mathbf{A})^{-1}\mathbf{A}^H . \tag{1}$$

The expectation operator is denoted by $E\{\cdot\}$.

We will multiply matrices in many different ways. Apart from the usual multiplication $\mathbf{AB}$, we will use $\mathbf{A} \odot \mathbf{B}$ to denote the Hadamard product (element-wise multiplication), and $\mathbf{A} \otimes \mathbf{B}$ to denote the Kronecker product,

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}.$$

We will also use the Khatri-Rao or column-wise Kronecker product of two matrices: let $\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, \cdots]$ and $\mathbf{B} = [\mathbf{b}_1, \mathbf{b}_2, \cdots]$, then

$$\mathbf{A} \circ \mathbf{B} = [\mathbf{a}_1 \otimes \mathbf{b}_1,\ \mathbf{a}_2 \otimes \mathbf{b}_2,\ \cdots].$$

Depending on the context, $\mathrm{diag}(\cdot)$ converts a vector to a diagonal matrix with the elements of the vector placed on the main diagonal, or converts a general matrix to a diagonal matrix by selecting its main diagonal. Further, $\mathrm{vec}(\cdot)$ converts a matrix to a vector by stacking the columns of the matrix.

Properties of Kronecker products are listed in, e.g., [43]. We frequently use

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD} \tag{2}$$

$$\mathrm{vec}(\mathbf{ABC}) = (\mathbf{C}^T \otimes \mathbf{A})\mathrm{vec}(\mathbf{B}) \tag{3}$$

$$\mathrm{vec}(\mathbf{A}\,\mathrm{diag}(\mathbf{b})\,\mathbf{C}) = (\mathbf{C}^T \circ \mathbf{A})\mathbf{b}. \tag{4}$$

Property (3) is used to move a matrix $\mathbf{B}$ from the middle of an equation to the right of it, exploiting the linearity of the product. Property (4) is a special case of it, to be used if $\mathbf{B}$ is a diagonal matrix: in that case $\mathrm{vec}(\mathbf{B})$ has many zero entries, and we can omit the corresponding columns of $\mathbf{C}^T \otimes \mathbf{A}$, leaving only the columns of the Khatri-Rao product $\mathbf{C}^T \circ \mathbf{A}$. A special case of (3) is

$$\mathrm{vec}(\mathbf{a}\mathbf{a}^H) = \bar{\mathbf{a}} \otimes \mathbf{a} \tag{5}$$

which shows how a rank-1 matrix $\mathbf{a}\mathbf{a}^H$ is related to a vector with a specific "Kronecker structure".

3 Basic Concepts of Interferometry; Data Model

The concept of interferometry is illustrated in Fig. 2. An interferometer measures the spatial coherency of the incoming electromagnetic field. This is done by correlating the signals from the individual receivers with each other. The correlation of each pair of receiver outputs provides the amplitude and phase of the spatial coherence function for the *baseline* defined by the vector pointing from the first to the second

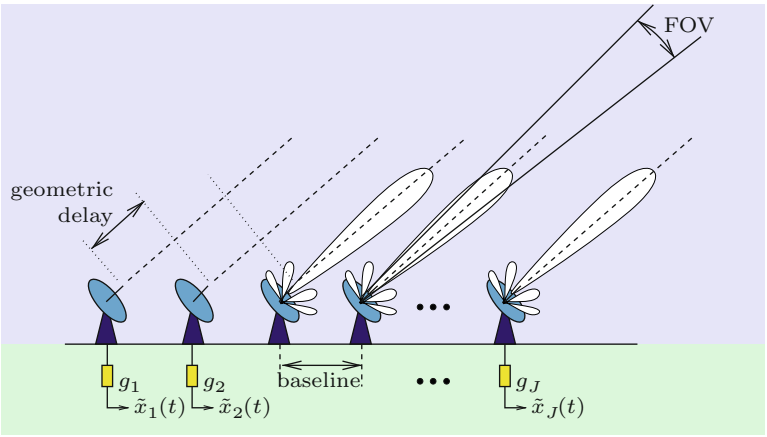


Fig. 2 Schematic overview of a radio interferometer

receiver in a pair. In radio astronomy, these correlations are called the *visibilities*. In this section, we describe the data acquisition in detail and construct a suitable data model.

3.1 Data Acquisition

Assume that there are J receiving elements. Depending on the context, a receiving element can be a telescope dish, a single antenna within a subarray (usually referred to as a *station*) or a beamformed subarray. The RF signal from the jth telescope, $\tilde{x}_j(t)$ is first moved to baseband where it is denoted by $x_j(t)$, then sampled and split into narrow subbands, e.g., of 100 kHz each, such that the narrowband condition holds. This condition states that the maximal geometrical delay across the array should be fairly representable by a phase shift of the complex baseband signal, and this property is discussed in more detail in the next subsection. The resulting signal is called $x_j(n,k)$, for the jth telescope, nth time bin, and for the subband frequency centered at RF frequency f_k. The J signals can be stacked into a $J \times 1$ vector $\mathbf{x}(n,k)$.

For each short-term integration (STI) interval m and each subband k, a covariance matrix estimate is formed by *integrating* (summing or averaging) the cross-correlation products $\mathbf{x}(n,k)\mathbf{x}^H(n,k)$ over N subsequent samples,

$$\hat{\mathbf{R}}_{m,k} = \frac{1}{N} \sum_{n=(m-1)N}^{mN-1} \mathbf{x}(n,k)\mathbf{x}^H(n,k)\,, \tag{6}$$

This processing chain is summarized in Fig. 3.

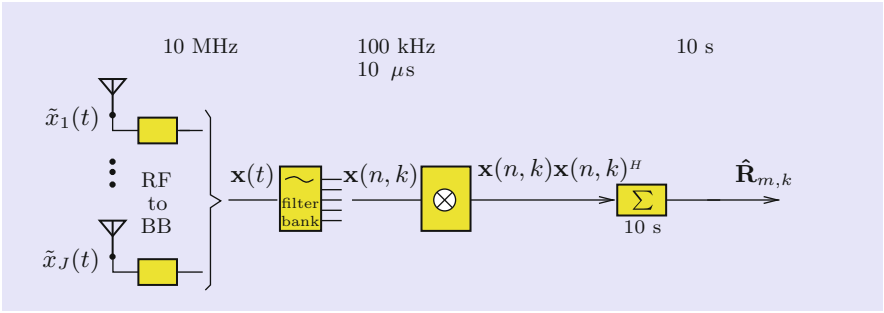


Fig. 3 The processing chain to obtain covariance data

The duration of an STI depends on the stationarity of the data, which is limited by factors like Earth rotation and the diameter of the array. For the LOFAR, a typical value for the STI is 1–10 s. A complete observation can last from a few minutes to a full night, i.e., more than 12 h. The resulting number of samples N in a snapshot observation is equal to the product of bandwidth and integration time and typically ranges from 10^3 (1 s, 1 kHz) to 10^6 (10 s, 100 kHz) in radio astronomical applications.

3.2 Complex Baseband Signal Representation

Before we can derive a data model, we need to include some more details on the RF to baseband conversion. In signal processing, signals are usually represented by their low pass equivalents, which is a suitable representation for narrowband signals in a digital communication system, and also applicable in the radio astronomy context. A complex valued bandpass signal, also called the *complex baseband signal*, with center frequency f_c may be written as

$$\tilde{s}(t) = s(t)e^{\mathrm{j}2\pi f_c t} \tag{7}$$

Suppose that the bandpass signal $\tilde{s}(t)$ is delayed by a time τ. This can be written as

$$\tilde{s}_\tau(t) := \tilde{s}(t-\tau) = s(t-\tau)e^{\mathrm{j}2\pi f_c(t-\tau)} = s(t-\tau)e^{-\mathrm{j}2\pi f_c\tau}e^{\mathrm{j}2\pi f_c t}\,.$$

The complex envelope of the delayed signal is thus $s_\tau(t) = s(t-\tau)e^{-\mathrm{j}2\pi f_c\tau}$. Let B be the bandwidth of the complex envelope (the baseband signal) and let $S(f)$ be its Fourier transform. We then have

$$s(t-\tau) = \int_{-B/2}^{B/2} S(f) e^{-\mathrm{j}2\pi f\tau} e^{\mathrm{j}2\pi ft} df \;\approx\; \int_{-B/2}^{B/2} S(f) e^{\mathrm{j}2\pi ft} df = s(t)$$

where the approximation $e^{-\mathrm{j}2\pi f\tau} \approx 1$ is valid if $|2\pi f\tau| \ll 1$ for all frequencies $|f| \leq \frac{B}{2}$. Ignoring a factor π, the resulting condition $B\tau \ll 1$ is called the narrowband condition. The quantitative interpretation of "much less than one" depends on the SNR of the received signals [67] and the sensitivity loss considered acceptable [9]. Under this condition, we have for the complex envelope $s_\tau(t)$ of the delayed bandpass signal $\tilde{s}_\tau(t)$ that

$$s_\tau(t) \approx s(t) e^{-\mathrm{j}2\pi f_c \tau} \qquad \text{for } B\tau \ll 1\,.$$

The conclusion is that, for narrowband signals, time delays smaller than the inverse bandwidth may be represented as phase shifts of the complex envelope. Phased array processing heavily depends on this step. For radio astronomy, the maximal delay τ is equal to the maximal geometric delay, which can be related to the diameter of the array. The bandwidth B is the bandwidth of each subband f_k in the RF processing chain that we discussed in the previous subsection.

3.3 Data Model

We return to the radio astronomy context. For our purposes, it is convenient to model the sky as consisting of a collection of Q spatially discrete point sources, with $s_q(n,k)$ the signal of the qth source at time sample n and frequency f_k.

The signal received at the jth antenna is a sum of delayed source signals, where the delays are geometric delays that depend on the direction under which each of the signals is observed. In the previous subsection, we saw that under the narrowband condition a delay of a narrowband signal $s(t,k)$ by τ can be represented by a phase shift:

$$s_\tau(t,k) = e^{-\mathrm{j}2\pi f_k \tau} s(t,k)$$

which takes the form of a multiplication of $s(t,k)$ by a complex number. Let $\mathbf{z}_j = [x_j, y_j, z_j]^T$ be the location of the jth antenna. Further, let $\mathbf{l}_q$ be a unit-length direction vector pointing into the direction of the qth source.

The geometrical delay τ at antenna j for a signal coming from direction $\mathbf{l}_q$ can be computed as follows. For a signal traveling directly from the origin of the coordinate system used to specify the antenna locations to antenna j, the delay is the distance from the origin to the jth antenna divided by c, the speed of light. For any other direction, the delay depends on the cosine of the angle of incidence (compared to the baseline vector) at observing time n, and is thus described by the inner product

of the location vector with the direction vector, i.e., $\tau_{q,j}(n) = \mathbf{z}_j \cdot \mathbf{l}_q(n)/c$. Overall, the phase factor representing the geometric delay is

$$a_{j,q}(n,k) = e^{-\mathrm{j}2\pi f_k \tau_{q,j}(n)} = e^{-\frac{2\pi \mathrm{j} f_k}{c}\mathbf{z}_j^T \mathbf{l}_q(n)}. \tag{8}$$

The coordinates of source direction vectors $\mathbf{l}_q$ are expressed as[1] (ℓ, m, n), where ℓ, m and n are direction cosines and $n = \sqrt{1-\ell^2-m^2}$ due to the normalization. There are several conventions and details regarding coordinate systems [52], but they are not of concern for us here.

Besides the phase factor $a_{q,j}(n,k)$, the received signals are also affected by the direction dependent response of the receiving element $b_j(\mathbf{l},n,k)$ and the direction independent complex valued receiver path gain $g_j(n,k)$. The function $b_j(\mathbf{l},n,k)$ is referred to as the *primary beam* to distinguish it from the array beam and the *point spread function* or *dirty beam* that results from beamforming over a full synthesis observation (more about this later). The general shape of the primary beam is known from (electromagnetic) modelling during the design of the telescope. If that model is not sufficiently accurate, it needs to be calibrated. Together with the tropospheric and ionospheric propagation conditions, the primary beam determines the direction dependent gain $g^d_{j,q}(n,k)$ of the jth receiving element. The signal $x_j(n,k)$ received by the jth receiving element can thus be described by

$$x_j(n,k) = g_j(n,k)\sum_{q=1}^{Q} g^d_{j,q}(n,k)a_{j,q}(n,k)s_q(n,k) + n_j(n,k), \tag{9}$$

where $n_j(n,k)$ denotes the additive noise in the jth receive path.

We can stack the phase factors $a_{j,q}(n,k)$ into an *array response vector* for each source as

$$\mathbf{a}_q(n,k) = \left[a_{1,q}(n,k), \cdots, a_{J,q}(n,k)\right]^T. \tag{10}$$

In a similar way, we can stack the direction independent gains $g_j(n,k)$ into a vector $\mathbf{g}(n,k)$, stack the direction dependent gains $g^d_{j,q}(n,k)$ into a vector for each source $\mathbf{g}^d_q(n,k)$ and stack the additive noise signals in a vector $\mathbf{n}(n,k)$. With these conventions, we can formulate the data model for the *array signal vector* as

$$\mathbf{x}(n,k) = \mathbf{g}(n,k) \odot \sum_{q=1}^{Q} \mathbf{g}^d_q(n,k) \odot \mathbf{a}_q(n,k)s_q(n,k) + \mathbf{n}(n,k). \tag{11}$$

For convenience of notation, we introduce the gain matrix

[1] With abuse of notation, as m, n are not related to the time variables used earlier.

$$\mathbf{G}(n,k) = \left[\mathbf{g}(n,k) \odot \mathbf{g}_q^d(n,k), \cdots, \mathbf{g}(n,k) \odot \mathbf{g}_Q^d(n,k)\right].$$

As we will see in Sect. 5, this gain matrix may have a specific structure depending on a priori knowledge about the direction independent gains and the direction dependent gains. This structure can then be exploited during calibration. We can also stack the array response vectors into an array response matrix $\mathbf{A}(n,k) = \left[\mathbf{a}_1(n,k), \cdots \mathbf{a}_Q(n,k)\right]^T$. These conventions allow us to write Eq. (11) as

$$\mathbf{x}(n,k) = (\mathbf{G}(n,k) \odot \mathbf{A}(n,k))\,\mathbf{s}(n,k) + \mathbf{n}(n,k), \tag{12}$$

where $\mathbf{s}(n,k) = \left[s_1(n,k), \cdots s_Q(n,k)\right]^T$.

For convenience of notation, we will in future usually drop the dependence on the frequency f_k (index k) from the notation. Previously, in (6), we defined correlation estimates $\hat{\mathbf{R}}_m$ as the output of the data acquisition process, where the time index m corresponds to the mth STI interval, such that $(m-1)N \leq n \leq mN$. Due to Earth rotation, the vectors $\mathbf{a}_q(n)$ change slowly with time, but we assume that within an STI it can be considered constant and can be represented, with some abuse of notation, by $\mathbf{a}_q(m)$. In that case, $\mathbf{x}(n)$ is wide sense stationary over the STI, and a single STI covariance matrix is defined as

$$\mathbf{R}_m = E\{\mathbf{x}(n)\,\mathbf{x}^H(n)\}, \qquad m = \left\lceil \frac{n}{N} \right\rceil \tag{13}$$

where $\mathbf{R}_m$ has size $J \times J$. Each element of $\mathbf{R}_m$ represents the interferometric correlation along the baseline vector between the two corresponding receiving elements. It is estimated by STI sample covariance matrices $\hat{\mathbf{R}}_m$ defined in (6), and our stationarity assumptions imply $E\{\hat{\mathbf{R}}_m\} = \mathbf{R}_m$.

We will model the source signals $s_q(n,k)$ and the noise signals $n_j(n,k)$ as zero mean white Gaussian random processes sampled at the Nyquist rate. We will also assume that the source signals and noise signals are mutually uncorrelated. With these assumptions, we find, by substituting Eq. (12) into Eq. (13), that

$$\begin{aligned}
\mathbf{R}_m &= E\left\{(\mathbf{G}_m \odot \mathbf{A}_m\mathbf{s}(n) + \mathbf{n}(n))\,(\mathbf{G}_m \odot \mathbf{A}_m\mathbf{s}(n) + \mathbf{n}(n))^H\right\} \\
&= (\mathbf{G}_m \odot \mathbf{A}_m)\,E\left\{\mathbf{s}(n)\mathbf{s}^H(n)\right\}(\mathbf{G}_m \odot \mathbf{A}_m)^H + E\left\{\mathbf{n}(n)\mathbf{n}^H(n)\right\} \\
&= (\mathbf{G}_m \odot \mathbf{A}_m)\,\mathbf{\Sigma}_s\,(\mathbf{G}_m \odot \mathbf{A}_m)^H + \mathbf{\Sigma}_n,
\end{aligned} \tag{14}$$

where $\mathbf{\Sigma}_s = \mathrm{diag}\,(\boldsymbol{\sigma}_s)$ with $\boldsymbol{\sigma}_s = [\sigma_1^2, \cdots, \sigma_Q^2]^T$ is the source covariance matrix and $\mathbf{\Sigma}_n = \mathrm{diag}\,(\boldsymbol{\sigma}_n)$ with $\boldsymbol{\sigma}_n = [\sigma_{n,1}^2, \cdots, \sigma_{n,J}^2]^T$ is the noise covariance matrix. In radio astronomy, the covariance data model described in Eq. (14) is usually referred to as the *measurement equation*.

3.4 Radio Interferometric Imaging Concepts

Under ideal circumstances, the array response matrix $\mathbf{A}_m$ is not perturbed by the gain matrix $\mathbf{G}_m$, i.e., we have $\mathbf{G}_m = \mathbf{1}\mathbf{1}^H$ where $\mathbf{1}$ denotes a vector of ones of appropriate size. The columns of $\mathbf{A}_m$ are given by Eq. (8). Its entries represent the phase shifts due to the geometrical delays associated with the array and source geometry. By adding the gain matrix $\mathbf{G}_m$, we can introduce directional disturbances due to non-isotropic antennas, unequal antenna gains and disturbances due to ionospheric effects.

Assuming ideal conditions and ignoring the additive noise, a single element of the array covariance matrix, usually referred to as a *visibility*, can be written as

$$(\mathbf{R}_m)_{ij} = \sum_{q=1}^{Q} a_{i,q}\overline{a_{j,q}}\sigma_q^2 = \sum_{q=1}^{Q} I\left(\mathbf{l}_q\right) e^{-\mathrm{j}\frac{2\pi}{\lambda}\left(\mathbf{z}_i(m)-\mathbf{z}_j(m)\right)^T\mathbf{l}_q}. \tag{15}$$

where $I(\mathbf{l}_q) = \sigma_q^2$ is the brightness (power) in direction $\mathbf{l}_q$. The function $I(\mathbf{l})$ is the brightness image (or *map*) of interest: it is this function that is shown when we refer to a radio-astronomical image like Fig. 1. It is a function of the direction vector $\mathbf{l}$: this is a 3D vector, but due to its normalization it depends on only two parameters. We could e.g., show $I(\cdot)$ as function of the direction cosines (ℓ, m), or of the corresponding angles.

For our discrete point-source model, the brightness image is

$$I(\mathbf{l}) = \sum_{q=1}^{Q} \sigma_q^2\, \delta(\mathbf{l} - \mathbf{l}_q) \tag{16}$$

where $\delta(\cdot)$ is a Kronecker delta, and the direction vector $\mathbf{l}$ is mapped to the location of "pixels" in the image (various transformations are possible). Only the pixels $\mathbf{l}_q$ are nonzero, and have value equal to the source variance σ_q^2.

The vector $\mathbf{z}_i(m) - \mathbf{z}_j(m)$ is the *baseline*: the (normalized) vector pointing from telescope i to telescope j. In radio astronomy, it is usually expressed in coordinates denoted by $\mathbf{u}_{ij} = (u, v, w)$ and normalized by the wavenumber, i.e., $\mathbf{u}_{ij}(m) = (2\pi/\lambda)(\mathbf{z}_i(m) - \mathbf{z}_j(m))$. The objective in telescope design is often to have as many different baselines as possible. In that case the entries of $\mathbf{R}_m$ are different and non-redundant. As the Earth turns, the baselines also turn, thus giving rise to new baseline directions. We will see later that the set of baselines during an observation determines the spatial sampling function by which the incoming wave field is sampled, with important implications on the quality of the resulting image.

Equation (15) describes the relation between the visibility model and the desired image, and it has the form of a Fourier transform; it is known in radio astronomy as the Van Cittert-Zernike theorem [49, 52]. Image formation (*map making*) is essentially the inversion of this relation. Unfortunately, we have only a finite set

of observations, therefore we can only obtain a *dirty image*: if we apply the inverse Fourier transformation to the measured correlation data, we obtain

$$\hat{I}_D(\mathbf{l}) := \sum_{i,j,m} \left(\hat{\mathbf{R}}_m\right)_{ij} e^{\mathrm{j}\mathbf{u}_{ij}^T(m)\mathbf{l}_q} \tag{17}$$

In terms of the measurement data model (15), the "expected value" of the image is obtained by replacing $\hat{\mathbf{R}}_m$ by $\mathbf{R}_m$, or

$$\begin{aligned} I_D(\mathbf{l}) &:= \sum_{i,j,m} (\mathbf{R}_m)_{ij}\, e^{\mathrm{j}\mathbf{u}_{ij}^T(m)\mathbf{l}} \\ &= \sum_{i,j,m} \sum_q \sigma_q^2 e^{\mathrm{j}\mathbf{u}_{ij}^T(m)(\mathbf{l}-\mathbf{l}_q)} \\ &= \sum_q I(\mathbf{l}_q) B(\mathbf{l}-\mathbf{l}_q) \\ &= I(\mathbf{l}) * B(\mathbf{l}), \end{aligned} \tag{18}$$

where the *dirty beam* is given by

$$B(\mathbf{l}) := \sum_{i,j,m} e^{\mathrm{j}\mathbf{u}_{ij}^T(m)\mathbf{l}}. \tag{19}$$

The dirty image $I_D(\mathbf{l})$ is the desired "true" image $I(\mathbf{l})$ convolved with the dirty beam $B(\mathbf{l})$: every point source excites a beam $B(\mathbf{l}-\mathbf{l}_q)$ centered at its location $\mathbf{l}_q$. The effect of this is that the true image gets blurred, thus limiting its resolution. Note that $B(\mathbf{l})$ is a known function: it only depends on the locations of the telescopes, or rather the set of telescope baselines $\mathbf{u}_{ij}(m) = (2\pi/\lambda)(\mathbf{z}_i(m) - \mathbf{z}_j(m))$.

Note that Eq. (17) has the form of a Fourier transform, although it has been defined on (u, v, w) samples that are non-uniformly spaced. To be able to use the computationally efficient fast Fourier transform (FFT), astronomy software first applies a *gridding* operation that interpolates and resamples the visibilities onto a regular grid, after which the FFT can be used to obtain the dirty image [49, 52]. This essentially implements a non-uniform FFT as used in other science communities [19].

As an example, the antenna configuration for the six stations forming the core of the LOFAR and the resulting single-STI dirty beam is shown in Fig. 4. The dirty beam has heavy sidelobes as high as -10 dB. A resulting dirty image (in dB scale) is shown in Fig. 5. In this image, we see the complete sky, in (ℓ, m) coordinates, where the reference direction is pointing towards zenith. The strong visible sources are Cassiopeia A and Cygnus A, also visible is the Milky Way. The image was obtained by averaging 259 STIs, each consisting of 1 s data in a single frequency channel of 195 kHz wide at a central frequency of 58.9 MHz.

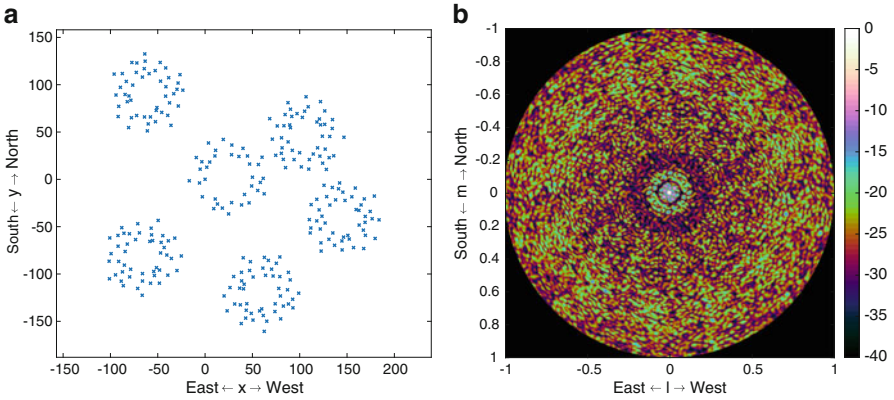


Fig. 4 (**a**) Coordinates of the antennas in the LOFAR Superterp, which defines the spatial sampling function, and (**b**) the resulting *dirty beam* in dB scale

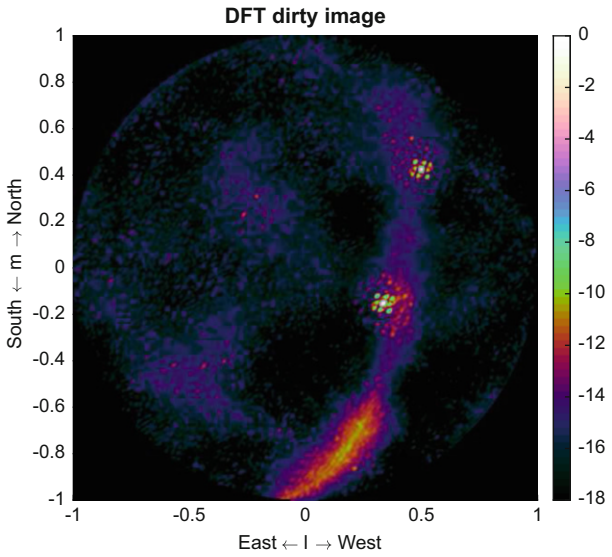


Fig. 5 Dirty image following (18), using LOFAR Superterp data

The dirty beam is essentially a non-ideal point spread function due to finite and non-uniform spatial sampling: we only have a limited set of baselines. The dirty beam usually has a main lobe centered at $\mathbf{l} = \mathbf{0}$, and many side lobes. If we would have a large number of telescopes positioned in a uniform rectangular grid, the dirty beam would be a 2-D sinc-function (similar to a boxcar taper in time-domain sampling theory). The resulting beam size is inversely proportional to the

aperture (diameter) of the array. This determines the *resolution* in the dirty image. The sidelobes of the beam give rise to confusion between sources: it is unclear whether a small peak in the image is caused by the main lobe of a weak source, or the sidelobe of a strong source. Therefore, attempts are made to design the array such that the sidelobes are low. It is also possible to introduce weighting coefficients ("tapers") in (18) to obtain an acceptable beamshape.

Another aspect is the summation over m (STI intervals) in (19), where the rotation of the Earth is used to obtain essentially many more antenna baselines. This procedure is referred to as *Earth rotation synthesis* as more (u, v, w) sampling points are obtained over time. The effect of this is that the sidelobes tend to get averaged out, to some extent. Many images are also formed by averaging over a small number of frequency bins (assuming the σ_q^2 are constant over these frequency bins), which enters into the equations in exactly the same way: Replace $\mathbf{z}_i(m)$ by $\mathbf{z}_i(m, k)$ and also sum over the frequency index k.

4 Image Reconstruction

The goal of image reconstruction is to obtain an estimate of the true image $I(\mathbf{l})$. Many approaches to this problem have been proposed, which can be divided into two classes. The first is a non-parametric approach that starts from the dirty image. Since the dirty image is the convolution of the true image by the dirty beam, this reduces the image reconstruction problem to a *deconvolution* problem. Deconvolution is the process of recovering $I(\mathbf{l})$ from $I_D(\mathbf{l})$ using knowledge of the dirty beam and thus to obtain the high-resolution "clean" image. A standard algorithm for doing this is CLEAN [27] and variants; however, many other algorithms are possible, depending on the underlying model assumptions and on a trade-off between accuracy and numerical complexity.

The second class of approaches is to consider image reconstruction as an estimation problem in which an unknown set of parameters describing $I(\mathbf{l})$ need to be extracted from the measured visibilities collected in the measured array covariance matrices $\hat{\mathbf{R}}_m$. This "model matching" approach is discussed in more detail in Sect. 4.4.

After a telescope has been designed and built, algorithms for image formation are the most important topic for signal processing. Careful techniques can increase the dynamic range (ratio between powers of the strongest and the weakest features in the image) by several orders of magnitude. However, the numerical complexity is often large, and high-resolution images require dedicated hardware solutions and sometimes even supercomputers. In this section, we will describe some of the algorithms. Additional overviews are available in [13, 14, 33, 36], as well as in the books [4, 52].

4.1 Constructing Dirty Images

4.1.1 Beamforming Formulation

Previously (Eq. (17)), we formulated the dirty image as the inverse Fourier transform of the measured correlations. Here, we will interpret this process as beamforming. Once we have this formulation, we may derive many other dirty images via beamforming techniques. For simplicity of notation, we assume from now on that only a single STI snapshot is used in the imaging, hence we also drop the time index m from the equations. The results can easily be extended.

The imaging process transforms the covariances of the received signals to an image of the source structure within the field-of-view of the receivers. In array processing terms, it can be described as follows [33]. Assume a data model as in (12) with all gain factors equal to unity, and recall the definition of the array response vector $\mathbf{a}(\mathbf{l})$ in (8) and (10) (using yet another change of notation to emphasize now that $\mathbf{a}$ is a function of the source direction $\mathbf{l}$). There are J antennas. To determine the power of a signal arriving from a particular direction $\mathbf{l}$, a weight vector

$$\mathbf{w}(\mathbf{l}) = \frac{1}{J}\mathbf{a}(\mathbf{l}) = \frac{1}{J}e^{-\mathrm{j}\frac{2\pi}{\lambda}\mathbf{Z}^T\mathbf{l}}, \tag{20}$$

where $\mathbf{Z} = [\mathbf{z}_1, \cdots, \mathbf{z}_J]$, is applied to the array signal vector $\mathbf{x}(n)$. The operation $y(n) = \mathbf{w}^H\mathbf{x}(n)$ is generally called beamforming. The choice $\mathbf{w} = \mathbf{a}$ precisely compensates the geometric phase delays so that the antenna signals are added in-phase. This can be regarded as a spatially matched filter, or *conjugate field match*. The (often omitted) scaling by $1/J$ ensures the correct scaling of the output power. Indeed, the output power of a beamformer is, generally,

$$E\{|y|^2\} = \mathbf{w}^H E\{\mathbf{x}\mathbf{x}^H\}\mathbf{w} = \mathbf{w}^H\mathbf{R}\mathbf{w}\,.$$

For a data model consisting of a single source with power σ^2 arriving from direction $\mathbf{a}(\mathbf{l})$, i.e., $\mathbf{x}(n) = \mathbf{a}(\mathbf{l})s(n)$, we have, with $\mathbf{w} = \frac{1}{J}\mathbf{a}(\mathbf{l})$,

$$E\{|y|^2\} = \mathbf{w}^H(\mathbf{a}\sigma^2\mathbf{a}^H)\mathbf{w} = \sigma^2\frac{\mathbf{a}^H\mathbf{a}}{J}\frac{\mathbf{a}^H\mathbf{a}}{J} = \sigma^2\,. \tag{21}$$

Thus, the matched beamformer corrects precisely the signal delays (phase shifts) present in $\mathbf{a}(\mathbf{l})$, when $\mathbf{w}$ matches $\mathbf{a}(\mathbf{l})$, i.e. the beamformer is pointed into the same direction as the source. If the beamformer is pointed into other directions, the response is usually much smaller.

Using the beamformer to scan over all pixels $\mathbf{l}$ in an image, we can create an image via beamforming as

$$\hat{I}_{BF}(\mathbf{l}) = \mathbf{w}(\mathbf{l})^H\hat{\mathbf{R}}\mathbf{w}(\mathbf{l}) \tag{22}$$

and the corresponding model for this image is

$$I_{BF}(\mathbf{l}) = \mathbf{w}(\mathbf{l})^H \mathbf{R} \mathbf{w}(\mathbf{l}) \,. \tag{23}$$

The matched filter corresponds to weights $\mathbf{w}(\mathbf{l})$ defined as in (20). Except for a factor J^2, the image $I_{BF}(\mathbf{l})$ is identical to the dirty image $I_D(\mathbf{l})$ defined in (18) for this choice! Indeed, starting from (18), we can write

$$I_D(\mathbf{l}) = \sum_{i,j} R_{ij}\, e^{\mathrm{j}\mathbf{u}_{ij}^T \mathbf{l}} = \sum_{i,j} \overline{a_i}(\mathbf{l}) R_{ij} a_j(\mathbf{l}) = \mathbf{a}(\mathbf{l})^H \mathbf{R} \mathbf{a}(\mathbf{l})$$

which is the beamforming image obtained using $\mathbf{w}(\mathbf{l}) = \mathbf{a}(\mathbf{l})$. The response to a single source at the origin is

$$\begin{aligned} B(\mathbf{l}) &= \mathbf{a}(\mathbf{l})^H \mathbf{a}(\mathbf{0}) \mathbf{a}(\mathbf{0})^H \mathbf{a}(\mathbf{l}) \\ &= \mathbf{a}(\mathbf{l})^H \mathbf{1}\mathbf{1}^H \mathbf{a}(\mathbf{l}) \\ &= \mathbf{1}^H [\mathbf{a}(\mathbf{l}) \mathbf{a}(\mathbf{l})^H] \mathbf{1} \\ &= \sum_{i,j} e^{\mathrm{j}\mathbf{u}_{ij}^T \mathbf{l}} \end{aligned}$$

which is the dirty beam defined in (19), now written in beamforming notation. It typically has a spike at $\mathbf{l} = \mathbf{0}$, and many sidelobes, depending on the spatial sampling function. We have already seen that these sidelobes limit the resolution, as they can be confused with (or mask) other sources.

So far, we looked at the response to a source, but ignored the effect of the noise on an image. In the beamforming formulation, the response to a data set which only consists of noise, or $\mathbf{R} = \mathbf{\Sigma}_n$ is

$$I_n(\mathbf{l}) = \mathbf{w}(\mathbf{l})^H \mathbf{\Sigma}_n \mathbf{w}(\mathbf{l}) \,.$$

Suppose that the noise is spatially white, $\mathbf{\Sigma}_n = \sigma_n^2 \mathbf{I}$, and that we use the matched beamformer (20), we obtain

$$I_n(\mathbf{l}) = \sigma_n^2 \frac{\mathbf{a}(\mathbf{l})^H}{J} \frac{\mathbf{a}(\mathbf{l})}{J} = \sigma_n^2 \frac{\|\mathbf{a}(\mathbf{l})\|^2}{J^2} = \frac{\sigma_n^2}{J}, \tag{24}$$

since all entries of $\mathbf{a}(\mathbf{l})$ have unit magnitude. As this is a constant, the image will be "flat". For a general data set, the responses to the sources and to the noise will be added. Comparing (21)–(24), we see that the noise is suppressed by a factor J compared to a point source signal coming from a specific direction. This is the *array gain*. If we use multiple STIs and/or frequencies f_k, the array gain can be larger than J.

4.1.2 Constructing Dirty Images by Adaptive Beamforming

Now that we have made the connection of the dirty image to beamforming, we can apply a range of other beamforming techniques instead of the matched filter, such as the class of spatially adaptive beamformers. In fact, these can be considered as 2D spatial-domain versions of (now classical) spectrum estimation techniques for estimating the power spectral density of a random process (viz. [26]), and the general idea is that we can obtain a higher resolution if the sidelobes generated by strong sources are made small.

As an example, the "minimum variance distortionless response" (MVDR) beamformer is defined such that the response towards the direction of interest $\mathbf{l}$ is unity, but signals from other directions are suppressed as much as possible, i.e.,

$$\mathbf{w}(\mathbf{l}) = \arg\min_{\mathbf{w}} \ \mathbf{w}^H \mathbf{R} \mathbf{w}\,, \qquad \text{such that} \quad \mathbf{w}^H \mathbf{a}(\mathbf{l}) = 1.$$

This problem can be solved in various ways. For example, after making a transformation $\mathbf{w}' := \mathbf{R}^{1/2}\mathbf{w}$, $\mathbf{a}' := \mathbf{R}^{-1/2}\mathbf{a}$, the problem becomes

$$\mathbf{w}'(\mathbf{l}) = \arg\min_{\mathbf{w}'} \ \|\mathbf{w}'\|^2\,, \qquad \text{such that} \quad \mathbf{w}'^H \mathbf{a}'(\mathbf{l}) = 1.$$

To minimize the norm of $\mathbf{w}'$, it should be aligned to $\mathbf{a}'$, i.e., $\mathbf{w}' = \alpha \mathbf{a}'$, and the solution is $\mathbf{w}' = \mathbf{a}'/(\mathbf{a}'^H \mathbf{a}')$. In terms of the original variables, the solution is then

$$\mathbf{w}(\mathbf{l}) = \frac{\mathbf{R}^{-1}\mathbf{a}(\mathbf{l})}{\mathbf{a}(\mathbf{l})^H \mathbf{R}^{-1} \mathbf{a}(\mathbf{l})}, \tag{25}$$

and the resulting MVDR dirty image can thus be described as

$$I_{MVDR}(\mathbf{l}) = \mathbf{w}(\mathbf{l})^H \mathbf{R} \mathbf{w}(\mathbf{l}) = \frac{1}{\mathbf{a}(\mathbf{l})^H \mathbf{R}^{-1} \mathbf{a}(\mathbf{l})}\,. \tag{26}$$

For a point-source model, this image will have a high resolution: two sources that are closely spaced will be resolved. The corresponding beam responses to different sources will in general be different: the beamshape is spatially varying. While we may represent $I_{MVDR}(\mathbf{l})$ as a convolution of the true image with a dirty beam, this is now a spatially varying convolution (viz. the convolution in a linear time-varying system). Deconvolution is still possible but has to take this into account.

Another consequence of the use of an adaptive beamformer is that the output noise power is not spatially uniform. Consider the data model $\mathbf{R} = \mathbf{A}\boldsymbol{\Sigma}_s \mathbf{A}^H + \boldsymbol{\Sigma}_n$, where $\boldsymbol{\Sigma}_n = \sigma_n^2 \mathbf{I}$ is the noise covariance matrix, then at the output of the beamformer the noise power is, using (25),

$$I_n(\mathbf{l}) = \mathbf{w}(\mathbf{l})^H \mathbf{R}_n \mathbf{w}(\mathbf{l}) = \frac{\mathbf{a}(\mathbf{l})^H \mathbf{R}^{-1} (\sigma_n^2 \mathbf{I}) \mathbf{R}^{-1} \mathbf{a}(\mathbf{l})}{[\mathbf{a}(\mathbf{l})^H \mathbf{R}^{-1} \mathbf{a}(\mathbf{l})]^2} = \sigma_n^2 \frac{\mathbf{a}(\mathbf{l})^H \mathbf{R}^{-2} \mathbf{a}(\mathbf{l})}{[\mathbf{a}(\mathbf{l})^H \mathbf{R}^{-1} \mathbf{a}(\mathbf{l})]^2}\,.$$

Thus, the output noise power is direction dependent.

As a remedy to this, a related beamformer which satisfies the constraint $\mathbf{w}(\mathbf{l})^H \mathbf{w}(\mathbf{l}) = 1$ (and therefore has spatially uniform output noise) is obtained by using a different scaling of the MVDR beamformer:

$$\mathbf{w}(\mathbf{l}) = \mu \mathbf{R}^{-1} \mathbf{a}(\mathbf{l}) , \qquad \mu = \frac{1}{[\mathbf{a}(\mathbf{l})^H \mathbf{R}^{-2} \mathbf{a}(\mathbf{l})]^{1/2}}.$$

This beamformer is known as the "Adapted Angular Response" (AAR) [8]. The resulting image is

$$I_{AAR}(\mathbf{l}) = \mathbf{w}(\mathbf{l})^H \mathbf{R} \mathbf{w}(\mathbf{l}) = \frac{\mathbf{a}(\mathbf{l})^H \mathbf{R}^{-1} \mathbf{a}(\mathbf{l})}{\mathbf{a}(\mathbf{l})^H \mathbf{R}^{-2} \mathbf{a}(\mathbf{l})}.$$

It has a high resolution and suppresses sidelobe interference under the white noise constraint.

Example MVDR and AAR dirty images using the same LOFAR stations as before are shown in Fig. 6. Comparing to Fig. 5, we observe that, as predicted, the sidelobe suppression in the MVDR and AAR dirty images is much better than the original matched beamformer dirty image. The images have a higher contrast and it appears that some additional point sources emerge as the result of lower sidelobe levels. This is especially true for the AAR dirty image.

4.2 Deconvolution

Having obtained a dirty image, we then attempt to recover the true image via *deconvolution*: inverting the effect of the (known) dirty beam.

4.2.1 The CLEAN Algorithm

A popular method for deconvolution is the CLEAN algorithm [27]. It was proposed for the classical, matched beamformer dirty image $I_D(\mathbf{l})$ defined in (17). From $I_D(\mathbf{l})$ and the known dirty beam $B(\mathbf{l})$, the desired image $I(\mathbf{l})$ is obtained via a sequential Least Squares fitting method. The algorithm is based on the assumption that the sky is mostly empty, and consists of a set of discrete point sources. The brightest source is estimated first, its contribution is subtracted from the dirty image, then the next brightest source is subtracted, etc.

The algorithm further uses the fact that $B(\mathbf{l})$ has its peak at the origin. Inside the loop, a candidate location $\mathbf{l}_q$ is selected as the location of the largest peak in $I_D(\mathbf{l})$, the corresponding power $\hat{\sigma}_q^2$ is estimated, and subsequently a small multiple

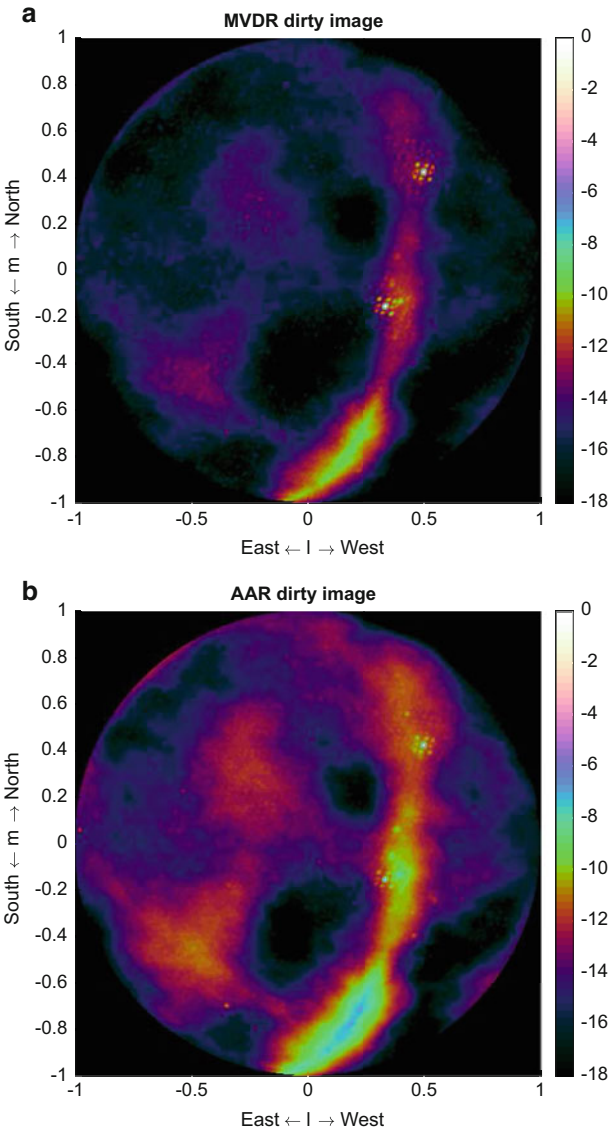


Fig. 6 Dirty images corresponding to the (**a**) MVDR and (**b**) AAR beamformers

of $\hat{\sigma}_q^2 B(\mathbf{l}-\mathbf{l}_q)$ is subtracted from $I_D(\mathbf{l})$. The objective is to minimize the residual, until it converges to the noise level:

$$
\begin{aligned}
&q = 0\\
&\text{while } I_D(\mathbf{l}) \text{ is not noise-like:}\\
&\left[\begin{array}{ll}
q & = q+1\\
\mathbf{l}_q & = \arg\max_{\mathbf{l}} I_D(\mathbf{l})\\
\hat{\sigma}_q^2 & = I_D(\mathbf{l}_q)/B(\mathbf{0})\\
I_D(\mathbf{l}) & := I_D(\mathbf{l}) - \gamma\hat{\sigma}_q^2 B(\mathbf{l}-\mathbf{l}_q)\,, \quad \forall \mathbf{l}
\end{array}\right.\\
&I_{clean}(\mathbf{l}) = I_D(\mathbf{l}) + \textstyle\sum_q \gamma\hat{\sigma}_q^2 B_{synth}(\mathbf{l}-\mathbf{l}_q), \quad \forall \mathbf{l}\,.
\end{aligned}
$$

The scaling parameter $\gamma \leq 1$ is called the loop gain; for accurate convergence it should be small because the estimated location of the peak is at a grid point, whereas the true location of the peak may be in between grid points. $B_{synth}(\mathbf{l})$ is a "synthetic beam", usually a Gaussian bell-shape with about the same beam width as the main lobe of the dirty beam; it is introduced to mask the otherwise high artificial resolution of the image.

In current imaging systems, instead of the subtractions on the dirty image, it is considered more accurate to do the subtractions on the sample covariance matrix $\hat{\mathbf{R}}$ instead,

$$\hat{\mathbf{R}} := \hat{\mathbf{R}} - \gamma\hat{\sigma}_q^2 \mathbf{a}(\mathbf{l}_q)\mathbf{a}(\mathbf{l}_q)^H$$

and then to recompute the dirty image. Computing a dirty image is the most expensive step in this loop, therefore usually a number of peaks are estimated from the dirty image together, the covariance is updated for this ensemble, and then the residual image is recomputed.

4.2.2 CLEAN Using Other Dirty Images

Instead of the matched beamformer dirty image $I_D(\mathbf{l})$, we can use other beamformed dirty images in the CLEAN loop, for example the MVDR dirty image. Due to its high resolution, the location of sources is better estimated than using the original dirty image (and the location estimate can be further improved by searching for the true peak on a smaller grid in the vicinity of the location of the maximum). A second modification to the CLEAN loop is also helpful: suppose that the location of the brightest source is $\mathbf{l}_q$, then the corresponding power α_q should be estimated by minimizing the residual $\|\mathbf{R} - \alpha\mathbf{a}(\mathbf{l}_q)\mathbf{a}(\mathbf{l}_q)^H\|^2$. This can be done in closed form: using (5) we find

$$\|\mathbf{R} - \alpha\mathbf{a}(\mathbf{l}_q)\mathbf{a}(\mathbf{l}_q)^H\| = \|\mathrm{vec}(\mathbf{R}) - \alpha[\bar{\mathbf{a}}(\mathbf{l}_q) \otimes \mathbf{a}(\mathbf{l}_q)]\|\,.$$

The optimal least squares solution for α is, using (1), (3) and (2) in turn,

$$\begin{aligned}\alpha_q &= [\bar{\mathbf{a}}(\mathbf{l}_q) \otimes \mathbf{a}(\mathbf{l}_q)]^{\dagger} \mathrm{vec}(\mathbf{R}) \\ &= \frac{[\bar{\mathbf{a}}(\mathbf{l}_q) \otimes \mathbf{a}(\mathbf{l}_q)]^H \mathrm{vec}(\mathbf{R})}{[\bar{\mathbf{a}}(\mathbf{l}_q) \otimes \mathbf{a}(\mathbf{l}_q)]^H [\bar{\mathbf{a}}(\mathbf{l}_q) \otimes \mathbf{a}(\mathbf{l}_q)]} \\ &= \frac{\mathbf{a}(\mathbf{l}_q)^H \mathbf{R} \mathbf{a}(\mathbf{l}_q)}{[\mathbf{a}(\mathbf{l}_q)^H \mathbf{a}(\mathbf{l}_q)]^2} \\ &= \frac{\mathbf{a}(\mathbf{l}_q)^H \mathbf{R} \mathbf{a}(\mathbf{l}_q)}{J^2},\end{aligned}$$

which is the power estimate of the matched filter. In the CLEAN loop, $\mathbf{R}$ should be replaced by its estimate $\hat{\mathbf{R}}$ minus the estimated components until q, and also a constraint that α_q is to be positive should be included. This method was proposed in [3].

Using the AAR dirty image in the CLEAN loop is also possible, and the resulting CLEANed image was called LS-MVI in [3].

4.3 Matrix Formulations

Because our data model is linear, it is beneficial to represent the covariance model and all subsequent operations on it in a linear algebra framework. In this more abstract formulation, details are hidden and it becomes easier to recognize the connection of image formation to standard formulations and more generic approaches, such as matrix inversion and parametric estimation techniques.

4.3.1 Matrix Formulation of the Data Model

Let us start again from the data model given by Eq. (12) assuming an ideal situation, in which all gain factors are unity. For simplicity, we consider only a single frequency bin and STI interval, but all results can be generalized straightforwardly. The model for the signals arriving at the antenna array is thus

$$\mathbf{x}(n) = \mathbf{A}\mathbf{s}(n) + \mathbf{n}(n)$$

and the covariance of $\mathbf{x}$ is (viz. (14))

$$\mathbf{R} = \mathbf{A}\boldsymbol{\Sigma}_s\mathbf{A}^H + \boldsymbol{\Sigma}_n\,.$$

We have available a sample covariance matrix

$$\hat{\mathbf{R}} = \frac{1}{N}\sum_n \mathbf{x}(n)\mathbf{x}(n)^H$$

which serves as the input data for the imaging step. Let us now vectorize this data model by defining

$$\hat{\mathbf{r}} = \mathrm{vec}(\hat{\mathbf{R}}), \qquad \mathbf{r} = \mathrm{vec}(\mathbf{R})$$

where $\mathbf{r}$ has the data model (using (4))

$$\mathbf{r} = (\bar{\mathbf{A}} \circ \mathbf{A})\boldsymbol{\sigma}_s + \mathrm{vec}(\boldsymbol{\Sigma}_n).$$

If $\boldsymbol{\Sigma}_n$ is diagonal, we can write $\mathrm{vec}(\boldsymbol{\Sigma}_n) = (\mathbf{I}\circ\mathbf{I})\boldsymbol{\sigma}_n$, where $\boldsymbol{\sigma}_n$ is a vector containing the diagonal entries of $\boldsymbol{\Sigma}_n$. Define $\mathbf{M}_s = \bar{\mathbf{A}} \circ \mathbf{A}$ and $\mathbf{M}_n = \mathbf{I} \circ \mathbf{I}$. Then

$$\mathbf{r} = \mathbf{M}_s\boldsymbol{\sigma}_s + \mathbf{M}_n\boldsymbol{\sigma}_n = [\mathbf{M}_s \;\; \mathbf{M}_n]\begin{bmatrix}\boldsymbol{\sigma}_s \\ \boldsymbol{\sigma}_n\end{bmatrix} = \mathbf{M}\boldsymbol{\sigma}. \tag{27}$$

In this formulation, several modifications can be introduced. E.g., a non-diagonal noise covariance matrix $\boldsymbol{\Sigma}_n$ will lead to a more general $\mathbf{M}_n$, while if $\boldsymbol{\Sigma}_n = \sigma_n^2\mathbf{I}$, we have $\mathbf{M}_n = \mathrm{vec}(\mathbf{I})$ and $\boldsymbol{\sigma}_n = \sigma_n^2$. Some other options are discussed in [47]. Also, if we have already an estimate of $\boldsymbol{\sigma}_n$, we can subtract it and write the model as

$$\mathbf{r}' := \mathbf{r} - \mathbf{M}_n\boldsymbol{\sigma}_n = \mathbf{M}_s\boldsymbol{\sigma}_s \tag{28}$$

The available measurements $\hat{\mathbf{r}}$ should be modified in the same way. This model is similar to (27), with the advantage that the number of unknown parameters in $\boldsymbol{\sigma}$ is smaller.

We can further write

$$\hat{\mathbf{r}} \;=\; \mathbf{r} + \mathbf{w} \;=\; \mathbf{M}\boldsymbol{\sigma} + \mathbf{w}, \tag{29}$$

where $\hat{\mathbf{r}}$ is the available "measurement data", $\mathbf{r}$ is its mean (expected value), and $\mathbf{w}$ is additive noise due to finite samples. It is not hard to derive that (for Gaussian signals) the covariance of this noise is [47]

$$\mathbf{C}_w = E(\hat{\mathbf{r}} - \mathbf{r})(\hat{\mathbf{r}} - \mathbf{r})^H = \frac{1}{N}(\bar{\mathbf{R}} \otimes \mathbf{R})$$

where N is the number of samples on which $\hat{\mathbf{R}}$ is based. We have thus written our original data model on $\mathbf{x}$ as a similar data model on $\hat{\mathbf{r}}$. Many estimation techniques from the literature that are usually applied to data models for $\mathbf{x}$ can be applied to the data model for $\mathbf{r}$. Furthermore, it is straightforward to extend this vectorized

formulation to include multiple snapshots over time and frequency to increase the amount of measurement data and thus to improve the imaging result: Simply stack the covariance data in $\hat{\mathbf{r}}$ and include the model structure in $\mathbf{M}$; note that $\boldsymbol{\sigma}$ remains unchanged. Similarly, assuming a diagonal noise covariance matrix, astronomers often drop the autocorrelation terms (diagonal of $\hat{\mathbf{R}}$), rather than attempting to do the subtraction in (28); this corresponds to dropping rows in $\mathbf{M}$ and corresponding rows in $\mathbf{M}_s$, and leads to a model similar to (28) but without the autocorrelation terms.

The unknown parameters in the data model are, first of all, the powers $\boldsymbol{\sigma}$. These appear linear in the model. Regarding the positions of the sources, we can consider two cases:

1. We consider a point source model with a "small" number of sources. In that case, $\mathbf{A} = \mathbf{A}(\boldsymbol{\theta})$ and $\mathbf{M} = \mathbf{M}(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is some parameterization of the unknown locations of the sources (the position vectors $\mathbf{l}_q$ for each source). These enter in a nonlinear way into the model $\mathbf{M}(\boldsymbol{\theta})$. The image $I(\mathbf{l})$ is constructed following (16), usually convolved with a synthetic beam $B_{synth}(\mathbf{l})$ to make the image look nicer. The resulting estimation techniques are very much related to *direction of arrival* (DOA) estimation in array signal processing, with a rich literature.
2. Alternatively, we consider a model where, for each pixel in the image, we assume a corresponding point source: the source positions $\mathbf{l}_q$ directly correspond to the pixels in the image. This can lead to a large number of sources. With the locations of the pixels predetermined, $\mathbf{M}$ is a priori known and not a function of $\boldsymbol{\theta}$, but $\mathbf{M}$ will have many columns (one for each pixel-source). The image $I(\mathbf{l})$ has a one-to-one relation to the source power vector $\boldsymbol{\sigma}_s$, we can thus regard $\boldsymbol{\sigma}_s$ as the image in this case.

We need to pose several requirements on $\mathbf{M}$ or $\mathbf{M}(\boldsymbol{\theta})$ to ensure identifiability. First of all, in the first case we must have $\mathbf{M}(\boldsymbol{\theta}) = \mathbf{M}(\boldsymbol{\theta}') \rightarrow \boldsymbol{\theta} = \boldsymbol{\theta}'$, otherwise we cannot uniquely find $\boldsymbol{\theta}$ from $\mathbf{M}$. Furthermore, for both cases we will require that $\mathbf{M}$ is a tall matrix (more rows than columns) and has full column rank, so that it has a left inverse (this will allow to estimate $\boldsymbol{\sigma}$). This puts a limit on the number of sources in the image (number of columns of $\mathbf{M}$) in relation to the number of observations (rows). If more snapshots (STIs) and/or multiple frequencies are available, as is the case in practice, then $\mathbf{M}$ will become taller, and more sources can be estimated thus increasing the resolution. If $\mathbf{M}$ is not tall, then there are some ways to generalize this using prior knowledge on the image, e.g. via the context of compressive sampling where we can have $\mathbf{M}$ wide as long as $\boldsymbol{\sigma}$ is sparse [59], which we will briefly discuss in Sect. 4.5.5.

For the moment, we will continue with the second formulation: one source per pixel, fewer pixels than available correlation data.

4.3.2 Matrix Formulation of Imaging via Beamforming

Let us now again interpret the "beamforming image" (22) as a linear transformation on the covariance data $\hat{\mathbf{r}}$. We can stack all image values $I(\mathbf{l})$ over all pixels $\mathbf{l}_q$ into a single vector $\mathbf{i}$, and similarly, we can collect the weights $\mathbf{w}(\mathbf{l})$ over all pixels into a single matrix $\mathbf{W} = [\mathbf{w}(\mathbf{l}_1), \cdots, \mathbf{w}(\mathbf{l}_Q)]$. From (3), we know that $\mathbf{w}^H \mathbf{R} \mathbf{w} = (\overline{\mathbf{w}} \otimes \mathbf{w})^H \mathrm{vec}(\hat{\mathbf{R}})$, so that we can write

$$\hat{\mathbf{i}}_{BF} = (\overline{\mathbf{W}} \circ \mathbf{W})^H \hat{\mathbf{r}} \,. \tag{30}$$

We saw before that the dirty image is obtained if we use the matched filter. In this case, we have $\mathbf{W} = \frac{1}{J}\mathbf{A}$, where $\mathbf{A}$ contains the array response vectors $\mathbf{a}(\mathbf{l})$ for every pixel $\mathbf{l}_q$ of interest. In this case, the image is

$$\hat{\mathbf{i}}_D = \frac{1}{J^2}(\bar{\mathbf{A}} \circ \mathbf{A})^H \hat{\mathbf{r}} = \frac{1}{J^2}\mathbf{M}_s^H \hat{\mathbf{r}} \,. \tag{31}$$

The expected value of the image is obtained by using $\mathbf{r} = \mathbf{M}\boldsymbol{\sigma}$:

$$\mathbf{i}_D = \frac{1}{J^2}\mathbf{M}_s^H \mathbf{M}\boldsymbol{\sigma} = \frac{1}{J^2}(\mathbf{M}_s^H \mathbf{M}_s)\boldsymbol{\sigma}_s + \frac{1}{J^2}(\mathbf{M}_s^H \mathbf{M}_n)\boldsymbol{\sigma}_n \,.$$

The quality or "performance" of the image, or how close $\hat{\mathbf{i}}_D$ is to $\mathbf{i}_D$, is related to its covariance,

$$\mathrm{cov}(\hat{\mathbf{i}}_D) = E\{(\hat{\mathbf{i}}_D - \mathbf{i}_D)(\hat{\mathbf{i}}_D - \mathbf{i}_D)^H\} = \frac{1}{J^4}\mathbf{M}_s^H \mathbf{C}_w \mathbf{M}_s$$

where $\mathbf{C}_w = \frac{1}{N}(\bar{\mathbf{R}} \otimes \mathbf{R})$ is the covariance of the noise on the covariance data. Since usually the astronomical sources are much weaker than the noise (often at least by a factor 100), we can approximate $\mathbf{R} \approx \boldsymbol{\Sigma}_n$. If the noise is spatially white, $\boldsymbol{\Sigma}_n = \sigma_n^2 \mathbf{I}$, we obtain for the covariance of $\hat{\mathbf{i}}_D$

$$\mathrm{cov}(\hat{\mathbf{i}}_D) \approx \frac{\sigma_n^4}{J^4 N}\mathbf{M}_s^H \mathbf{M}_s \,.$$

The variance in the image is given by the diagonal of this expression. From this and the structure of $\mathbf{M}_s = (\bar{\mathbf{A}} \circ \mathbf{A})$ and the structure of $\mathbf{A}$, we can see that the variance on each pixel in the dirty image is constant, $\sigma_n^4/(J^2 N)$, but that the noise on the image is correlated, possibly leading to visible structures in the image. This is a general phenomenon. Similar equations can be derived for the MVDR image and the AAR image.

4.4 Parametric Image Estimation

In Sect. 4.2, we discussed various deconvolution algorithms based on the CLEAN algorithm. This algorithm uses a successive approximation of the dirty image using a point source model. Alternatively, we take a model-based approach. The imaging problem is formulated as a parametric estimation problem where certain parameters (source locations, powers, noise variance) are unknown and need to be estimated. Although we start from a Maximum Likelihood formulation, we will quickly arrive at a more feasible Least Squares approach. The discussion was presented in [45] and follows to some extent [47], which is a general array processing approach to a very similar problem and can be read for further details.

4.4.1 Weighted Least Squares Imaging

The image formation problem can be formulated as a maximum likelihood (ML) estimation problem, and solving this problem should provide a statistically efficient estimate of the parameters. Since all signals are assumed to be i.i.d. Gaussian signals, the derivation is standard and the ML estimates are obtained by minimizing the negative log-likelihood function [47]

$$\{\hat{\boldsymbol{\sigma}}, \hat{\boldsymbol{\theta}}\} = \arg\min_{\boldsymbol{\sigma},\boldsymbol{\theta}} \ \ln|\mathbf{R}(\boldsymbol{\sigma},\boldsymbol{\theta})| + \mathrm{tr}\left(\mathbf{R}^{-1}(\boldsymbol{\sigma},\boldsymbol{\theta})\hat{\mathbf{R}}\right) \tag{32}$$

where $|\cdot|$ denotes the determinant. $\mathbf{R}(\boldsymbol{\sigma},\boldsymbol{\theta})$ is the model, i.e., $\mathrm{vec}(\mathbf{R}(\boldsymbol{\sigma},\boldsymbol{\theta})) = \mathbf{r} = \mathbf{M}(\boldsymbol{\theta})\boldsymbol{\sigma}$, where $\boldsymbol{\theta}$ parameterizes the source locations, and $\boldsymbol{\sigma}$ their intensities.

We will first consider the overparameterized case, where $\boldsymbol{\theta}$ is a (known) list of all pixel coordinates in the image, and each pixel corresponds to a source. In this case, $\mathbf{M}$ is a priori known, the model is linear, and the ML problem reduces to a Weighted Least Squares (WLS) problem to match $\hat{\mathbf{r}}$ to the model $\mathbf{r}$:

$$\hat{\boldsymbol{\sigma}} = \arg\min_{\boldsymbol{\sigma}} \|\mathbf{C}_w^{-1/2}(\hat{\mathbf{r}} - \mathbf{r})\|_2^2 = \arg\min_{\boldsymbol{\sigma}} \ (\hat{\mathbf{r}} - \mathbf{M}\boldsymbol{\sigma})^H \mathbf{C}_w^{-1}(\hat{\mathbf{r}} - \mathbf{M}\boldsymbol{\sigma}) \tag{33}$$

where we fit the "data" $\hat{\mathbf{r}}$ to the model $\mathbf{r} = \mathbf{M}\boldsymbol{\sigma}$. The correct weighting is the inverse of the covariance of the residual, $\mathbf{w} = \hat{\mathbf{r}} - \mathbf{r}$, i.e., the noise covariance matrix $\mathbf{C}_w = \frac{1}{N}(\bar{\mathbf{R}} \otimes \mathbf{R})$. For this, we may also use the estimate $\hat{\mathbf{C}}_w$ obtained by using $\hat{\mathbf{R}}$ instead of $\mathbf{R}$. Using the assumption that the astronomical sources are much weaker than the noise we could contemplate to use $\mathbf{R} \approx \boldsymbol{\Sigma}_n$ for the weighting. If the noise is spatially white, $\boldsymbol{\Sigma}_n = \sigma_n^2\mathbf{I}$, the weighting can then even be omitted.

The solution of (33) is obtained by applying the pseudo-inverse,

$$\hat{\boldsymbol{\sigma}} = [\mathbf{C}_w^{-1/2}\mathbf{M}]^{\dagger}\mathbf{C}_w^{-1/2}\hat{\mathbf{r}} = (\mathbf{M}^H\mathbf{C}_w^{-1}\mathbf{M})^{-1}\mathbf{M}^H\mathbf{C}_w^{-1}\hat{\mathbf{r}} =: \mathbf{M}_d^{-1}\hat{\boldsymbol{\sigma}}_d \tag{34}$$

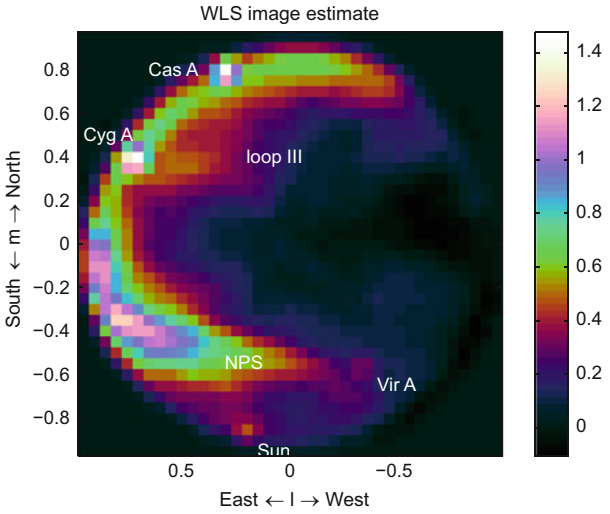


Fig. 7 Image corresponding to the WLS formulation (34)

where

$$\mathbf{M}_d := \mathbf{M}^H \mathbf{C}_w^{-1} \mathbf{M}, \qquad \hat{\boldsymbol{\sigma}}_d := \mathbf{M}^H \mathbf{C}_w^{-1} \hat{\mathbf{r}}.$$

Here, we can consider the term $\hat{\boldsymbol{\sigma}}_d = \mathbf{M}^H \mathbf{C}_w^{-1} \hat{\mathbf{r}}$ as a "dirty image": it is comparable to (31), although we have introduced a weighting by $\mathbf{C}_w^{-1}$ and estimate the noise covariance parameters $\boldsymbol{\sigma}_n$ as well as the source powers in $\boldsymbol{\sigma}_s$ (the actual image). The factor $1/J^2$ in (31) can be seen as a crude approximation of $\mathbf{M}_d^{-1}$.

Figure 7 shows an example WLS image for a single LOFAR station. The image was obtained by deconvolving the dirty image from 25 STIs, each consisting of 10 s data in 25 frequency channels of 156 kHz wide taken from the band 45–67 MHz, avoiding the locally present radio interference. As this shows data from a single LOFAR station, with a relatively small maximal baseline (65 m), the resolution is limited and certainly not representative of the capabilities of the full LOFAR array. The resolution (number of pixels) in this image is kept limited (about 1000) for reasons discussed below.

The term $\mathbf{M}_d^{-1} = (\mathbf{M}^H \mathbf{C}_w^{-1} \mathbf{M})^{-1}$ is a deconvolution operation. This inversion can only be carried out if the deconvolution matrix $\mathbf{M}_d = \mathbf{M}^H \mathbf{C}_w^{-1} \mathbf{M}$ is not rank deficient. This requires at least that $\mathbf{M}$ is a tall matrix ("less pixels than observations" in case we take one source per pixel). Thus, high resolution WLS imaging is only possible if a limited number of sources is present. The condition number of $\mathbf{M}_d$, i.e., the ratio of the largest to the smallest eigenvalue of $\mathbf{M}_d$, gives important information on our ability to compute its inverse: LS theory tells us that the noise on $\hat{\boldsymbol{\sigma}}_d$ could, in the worst case, be magnified by this factor. The optimal (smallest) condition number of any matrix is 1, which is achieved if $\mathbf{M}_d$ is a scaling of the identity matrix, or if

the columns of $\mathbf{C}_w^{-1/2}\mathbf{M}$ are all orthogonal to each other. If the size of $\mathbf{M}$ becomes less tall, then the condition number of $\mathbf{M}_d$ becomes larger (worse), and once it is a wide matrix, $\mathbf{M}$ is singular and the condition number will be infinite. Thus, we have a trade-off between the resolution (number of pixels in the image) and the noise enhancement.

The definition of $\mathbf{M}_d$ shows that it is not data dependent, and it can be precomputed for a given telescope configuration and observation interval. It is thus possible to explore this trade-off beforehand. To avoid numerical instabilities (noise enhancement), we would usually compute a regularized inverse or pseudo-inverse of this matrix, e.g., by first computing the eigenvalue decomposition

$$\mathbf{M}_d = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^H$$

where $\mathbf{U}$ contains the (orthonormal) eigenvectors and $\boldsymbol{\Lambda}$ is a diagonal matrix containing the eigenvalues, sorted from large to small. Given a threshold ϵ on the eigenvalues, we can define $\tilde{\boldsymbol{\Lambda}}$ to be a diagonal matrix containing only the eigenvalues larger than ϵ, and $\tilde{\mathbf{U}}$ a matrix containing the corresponding eigenvectors. The ϵ-threshold pseudo-inverse is then given by

$$\mathbf{M}_d^{\dagger} := \tilde{\mathbf{U}}\tilde{\boldsymbol{\Lambda}}^{-1}\tilde{\mathbf{U}}^H$$

and the resulting image is

$$\boldsymbol{\sigma} = \tilde{\mathbf{U}}\tilde{\boldsymbol{\Lambda}}^{-1}\tilde{\mathbf{U}}^H\boldsymbol{\sigma}_d\,. \tag{35}$$

This can be called the "Karhunen-Loève" image, as the rank reduction is related to the Karhunen-Loève transform (KLT). It corresponds to selecting an optimal (Least Squares) set of basis vectors on which to project a certain data set, here $\boldsymbol{\sigma}_d$.

An example KLT image is shown in Fig. 8. In this image, the number of pixels is much larger than before in Fig. 7 (about 9000), but the rank of the matrix $\mathbf{M}_d$ is truncated at $1/200$ times the largest eigenvalue, leaving about 1300 out of 9000 image components. The result is not quite satisfactory: the truncation to a reduced basis results in annoying ripple artefacts in the image.

Computing the eigenvalue decomposition for large matrices is complex. A computationally simpler alternative is to compute a regularized inverse of $\mathbf{M}_d$, i.e., to take the inverse of $\mathbf{M}_d + \epsilon\mathbf{I}$. This should yield similar (although not identical) results.

If we use the alternative sky model where we assume a point source model with a "small" number of sources ($\mathbf{M} = \mathbf{M}(\boldsymbol{\theta})$), then the conditioning of $\mathbf{M}_d$, and thus the performance of the deconvolution, is directly related to the number of sources and their spatial distribution.

The performance of the method is assessed by looking at the covariance of the resulting image (plus noise parameters) $\hat{\boldsymbol{\sigma}}$ in (34). It is given by

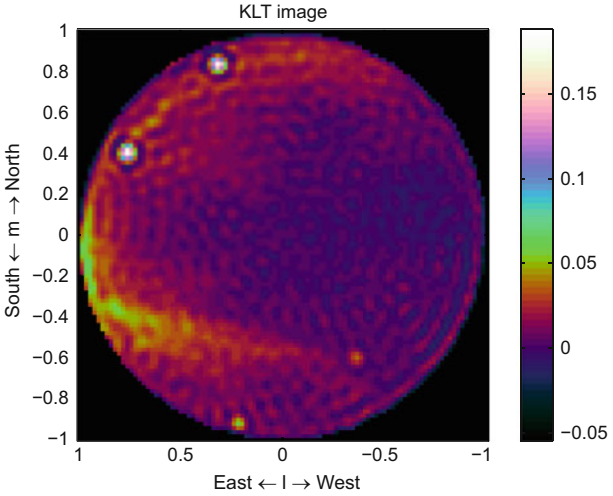


Fig. 8 Image corresponding to the KLT solution (35)

$$\begin{aligned}\mathbf{C}_{\boldsymbol{\sigma}} &= (\mathbf{M}^H\mathbf{C}_w^{-1}\mathbf{M})^{-1}\mathbf{M}^H\mathbf{C}_w^{-1}(\mathbf{C}_w)\mathbf{C}_w^{-1}\mathbf{M}(\mathbf{M}^H\mathbf{C}_w^{-1}\mathbf{M})^{-1} \\ &= (\mathbf{M}^H\mathbf{C}_w^{-1}\mathbf{M})^{-1} \quad = \quad \mathbf{M}_d^{-1}\,.\end{aligned}$$

This again shows that the performance of the imaging method follows directly from the conditioning of the deconvolution matrix $\mathbf{M}_d$. If $\mathbf{M}_d$ is sufficiently well conditioned, the noise on the image is limited, otherwise it may be large. The formulation also shows that the pixels in the image are correlated ($\mathbf{M}_d$ is in general not diagonal), as we obtained before for the dirty image.

Similarly, if we use the pseudo-inverse $\mathbf{M}_d^\dagger = \tilde{\mathbf{U}}\tilde{\boldsymbol{\Lambda}}^{-1}\tilde{\mathbf{U}}^H$ for the deconvolution, then we obtain $\mathbf{C}_{\boldsymbol{\sigma}} = \mathbf{M}_d^\dagger$. In this case, the noise enhancement depends on the chosen threshold ϵ. Also, the rank of $\mathbf{C}_{\boldsymbol{\sigma}}$ depends on this threshold, and since it is not full rank, the number of independent components (sources) in the image is smaller than the number of shown pixels: the rank reduction defines a form of interpolation.

Using a rank truncation for radio astronomy imaging was already suggested in [10]. Unfortunately, if the number of pixels is large, this technique by itself is not sufficient to obtain good images, e.g., the resulting pixels may not all be positive, which is unplausible for an intensity image. Thus, the overparameterized case requires additional constraints; some options are discussed in Sects. 4.5.4 and 4.5.5.

4.4.2 Estimating the Position of the Sources

Let us now consider the use of the alternative formulation, where we write $\mathbf{A} = \mathbf{A}(\boldsymbol{\theta})$ and $\mathbf{M} = \mathbf{M}(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ captures the positions of the limited number of sources in

the image. In this case, we have to estimate both $\boldsymbol{\sigma}$ and $\boldsymbol{\theta}$. If we start again from the ML formulation (32), it does not seem feasible to solve this minimization problem in closed form. However, we can again resort to the WLS covariance matching problem and solve instead

$$\begin{aligned}\{\hat{\boldsymbol{\sigma}}, \hat{\boldsymbol{\theta}}\} &= \arg\min_{\boldsymbol{\sigma},\boldsymbol{\theta}} \|\mathbf{C}_w^{-1/2}[\hat{\mathbf{r}} - \mathbf{r}(\boldsymbol{\sigma}, \boldsymbol{\theta})]\|^2 \\ &= \arg\min_{\boldsymbol{\sigma},\boldsymbol{\theta}} [\hat{\mathbf{r}} - \mathbf{M}(\boldsymbol{\theta})\boldsymbol{\sigma}]^H \mathbf{C}_w^{-1}[(\hat{\mathbf{r}} - \mathbf{M}(\boldsymbol{\theta})\boldsymbol{\sigma}]\,. \end{aligned} \tag{36}$$

It is known that the resulting estimates are, for a large number of samples, equivalent to ML estimates and therefore asymptotically efficient [47].

The WLS problem is separable: suppose that the optimal $\boldsymbol{\theta}$ is known, so that $\mathbf{M} = \mathbf{M}(\boldsymbol{\theta})$ is known, then the corresponding $\boldsymbol{\sigma}$ will satisfy the solution which we found earlier:

$$\hat{\boldsymbol{\sigma}} = (\mathbf{M}^H \mathbf{C}_w^{-1} \mathbf{M})^{-1} \mathbf{M}^H \mathbf{C}_w^{-1} \hat{\mathbf{r}}\,.$$

Substituting this solution back into the problem, we obtain

$$\begin{aligned}\hat{\boldsymbol{\theta}} &= \arg\min_{\boldsymbol{\theta}}\ \hat{\mathbf{r}}^H [\mathbf{I} - \mathbf{M}(\boldsymbol{\theta})(\mathbf{M}(\boldsymbol{\theta})^H \mathbf{C}_w^{-1} \mathbf{M}(\boldsymbol{\theta}))^{-1} \mathbf{M}(\boldsymbol{\theta})^H \mathbf{C}_w^{-1}]^H \ \cdot \\ &\qquad \cdot \mathbf{C}_w^{-1} \cdot [\mathbf{I} - \mathbf{M}(\boldsymbol{\theta})(\mathbf{M}(\boldsymbol{\theta})^H \mathbf{C}_w^{-1} \mathbf{M}(\boldsymbol{\theta}))^{-1} \mathbf{M}(\boldsymbol{\theta})^H \mathbf{C}_w^{-1}] \hat{\mathbf{r}} \\ &= \arg\min_{\boldsymbol{\theta}}\ \hat{\mathbf{r}}^H \mathbf{C}_w^{-1/2} (\mathbf{I} - \boldsymbol{\Pi}(\boldsymbol{\theta})) \mathbf{C}_w^{-1/2} \hat{\mathbf{r}} \\ &= \arg\max_{\boldsymbol{\theta}}\ \hat{\mathbf{r}}^H \mathbf{C}_w^{-1/2} \boldsymbol{\Pi}(\boldsymbol{\theta}) \mathbf{C}_w^{-1/2} \hat{\mathbf{r}} \end{aligned}$$

where $\boldsymbol{\Pi}(\boldsymbol{\theta}) = \mathbf{C}_w^{-1/2} \mathbf{M}(\boldsymbol{\theta}) \big(\mathbf{M}(\boldsymbol{\theta})^H \mathbf{C}_w^{-1} \mathbf{M}(\boldsymbol{\theta})\big)^{-1} \mathbf{M}(\boldsymbol{\theta})^H \mathbf{C}_w^{-1/2}$.

$\boldsymbol{\Pi}(\boldsymbol{\theta})$ is an orthogonal projection: $\boldsymbol{\Pi}^2 = \boldsymbol{\Pi}$, $\boldsymbol{\Pi}^H = \boldsymbol{\Pi}$. The projection is onto the column span of $\mathbf{M}'(\boldsymbol{\theta}) := \mathbf{C}_w^{-1/2} \mathbf{M}(\boldsymbol{\theta})$. The estimation of the source positions $\boldsymbol{\theta}$ is nonlinear. It could be obtained iteratively using a Newton iteration (cf. [47]). The sources can also be estimated sequentially [47], which provides an alternative to the CLEAN algorithm.

4.4.3 Preconditioned WLS

WLS imaging can be improved using preconditioning, and this has an interesting relation to the adaptive beamforming techniques discussed earlier. From this point forward we assume that an estimate of the noise has been subtracted from the images as in (28) such that $\mathbf{M} = \mathbf{M}_s$ and $\boldsymbol{\sigma} = \boldsymbol{\sigma}_s$.

If $\mathbf{M}$ has full column rank then $\mathbf{H}_{\mathrm{LS}} := \mathbf{M}^H\mathbf{M}$ and $\mathbf{H}_{\mathrm{WLS}} := \mathbf{M}^H\mathbf{C}_w^{-1}\mathbf{M}$ are non-singular and there exists a unique solution to LS and WLS. For example the solution to the LS imaging becomes

$$\boldsymbol{\sigma} = \mathbf{H}_{\mathrm{LS}}^{-1}\hat{\boldsymbol{\sigma}}_{\mathrm{D}} \tag{37}$$

where $\hat{\boldsymbol{\sigma}}_{\mathrm{D}} = \mathbf{M}^H\hat{\mathbf{r}}$ is the estimated dirty image. Unfortunately, if the number of pixels is large then $\mathbf{H}_{\mathrm{LS}}$ and $\mathbf{H}_{\mathrm{WLS}}$ become ill-conditioned or even singular. Generally, we need to improve the conditioning of the deconvolution matrices and to find appropriate regularizations.

One way to improve the conditioning of a matrix is by applying a preconditioner. The most widely used and simplest preconditioner is the Jacobi preconditioner [1] which, for any matrix $\mathbf{M}$, is given by $[\mathrm{diag}(\mathbf{M})]^{-1}$. Let $\mathbf{D}_{\mathrm{WLS}} = \mathrm{diag}(\mathbf{H}_{\mathrm{WLS}})$, then by applying this preconditioner to $\mathbf{H}_{\mathrm{WLS}}$ we obtain

$$[\mathbf{D}_{\mathrm{WLS}}^{-1}\mathbf{H}_{\mathrm{WLS}}]\boldsymbol{\sigma} = \mathbf{D}_{\mathrm{WLS}}^{-1}\hat{\boldsymbol{\sigma}}_{\mathrm{WLS}} \tag{38}$$

where $\hat{\boldsymbol{\sigma}}_{\mathrm{WLS}} = \mathbf{M}^H\mathbf{C}_w^{-1}\hat{\mathbf{r}}$. We take a closer look at $\mathbf{D}_{\mathrm{WLS}}^{-1}\hat{\boldsymbol{\sigma}}_{\mathrm{WLS}}$. For a single STI

$$\begin{aligned}\mathbf{H}_{\mathrm{WLS}} &= (\bar{\mathbf{A}} \circ \mathbf{A})^H(\hat{\mathbf{R}}^{-T} \otimes \hat{\mathbf{R}}^{-1})(\bar{\mathbf{A}} \circ \mathbf{A}) \\ &= (\mathbf{A}^T\hat{\mathbf{R}}^{-T}\bar{\mathbf{A}}) \odot (\mathbf{A}^H\hat{\mathbf{R}}^{-1}\mathbf{A})\end{aligned}$$

and

$$\mathbf{D}_{\mathrm{WLS}}^{-1} = \begin{bmatrix} \frac{1}{(\mathbf{a}_1^H\hat{\mathbf{R}}^{-1}\mathbf{a}_1)^2} & & \\ & \ddots & \\ & & \frac{1}{(\mathbf{a}_Q^H\hat{\mathbf{R}}^{-1}\mathbf{a}_Q)^2} \end{bmatrix}, \tag{39}$$

where we have assumed that $\mathbf{a}_i$ is normalized by a factor $1/\sqrt{J}$ such that $\mathbf{a}_i^H\mathbf{a}_i = 1$. This means that

$$\begin{aligned}\mathbf{D}_{\mathrm{WLS}}^{-1}\hat{\boldsymbol{\sigma}}_{\mathrm{WLS}} &= \mathbf{D}_{\mathrm{WLS}}^{-1}\left((\hat{\mathbf{R}}^{-T} \otimes \hat{\mathbf{R}}_{\mathrm{I}}^{-1})(\bar{\mathbf{A}} \circ \mathbf{A})\right)^H\hat{\mathbf{r}} \\ &= (\hat{\mathbf{R}}^{-T}\bar{\mathbf{A}}\mathbf{D}_{\mathrm{WLS}}^{-1/2} \circ \hat{\mathbf{R}}^{-1}\mathbf{A}\mathbf{D}_{\mathrm{WLS}}^{-1/2})^H\hat{\mathbf{r}}\end{aligned}$$

which is equivalent to a dirty image that is obtained by applying a beamformer of the form

$$\mathbf{w}_i = \frac{1}{\mathbf{a}_i^H\hat{\mathbf{R}}^{-1}\mathbf{a}_i}\hat{\mathbf{R}}^{-1}\mathbf{a}_i \tag{40}$$

to both sides of $\hat{\mathbf{R}}$ and stacking the results, $\hat{\sigma}_i = \mathbf{w}_i^H \hat{\mathbf{R}} \mathbf{w}_i$, of each pixel into a vector. This beamformer is the MVDR beamformer which we have introduced before! This shows that the Preconditioned WLS (PWLS) image (motivated from its connection to the maximum likelihood) is expected to exhibit the features of high-resolution beamforming associated with the MVDR. The PWLS was introduced in [45].

4.5 Constraints on the Image

Another approach to improve the conditioning of a problem is to introduce appropriate constraints on the solution. Typically, image formation algorithms exploit external information regarding the image in order to regularize the ill-posed problem. For example maximum entropy techniques [21] impose a smoothness condition on the image while the CLEAN algorithm [27] exploits a point source model wherein most of the image is empty, and this has recently been connected to sparse optimization techniques [59].

4.5.1 Non-negativity Constraint

A lower bound on the image is almost trivial: each pixel in the image represents the intensity at a certain direction, hence is non-negative. This is physically plausible, and to some extent already covered by CLEAN [41]. It is an explicit condition in a Non-Negative Least Squares (NNLS) formulation [10], which searches for a Least Squares fit while requiring that the solution $\boldsymbol{\sigma}$ has all entries $\sigma_i \geq 0$:

$$\begin{aligned} &\min_{\boldsymbol{\sigma}} \ \|\hat{\mathbf{r}} - \mathbf{M}\boldsymbol{\sigma}\|^2 \\ &\text{subject to } \mathbf{0} \leq \boldsymbol{\sigma} \end{aligned} \tag{41}$$

4.5.2 Dirty Image as Upper Bound

A second constraint follows if we also know an upper bound $\boldsymbol{\gamma}$ such that $\boldsymbol{\sigma} \leq \boldsymbol{\gamma}$, which will bound the pixel intensities from above. We will propose several choices for $\boldsymbol{\gamma}$.

By closer inspection of the ith pixel of the matched beamformer dirty image $\hat{\boldsymbol{\sigma}}_{\mathrm{D}}$, we note that its expected value is given by

$$\sigma_{\mathrm{D},i} = \mathbf{a}_i^H \mathbf{R} \mathbf{a}_i \, .$$

Using normalization $\mathbf{a}_i^H \mathbf{a}_i = 1$, we obtain

$$\sigma_{\mathrm{D},i} = \sigma_i + \mathbf{a}_i^H \mathbf{R}_r \mathbf{a}_i \, , \tag{42}$$

where

$$\mathbf{R}_r = \sum_{j \neq i} \sigma_j \mathbf{a}_j \mathbf{a}_j^H + \mathbf{R}_\mathbf{n} \tag{43}$$

is the contribution of all other sources and the noise. Note that $\mathbf{R}_r$ is positive-(semi)definite. Thus, (42) implies $\sigma_{\mathrm{D},i} \geq \sigma_i$ which means that the expected value of the matched beamformer dirty image forms an upper bound for the desired image, or

$$\boldsymbol{\sigma} \leq \boldsymbol{\sigma}_\mathrm{D} \,. \tag{44}$$

We can extend this concept to a more general beamformer $\mathbf{w}_i$. The output power of this beamformer, in the direction of the ith pixel, becomes

$$\sigma_{\mathbf{w},i} = \mathbf{w}_i^H \mathbf{R} \mathbf{w}_i = \sigma_i \mathbf{w}_i^H \mathbf{a}_i \mathbf{a}_i^H \mathbf{w}_i + \mathbf{w}_i^H \mathbf{R}_r \mathbf{w}_i \,. \tag{45}$$

If we require that

$$\mathbf{w}_i^H \mathbf{a}_i = 1 \tag{46}$$

we have

$$\sigma_{\mathbf{w},i} = \sigma_i + \mathbf{w}_i^H \mathbf{R}_r \mathbf{w}_i \,. \tag{47}$$

As before, the fact that $\mathbf{R}_r$ is positive definite implies that

$$\sigma_i \leq \sigma_{\mathbf{w},i} \,. \tag{48}$$

We can easily verify that the matched filter weights $\mathbf{w}_{\mathrm{D},i}$ as given in (20) satisfy (46) and, hence, that the resulting dirty image $\sigma_{\mathrm{D},i}$ is a specific upper bound.

4.5.3 Tightest Upper Bound

The next question is: What is the tightest upper bound for σ_i that we can construct using linear beamforming?

We can translate the problem of finding the tightest upper bound to the following optimization question:

$$\sigma_{\mathrm{opt},i} = \min_{\mathbf{w}_i} \mathbf{w}_i^H \mathbf{R} \mathbf{w}_i \tag{49}$$

$$\text{s.t. } \mathbf{w}_i^H \mathbf{a}_i = 1$$

where $\sigma_{\text{opt},i}$ would be this tightest upper bound. This optimization problem is exactly the same as the one used in Sect. 4.1.2 to obtain the MVDR beamformer. Hence

$$\mathbf{w}_i = \frac{1}{\mathbf{a}_i^H \mathbf{R}^{-1} \mathbf{a}_i} \mathbf{R}^{-1} \mathbf{a}_i .$$

This means that for a single STI the MVDR image is the tightest upper bound that can be constructed using beamformers.

Note that $\mathbf{w}_{\text{D},i}$ also satisfies the constraint in (46), i.e. $\mathbf{w}_{\text{D},i}^H \mathbf{a}_i = \mathbf{a}_i^H \mathbf{a}_i = 1$, but does not necessary minimize the output power $\mathbf{w}_i^H \mathbf{R} \mathbf{w}_i$, therefore the MVDR dirty image is smaller than the matched beamformer dirty image: $\boldsymbol{\sigma}_{\text{MVDR}} \leq \boldsymbol{\sigma}_{\text{D}}$. This relation also holds if $\mathbf{R}$ is replaced by the sample covariance $\hat{\mathbf{R}}$.

For multiple snapshots the tightest bound can be obtained by taking the minimum of the individual MVDR estimates [44]. The bound becomes

$$\sigma_{\text{opt},i} = \min_m \frac{1}{\mathbf{a}_{m,i} \mathbf{R}_m^{-1} \mathbf{a}_{m,i}} .$$

One problem with using this result in practice is that $\sigma_{\text{opt},i}$ depends on a single snapshot. Actual dirty images are based on the sample covariance matrix $\hat{\mathbf{R}}$ and hence they are random variables. If we use a sample covariance matrix $\hat{\mathbf{R}}$ instead of the true covariance matrix $\mathbf{R}$, this bound would be too noisy without any averaging. Hence we would like to find a beamformer that exhibits the same averaging behavior as the matched beamformer while being as tight as possible. Sardarabadi [44] shows that a modified multi-snapshot MVDR image can be defined as

$$\sigma_{\text{MVDR},i} = \frac{1}{\frac{1}{M} \sum_m \mathbf{a}_{m,i}^H \mathbf{R}_m^{-1} \mathbf{a}_{m,i}} , \tag{50}$$

which satisfies $\sigma_i \leq \sigma_{\text{MVDR},i} \leq \sigma_{\text{D},i}$ and produces a very tight bound.

4.5.4 Constrained WLS Imaging

Now that we have lower and upper bounds on the image, we can use these as constraints in the LS imaging problem to provide a regularization. The resulting constrained LS (CLS) imaging problem is

$$\begin{aligned} &\min_{\boldsymbol{\sigma}} \ \|\hat{\mathbf{r}} - \mathbf{M}\boldsymbol{\sigma}\|^2 \\ &\text{s.t. } \mathbf{0} \leq \boldsymbol{\sigma} \leq \boldsymbol{\gamma} \end{aligned} \tag{51}$$

where $\boldsymbol{\gamma}$ can be chosen either as $\boldsymbol{\gamma} = \boldsymbol{\sigma}_{\text{D}}$ for the matched beamformer dirty image or $\boldsymbol{\gamma} = \boldsymbol{\sigma}_{\text{MVDR}}$ for the MVDR dirty image.

The extension to constrained WLS leads to the problem formulation

$$\begin{aligned} &\min_{\boldsymbol{\sigma}} \|\mathbf{C}_w^{-1/2}\left(\hat{\mathbf{r}} - \mathbf{M}\boldsymbol{\sigma}\right)\|^2 \\ &\text{s.t. } \mathbf{0} \leq \boldsymbol{\sigma} \leq \boldsymbol{\gamma}\,. \end{aligned} \tag{52}$$

It is also recommended to include a preconditioner which, as was shown in Sect. 4.4.3, relates the WLS to the MVDR dirty image. However, because of the inequality constraints, (52) does not have a closed form solution and it is solved by an iterative algorithm. In order to have the relation between the WLS and MVDR dirty image during the iterations we introduce a change of variables of the form $\check{\boldsymbol{\sigma}} = \mathbf{D}\boldsymbol{\sigma}$, where $\check{\boldsymbol{\sigma}}$ is the new variable for the preconditioned problem and the diagonal matrix $\mathbf{D}$ is given in (39). The resulting constrained preconditioned WLS (CPWLS) optimization problem is

$$\begin{aligned} \check{\boldsymbol{\sigma}} = \arg\min_{\check{\boldsymbol{\sigma}}} &\|\mathbf{C}_w^{-1/2}\left(\hat{\mathbf{r}} - \mathbf{M}\mathbf{D}^{-1}\check{\boldsymbol{\sigma}}\right)\|^2 \\ &\text{s.t. } \mathbf{0} \leq \check{\boldsymbol{\sigma}} \leq \mathbf{D}\boldsymbol{\gamma} \end{aligned} \tag{53}$$

and the final image is found by setting $\boldsymbol{\sigma} = \mathbf{D}^{-1}\check{\boldsymbol{\sigma}}$. Here we used that $\mathbf{D}$ is a positive diagonal matrix so that the transformation to an upper bound for $\check{\boldsymbol{\sigma}}$ is correct. As mentioned, the dirty image that follows from the (unconstrained) Weighted Least Squares part of the problem is given by the MVDR image $\hat{\boldsymbol{\sigma}}_{\text{MVDR}}$.

These problems are convex and their solutions can be found using various numerical optimization techniques such as the active set method, as discussed in more detail in [45]. Some experimental results using non-negative constraints are shown in [23, 37, 51].

4.5.5 Imaging Using Sparse Reconstruction Techniques

Compressive sampling/sensing (CS) is a "new" topic, currently drawing wide attention. It is connected to random or non-uniform sampling, and as such, it has been used in radio astronomy for a long time. In the CS community, the recovery of full information from undersampled data is the central problem, and to regularize this problem, the main idea has been to exploit the sparsity of the solution: the number of nonzero entries in the solution is supposed to be small. This is measured by the ℓ_0-norm: $\|\boldsymbol{\sigma}\|_0$ is the number of nonzero entries in $\boldsymbol{\sigma}$. Optimizing using this norm is difficult, and therefore as a surrogate, the ℓ_1-norm is used.

To introduce this, let us start from the Least Squares formulation, and consider the KLT regularization. This constrains the solution image to lie on a basis determined by the dominant column span of $\mathbf{M}$ (possibly giving rise to artefacts). It is straightforward to show that this regularization is connected to adding a regularization term

$$\min_{\boldsymbol{\sigma}} \|\hat{\mathbf{r}} - \mathbf{M}\boldsymbol{\sigma}\|_2^2 + \lambda\|\boldsymbol{\sigma}\|_2$$

where λ is related to the truncation threshold used in the KLT. The used norm on $\boldsymbol{\sigma}$ is ℓ_2, the sum of squares, or the total "energy" of the image.

An alternative to this is to use a regularization term $\|\boldsymbol{\sigma}\|_1$ based on the ℓ_1 norm of $\boldsymbol{\sigma}$, or the sum of absolute values [35, 59]. The resulting problem is

$$\min_{\boldsymbol{\sigma}} \|\hat{\mathbf{r}} - \mathbf{M}\boldsymbol{\sigma}\|_2^2 + \lambda\|\boldsymbol{\sigma}\|_1$$

An alternative formulation of this problem is

$$\min_{\boldsymbol{\sigma}} \|\boldsymbol{\sigma}\|_1 \quad \text{subject to} \quad \|\hat{\mathbf{r}} - \mathbf{M}\boldsymbol{\sigma}\|_2^2 \leq \epsilon$$

where ϵ is threshold on the residual noise. Like for KLT, the results depend on the chosen noise threshold ϵ (or regularization parameter λ).

Minimizing the ℓ_1-norm is known to promote the sparsity of the solution vector. The implied sparsity assumption in the model poses that the sky is mostly empty. Although it has already long been suspected that CLEAN is related to ℓ_1-optimization [41] (in fact, it is now recognized as a Matching Pursuit algorithm [39]), CS theory states the general conditions under which this assumption is likely to recover the true image [35, 59]. Extensions are needed in case of extended emissions [37]. As images may consist of sources with different source structures, different sources may be best represented, i.e., best compressible, by different bases. This is the basic idea behind the Sparsity Averaging Reweighted Analysis (SARA) algorithm, which aims to find the sparsest representation using an overdetermined dictionary composed of multiple complete bases [11, 12].

4.5.6 Comparison of Regularization Techniques

In this section, we discussed a number of constraints to regularize the ill-posed inverse imaging problem: non-negativity, upper bound, and sparsity of the image. This can be combined into a single problem,

$$\begin{aligned} \min_{\check{\boldsymbol{\sigma}}} \ & \|\mathbf{C}_w^{-1/2}\left(\hat{\mathbf{r}} - \mathbf{M}\mathbf{D}^{-1}\check{\boldsymbol{\sigma}}\right)\|^2 + \lambda\|\mathbf{D}^{-1}\boldsymbol{\sigma}\| \\ & \text{s.t. } \mathbf{0} \leq \check{\boldsymbol{\sigma}} \leq \mathbf{D}\boldsymbol{\gamma} \end{aligned} \tag{54}$$

where $\mathbf{D}$ is an optional preconditioner, the resulting image is $\boldsymbol{\sigma} = \mathbf{D}^{-1}\check{\boldsymbol{\sigma}}$, and the norm is either ℓ_1 or ℓ_2. Many variations on this problem are possible. Taken by itself, the non-negativity constraint is already known to be a strong constraint for regularization. It can even be shown that, when certain conditions are satisfied, the non-negativity constraint alone already promotes a sparse solution [20]. In cases where there is a combination of sparse and extended structures in the image, an ℓ_2 regularization might be more appropriate.

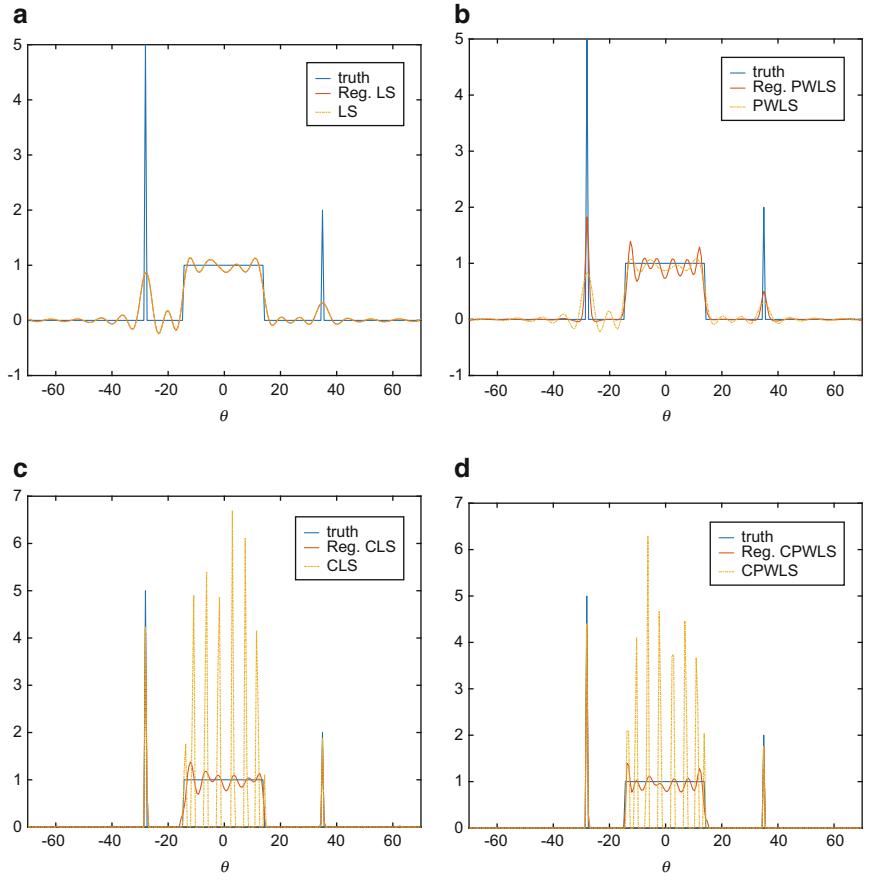


Fig. 9 Solutions for different algorithms with and without regularization; (**a**) Unconstrained LS. (**b**) Unconstrained PWLS. (**c**) Constrained LS. (**d**) Constrained PWLS

To illustrate the effects of regularization, constraints, and preconditioning, we consider a 1D "image" reconstruction example. A uniform linear array (ULA) with 20 receivers is simulated. The array is exposed to two point sources with magnitudes 5 and 2 and an extended rectangular source with a magnitude of 1. Because it is a ULA, $\text{rank}(\mathbf{M}) = 2J - 1 = 39$, while the number of pixels is $Q = 245$. This shows that $\mathbf{H}_{\text{LS}} = \mathbf{M}^H\mathbf{M}$ is singular. We use ℓ_2-norm regularization with a regularization coefficient $\lambda = 1/\sqrt{N}$ where $N = 1000$ is the number of samples in a single STI.

Figure 9 shows the result of the various estimation techniques with and without bound constraints and regularization. Figure 9a shows the result of standard LS with and without regularization, Fig. 9b shows similar results for unconstrained Preconditioned WLS, Fig. 9c incorporates the bound constraints for the LS problem, and Fig. 9d shows the results for CPWLS.

The figures show the following:

- Both standard LS and PWLS are unable to recover the point sources and suffer from high sidelobe levels. The regularization does not seem to affect the LS solution while it improves the sidelobe behavior in the PWLS solution at the cost of less accurate estimates for the extended structure.
- Both Constrained LS and Constrained PWLS without regularization attempt to model the extended structure using a series of point sources. This is the consequence of the non-negativity constraint which tends to promote sparsity.
- For CLS and CPWLS an ℓ_2-norm regularization helps with the recovery of the extended structure. The value of $\lambda = 1/\sqrt{N}$ seems to be a good balance for both extended and point sources.

5 Calibration

5.1 Non-ideal Measurements

In the previous section we showed that there are many options to make an image from radio interferometric measurements. However, we assumed that these measurements were done under ideal circumstances, such that the gain matrix in our data model given by (14) only contains ones. In practice, there are several effects that make matters more complicated causing $\mathbf{G} \neq \mathbf{1}\mathbf{1}^H$ (where we omitted the STI index m for convenience of notation as we will initially consider calibration on a per-STI basis). These effects need to be estimated and corrected for in a process called *calibration*. For this, some reference information is needed. In this section, we will assume that the locations and powers of Q reference sources are known, where Q can be small (order 1 to 10) or large (up to a complete image). In practice, calibration is an integral part of the imaging step, and not a separate phase as we will see in Sect. 6. The model given by (14) is not identifiable in its generality unless we make some assumptions on the structure of $\mathbf{G}$ (in the form of a suitable parameterization) and describe how it varies with time and frequency, e.g., in the form of (stochastic) models for these variations.

The effects captured by the gain matrix $\mathbf{G}$ can be subdivided in instrumental effects and propagation effects. We start by describing a few basic effects as understanding those will help to establish a suitable representation of the gain matrix.

5.1.1 Instrumental Effects

The instrumental effects consist of the directional response of the receiving elements (antennas) and the direction-independent electronic gains and phases of the receivers.

The directional response or *primary beam* of the receiving elements in the array can be described by a function $b_j(\mathbf{l})$, where we have assumed that this function is constant over the time and frequency span of the STI. It is generally assumed that the primary beam is equal for all elements in the array. With Q point sources, we will collect the resulting samples of the primary beam into a vector $\mathbf{b} = [b(\mathbf{l}_1), \cdots, b(\mathbf{l}_Q)]^T$. These coefficients are seen as gains that (squared) will multiply the source powers σ_q^2. The general shape of the primary beam $b(\mathbf{l})$ is known from electromagnetic modeling during the design of the telescope. If this is not sufficiently accurate, then it has to be calibrated, which is typically done off-line in the lab.

Next, each receiver element in the array is connected to a receiver chain (low-noise amplifier, bandpass filter, down-modulator), and initially the direction-independent electronic gains and phases of each receiver chain are unknown and have to be estimated. They are generally different from element to element. We thus have an unknown vector $\mathbf{g}$ (size $J \times 1$) with complex entries that each multiply the output signal of each telescope. As the direction independent gains are identical for all Q sources while the direction dependent response is identical for all elements, the gain matrix can be factored as $\mathbf{G} = \mathbf{g}\mathbf{b}^H$. By introducing the diagonal matrices $\boldsymbol{\Gamma} = \text{diag}(\mathbf{g})$ and $\mathbf{B} = \text{diag}(\mathbf{b})$, we can write $\mathbf{G} \odot \mathbf{A} = \boldsymbol{\Gamma}\mathbf{A}\mathbf{B}$.

Also the noise powers of each element are unknown and generally unequal to each other. We will still assume that the noise is independent from element to element. We can thus model the noise covariance matrix by an (unknown) diagonal $\boldsymbol{\Sigma}_n$.

For instrumental calibration, we can thus reformulate our data model in (14) to

$$\mathbf{R} = (\boldsymbol{\Gamma}\mathbf{A}\mathbf{B})\boldsymbol{\Sigma}_s(\mathbf{B}^H\mathbf{A}^H\boldsymbol{\Gamma}^H) \; + \; \boldsymbol{\Sigma}_n \tag{55}$$

Usually, $\boldsymbol{\Gamma}$ and $\mathbf{B}$ are considered to vary only slowly with time m and frequency k.

5.1.2 Propagation Effects

Ionospheric and tropospheric turbulence cause time-varying refraction and diffraction, which has a profound effect on the propagation of radio waves. In the simplest case, the ionosphere is modeled as a thin layer at some height (say 100 km) above the Earth, causing delays that can be represented as phase shifts. At the low frequencies used for LOFAR, this effect is more pronounced. Generally it is first assumed that the ionosphere is "constant" over about 10 km and about 10 s. A better model is to model the ionospheric delay as a "wedge", a linear function of the distance between piercing points (the intersection of the direction vectors $\mathbf{l}_q$ with the ionospheric phase screen). As illustrated in Fig. 10, this modifies the geometric delays, leading to a shift in the apparent position of the sources. For larger distances, higher-order functions are needed to model the spatial behaviour of the ionosphere, and if left uncorrected, the resulting image distortions are comparable to the distortions one sees when looking at lights at the bottom of a swimming pool.

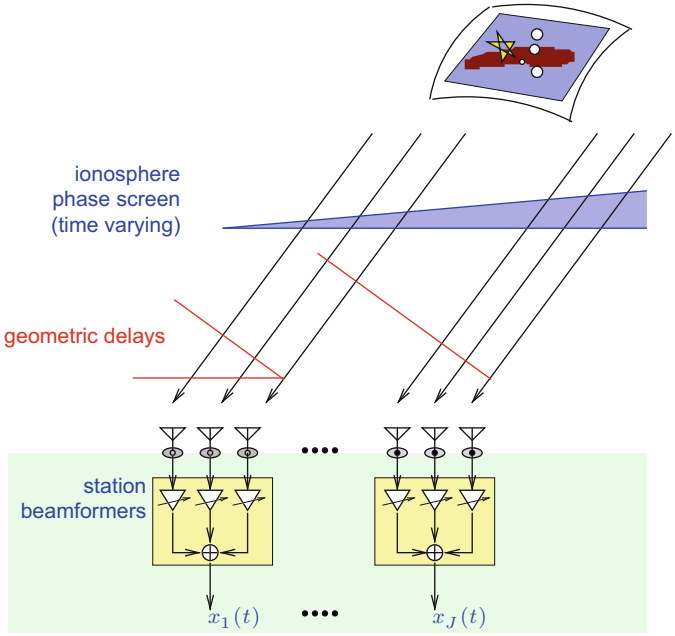


Fig. 10 A radio interferometer where stations consisting of phased array elements replace telescope dishes. The ionosphere adds phase delays to the signal paths. If the ionospheric electron density has the form of a *wedge*, it will simply shift the apparent positions of all sources

Previously, we described the array response matrix $\mathbf{A}$ as a function of the source direction vectors $\mathbf{l}_q$, and we wrote $\mathbf{A}(\boldsymbol{\theta})$ where the vector $\boldsymbol{\theta}$ was a suitable parameterization of the $\mathbf{l}_q$ (typically two direction cosines per source). If a linear model for the ionospheric disturbance is sufficient, then it is sufficient to replace $\mathbf{A}(\boldsymbol{\theta})$ by $\mathbf{A}(\boldsymbol{\theta}')$, where $\boldsymbol{\theta}'$ differs from $\boldsymbol{\theta}$ due to the shift in apparent direction of each source.

The modified data model that captures the above effects is thus

$$\mathbf{R} = (\boldsymbol{\Gamma}\mathbf{A}(\boldsymbol{\theta}')\mathbf{B})\boldsymbol{\Sigma}_s(\mathbf{B}^H\mathbf{A}(\boldsymbol{\theta}')^H\boldsymbol{\Gamma}^H) \;+\; \boldsymbol{\Sigma}_n\,. \tag{56}$$

In the next subsection, we will first describe how models of the form (55) or (56) can be identified. This step will serve as a stepping stone in the identification of a more general $\mathbf{G}$.

5.2 Calibration Algorithms

5.2.1 Estimating the Element Gains and Directional Responses

Let us assume a model of the form (55), where there are Q dominant calibration sources within the field of view. For these sources, we assume that their positions and source powers are known with sufficient accuracy from tables, i.e., we assume that $\mathbf{A}$ and $\mathbf{\Sigma}_s$ are known. We can then write (55) as

$$\mathbf{R} = \boldsymbol{\Gamma}\mathbf{A}\mathbf{\Sigma}\mathbf{A}^H\boldsymbol{\Gamma}^H + \mathbf{\Sigma}_n \tag{57}$$

where $\mathbf{\Sigma} = \mathbf{B}\mathbf{\Sigma}_s\mathbf{B}$ is a diagonal matrix with apparent source powers. With $\mathbf{B}$ unknown, $\mathbf{\Sigma}$ is unknown, but estimating $\mathbf{\Sigma}$ is precisely the problem we studied in Sect. 4 when we discussed imaging. Thus, once we have estimated $\mathbf{\Sigma}$ and know $\mathbf{\Sigma}_s$, we can easily estimate the directional gains $\mathbf{B}$. The problem thus reduces to estimate the diagonal matrices $\boldsymbol{\Gamma}$, $\mathbf{\Sigma}$ and $\mathbf{\Sigma}_n$ from a model of the form (57).

For some cases, e.g., arrays where the elements are traditional telescope dishes, the field of view is quite narrow (degrees) and we may assume that there is only a single calibrator source in the observation. Then $\mathbf{\Sigma} = \sigma^2$ is a scalar and the problem reduces to

$$\mathbf{R} = \mathbf{g}\sigma^2\mathbf{g}^H + \mathbf{\Sigma}_n$$

and since $\mathbf{g}$ is unknown, we could even absorb the unknown σ in $\mathbf{g}$ (it is not separately identifiable). The structure of $\mathbf{R}$ is a rank-1 matrix $\mathbf{g}\sigma^2\mathbf{g}^H$ plus a diagonal $\mathbf{\Sigma}_n$. This is recognized as a "rank-1 factor analysis" model in multivariate analysis theory [32, 40]. Given $\mathbf{R}$, we can solve for $\mathbf{g}$ and $\mathbf{\Sigma}_n$ in several ways [6, 7, 64]. For example, any submatrix away from the diagonal is only dependent on $\mathbf{g}$ and is rank 1. This allows direct estimation of $\mathbf{g}$. This property is related to the gain and phase closure relations often used in the radio astronomy literature for calibration (in particular, these relations express that the determinant of any 2×2 submatrix away from the main diagonal will be zero, which is the same as saying that this submatrix is rank 1).

In general, there are more calibrator sources (Q) in the field of view, and we have to solve (57). A simple idea is to resort to an Alternating Least Squares approach. If $\boldsymbol{\Gamma}$ would be known, then we can correct $\mathbf{R}$ for it, so that we have precisely the same problem as we considered before, (33), and we can solve for $\mathbf{\Sigma}$ and $\mathbf{\Sigma}_n$ using the techniques discussed in Sect. 4.4.1. Alternatively, with $\mathbf{\Sigma}$ known, we can say we know a reference model $\mathbf{R}_0 = \mathbf{A}\mathbf{\Sigma}\mathbf{A}^H$, and the problem is to identify the element gains $\boldsymbol{\Gamma} = \mathrm{diag}(\mathbf{g})$ from a model of the form

$$\mathbf{R} = \boldsymbol{\Gamma}\mathbf{R}_0\boldsymbol{\Gamma}^H + \mathbf{\Sigma}_n$$

or, after applying the vec(·)-operation,

$$\text{vec}(\mathbf{R}) = \text{diag}(\text{vec}(\mathbf{R}_0))(\overline{\mathbf{g}} \otimes \mathbf{g}) + \text{vec}(\mathbf{\Sigma}_n)\,.$$

This leads to the Least Squares problem

$$\hat{\mathbf{g}} = \arg\min_{\mathbf{g}} \|\text{vec}(\hat{\mathbf{R}} - \mathbf{\Sigma}_n) - \text{diag}(\text{vec}(\mathbf{R}_0))(\overline{\mathbf{g}} \otimes \mathbf{g})\|^2\,.$$

This problem cannot be solved in closed form. Alternatively, we can first solve an unstructured problem: define $\mathbf{x} = \overline{\mathbf{g}} \otimes \mathbf{g}$ and solve

$$\hat{\mathbf{x}} = \text{diag}(\text{vec}(\mathbf{R}_0))^{-1}\text{vec}(\hat{\mathbf{R}} - \mathbf{\Sigma}_n)$$

or equivalently, if we define $\mathbf{X} = \mathbf{g}\mathbf{g}^H$,

$$\hat{\mathbf{X}} = (\hat{\mathbf{R}} - \mathbf{\Sigma}_n) \oslash \mathbf{R}_0.$$

where $\oslash$ denotes an element-wise matrix division. After estimating the unstructured $\mathbf{X}$, we enforce the rank-1 structure $\mathbf{X} = \mathbf{g}\mathbf{g}^H$, via a rank-1 approximation, and find an estimate for $\mathbf{g}$. The element-wise division can lead to noise enhancement; this is remediated by only using the result as an initial estimate for a Gauss-Newton iteration [22] or by formulating a *weighted* least squares problem instead [61, 64].

With $\mathbf{g}$ known, we can again estimate $\mathbf{\Sigma}$ and $\mathbf{\Sigma}_n$, and make an iteration. Overall we then obtain an alternating least squares solution. A more optimal solution can be found by solving the overall problem (57) as a covariance matching problem with a suitable parameterization, and the more general gradient descent algorithms (e.g., Gauss-Newton and Levenberg-Marquardt) presented in [47] lead to an asymptotically unbiased and statistically efficient solution.

For large arrays, Gauss-Newton iterations or weighted least squares approaches become computationally expensive as they scale cubicly with the number of receiving elements in the array. Several people have therefore proposed an iterative alternating direction implicit (ADI) method [25, 42, 50], which was demonstrated to have robust convergence and to be statistically efficient for typical scenarios encountered in radio astronomy in which the noise powers dominate over the source powers and are very similar for all elements in the array [50].

The resulting calibration algorithms are one step in the classical self-calibration (SelfCal) algorithm [15, 48] widely used in the radio astronomy literature, in particular for a single calibrator source. In the calibration step of SelfCal, $\mathbf{R}_0$ is a reference model, obtained from the best known map at that point in the iteration. Next, in the imaging step of SelfCal, the calibration results are used to correct the data $\hat{\mathbf{R}}$ and the next best image is constructed. This leads to a new reference model $\mathbf{R}_0$, etc.

5.2.2 Estimating the Ionospheric Perturbation

The more general calibration problem (56) follows from (55) by writing $\mathbf{A} = \mathbf{A}(\boldsymbol{\theta}')$ where $\boldsymbol{\theta}'$ are the apparent source locations. This problem can be easily solved in quite the same way: in the alternating least squares problem we solve for $\mathbf{g}$, $\boldsymbol{\theta}'$, $\boldsymbol{\sigma}_s$ and $\boldsymbol{\sigma}_n$ in turn, keeping the other parameters fixed at their previous estimates. After that, we can relate the apparent source locations to the (known) locations of the calibrator sources $\boldsymbol{\theta}$.

The resulting phase corrections $\mathbf{A}'$ to relate $\mathbf{A}(\boldsymbol{\theta}')$ to $\mathbf{A}(\boldsymbol{\theta})$ via $\mathbf{A}(\boldsymbol{\theta}') = \mathbf{A}(\boldsymbol{\theta}) \odot \mathbf{A}'$ give us an estimate of the ionospheric phase screen in the direction of each source. These "samples" can then be interpolated to obtain a phase screen model for the entire field of view. This method is limited to the regime where the phase screen can be modeled as a linear gradient over the array. An implementation of this algorithm is called Field-Based Calibration [16].

Other techniques are based on "peeling" [42]. In this method of successive estimation and subtraction, calibration parameters are obtained for the brightest source in the field. The source is then removed from the data, and the process is repeated for the next brightest source. This leads to a collection of samples of the ionosphere, to which a model phase screen can be fitted.

5.2.3 Estimating the General Model

In the more general case (14), viz.

$$\mathbf{R} = (\mathbf{G} \odot \mathbf{A})\boldsymbol{\Sigma}_s(\mathbf{G} \odot \mathbf{A})^H + \boldsymbol{\Sigma}_n \,,$$

we have an unknown full matrix $\mathbf{G}$. We assume $\mathbf{A}$ and $\boldsymbol{\Sigma}_s$ known. Since $\mathbf{A}$ element-wise multiplies $\mathbf{G}$ and $\mathbf{G}$ is unknown, we might as well omit $\mathbf{A}$ from the equations without loss of generality. For the same reason also $\boldsymbol{\Sigma}_s$ can be omitted. This leads to a problem of the form

$$\mathbf{R} = \mathbf{G}\mathbf{G}^H + \boldsymbol{\Sigma}_n \,,$$

where the $J \times Q$ matrix $\mathbf{G}$ and $\boldsymbol{\Sigma}_n$ (diagonal) are unknown. This problem is known as a rank-Q factor analysis problem. Note that if the noise would be spatially white ($\boldsymbol{\Sigma}_n = \sigma_n^2\mathbf{I}$), then $\mathbf{G}$ can be solved from an eigenvalue decomposition of $\mathbf{R}$, up to a unitary factor at the right.

The more general Factor Analysis problem is a classical problem in multivariate statistics that has been studied since the 1930s [32, 40]. Currently, FA is an important and popular tool for latent variable analysis with many applications in various fields of science [2]. However, its application within the signal processing community has been surprisingly limited. The problem can be regarded as a special case of covariance matching, studied in detail in [47]. Thus, the problem can be

solved using Gauss-Newton iterations. The current algorithms are robust and have a computational complexity similar to that of an eigenvalue decomposition of $\mathbf{R}$ [44].

It is important to note that $\mathbf{G}$ can be identified only up to a unitary factor $\mathbf{V}$ at the right: $\mathbf{G}' = \mathbf{GV}$ would also be a solution. This factor makes the gains unidentifiable unless we introduce more structure to the problem. To make matters worse, note that this problem is used to fine-tune earlier coarser models (56). At this level of accuracy, the number of dominant sources Q is often not small anymore, and at some point $\mathbf{G}$ is not identifiable: counting number of equations and unknowns, we find that the maximum factor rank is limited by $Q < J - \sqrt{J}$.

As discussed in [46] and studied in more detail in [55], more structure needs to be introduced to be able to solve the problem. Typically, what helps is to consider the problem for a complete observation (rather than for a single snapshot $\mathbf{R}$) where we have many different frequencies f_k and time intervals m. The directional response matrix $\mathbf{A}_{m,k}$ varies with m and k in a known way, and the instrumental gains $\mathbf{g}$ and $\mathbf{b}$ are relatively constant. The remaining part of $\mathbf{G} = \mathbf{g}\mathbf{b}^H \odot \mathbf{A}'$ is due to the ionospheric perturbations, and models can be introduced to describe its fluctuation over time, frequency, and space using some low order polynomials. We can also introduce stochastic knowledge that describe a correlation of parameters over time and space.

For LOFAR, a complete calibration method that incorporates many of the above techniques was recently proposed in [28]. In general, calibration and imaging need to be considered in unison, leading to many potential directions, approaches, and solutions. Once calibration reaches the stage of full image calibration at the full resolution, we basically try to identify a highly detailed parametric model using gradient descent techniques. The computational complexity can be very high. To limit this, SAGEcal [31] clusters parameters into non-overlapping sets associated with different directions on the sky, solves the "independent" problems separately, and then combines in a parameter-fusing step. Distributed SAGEcal [66] also exploits parallelism such as continuity over time and frequency, again solving "independent" problems separately in parallel, followed by a fusion step.

6 A Typical Signal Processing Pipeline

To conclude this chapter, we discuss how calibration and imaging techniques are put together to form an imaging pipeline. We do this using a pipeline developed to guide the design of the SKA computing systems [29, 30] as an example. If the receiving elements of such a system are phased array stations, as is the case for the low-frequency system of the SKA, an end-to-end imaging pipeline consists of three stages of processing: Station Beamforming, processing in the Central Signal Processor (CSP), and the Science Data Processor (SDP). Block diagrams for each stage are shown in Figs. 11, 12 and 13.

Figure 11 shows a typical block diagram for signal processing within a phased array station. The signals from the receiving elements within a station are digitized

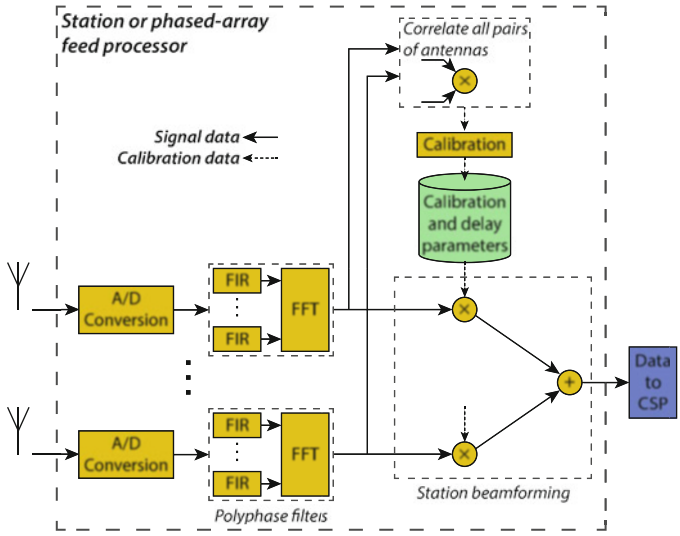


Fig. 11 Typical block diagram for signal processing within a phased array station [29, 30]

and combined into a single beamformed output, providing a well-defined beam on the sky. This is usually done by a standard delay beamformer by applying weights as described in (20). As the delays are represented by phase shifts, the signals need to be narrowband with respect to this delay. This is ensured by splitting the digitized signal of each receiver path into multiple coarse frequency channels (typically order (a few) 100 kHz wide) by a polyphase filter bank. The time series produced for each of these coarse channels can also be fed into a correlator to produce array covariance matrices for the station. These covariance matrices can be used to perform calibration. Usually, this only concerns direction independent gain calibration as described in Sect. 5.2.1. Those calibration solutions can be used to adapt the beamformer weights to correct for complex valued gain differences between receive paths. The beamformed output of each phased array station is sent to the CSP for further processing.

Figure 12 shows the block diagram for the signal processing within the CSP of the SKA. The goal of the CSP is to combine data from the receiving elements of the SKA interferometer by correlating its input signals. As the signals can be integrated after correlation, this step can significantly reduce the data volume using relatively simple operations. The input signals are either beamformed signals from phased array stations or coarsely channelized signals from reflector dishes. As the longest baselines of the SKA interferometer are much longer than the size of an individual station, much narrower frequency channels are required to satisfy the narrowband assumption discussed in Sect. 3.2. This is achieved by a second polyphase filter bank, which splits the coarse frequency channels further into fine channels (typically order 1 kHz wide). Any residual time delay across the array remaining after the coarse delay correction done by shifting time series with respect

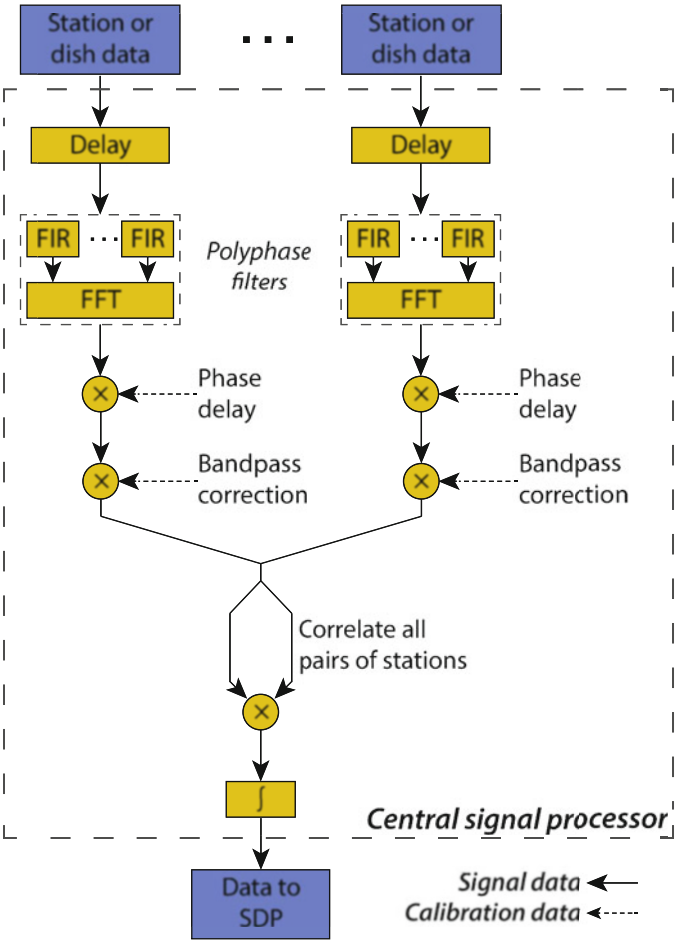


Fig. 12 Block diagram for data processing in the Central Signal Processor (CSP) of the SKA [29, 30]

to each other before the polyphase filter bank, is then corrected by applying an appropriate phase rotation. As the power received in individual frequency channels may vary significantly across frequency due to the intrinsic spectrum of most astronomical sources and the gain characteristics of the instrument, a bandpass correction is applied to equalize the power across frequency before the signals are correlated. After correlation, the data is integrated into STIs and data corrupted by radio frequency interference (RFI) is flagged before the data is transferred to the SDP.

A block diagram for an imaging pipeline within the SDP is shown in Fig. 13. After some pre-processing, consisting of demixing, integration and initial calibration, a self-calibration and imaging cycle is started.

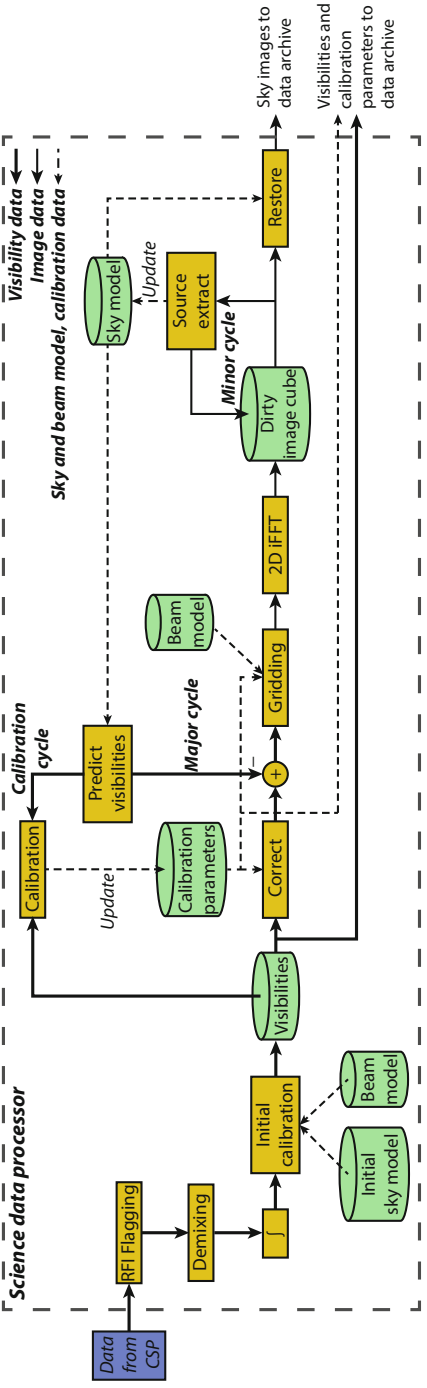


Fig. 13 Block diagram for the imaging pipeline in the Science Data Processor of the SKA [29, 30]

We first discuss the pre-processing steps. A few exceptionally bright astronomical radio sources, like Cas A and Cyg A, are so bright that their signature can be detected in the data even in observations on fields that are at a considerable distance from these sources. This is mitigated by applying phase rotation (effectively applying beamforming weights to the visibilities without adding them together) towards these sources, estimating and subtracting their response, and undoing the phase rotation again. This process is called *demixing*. After demixing, further integration is possible, which reduces the computational burden in further stages of the pipeline. Initial calibration usually consists of direction independent calibration of the complex valued gains of the individual receive paths in the interferometer array. The algorithms used here are very similar to those exploited in the station calibration mentioned before.

After initial calibration, the self-calibration and imaging cycle is entered, which is the main part of the SDP imaging pipeline. It starts by computing the residual visibilities obtained after subtracting the best available model for the visibilities based on the current best knowledge of calibration parameters and sky model from the measured visibilities. A dirty image is made from the residual visibilities. The required operations (17) are essentially a Fourier transform, but on non-uniformly sampled data. To be able to use the fast Fourier transform (required because this step is the most expensive in the entire processing pipeline), the residual visibilities are *gridded* onto a uniform grid, after which the inverse FFT is applied. Other computationally efficient implementations for non-uniform fast Fourier transforms may be considered. As this processing step is similar in many other image formation instruments (e.g., geophysics [19] and MRI), the available literature is rich.

Iterative algorithms such as CLEAN are used to find and subtract new sources in the residual image. This is referred to as the *minor cycle*. The new source components are added to the sky model, which is then used in the next iteration of the self-calibration and imaging cycle, the *major cycle*. Once this process has converged sufficiently, the sky model (deconvolved image) is added to the residual image, which should ideally only contain noise at this stage. That result is then presented as the final image. Since the major cycle is very expensive, the usual approach is to detect thousands of sources in each minor cycle, and to run the major cycle less than 10 times.

7 Concluding Remarks and Further Reading

In this chapter, we presented a signal processing viewpoint on radio astronomy. We showed how, with the right translations, the "measurement equations" are connected to covariance matrix data models used in the phased array signal processing literature. In this presentation, the resulting data models are very compact and clean, in the sense that the most straightforward covariance data models, widely studied in the signal processing literature as theoretical models, already seem valid. This is because far field assumptions clearly hold, and the propagation channels are very

simple (no multipath), in contrast to other array processing applications such as seismology, synthetic aperture radar, or biomedical tomography.

However, this does not mean that radio astronomy is a "simple" application: data volumes are massive, and the requirements on resolution and accuracy are mind-boggling. Current telescopes, developed in the 1970s, start with signals sampled at 1–2 bits accuracy (because anyway the signals are mostly noise), and after data reduction and map making routinely end up with images with a dynamic range of 10^5.

So far, radio astronomy has done very well without explicit connection to the array signal processing literature. However, we expect that, by making this connection, a wealth of new insights and access to "new" algorithms can be obtained. This will be beneficial, and possibly essential, for the development of new instruments like LOFAR and SKA.

For further reading we suggest, first of all, the classical radio astronomy textbooks, e.g., by Thompson et al. [52] and by Perley et al. [49]. The August 2009 issue of the *Proceedings of the IEEE* was devoted to the presentation of new instruments. The January 2010 issue of *IEEE Signal Processing Magazine* gave a signal processing perspective. For general insights into imaging and deconvolution, we suggest Blahut [4].

Challenges for signal processing lie in (1) imaging, (2) calibration, (3) interference suppression. These problems are really intertwined. It is interesting to note that, especially for calibration and interference suppression, factor analysis is an essential tool. Our contributions in these areas have appeared in [3, 6, 33, 34, 36, 55, 57, 62–64] and are summarized in the PhD theses [5, 44, 54, 60], which should provide ample details for further reading.

References

1. Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., der Vorst, H.V.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition. SIAM, Philadelphia, PA (1994)
2. Bartholomew, D.J., Knott, M., Moustaki, I.: Latent Variable Models and Factor Analysis: A Unified Approach. John Wiley and Sons (2011)
3. Ben-David, C., Leshem, A.: Parametric high resolution techniques for radio astronomical imaging. IEEE Journal of Selected Topics in Signal Processing **2**(5), 670–684 (2008)
4. Blahut, R.E.: Theory of remote image formation. Cambridge University Press (2004). ISBN 0521553733
5. Boonstra, A.J.: Radio frequency interference mitigation in radio astronomy. Ph.D. thesis, TU Delft, Dept. EEMCS (2005). ISBN 90-805434-3-8
6. Boonstra, A.J., van der Veen, A.J.: Gain calibration methods for radio telescope arrays. IEEE Transactions on Signal Processing **51**(1), 25–38 (2003)
7. Boonstra, A.J., Wijnholds, S.J., van der Tol, S., Jeffs, B.: Calibration, sensitivity and RFI mitigation requirements for LOFAR. In: IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). Philadelphia (Penn.), USA (2005)
8. Borgiotti, G.B., Kaplan, L.J.: Superresolution of uncorrelated interference sources by using adaptive array techniques. IEEE Transactions on Antennas and Propagation **27**, 842–845 (1979)

9. Bridle, A.H., Schwab, F.R.: Bandwidth and Time-Average Smearing. In: G.B. Taylor, C.L. Carilli, R.A. Perley (eds.) Synthesis Imaging in Radio Astronomy II, *Astronomical Society of the Pacific Conference Series*, vol. 180, chap. 18, pp. 371–382. Astronomical Society of the Pacific (1999)
10. Briggs, D.S.: High fidelity deconvolution of moderately resolved sources. Ph.D. thesis, New Mexico Inst. of Mining and Technology, Socorro (NM) (1995)
11. Carrillo, R.E., McEwen, J.D., Wiaux, Y.: Sparsity averaging reweighted analysis (SARA): a novel algorithm for radio-interferometric imaging. Monthly Notices of the Royal Astronomical Society **426**(2), 1223–1234 (2012)
12. Carrillo, R.E., McEwen, J.D., Wiaux, Y.: PURIFY: a new approach to radio-interferometric imaging. Monthly Notices of the Royal Astronomical Society **439**(4), 3591–3604 (2014)
13. Cornwell, T., Braun, R., Brigss, D.S.: Deconvolution. In: G.B. Taylor, C.L. Carilli, R.A. Perley (eds.) Synthesis Imaging in Radio Astronomy II, *Astronomical Society of the Pacific Conference Series*, vol. 180, pp. 151–170. Astronomical Society of the Pacific (1999)
14. Cornwell, T.J.: Multiscale CLEAN deconvolution of radio synthesis images. IEEE Journal of Selected Topics in Signal Processing **2**(5), 793–801 (2008)
15. Cornwell, T.J., Wilkinson, P.N.: A new method for making maps with unstable radio interferometers. Monthly Notices of the Royal Astronomical Society **196**, 1067–1086 (1981)
16. Cotton, W.D., et al.: Beyond the isoplanatic patch in the VLA Low-frequency Sky Survey. In: Proceedings of the SPIE, vol. 5489, pp. 180–189. Glasgow (2004)
17. Dewdney, P.E., Braun, R.: SKA1-low configuration coordinates - complete set. Tech. Rep. SKA-TEL-SKO-0000422, SKA Office, Manchester (UK) (2016)
18. Dewdney, P.E., Hall, P.J., Schilizzi, R.T., Lazio, T.J.L.W.: The square kilometre array. Proceedings of the IEEE **97**(8), 1482–1496 (2009)
19. Duijndam, A.J.W., Schonewille, M.A.: Nonuniform fast Fourier transform. Geophysics **64**(2), 539–551 (1999)
20. Foucart, S., Koslicki, D.: Sparse recovery by means of nonnegative least squares. IEEE Signal Processing Letters **21**(4), 498–502 (2014)
21. Frieden, B.: Restoring with maximum likelihood and maximum entropy. Journal of the Optical Society of America **62**, 511–518 (1972)
22. Fuhrmann, D.R.: Estimation of sensor gain and phase. IEEE Transactions on Signal Processing **42**(1), 77–87 (1994)
23. Garsden, H., et al.: LOFAR sparse image reconstruction. Astronomy & Astrophysics **575**(A90), 1–18 (2015)
24. van Haarlem, M.P., et al.: LOFAR: The low frequency array. Astronomy & Astrophysics **556**(A2), 1–53 (2013)
25. Hamaker, J.P.: Understanding radio polarimetry - iv. the full-coherency analogue of scalar self-calibration: Self-alignment, dynamic range and polarimetric fidelity. Astronomy & Astrophysics Supplement **143**(3), 515–534 (2000)
26. Hayes, M.H.: Statistical Digital Signal Processing and Modeling. John Wiley and Sons (1996)
27. Hogbom, J.A.: Aperture synthesis with non-regular distribution of interferometer baselines. Astronomy and Astrophysics Suppl. **15**, 417–426 (1974)
28. Intema, H.T., et al.: Ionospheric calibration of low frequency radio interferometric observations using the peeling scheme. I. Method description and first results. Astronomy & Astrophysics **501**(3), 1185–1205 (2009)
29. Jongerius, R.: Exascale computer system design: The square kilometre array. Ph.D. thesis, Eindhoven University of Technology (2016). ISBN 978-90-386-4136-2
30. Jongerius, R., Wijnholds, S., Nijboer, R., Corporaal, H.: An end-to-end computing model for the square kilometre array. IEEE Computer **47**(9), 48–54 (2014)
31. Kazemi, S., Yatawatta, S., Zaroubi, S., Lampropoulos, P., de Bruyn, A.G., Koopmans, L.V.E., Noordam, J.: Radio interferometric calibration using the sage algorithm. Monthly Notices of the Royal Astronomical Society **414**(2), 1656 (2011)
32. Lawley, D.N., Maxwell, A.E.: Factor Analysis as a Statistical Method. Butterworth & Co, London (1971)

33. Leshem, A., van der Veen, A.J.: Radio-astronomical imaging in the presence of strong radio interference. IEEE Transactions on Information Theory **46**(5), 1730–1747 (2000)
34. Leshem, A., van der Veen A. J., Boonstra, A.J.: Multichannel interference mitigation technique in radio astronomy. Astrophysical Journal Supplements **131**(1), 355–374 (2000)
35. Levanda, R., Leshem, A.: Radio astronomical image formation using sparse reconstruction techniques. In: IEEE 25th convention of Elec. Electron. Eng. in Israel (IEEEI 2008), pp. 716–720 (2008)
36. Levanda, R., Leshem, A.: Synthetic aperture radio telescopes. IEEE Signal Processing Magazine **27**(1), 14–29 (2010)
37. Li, F., Cornwell, T.J., de Hoog, F.: The application of compressive sampling to radio astronomy; I deconvolution. Astronomy and Astrophysics **528**(A31), 1–10 (2011)
38. Lonsdale, C., et al.: The Murchison Widefield Array: Design overview. Proceedings of the IEEE **97**(8), 1497–1506 (2009)
39. Mallat, S.G., Zhang, Z.: Matching pursuits with time-frequency dictionaries. IEEE Transactions on Signal Processing **41**(12), 3397–3415 (1993)
40. Mardia, K.V., Kent, J.T., Bibby, J.M.: Multivariate Analysis. Academic Press, New York (1979)
41. Marsh, K.A., Richardson, J.M.: The objective function implicit in the CLEAN algorithm. Astronomy and Astrophysics **182**(1), 174–178 (1987)
42. Mitchell, D.A., et al.: Real-time calibration of the Murchison Widefield Array. IEEE Journal of Selected Topics in Signal Processing **2**(5), 707–717 (2008)
43. Moon, T.K., Stirling, W.C.: Mathematical Methods and Algorithms for Signal Processing. Prentice Hall (2000). ISBN 0201361868
44. Mouri Sardarabadi, A.: Covariance matching techniques for radio astronomy calibration and imaging. Ph.D. thesis, TU Delft, Dept. EEMCS (2016)
45. Mouri Sardarabadi, A., Leshem, A., van der Veen, A.J.: Radio astronomical image formation using constrained least squares and Krylov subspaces. Astronomy & Astrophysics **588**, A95 (2016)
46. Noordam, J.E.: Generalized self-calibration for LOFAR. In: XXVIIth General Assembly of the International Union of Radio Science (URSI). Maastricht (The Netherlands) (2002)
47. Ottersten, B., Stoica, P., Roy, R.: Covariance matching estimation techniques for array signal processing applications. Digital Signal Processing, A Review Journal **8**, 185–210 (1998)
48. Pearson, T.J., Readhead, A.C.S.: Image formation by self-calibration in radio astronomy. Annual Review of Astronomy and Astrophysics **22**, 97–130 (1984)
49. Perley, R.A., Schwab, F.R., Bridle, A.H.: Synthesis Imaging in Radio Astronomy, *Astronomical Society of the Pacific Conference Series*, vol. 6. BookCrafters Inc. (1994)
50. Salvini, S., Wijnholds, S.J.: Fast gain calibration in radio astronomy using alternating direction implicit methods: Analysis and applications. Astronomy & Astrophysics **571**(A97), 1–14 (2014)
51. Schwardt, L.C.: Compressed sensing imaging with the KAT-7 array. In: International Conference on Electromagnetics in Advanced Applications (ICEAA), pp. 690–693 (2012)
52. Thompson, A.R., Moran, J.M., Swenson, G.W.: Interferometry and Synthesis in Radio Astronomy, 2nd edn. Wiley, New York (2001)
53. Tingay, S.J., et al.: The Murchison widefield array: The square kilometre array precursor at low radio frequencies. Publications of the Astronomical Society of Australia **30**(7) (2013)
54. van der Tol, S.: Bayesian estimation for ionospheric calibration in radio astronomy. Ph.D. thesis, TU Delft, Dept. EEMCS (2009)
55. van der Tol, S., Jeffs, B.D., van der Veen, A.J.: Self-calibration for the LOFAR radio astronomical array. IEEE Transactions on Signal Processing **55**(9), 4497–4510 (2007)
56. Turner, W.: SKA phase 1 system requirements specification. Tech. Rep. SKA-TEL-SKO-0000008, SKA Office, Manchester (UK) (2016)
57. van der Veen, A.J., Leshem, A., Boonstra, A.J.: Array signal processing for radio astronomy. Experimental Astronomy **17**(1–3), 231–249 (2004)
58. de Vos, M., Gunst, A., Nijboer, R.: The LOFAR telescope: System architecture and signal processing. Proceedings of the IEEE **97**(8), 1431–1437 (2009)

59. Wiaux, Y., Jacques, L., Puy, G., Scaife, A.M.M., Vandergheynst, P.: Compressed sensing imaging techniques for radio interferometry. Monthly Notices of the Royal Astronomical Society **395**, 1733–1742 (2009)
60. Wijnholds, S.J.: Fish-eye observing with phased array radio telescopes. Ph.D. thesis, TU Delft, Dept. EEMCS (2010). ISBN 978-90-9025180-6
61. Wijnholds, S.J., Boosntra, A.J.: A multisource calibration method for phased array telescopes. In: Fourth IEEE Workshop on Sensor Array and Multi-channel Processing (SAM). Waltham (Mass.), USA (2006)
62. Wijnholds, S.J., van der Tol, S., Nijboer, R., van der Veen, A.J.: Calibration challenges for the next generation of radio telescopes. IEEE Signal Processing Magazine **27**(1), 32–42 (2010)
63. Wijnholds, S.J., van der Veen, A.J.: Fundamental imaging limits of radio telescope arrays. IEEE Journal of Selected Topics in Signal Processing **2**(5), 613–623 (2008)
64. Wijnholds, S.J., van der Veen, A.J.: Multisource self-calibration for sensor arrays. IEEE Transactions on Signal Processing **57**(9), 3512–3522 (2009)
65. Wise, M.W., Rafferty, D.A., McKean, J.P.: Feedback at the working surface: A joint X-ray and low-frequency radio spectral study of the Cocoon Shock in Cygnus A. In: 13th Meeting of the American Astronomical Society's High Energy Astrophysics Division (HEAD), pp. 88–89 (2013)
66. Yatawatta, S.: Distributed radio interferometric calibration. Monthly Notices of the Royal Astronomical Society **449**(4), 4506 (2015)
67. Zatman, M.: How narrow is narrowband. IEE Proc. Radar, Sonar and Navig. **145**(2), 85–91 (1998)

Distributed Smart Cameras and Distributed Computer Vision

Marilyn Wolf and Jason Schlessman

Abstract Distributed smart cameras are multiple-camera systems that perform computer vision tasks using distributed algorithms. Distributed algorithms scale better to large networks of cameras than do centralized algorithms. However, new approaches are required to many computer vision tasks in order to create efficient distributed algorithms. This chapter motivates the need for distributed computer vision, surveys background material in traditional computer vision, and describes several distributed computer vision algorithms for calibration, tracking, and gesture recognition.

1 Introduction

Distributed smart cameras have emerged as an important category of distributed sensor and signal processing systems. Distributed sensors in other media have been important for quite some time, but recent advances have made the deployment of large camera systems feasible. The unique properties of imaging add new classes of problems that are not apparent in unidimensional and low-rate sensors. **Physically distributed cameras** have been used in computer vision for quite some time to handle two problems: **occlusion** and **pixels-on-target**. Cameras at different locations expose and occlude different parts of the scene. Their imagery can be combined to create a more complete model of the scene. Pixels-on-target refers to the resolution with which a part of the scene is captured, which in most applications is primarily limited by sensor resolution and not by optics. Wide-angle lenses cover a large area but at low resolution for any part of the scene. Imagery from multiple cameras can be combined to provide both extended coverage

M. Wolf (✉)
School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, USA
e-mail: wolf@ece.gatech.edu

J. Schlessman
Department of Electrical Engineering, Princeton University, Princeton, NJ, USA

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_10

and adequate resolution. Distributed smart cameras combine physically distributed cameras and distributed algorithms. Early approaches to distributed-camera-based computer vision used server-based, centralized algorithms. While such algorithms are often easier to conceive and implement, they do not scale well. Properly-designed distributed algorithms scale to handle much larger camera networks. VLSI technology has aided both the image-gathering and computational abilities of distributed smart camera systems. Moore's Law has progressed to the point where very powerful multiprocessors can be put on a single chip at very low cost [32]. The same technology has also provided cheap and powerful image sensors, particularly in the case of CMOS image sensors [33]. Distributed smart cameras have been used for a variety of applications, including tracking, gesture recognition, and target identification. Networks of several hundred cameras have been tested. Over time, we should expect to see much larger networks both tested and deployed. Surveillance is one application that comes to mind. While surveillance and security are a large application—analysts estimate that 25 million security cameras are installed in the United States—that industry moves at a relatively slow pace. Health care, traffic analysis, and entertainment are other important applications of distributed smart cameras. After starting in the mid-1990s, research on distributed smart cameras has progressed rapidly.

We start with a review of some techniques from computer vision that were not specifically developed for distributed systems but have been used as components in distributed systems. Section 3 reviews early research in distributed smart cameras. Section 4 considers the types of challenges created by distributed smart cameras. We next consider calibration of cameras in Sect. 5, followed by algorithms for tracking in Sect. 6 and gesture recognition in Sect. 7. Section 8 discusses computing platforms suitable for real-time distributed computer vision.

2 Approaches to Computer Vision

Several algorithms that used in traditional computer vision problems such as tracking also play important roles as components in distributed computer vision systems. In this section we briefly review some of those algorithms. Tracking refers to the target or object of interest as **foreground** and non-interesting objects as **background** (even though this usage is at variance with the terminology of theater). Many tracking algorithms assume that the background is relatively static and use a separate step, known as **background elimination** or **background subtraction**, to eliminate backgrounds. The simplest background subtraction algorithm simply compares each pixel in the current frame to the corresponding pixel in a reference frame that does not include any targets. If the current-frame pixel is equal the reference-frame pixel, then it is marked as background. This algorithm is computationally cheap but not very accurate. Even very small amounts of movement in the background can cause erroneous classification; the classic example of extraneous motion is the blowing of tree leaves in the wind. The mixture-of-Gaussians approach [14] provides much

more results. Mixture-of-Gaussian systems also use multiple models so more than one candidate model can be kept alive at a time. The algorithm proceeds in three steps: compare Gaussians of each model to find matches; update Gaussian mean and variance; update weights. Given a pixel value as a vector of luminance (Y) and chrominance (C_b, C_r) information, $X \in (Y, Cb, Cr)$ and $\alpha_x = \sqrt{\frac{|X-\mu_x|}{\sigma_x}}$, we can compare a current-frame and reference-frame pixels using a threshold T:

$$\left(\frac{\alpha_Y}{a_Y}\right)^2 + \left(\frac{\alpha_{Cb}}{a_{Cb}}\right)^2 + \left(\frac{\alpha_{Cr}}{a_{Cr}}\right)^2 < T \tag{1}$$

Matching weights are updated as follows, where n is the number of frames over which this Gaussian distribution has been active:

$$N = \begin{cases} n, & n < N_{max} \\ N_{max}, & n \geq N_{max} \end{cases} \tag{2}$$

$$\mu_{Xn} = \mu_{Xn-1} + \frac{(X_n - \mu_{Xn-1})}{N} \tag{3}$$

$$\sigma_{Xn} = \sigma_{Xn-1} + \frac{(X_n - \mu_{Xn-1})(X_n - \mu_{Xn}) - \sigma_{Xn-1}}{N} \tag{4}$$

A weight w_m evaluates the system's confidence in that model and pixel position:

$$w_m = \begin{cases} w_{m-1} + \frac{(\rho - w_{m-1})}{m}, & m < M_{max} \\ w_{m-1} + \frac{(\rho - w_{m-1})}{M_{max}}, & m \geq M_{max} \end{cases} \tag{5}$$

An appearance model is used to compare the appearance of targets in different frames; appearance models are useful both in a single-camera system to ensure tracking continuity or between cameras to compare views. Appearance models may make use of shape and/or color. Bounding box is a simple shape model, but more complex curve-based models may also be used. A color or luminance histogram is often used to represent detail within the image. One common method for comparing two histograms is the Bhattacharyya distance. More than one appearance model may be necessary since many interesting targets change in both shape and color as a function of the observer's position. The two major approaches to single-camera tracking are Kalman filtering [6] and particle filters. Particle filters [8] use Monte Carlo methods to estimate a probability distribution. Weighted particles represent samples of the hidden states as weighted by Bayesian estimates of probability masses. Coates [10] describes a distributed algorithms for computing particle filters. In their architecture, each sensor node ran one particle filter, with random number generators on each node that were synchronized. They described two approaches to the problem of distributing observations over the network, one parametric and one based on adaptive encoding. Sheng et al. [27] developed two distributed particle filter algorithms: one that exchanges information between cliques (nearby sensors

whose signals tend to be correlated) and another in which cliques compute partial estimates based on local information and forward their estimates to a fusion center.

3 Early Work in Distributed Smart Cameras

The DARPA-sponsored Video Surveillance and Monitoring (VSAM) program was one of the first efforts to develop distributed computer vision systems. A system developed at Carnegie Mellon University [11] developed a cooperative tracking system in which tracking targets were handed off from camera to camera. Each sensor processing unit (SPU) classified targets into categories such as *human* or *vehicle*. At the MIT Media Lab, Mallet and Bove [20] developed a distributed camera network that could hand-off tracking targets in real time. Their camera network consisted of small cameras mounted on tracks in the ceiling of a room. The cameras would move to improve their view of subjects based on information from other cameras as well as their own analysis. Lin et al. [19] developed a distributed system for gesture recognition that fuses data after some image processing using a peer-to-peer protocol. That system will be described in more detail in Sect. 7. The distributed tracker of Bramberger et al. [7] passed off a tracking task from camera to camera as the target moved through the scene. Each camera ran its own tracker. Handoffs were controlled by a peer-to-peer protocol.

4 Challenges

A distributed smart camera is a data fusion system—samples from cameras are captured, features are extracted and combined, and results are classified. There is more than one way to perform these steps, providing a rich design space. We can identify several axes on which the design space of distributed smart cameras can be analyzed:

- How abstract is the data being fused: pixel, small-scale feature, shape, etc.? What methods are used to fuse data?
- What groups/cliques of sensors combine their data? For example, groups may be formed by network connectivity, location, or signal characteristics. How does the group structure evolve as the scene changes?
- How sparse in time and space is the data?

We can also identify some axes based on the underlying technology:

- Number of cameras.
- Heterogeneity of cameras.
- Fixed and/or moving cameras.
- Fields-of-view (overlapping, non-overlapping).

- Server-assisted (directories, etc.) vs. peer-to-peer.
- Data fusion level: pixels (server-based) vs. mid-level vs. high-level

The simplest form of data fusion is to directly combine pixels. Image stitching algorithms [21] can be used to merge several images into a larger image. Such algorithms are more often used for photography—that is, directly-viewed images—than for computer vision. Thanks to the large amount of data involved, some amount of preprocessing is often done before merging data. Features can be extracted using a variety of methods. Scale-invariant feature transform (SIFT) is a widely used feature extraction algorithm. A set of feature vectors is extracted from the image as min/max of the difference-of-Gaussians function to smoothed and resampled images. The features are then clustered to identify corresponding features in the images to be compared. These features can be combined in a variety of ways using distributed algorithms, either tree-based or graph-based. Group structure is an issue that cuts across image processing and distributed algorithms. The natural structure of the problem suggests certain groupings of data that are reflected in the formulas that describe the image processing being performed. Distributed algorithms and networks also encourage grouping. In many cases, the groupings suggested by those two disciplines are complementary but in some cases their recommendations conflict.

Group structure in networks is often based on network distance as illustrated in Fig. 1. Network distance is determined in part in a wireless network by the transmission radius of a node, which determines what other nodes can be reached directly. It is also determined by the path length between nodes in a network. In contrast, group structure for image processing algorithms is determined by field-of-view as illustrated in Fig. 2. When cameras have overlapping fields of view, a target is in general visible to several cameras, These cameras are not always physically adjacent—in fact, to battle occlusion, cameras at widely different angles provide the best coverage. The cameras also may not be close in the network sense. However, these cameras do need to talk to each other. In the case of non-overlapping fields-of-view, as shown in Fig. 3, communication between nodes is determined by the possible or likely paths of targets, which may or may not correspond to network topology. Features from multiple sensors need to be combined at some point. Where

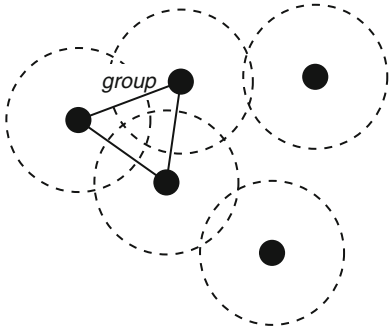


Fig. 1 Groups in wireless networks

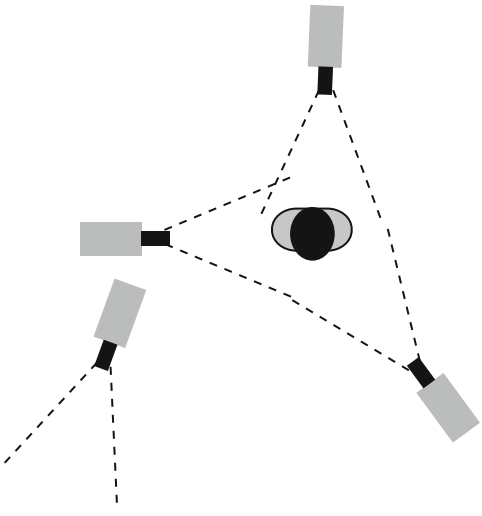


Fig. 2 Groups in overlapping field-of-view image processing

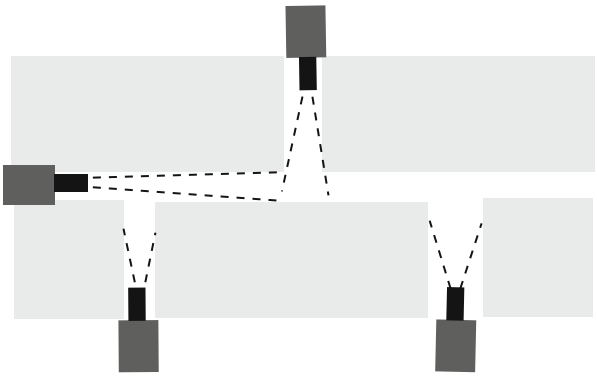


Fig. 3 Groups in non-overlapping cameras

they are combined and how they are combined are both factors. We will use two example systems to illustrate different approaches to feature analysis, fusion, and classification. A separate chapter in this book describes sensor networks in more detail.

5 Camera Calibration

Several aspects of the camera network need to be calibrated:

- **Intrinsic** calibration determines camera parameters such as focal length and color response.

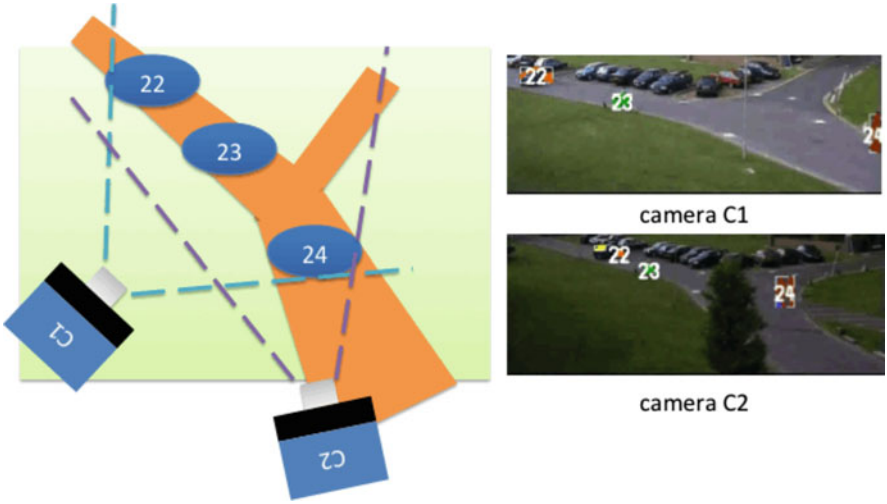


Fig. 4 Overlapping fields of view

- **Extrinsic** spatial calibration determines the position of the camera relative to the scene that it views and, in the case of camera networks, to the other cameras in the system.
- **Temporal** calibration determines the time at which frames are captured and, in the case of camera networks, how the cameras are synchronized relative to each other.

Some calibration problems for camera networks are similar to the calibration problems for a single camera. Camera networks also introduce new calibration problems. Without proper calibration, we cannot properly compare data or analysis results between cameras.

Figure 4 shows a simple example of extrinsic calibration. The two cameras have different views of the road and the tracking subjects.

The book by Hartley and Zisserman [13] provides a thorough discussion of camera calibration. A thorough review of the subject is beyond the scope of this article; we concentrate here on distributed algorithms to solve calibration problems. However, we can identify a few basic techniques for calibration. Intrinsic calibration is necessary to determine the relationship between image features and objects in world coordinates. When we have multiple cameras, we need to determine the relationships between the cameras as well as the internal parameters of each camera. The **fundamental matrix** describes the relationship between two views of a point in space. The **three-view geometry** problem determines the relationships between three cameras, which is based on the correspondence between two lines and a point. The **trifocal tensor** describes the required relationship, along with a set of internal constraints. The **four-view geometry** problem is yet more complex and is often

solved using the simpler affine camera model. This problem can be generalized to the n-camera case.

Radke et al. [25] developed a distributed algorithm for the metric calibration of camera networks. External calibration of a camera determines the position of the camera in world coordinates using a rotation matrix and translation vector. Intrinsic parameters include focal length, location of the principal point, and skew. Calibrating cameras through image processing is usually more feasible than by directly measuring camera position. Devarajan et al. model the camera network using a communication graph and a vision graph. The communication graph is based on network connectivity—it has an edge between nodes that directly communicate; this graph can be constructed using standard ad-hoc network techniques. The vision graph is based on signal characteristics—it has an edge between two nodes that have overlapping fields-of-view; this graph needs to be constructed during the calibration process. Each camera is described by a 3×4 matrix P_i that gives the rotation matrix and optical center of the camera. The intrinsic parameter matrix for camera i is known as K_i. A set of points $X = \{X_1, \ldots, X_n\}$ are used as reference points for calibration; these points can be selected using a variety of methods, such as SIFT. Two cameras have overlapping fields-of-view if they can both see a certain number of the Xs.

The reconstruction is ambiguous and we need to estimate a matrix H that turns the reconstruction into a metric form (preserving angles, etc.). A bundle adjustment step uses nonlinear minimization to improve the calibration. Synchronization, also known as temporal calibration, is necessary to provide an initial time base for video analysis. Cameras may also need to be resynchronized during operation. Even professional cameras may not provide enough temporal stability for long-running experiments and low-cost cameras often display wide variations in frame rate. Lamport and Melliar-Smith [18] developed an algorithm for synchronizing clocks in distributed systems; such approaches are independent of the distributed camera application. Velipasalar and Wolf [30] developed a video-based algorithm for synchronization. The algorithm works by matching target positions between cameras, so it assumes that the cameras have previously been spatially calibrated. A set of frames is selected for comparison by each camera. For each frame, an anchor point is selected. The targets are assumed to be on a plane, so the anchor point is determined by dropping a line from the top of the bounding box to the ground plane. Given these reference points, a search algorithm compares the positions of the targets to minimize the distance $D_{i,j}^{1,2}$ between frame sequences 1 and 2 at offsets i and j.

Pollefeys et al. [23] developed an algorithm for combined spatial and temporal calibration of camera networks. They use a moving target, such as a person, to provide data for calibration in the form of boundary points on the target's silhouette. They use RANSAC to match points from frames from two videos. The RANSAC search evaluates both the epipolar geometry and the temporal offset between the frames. They first generate a consistent set of fundamental matrix for three cameras.

They then add cameras one at a time until either all cameras have been calibrated or the errors associated with a camera's calibration are too large.

Porikli and Divakaran [24] calibrate color models of cameras by recording images of identical objects from each camera and then computing a correlation matrix of the histograms generated for the object from each camera.

6 Tracking

Tracking models the movement of targets over an extended period. The tracking problem can take different forms when several cameras are used, depending on whether the fields-of-views of the cameras overlap.

6.1 Tracking with Overlapping Fields-of-View

Velipasalar et al. [29] developed a peer-to-peer tracking system that fuses data at a higher level. At system calibration time, the cameras determine all other cameras that share overlapping fields-of-view. This information is used to build groups for sharing information about targets. Each camera runs its own tracker. As illustrated in Fig. 5, a camera maintains for each target in its field-of-view the position and appearance model of the target. The cameras then trade information about targets. The cameras that have a given target in their fields-of-view form a group. Cameras communicate every n frames, where n is chosen at design time. A more adaptive approach would be to adapt the communication rate based upon

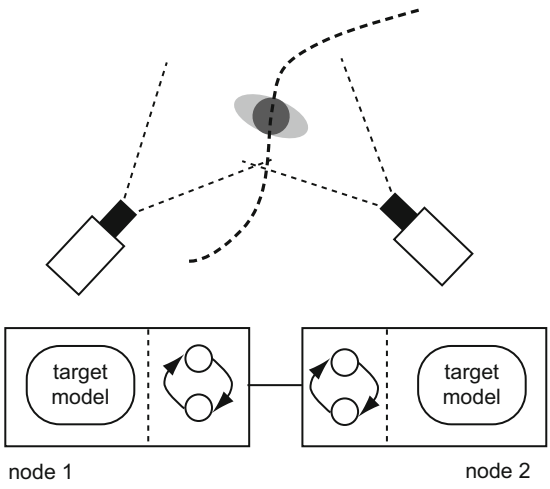


Fig. 5 Combining tracks during distributed tracking

the rate at which targets move. The group is built on top of a logical ring structure, which is commonly used to communicate in multiprocessors. The cameras in the group exchange information on the position of the target. Exchange allows cameras to estimate position more accurately. It also allows cameras whose view of the target is occluded to know where the target is during the occlusion interval. The protocol also negotiates an identity for each target. When a target enters the system, the cameras must also agree upon a common label for the target. They compare appearance models and positions to determine that they, in fact share a target. They then select a common label for that target.

Fleuret et al. [12] use dynamic programming to track multiple people in a scene. They discretize the planar floor into discrete regions and compute the most likely positions of people from frame to frame. They solve overlapping windows of 100 frames. Models for motion and color constrain the search space. They identify tracks one subject at a time to minimize the size of the search space.

6.2 Tracking in Sparse Camera Networks

In many environments, we cannot set up cameras whose fields-of-view cover the entire area of interest. In sparse camera networks, we must find alternative methods to correlate observations between cameras. The most common approach is to use Bayesian metrics to group observations into sets that correspond to paths. Search algorithms find the most likely assignments of observations to paths. A graph models the set of observation positions as nodes and the possible paths between those positions as edges.

Oh et al. [22] developed a Markov chain Monte Carlo (MCMC) algorithm for multi-target tracking (MCMC-MTT). We are given a set of target positions x_i and a set of observations y_i. We partition the available observations into a set of tracks $\tau = \{y_1, \ldots, y_t\}$. The set of all tracks for a given scene is ω. Given a new set of observations y_{t+1}, we want to assign them to tracks and possibly reassign existing observations to different tracks, generating an updated set of tracks ω' such that we maximize the posterior of ω'. Oh et al. showed that the posterior of ω' is:

$$P(\omega \mid Y) = \frac{1}{Z} \prod_{1 \le t \le T} p_z^{z_t} (1 - p_z)^{c_t} p_d^{d_t} (1 - p_d)^{u_t} \lambda_b^{a_t} \lambda_f^{f_t} \times \prod_{\tau \in \omega / \{\tau_0\}} \prod_{1 \le i \le |\tau| - 1} N(\tau(t_{i+1} \mid \bar{x}_{t_{i+1}}(\tau), b_{t_{i+1}}(\tau)) \tag{6}$$

In this formula, Z is a normalizing constant and $N()$ is the Gaussian density function with mean μ and covariance matrix Σ. In the approach of Oh et al., MCMC multi-target tracking makes use of several types of moves to generate new candidate tracks:

- Birth/death: A birth move creates a new track while death destroys an existing one.
- Split/merge: A split breaks one track into two pieces while a merge combines two tracks into one.
- Extension/reduction: Extension lengthens an existing track by adding new observations at the end of the track while reduction shortens a track.
- Track update: An update adds an observation to a track.
- Track switch: A track switch exchanges observations between two tracks.

A move is accepted or rejected with probability

$$A(\omega, \omega') = \min\left(1, \frac{\pi(\omega')q(\omega', \omega)}{\pi(\omega)q(\omega, \omega')}\right) \tag{7}$$

where $\pi(\omega)$ is the stationary distribution of the states and $q(\omega, \omega')$ is the proposal distribution.

Kim et al. [16] used a modified graph model to encapsulate information about priors that guides the search process. Entry and exit nodes are often connected by cycles of edges that model entry and exit. Such cycles can induce long paths to be generated that consist of the target shuttling between two nodes. A supernode models a set of nodes and associated edges. The supernode represents the prior knowledge that such cycles are unlikely. Kim and Wolf [17] developed a distributed algorithm to perform the search over candidate paths. A camera is able to communicate only with nearby cameras. Each camera estimates local paths for the targets using its own observations and observations obtained from nearby cameras. A maximum weighted bipartite matching algorithm is used to match observations with their immediate predecessors. The local paths are then concatenated to create global paths for the targets.

Song and Roy-Chowdhury [28] used a modified form of optimal path solving with random variables for weights. They compare the similarity between observations to compute a similarity score for a pair of observations. They then solve a maximum matching problem in a weighted bipartite graph. They handle variations among the observations by relaxing independence of the correspondences between features using a path smoothness function.

Javed et al. [15] model variations in appearance of targets from camera to camera due to a combination of camera parameters, illumination, and pose using brightness transfer functions (BTFs). They generated inter-camera BTFs for pairs of cameras during a training phase using object histograms. They then use inter-camera BTFs to modify the similarity metric for observations during computation of the track.

Tracking from pan-tilt-zoom (PTZ) cameras is more challenging. de Agapito et al. [1] developed an algorithm for calibrating using a set of images from a camera that rotates. They assume that pixels are square, that the images are not skewed, and that there is a known principal point. Given those constraints, they can formulate the homographies between the reference image and the other images from the camera.

Standard least-squares algorithms can be used to solve the system of equations. Del Bimbo et al. [4] perform real-time tracking taking into account the homography between the reference view and successive frames of the PTZ camera. They use a particle filter to estimate the camera parameters. They use a SIFT algorithm to extract points in the images.

7 Gesture Recognition

A gesture is a sequence of poses [34]. Each pose can be extracted from a frame. The sequence of poses can then be determined from the poses using a variety of techniques such as hidden Markov models.

Van den Bergh et al. [2] developed an algorithm for pose estimation. They extract a three-dimensional hull over the subject using voxel carving. They achieve real-time performance by using a fixed lookup table for each voxel that gives for each pixel the set of voxels that project onto that pixel. They use an example-based classifier to bin the 3D hulls into poses.

Lin et al. [19] developed a peer-to-peer algorithm for gesture recognition. The distributed system is based on a single-camera gesture recognition system [34]. That algorithm extracts shapes, builds a graph model of the relationships between the shapes, and then matches that graph against a library of graphs representing poses.

The basic challenge in extending the single-camera system to multiple cameras is that a subject may stand such that no single camera has a complete view. In general, data must be combined from multiple cameras before a graph can be built. A brute-force approach would be to share frames, but this entails a substantial bandwidth penalty. Lin's system combines the features created by pixel-level region identification. The regions from different cameras are easily fused but they can be transmitted at lower cost. As shown in Fig. 6, when the target appears in the view

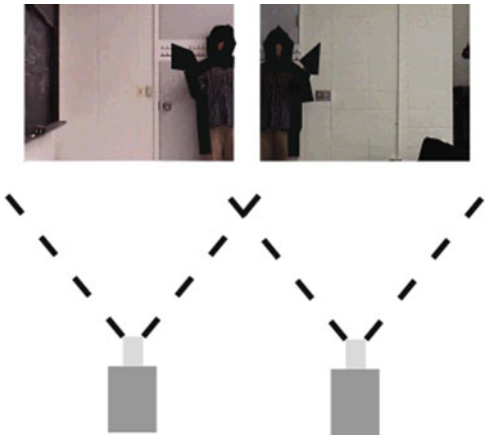


Fig. 6 Combining features for gesture recognition

of multiple cameras, each camera extracts the foreground object and extracts pixel boundaries for the foreground regions. A protocol determines which camera will fuse these boundaries and complete the recognition process. A group is formed by the cameras that can view the target. One member of the group, the lead node, has a token that denotes that it is responsible for fusion. The token moves to a different node when the target's position, as determined by the centerline of its bounding box, moves out of the field-of-view of the lead node. The token is passed to a camera whose field-of-view contains the target centerline.

8 Platform Architectures

The computing platform includes processing elements, interconnect, memory, and networking. The details of the platform can vary widely, but the fundamental assumptions made by most distributed computer vision algorithms include the nodes and network in a distributed system. Processing nodes are assumed to have reasonably powerful processors that can perform significant amounts of work in real time. RAM is generally not unlimited but enough is available for the required algorithms. Most systems assume that power is available to nodes for continuous operations, although it is possible to build a battery-operated system that will operate for a short interval. Network bandwidth and quality-of-service are an important consideration. Wireless networks generally do not provide enough bandwidth to allow a large number of cameras to stream video for processing in the cloud. Wired networks provide more bandwidth and better QoS but streaming video from multiple nodes will still require significant network resources. Distributed smart cameras that process video streams locally are motivated in part by networking limitations.

Heterogeneous platforms are commonly used for embedded computer vision. These platforms provide different types of computational units that can be applied to different parts of the system. In addition to CPUs, heterogeneous platforms may make use of graphics processing units (GPUs), digital signal processors (DSPs), and field-programmable gate arrays (FPGAs). The OpenCV library (http://opencv.org) has become a popular development environment for computer vision applications. OpenCV has been ported to a wide variety of platforms, including both CPUs and GPUs. Many platforms that target embedded computer vision applications include GPUs. These platforms fall into one of two categories: embedded GPUs such as the NVIDIA Tegra used in their Jetson platform is designed for mobile graphics and computer vision; smartphone platforms such as the Qualcomm Snapdragon family include both computational and wireless networking resources.

A GPU provides large number of floating-point units that can operate on local memory, features that can be exploited by computer vision algorithms. Modern GPUs use a multithreaded programming model to provide highly parallel execution. Yu and van Engelen [35] developed a GPU-based algorithm for importance sampling with parallelization of irregular wavefronts. They exploited properties of the probability distribution to determine an address calculation for accessing

importance function tables. Chen et al. [9] developed a hardware architecture for k-means clustering with hierarchical data sampling; their design included pipelined processors for logarithm and Bayesian Information Criterion computations. Bodensteiner and Arens [5] developed an algorithm for registration of 2D video to 3d LiDAR imagery; they used a GPU to perform backprojection of the video 2D features to the LiDAR models. Berjon et al. [3] developed a Bayesian classifier for moving object detection. They used region-of-interest masks to reduce computation time; they mapped the irregular set of RoI areas onto the GPU using a 1D list. Wang et al. [31] developed an MCMC-based learning algorithm for GPUs.

DSPs are instruction set processors optimized for streaming and array processing workloads found in digital signal processing. Very long instruction word (VLIW) architectures, which statically dispatch several instructions per cycle, are widely used in DSPs. Vector processors are also commonly used for signal processing workloads to take advantage of the vector and matrix operations commonly found in DSP and computer vision workloads.

Field-programmable gate arrays (FPGAs) can be used to build custom accelerators and heterogeneous architectures without resorting to a custom chip design. Typically an FPGA will be utilized either as a design and test platform for a final custom build or as a target reconfigurable fabric for these cases, respectively. Much like DSPs, these provide optimized processing ability, in this case with specialized processing units within the FPGA fabric. For example, high-speed integer multipliers and adders are found in many FPGA cells currently on the market. Furthermore, many FPGAs incorporate a simple embedded processor for algorithm flow control. The complexity in using this type of platform is in design overhead. An FPGA requires significantly more design consideration than the previously mentioned platforms. This is due to the inherent need for hardware and software task analysis. Also, the determined hardware tasks typically cannot be directly mapped to FPGA logic, and instead require algorithmic redesign and hardware timing scheme design and analysis.

All of these platforms raise the need for memory bandwidth analysis during architectural design and mapping. This is due to the platforms having a smaller footprint, and resultant lower amount of available storage on the board, not to mention on the chip itself. As vision algorithm complexity increases through use of larger pixel representation schemes, greater temporal frame storage, and also accuracy requirements which abound in real-time critical vision systems, the amount of data which needs to be both transferred on and off chip as well as stored on-chip intermediately increases dramatically. Schlessman [26] addresses each of these issues with the MEAD methodology for embedded architectural design, ultimately providing vision designers with a clearer concept of what platform architecture is best suited for the algorithm under consideration.

9 Summary

Distributed algorithms offer a new approach to computer vision. Nodes in a distributed system communicate explicitly through mechanisms that have non-zero cost in delay and energy. This constraint encourages us to consider algorithms that refine the representation of a scene in stages with predictable ways of combining data within a stage. Distributed algorithms are, in general, more difficult to develop and prove correct, but they also provide advantages such as robustness. Distributed signal processing may require new statistical representations of signals that can be efficiently shared over the network while preserving the useful properties of the underling signal. We also need a better understanding of the accuracy of the approximations underlying distributed signal processing algorithms—how do termination conditions affect the accuracy of the estimate, for example.

Acknowledgements This work was supported in part by the National Science Foundation under grant 0720536.

References

1. de Agapito, L., Hartley, R., Hayman, E.: Linear self-calibration of a rotating and zooming camera. In: Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on., vol. 1, pp. 2 vol. (xxiii+637+663) (1999). https://doi.org/10.1109/CVPR.1999.786911
2. den Bergh, M.V., Koller-Meier, E., Kehl, R., Gool, L.V.: Real-time 3d body pose estimation. In: H. Aghajan, A. Cavallaro (eds.) Multi-Camera Networks: Principles and Applications, chap. 14. Academic Press (2009)
3. Berjon, D., Cuevas, C., Moran, F., Garca, N.: Region-based moving object detection using spatially conditioned nonparametric models in a GPU. In: 2014 IEEE International Conference on Consumer Electronics (ICCE), pp. 359–360 (2014). https://doi.org/10.1109/ICCE.2014.6776041
4. Bimbo, A.D., Dini, F., Pernici, F., Grifoni, A.: Pan-tilt-zoom camera networks. In: H. Aghajan, A. Cavallaro (eds.) Multi-Camera Networks: Principles and Applications, chap. 8. Academic Press (2009)
5. Bodensteiner, C., Arens, M.: Real-time 2d video/3d lidar registration. In: Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012), pp. 2206–2209 (2012)
6. Boykov, V., Huttenlocher, D.: Adaptive Bayesian recognition in tracking rigid objects. In: Proceedings, IEEE Conference on Computer Vision and Pattern Recognition, pp. 697–704. IEEE (2000)
7. Bramberger, M., Quaritsch, M., Winkler, T., Rinner, B., Schwabach, H.: Integrating multi-camera tracking into a dynamic task allocation system for smart cameras. In: Proceedings of the IEEE Conference on Advanced Video and Signal Based Surveillance (AVSS 2005), pp. 474–479. IEEE (2005)
8. Candy, J.V.: Boostrap particle filtering. IEEE Signal Processing Magazine **73**, 73–85 (2007)
9. Chen, T.W., Chien, S.Y.: Flexible hardware architecture of hierarchical k-means clustering for large cluster number. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **19**(8), 1336–1345 (2011). https://doi.org/10.1109/TVLSI.2010.2049669

10. Coates, M.: Distributed particle filters for sensor networks. In: Information Processing in Sensor Networks, 2004. IPSN 2004. Third International Symposium on, pp. 99–107. IEEE (2004)
11. Collins, R.T., Lipton, A.J., Fujiyoshi, H., Kanade, T.: Algorithms for cooperative multisensory surveillance. Proceedings of the IEEE **89**(10), 1456–1477 (2001)
12. Fleuret, F., Berclaz, J., Lengagne, R., Fua, P.: Multicamera people tracking with a probabilistic occupancy map. Pattern Analysis and Machine Intelligence, IEEE Transactions on **30**(2), 267–282 (2008). https://doi.org/10.1109/TPAMI.2007.1174
13. Hartley, R.I., Zisserman, A.: Multiple View Geometry in Computer Vision, second edn. Cambridge University Press, ISBN: 0521540518 (2004)
14. Horprasesert, T., Harwood, D., Davis, L.S.: A statistical approach for real-time robust background subtraction and shadow detection. In: IEEE International Conference on Computer Vision FRAME-RATE Workshop (1999)
15. Javed, O., Shafique, K., Shah, M.: Appearance modeling for tracking in multiple non-overlapping cameras. In: Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on, vol. 2, pp. 26–33 (2005). https://doi.org/10.1109/CVPR.2005.71
16. Kim, H., Romberg, J., Wolf, W.: Multi-camera tracking on a graph using Markov chain monte carlo. In: Proceedings, 2009 ACM/IEEE International Conference on Distributed Smart Cameras. ACM (2009)
17. Kim, H., Wolf, M.: Distributed tracking in a large-scale network of smart cameras. In: Proceedings of the Fourth ACM/IEEE International Conference on Distributed Smart Cameras, p. 8–16. ACM Press (2010)
18. Lamport, L., Melliar-Smith, M.: Synchronizing clocks in the presence of faults. Journal of the ACM **32**(1), 52–78 (1985)
19. Lin, C.H., Lv, T., Wolf, W., Ozer, I.B.: A peer-to-peer architecture for distributed real-time gesture recognition. In: Proceedings, International Conference on Multimedia and Exhibition, pp. 27–30. IEEE (2004)
20. Mallett, J., Jr., V.M.B.: Eye society. In: Proceedings IEEE ICME 2003. IEEE (2003)
21. McMillan, L., Bishop, G.: Plenoptic modeling: an image-based rendering system. In: Proceedings, ACM SIGGRAPH, pp. 39–46. ACM (1995)
22. Oh, S., Russell, S., Sastry, S.: Markov chain monte carlo data association for general multiple-target tracking problems. In: Proc. 43rd IEEE Conf. Decision and Control (2004)
23. Pollefeys, M., Sinha, S.N., Guan, L., Franco, J.S.: Multi-view calibration, synchronization, and dynamic scene reconstruction. In: H. Aghajan, A. Cavallaro (eds.) Multi-Camera Networks: Principles and Applications, chap. 2. Academic Press (2009)
24. Porikli, F., Divakaran, A.: Multi-camera calibration, object tracking and query generation. Tech. Rep. TR-2003-100 (2003)
25. Radke, R., Devarajan, D., Cheng, Z.: Calibrating distributed camera networks. Proceedings of the IEEE **96**(10), 1625–1639 (2008)
26. Schlessman, J.: Methodology and architectures for embedded computer vision (2012). PhD Thesis, Department of Electrical Engineering, Princeton University, in preparation
27. Sheng, X., Hu, Y.H., Ramanathan, P.: Distributed particle filter with gmm approximation for multiple targets localization and tracking in wireless sensor network. In: Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on, pp. 181–188. IEEE (2005)
28. Song, B., Roy-Chowdhury, A.: Robust tracking in a camera network: A multi-objective optimization framework. Selected Topics in Signal Processing, IEEE Journal of **2**(4), 582–596 (2008). https://doi.org/10.1109/JSTSP.2008.925992
29. Veliapasalar, S., Schlessman, J., Chen, C.Y., Wolf, W.H., Singh, J.P.: A scalable clustered camera system for multiple object tracking. EURASIP Journal on Image and Video Processing **2008** (2008). Article ID 542808

30. Velipasalar, S., Wolf, W.H.: Frame-level temporal calibration of video sequences from unsynchronized cameras. Machine Vision and Applications Journal (DOI 10.1007/s00138-008-0122-6) (2008)
31. Wang, Y., Qian, W., Zhang, S., Liang, X., Yuan, B.: A learning algorithm for bayesian networks and its efficient implementation on gpus. IEEE Transactions on Parallel and Distributed Systems **27**(1), 17–30 (2016). https://doi.org/10.1109/TPDS.2014.2387285
32. Wolf, W.: High Performance Embedded Computing. Morgan Kaufman (2006)
33. Wolf, W.: Modern VLSI Design: IP-Based Design, fourth edn. PTR Prentice Hall (2009)
34. Wolf, W., Ozer, B., Lv, T.: Smart cameras as embedded systems. IEEE Computer **35**(9), 48–53 (2002)
35. Yu, H., van Engelen, R.: Importance sampling on Bayesian networks with deterministic causalities

Part II
Architectures

Arithmetic

Oscar Gustafsson and Lars Wanhammar

Abstract In this chapter fundamentals of arithmetic operations and number representations used in DSP systems are discussed. Different relevant number systems are outlined with a focus on fixed-point representations. Structures for accelerating the carry-propagation of addition are discussed, as well as multi-operand addition. For multiplication, different schemes for generating and accumulating partial products are presented. In addition to that, optimization for constant coefficient multiplication is discussed. Division and square-rooting are also briefly outlined. Furthermore, floating-point arithmetic and the IEEE 754 floating-point arithmetic standard are presented. Finally, some methods for computing elementary functions, e.g., trigonometric functions, are presented.

1 Number Representation

The way we select to represent our numbers has a profound impact on the corresponding computational units. Here we consider number representations based on a positional weight (radix) of two. It is worth noting that as the main application area considered here is digital signal processing (DSP), we will, where required, choose to consider numbers that are fractional rather than integer. This will mainly effect the numbering of the indices.

O. Gustafsson (✉) · L. Wanhammar
Department of Electrical Engineering, Linköping University, Linköping, Sweden
e-mail: oscar.gustafsson@liu.se; lars.wanhammar@liu.se

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_11

1.1 Binary Representation

An unsigned binary number, X, with W_f fractional bits can be written as

$$X = \sum_{i=1}^{W_f} x_i 2^{-i}, \tag{1}$$

where $x_i \in \{0, 1\}$. Denoting the weight of the least significant position as Q, in this case $Q = 2^{-W_f}$, one can see that the range of X is $0 \leq X \leq 1 - Q$. Q is sometimes referred to as unit of least significant position, *ulp*. As an example, the number 0.25_{10} is written using $W_f = 3$ as $.010_2$ and $Q = 2^{-3} = 0.125_{10}$.

1.2 Two's Complement Representation

To represent negative numbers, there are several different number representations proposed. The most common one is the two's complement (2C) representation. Here, a binary number, X, with W_f fractional bits is written as

$$X = -x_0 + \sum_{i=1}^{W_f} x_i 2^{-i}. \tag{2}$$

This gives a numerical range as $-1 \leq X \leq 1 - Q$. It is worth noting that the range is not symmetric. This will cause problems when implementing certain arithmetic operations as discussed later. The sign bit, x_0, is one if $X < 0$ and zero otherwise. A number -0.25_{10} is represented as 1.11_{2C} with $W_f = 2$, while 0.25_{10} is 0.01_{2C} .

A beneficial property of two's complement arithmetic is the fact that an arbitrary long sequence of numbers can be added in arbitrary order as long as the result is known to be in the range of the representation. Any overflows/underflows in the intermediate computations will cancel. This is related to the fact that computations in two's complement number representation are performed modulo $2W_S$, where W_S is the weight of the sign bit. For the representation in (2) $W_S = 1$ so all computations are performed modulo 2.

1.3 Redundant Representations

A redundant representation is a representation where a number may have more than one representation. As we will see later the fact that we can select the representation will provide a number of advantages, most importantly the ability to perform addition in constant time.

1.3.1 Signed-Digit Representation

In a signed-digit (SD) number representation, the digits may have either positive or negative sign. For a radix-2 representation we have $x_i \in \{-1, 0, 1\}$. Using W_f fractional bits as in (1) the range of X is $-1 + Q \leq X \leq 1 - Q$. A number may now have more than one representation. Consider the number 0.25_{10} which can be written as $.01_{SD}$ or $.1\bar{1}_{SD}$, where $\bar{1}$ is used to denote -1.

In some applications as, for instance, digital filters [30], FFTs [5] and DCTs [28] as well as general DSP algorithms [45], it is of interest to find a signed-digit representation with a minimum number of non-zero positions to simplify the corresponding multiplication (see Sect. 3.5). This is referred to as a minimum signed-digit (MSD) representation. However, in general it is non-trivial to determine if a representation is minimum. A specific minimum signed-digit representation is obtained if the constraint $x_i x_{i+1} = 0, \forall i$ is imposed. The resulting representation is called canonic signed-digit (CSD) representation and is apart from being minimum also unique (as the name indicates).

1.3.2 Carry-Save Representation

The carry-save representation stems from the ripple-carry adder, which will be further discussed in Sect. 2.1. Instead of propagating the carries in the addition, these bits are stored and the data is represented using two vectors. This also leaves an additional input of the full adder cells unused, so it is possible to add three vectors. A full adder cell is illustrated in Fig. 1b and the truth table is given in Fig. 1a. From this we can see that

$$s_i = \text{parity}\,(a_i \oplus b_i \oplus d_i) = a_i \oplus b_i \oplus d_i, \tag{3}$$

where $\oplus$ is the exclusive-OR operation, and

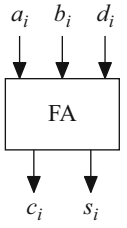
$$c_i = \text{majority}\,(a_i, b_i, d_i) = a_i b_i + a_i d_i + b_i d_i. \tag{4}$$

Fig. 1 Full adder: (**a**) truth table, (**b**) symbol

a

a_i	b_i	d_i	s_i	c_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

b



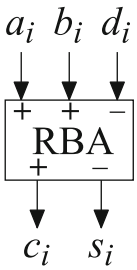


Fig. 2 Redundant binary adder with two positive and one negative inputs: symbol

a_i	b_i	d_i	s_i	c_i	Result
0	0	0	0	0	0
0	0	1	1	0	−1
0	1	0	1	1	1
0	1	1	0	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	0	1	2
1	1	1	1	1	1

Fig. 3 Redundant binary adder with two positive and one negative inputs: truth table

Assuming two's complement representation of the vectors, a number is represented as

$$X = S + C = -s_0 + \sum_{i=1}^{W_f} s_i 2^{-i} - c_0 + \sum_{i=1}^{W_f} c_i 2^{-i}. \tag{5}$$

It is possible to represent numbers in the range $-2 \leq X \leq 2 - 2Q$. However, it is common to let X span the same numerical range as the two's complement representation since it usually will be converted into a non-redundant representation. The carry-save representation can also be seen as a representation with radix-2 and digits $x_i \in \{0, 1, 2, 3\}$ where $x_i = 2c_i + s_i$ (Fig. 3).

The concept of carry-save representation can be generalized to binary redundant representations by replacing the relation $x_i = 2c_i + s_i$ with either $x_i = 2c_i - s_i$ where $x_i \in \{-1, 0, 1, 2\}$ or $x_i = -2c_i + s_i$ where $x_i \in \{-2, -1, 0, 1\}$. It is then possible to use cells corresponding to full adders, but with designated signs of the inputs, such that the bits with negative weights should be connected to '−'-inputs. This is illustrated in Fig. 3 while the computational rules for an adder with two positive and one negative input are shown in Fig. 2. Note that both types of representations needs to be used as otherwise there would be an imbalance as each adder produces one positive and one negative vector but inputs either one positive and two negative or two positive and one negative.

1.4 Shifting and Increasing the Word Length

When adding signed numbers it is for most number representations important that the vectors have the same lengths. For two's complement representation we must extend the sign by copying the sign-bit at the MSB side of the vector. On the LSB side it is enough to extend with zeros. It is worth noting that for one's complement the sign-bits are inserted at the LSB side.

The operation of shifting, i.e., multiplying or dividing by a power of two, is in fact the same as increasing the word length. Hence, shifting a two's complement number two positions to the right (dividing by 4) requires copying of the sign-bit to the two introduced positions.

For carry-save representation, the extension of the word length on the MSB side requires that the sign-bits of the carry and sum vectors are corrected to avoid what is referred to as carry overflow [38]. The origin of this effect is that once the carry and sum vectors are added there is an overflow in the addition. As the resulting carry bit from the MSB position is neglected the result still remains valid, in fact this is a required property for two's complement representation to work. However, if the vectors are shifted once before the final addition, the effect of the overflow will occur in an earlier position.

1.5 Negation

Negation is a useful operation in itself, but also in a subtraction, $Z = X - Y$, which can be seen as an addition of the negated value, $Z = X + (-Y)$.

To negate a two's complement number one inverts all the bits and add a one to the LSB position. As will be seen later on, this does not necessarily have to be performed explicitly, but can rather be integrated with an addition.

1.6 Finite Word Length Effects

In recursive algorithms, it is necessary to use non-linear operations to maintain a finite word length. This introduces small re-quantization errors, so called granularity errors. In addition, very large overflow errors will occur if the finite number range is exceeded. These errors will not only cause distortion, but may also be the cause of parasitic oscillations in recursive algorithms [6, 14, 47].

The effect of these errors depends on many factors, for example, type of quantization, algorithm, type of arithmetic, representation of negative numbers, and properties of the input signal[31]. The analysis of the influence of round-off errors in floating-point arithmetic is very complicated [32, 46, 58], because quantization of products as well as addition in floating-point arithmetic and subtraction causes

errors that depend on the signal values. Quantization errors in fixed-point arithmetic are with few exceptions independent of the signal values and can, therefore, be analyzed independently. Lastly, the high dynamic range provided by floating-point arithmetic is not really needed in good filter algorithms, since filter structures with low coefficient sensitivity also utilize the available number range efficiently.

Fixed-point arithmetic is predominately used in application-specific ICs since the required hardware is much simpler, faster, and consumes less power compared with floating-point arithmetic. We will, therefore, focus on fixed-point arithmetic.

1.6.1 Overflow Characteristics

A two's complement representation of negative numbers is usually employed in digital hardware. The overflow characteristic of the two's complement representation is shown in Fig. 4.

As discussed earlier, the largest and smallest numbers in two's complement representation are $1 - Q$ and -1, respectively. A two's complement number larger than $1 - Q$ will be interpreted as $x - 2$, while a number slightly smaller than -1 will be interpreted as $x - 2$. Hence, very large overflows errors are incurred. A common scheme to reduce the size of overflow errors and mitigate their harmful effect is to limit numbers outside the normal range to either the largest or smallest representable number. This scheme is referred to as *saturation arithmetic* and is shown in Fig. 5.

Many standard signal processors provide addition and subtraction instructions with inherent saturation. Another saturation scheme, which may be simpler to implement in hardware, is to invert all bits when overflow occurs. Using fixed-point arithmetic the signal levels need to be adjusted at the input of multiplications with non-integer coefficients. Note that, as discussed earlier, a sum of several numbers,

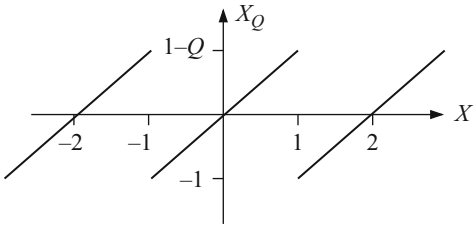


Fig. 4 Overflow characteristic of two's complement arithmetic

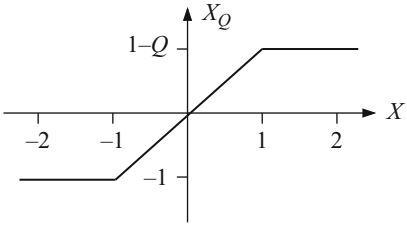


Fig. 5 Overflow characteristic of saturation arithmetic

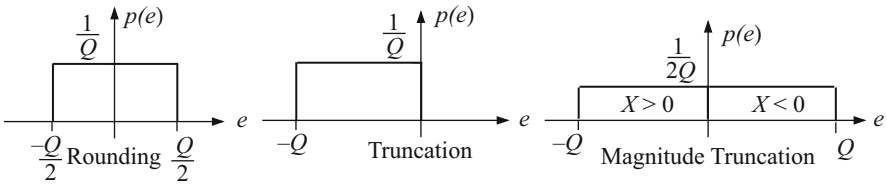


Fig. 6 Error distributions for fixed-point arithmetic

using two's complement representation, may have intermediate overflows as long as the final value (sum) is within the valid range.

1.6.2 Truncation

Quantizing a binary number, X, with infinite word length to a number, X_Q, with finite word length yields an error

$$e = X_Q - X. \tag{6}$$

Truncation of the binary number is performed by removing the bits with index $i > W_f$. The resulting error density distribution is shown in the center of Fig. 6. The variance is $\sigma^2 = \frac{Q^2}{12}$ and the mean value is $-Q/2$ where Q refer to the weight of the last bit position.

1.6.3 Rounding

Rounding is, in practice, performed by adding $2^{-(W_f+1)}$ to the non-quantized number before truncation. Hence, the quantized number is the nearest approximation to the original number. However, if the word length of X is $W_f + 1$, the quantized number should, in principle, be rounded upwards if the last bit is 1 and downwards if it is 0, in order to make the mean error zero. This special case is often neglected in practice. The resulting error density distribution, $p(e)$, is shown to the left in Fig. 6. The variance is $\sigma^2 = \frac{Q^2}{12}$ and the mean value is zero.

1.6.4 Magnitude Truncation

Magnitude truncation quantizes the number so that

$$\left|X_Q\right| \leq |X| \,. \tag{7}$$

Hence, $e \leq 0$ if $X \geq 0$ and $e \geq 0$ if $X \leq 0$. This operation can be performed by adding $2^{-(W_f+1)}$ before truncation if X is negative and 0 otherwise. That is, in two's complement representation adding the sign bit to the last position. The resulting error density distribution is shown to the right in Fig. 6. The error analysis of magnitude truncation becomes very complicated since the error and sign of the signal are correlated [31].

Magnitude truncation is needed to suppress parasitic oscillation in wave digital filters [14].

1.6.5 Quantization of Products

The effect of a quantization operation, except for magnitude truncation, can be modeled with a white additive noise source that is independent of the signal and with the error density functions as shown in Fig. 6. This model can be used if the signal varies from sample to sample over several quantization levels in an irregular way. However, the error density function is a discrete function if both the signal and the coefficient have finite word lengths. The difference is significant only if a few bits are discarded by the quantization. The mean value and variance for the errors are

$$m = \begin{cases} \frac{Q_c}{2} Q, & \text{rounding} \\ \frac{Q_c - 1}{2} Q, & \text{truncation} \end{cases} \tag{8}$$

and

$$\sigma_e 2 = k_e (1 - Q_c^2) \frac{Q^2}{12}, \tag{9}$$

where

$$k_e = \begin{cases} 1, & \text{rounding or truncation} \\ 4 - \frac{6}{\pi}, & \text{magnitude truncation} \end{cases}, \tag{10}$$

where Q and Q_c refer to the signal and coefficient, respectively. For long coefficient word lengths the average value is close to zero for rounding and $-Q/2$ for truncation. Correction of the average value and variance is only necessary for short coefficient word lengths, for example, for the scaling coefficients.

2 Addition

The operation of adding two or more numbers is in many ways the most fundamental arithmetic operation since most other operations in one or another way are based on addition.

The methods discussed here concerns either two's complement or unsigned representation. Then, the major problem is to efficiently speed up the carry-propagation in the adder. There are other schemes than those presented here, for more details we refer to e.g., [59]. It is also possible to perform addition in constant time using redundant number systems such as the previously discussed signed-digit or carry-save representations. An alternative is to use residue number systems (RNS), that split the carry-chain into several shorter ones [40].

2.1 Ripple-Carry Addition

The probably most straightforward way to perform addition of two numbers is to perform bit-by-bit addition using a full adder (FA) cell, shown in Fig. 1b, and propagate the carry bit to the next stage. This is called ripple-carry addition and is illustrated in Fig. 7. This type of adder can add both unsigned and two's complement numbers. However, for two's complement numbers the result must have the same number of bits as the inputs, while for unsigned numbers the carry bit acts as a possible additional bit to increase the word length. The operation of adding two two's complement numbers is outlined in Fig. 8 for the example of numbers $-53/256$ and $94/256$.

The major drawback with the ripple-carry adder is that the worst case delay is proportional to the word length. Also, typically the ripple-carry adder will produce many glitches due to the full adder cells having to wait for the correct carry. This situation is improved if the delay for the carry bit is smaller than that of the sum bit [22]. However, due to the simple design the energy per computation is still reasonably small [59].

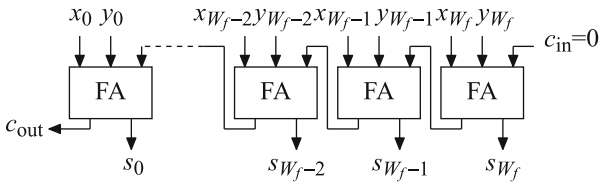


Fig. 7 Ripple-carry adder

Fig. 8 Example of addition in two's complement arithmetic using a ripple-carry adder

Value	0	1	2	3	4	5	6	7	Signal
–53/256	1	1	0	0	1	0	1	1	x_i
94/256	0	1	0	1	1	1	1	0	y_i
	1	1	0	1	1	1	1	0	c_i
41/256	0	0	1	0	1	0	0	1	s_i

2.2 Carry-Lookahead Addition

To speed up the addition several different methods have been proposed, see, for instance, [59]. Methods typically referred to as carry-lookahead methods are based on the following observation. The carry output of a full adder sometimes depends on the carry input and sometimes it is determined without the need of the carry input. This is illustrated in Table 1. Based on this we can define the propagate signal, p_i, and the generate signal, g_i, as

$$p_i = a_i \oplus b_i \text{ and } g_i = a_i b_i. \tag{11}$$

Now, the carry output can be expressed as

$$c_{i-1} = g_i + p_i c_i. \tag{12}$$

For the next stage the expression becomes

$$c_{i-2} = g_{i-1} + p_{i-1}c_{i-1} = g_{i-1} + p_{i-1}(g_i + p_i c_i) = g_{i-1} + p_{i-1}g_i + p_{i-1}p_i c_i. \tag{13}$$

For $N+1$:th stage we have

$$\begin{aligned} c_{i-(N+1)} = \; & g_{i-N} + p_{i-N}g_{i-(N-1)} + p_{i-N}p_{i-N-1}g_{i-(N-2)} + \ldots \\ & + p_{i-(N-1)} \cdots p_{i-1}p_i c_i. \end{aligned} \tag{14}$$

The terms containing g_k and possibly p_k terms are called group generate, as they together acts as a merged generate signal for all the bits i to $i-N$. The subterm $p_{i-(N-1)} \cdots p_{i-1}p_i$ is similarly called group propagate. Both the group generate and group propagate signals are independent of any carry signal. Hence, (15) shows that it is possible to have the carry propagate N stages with a maximum delay of one AND-gate and one OR-gate as illustrated in Fig. 9. However, the complexity and delay of the precomputation network grows with N, and, hence, a careful design is required to not make the precomputation the new critical path.

Table 1 Cases for carry-propagation in a full adder cell

a_i	b_i	c_{i-1}	Case
0	0	0	No carry-propagation (kill)
0	1	c_i	Carry-propagation (propagate)
1	0	c_i	Carry-propagation (propagate)
1	1	1	Carry-generation (generate)

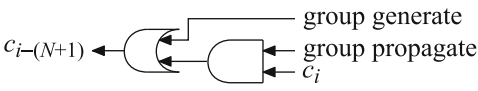


Fig. 9 Illustration of N-stage carry-lookahead carry propagation

The carry-lookahead approach can be generalized using dot-operators. Adders using dot-operators are often referred to as parallel prefix adders. The dot-operator operates on a pair of generate and propagate signals and is defined as

$$\begin{bmatrix} g_k \\ p_k \end{bmatrix} = \begin{bmatrix} g_i \\ p_i \end{bmatrix} \bullet \begin{bmatrix} g_j \\ p_j \end{bmatrix} \triangleq \begin{bmatrix} g_i + p_i g_j \\ p_i p_j \end{bmatrix}. \tag{15}$$

The group generate from position k to position l, $k < l$, can be denoted by $G_{k:l}$ and similarly the group propagate as $P_{k:l}$. These are then defined as

$$\begin{bmatrix} G_{k:l} \\ P_{k:l} \end{bmatrix} \triangleq \begin{bmatrix} g_k \\ p_k \end{bmatrix} \bullet \begin{bmatrix} g_{k+1} \\ p_{k+1} \end{bmatrix} \bullet \cdots \bullet \begin{bmatrix} g_l \\ p_l \end{bmatrix}. \tag{16}$$

The dot operator is associative but not commutative. Furthermore, the dot-operation is idempotent. This means that

$$\begin{bmatrix} g_k \\ p_k \end{bmatrix} = \begin{bmatrix} g_k \\ p_k \end{bmatrix} \bullet \begin{bmatrix} g_k \\ p_k \end{bmatrix}. \tag{17}$$

For the group generate and propagate signals this leads to that

$$\begin{bmatrix} G_{k:n} \\ P_{k:n} \end{bmatrix} = \begin{bmatrix} G_{k:l} \\ P_{k:l} \end{bmatrix} \bullet \begin{bmatrix} G_{m:n} \\ P_{m:n} \end{bmatrix}, \; k \leq l, m \leq n, m \leq l - 1. \tag{18}$$

This is illustrated in Fig. 10. Hence, we can form the group generate and group propagate signals by combining smaller, possibly overlapping, subgroup generate and propagate signals.

The carry signal in position k can be written as

$$c_k = G_{(k+1):l} + P_{(k+1):l} c_l \tag{19}$$

Similarly, the sum signal in position k for an adder using W_f fractional bits is then expressed according to (3) as

$$s_k = a_k \oplus b_k \oplus d_k = p_k \oplus (G_{(k+1):W_f} + P_{(k+1):W_f} c_{\text{in}}) = p_k \oplus c_k \tag{20}$$

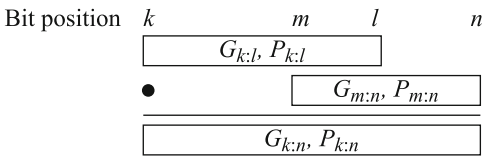


Fig. 10 Illustration of the idempotency property for group generate and propagate signals

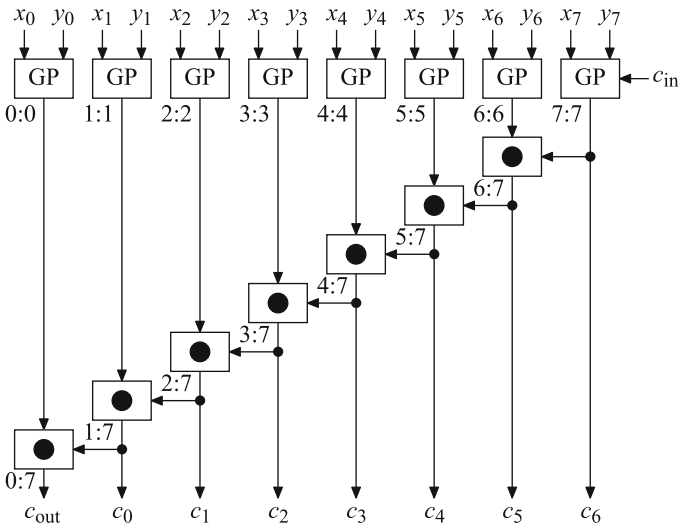


Fig. 11 Sequential computation of group generate and propagate signals

From this, one can see that it is of interest to compute all group generate and group propagate originating from the LSB position, i.e., $G_{k:W_f}$ and $P_{k:W_f}$ for $1 \leq k \leq W_f$.

A straightforward way of obtaining this is to do a sequential operation as shown in Fig. 11. However, again the delay will be linear in the word length, as for the ripple-carry adder. Indeed, the adder in Fig. 11 is a ripple-carry adder where the full adder cells explicitly compute p_i and g_i.

Based on the properties of the dot operator, we can possibly find ways to interconnect the adders such that the depth is reduced by computing different group generate and propagate signals in parallel. This is illustrated for an 8-bit adder in Fig. 12. This particular structure of interconnecting the dot-operators are referred to as a Ladner-Fischer parallel prefix adder [27]. Often one uses a simplified structure to represent the parallel prefix computation, as shown in Fig. 13a. Comparing Figs. 12 and 13a it is clear that dots in Fig. 13a correspond to dot-operators in Fig. 12. In fact, the parallel prefix graphs in Fig. 13 work for any associative operation.

Over the years there has been a multitude of different schemes for parallel prefix addition trading the depth, number of dot-product operations, and fan-out of the dot-product cells. In Fig. 13b–d, three of the earlier proposed schemes for 16-bit parallel prefix computations are illustrated, namely Ladner-Fischer [27], Kogge-Stone [24], and Brent-Kung [4], respectively. Unified views of all possible parallel prefix schemes have been proposed in [20, 23].

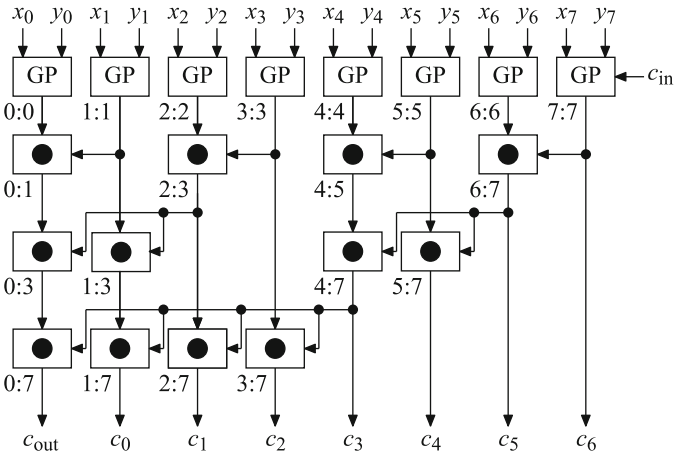


Fig. 12 Parallel computation of the group generate and propagate signals

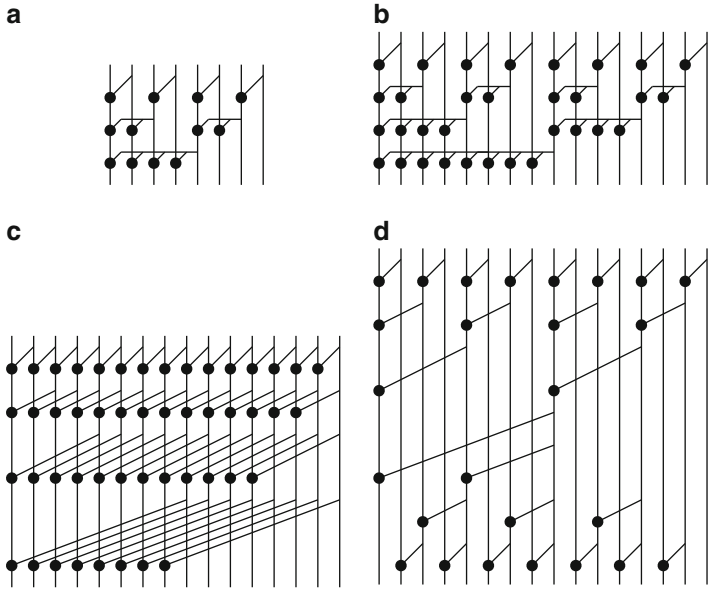


Fig. 13 Different parallel prefix schemes for an 8-bit Ladner-Fischer adder [27] as shown in Fig. 12 and for 16-bit adders: (**b**) Ladner-Fischer [27], (**c**) Kogge-Stone [24], and (**d**) Brent-Kung [4]

2.3 Carry-Select and Conditional Sum Addition

The fundamental idea of carry-select addition is to split the adder into two or more stages. For all stages except the stage with the least significant bits one uses two adders. It is assumed that the incoming carry bit is zero one of the adders and one for the other. Then, a multiplexer is used to select the correct result and carry to the next stage once the incoming carry is known. A two-stage carry-select adder is shown in Fig. 14. The length of the stages should be designed such that the delay of the stage is equivalent to the delay of the first stage plus the number of multiplexers that the carry signal passes through. Hence, the actual values are determined by the relative adder and multiplexer delays, as well as the fan-out of the multiplexer control signals.

For each smaller adder in the carry-select adder, it is possible to apply the same idea of splitting each smaller adder into even smaller adders. For example, each of the two k_1 bit adders can be split into two k_3 bits and one k_4 bits adders, where $k_1 = k_3 + k_4$, in a similar way. Note, however, that only four smaller adders are required instead of six as the same two k_3 bits adders can be used. If this is applied until only one-bit adders remain, we obtain a conditional sum adder. There are naturally, a wide range of intermediate adder structures based on the ideas of carry-select and conditional sum adders.

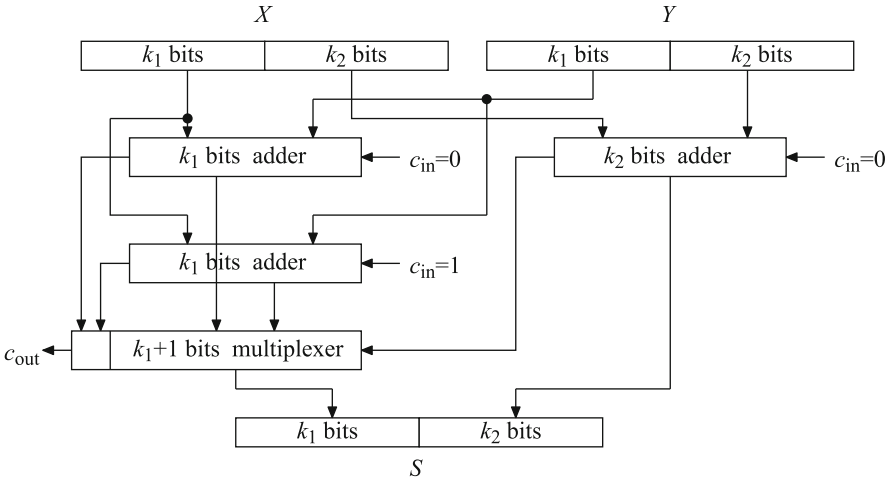


Fig. 14 Two-stage carry-select adder

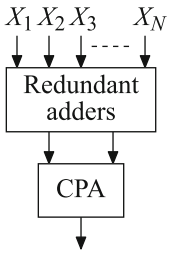


Fig. 15 Principle of a multi-operand adder

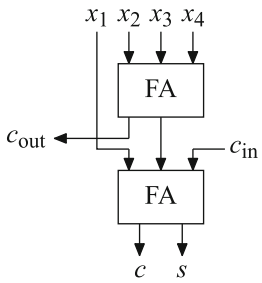


Fig. 16 4:2 compressor composed of full adders (3:2 counters)

2.4 Multi-Operand Addition

When several operands are to be added, it is beneficial to avoid several carry-propagations. Especially, when there are delay constraints it is inefficient to use several high-speed adders. Instead it is common to use a redundant intermediate representation and a fast final carry-propagation adder (CPA). The basic concept is illustrated in Fig. 15.

For performing multi-operand addition, either counters or compressors or a combination of counters and compressors can be used. A counter is a logic gate that takes a number of inputs, add them together and produce a binary representation of the output. The simplest counter is the full adder cell shown in Fig. 1b. In terms of counters, it is a 3:2 counter, e.g., it has three inputs and produces a 2-bit output word. This can be generalized to $n : k$ counters, having n inputs of the same weight and producing a k bit output corresponding to the number of ones in the input. Clearly, n and k must satisfy $n \leq 2^k - 1$ or equivalently $k \geq \lceil \log_2(n+1) \rceil$.

A compressor on the other hand does not produce a valid binary count of the number of input bits. However, it does reduce the number of partial products, but at the same time has several incoming and outgoing carries. The output carries should be generated without any dependence on the input carries. The most frequently used compressor is the 4:2 compressor shown in Fig. 16, which is realized using full adders. Clearly, there is no major advantage using 4:2 compressors that are implemented as in Fig. 16 compared to using 3:2 counters (full adders). However, other possible realizations are available. These should satisfy

$$x_1 + x_2 + x_3 + x_4 + c_{\text{in}} = s + 2c + 2c_{\text{out}} \tag{21}$$

and c_{out} should be independent of c_{in}. There exist realizations with lower logic depth compared to full adders, and, hence, the total delay of the multi-operand addition may be reduced using 4:2 compressors.

It is important to note that an n:k counter or compressor reduces the number of bits in the computation with exactly $n - k$. Hence, it is easy to estimate the required number of counters and compressors to perform any addition if the original number of bits to be added and the number of bits for the result are known. It should also be noted, that depending on the actual structure it is typically impossible to use only one type of compressors and adders. Specifically, half adders (or 2:2 counters) may sometimes be needed, despite not reducing the number of bits, to move bits to the correct weights for further additions.

3 Multiplication

The process of multiplication can be divided into three different steps: partial product generation that determines a number of bits to be added, summation of the generated partial products, and, for some of the summation structures, carry-propagation addition, usually called vector merging addition (VMA), as many summation structures produce redundant results.

3.1 Partial Product Generation

For unsigned binary representation, the partial product generation can be readily realized using AND-gates computing bit-wise multiplications as

$$Z = XY = \sum_{i=1}^{W_{fX}} x_i 2^{-i} \sum_{j=1}^{W_{fY}} y_j 2^{-j} = \sum_{i=1}^{W_{fX}} \sum_{j=1}^{W_{fY}} x_i y_j 2^{-i-j}. \tag{22}$$

This leads to a partial product array as shown in Fig. 17.

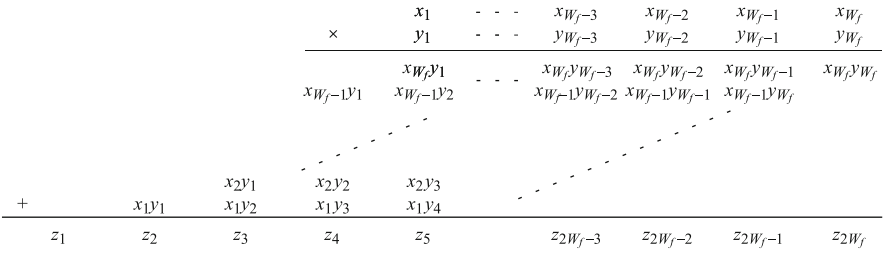


Fig. 17 Partial product array for unsigned multiplication

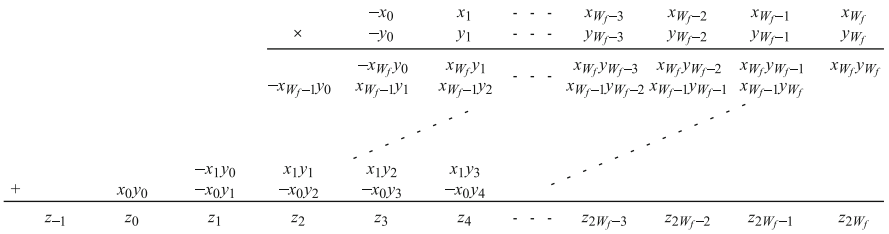


Fig. 18 Partial product array for two's complement multiplication

For two's complement data the result is very similar, except that the sign-bit causes some of the bits to have the negative sign. This can be seen from

$$\begin{aligned} Z &= XY \\ &= \left(-x_0 + \sum_{i=1}^{W_{fX}} x_i 2^{-i}\right)\left(-y_0 + \sum_{j=1}^{W_{fY}} y_j 2^{-j}\right) \\ &= x_0 y_0 - x_0 \sum_{j=1}^{W_{fY}} y_j 2^{-j} - y_0 \sum_{i=1}^{W_{fX}} x_i 2^{-i} + \sum_{i=1}^{W_{fX}} \sum_{j=1}^{W_{fY}} x_i y_j 2^{-i-j}. \end{aligned} \tag{23}$$

The corresponding partial product matrix is shown in Fig. 18.

3.1.1 Avoiding Sign-Extension

As previously stated, the word lengths of two two's complement numbers should be equal when performing the addition or subtraction. Hence, the straightforward way of dealing with the varying word lengths in two's complement multiplication is to sign-extend the partial results to obtain the same word length for all rows.

To avoid this excessive sign-extension it is possible to either perform the summation from top to bottom and perform sign-extension of the partial results to match the next row to be added. This is further elaborated in Sects. 3.2.1 and 3.2.2. However, if we want to be able to add the partial products in an arbitrary order using a multi-operand adder as discussed in Sect. 2.4, the following technique, proposed initially by Baugh and Wooley [2], can be used. Note that for a negative partial product we have $-p = \bar{p} - 1$. Hence, we can replace all negative partial products with an inverted version. Then, we need to subtract a constant value from the result, but as there will be several constants, one from each negative partial product, we can sum these up and form a single compensation vector to be added. When this is applied we get the partial product array as shown in Fig. 19.

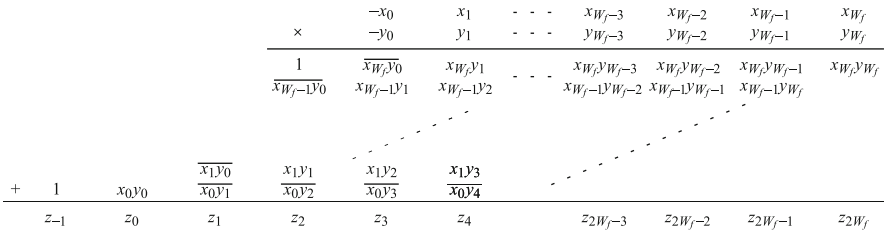


Fig. 19 Partial product array without sign-extension

Table 2 Rules for the radix-4 modified Booth encoding

x_{2k}	x_{2k+1}	x_{2k+2}	r_k	$d_{2k}d_{2k+1}$	Description
0	0	0	0	00	String of zeros
0	0	1	1	01	End of ones
0	1	0	1	01	Single one
0	1	1	2	10	End of ones
1	0	0	-2	$\bar{1}0$	Start of ones
1	0	1	-1	$0\bar{1}$	Start and end of ones ($\bar{1}0 + 01$)
1	1	0	-1	$0\bar{1}$	Start of ones
1	1	1	0	00	String of ones

3.1.2 Reducing the Number of Rows

As discussed in Sect. 1.3.1, it is possible to reduce the number of non-zero positions by using a signed-digit representation. It would be possible to use, e.g., a CSD representation to obtain a minimum number of non-zeros. However, the drawback is that the conversion from two's complement to CSD requires the carry-propagation. Furthermore, the worst case is that half of the positions are non-zero, and, hence, one would still need to design the multiplier to deal with this case.

Instead, it is possible to derive a signed-digit representation that is not necessarily minimum but has at most half of the positions being non-zero. This is referred to modified Booth encoding [33] and is often described as being a radix-4 signed-digit representation where the recoded digits $r_i \in \{-2, -1, 0, 1, 2\}$. An alternative interpretation is a radix-2 signed-digit representation where $d_i d_{i-1}$, $i \in \{W_f, W_{f-2}, W_{f-4}, \dots\}$. The logic rules for performing the modified Booth encoding are based on the idea of finding strings of ones and replace them as $011\dots11 = 100\dots0\bar{1}$ and are illustrated in Table 2. From this, one can see that there is at most one non-zero digit in each pair of digits ($d_{2k}d_{2k+1}$).

Now, to perform the multiplication, we must be able to possibly negate and multiply the operand with 0, 1, or 2. This can conceptually be performed as in Fig. 20. As discussed earlier, the negation is typically performed by inverting the bits and add a one in the column corresponding to the LSB position. The partial product array for a multiplier using the modified Booth encoding is shown in Fig. 21.

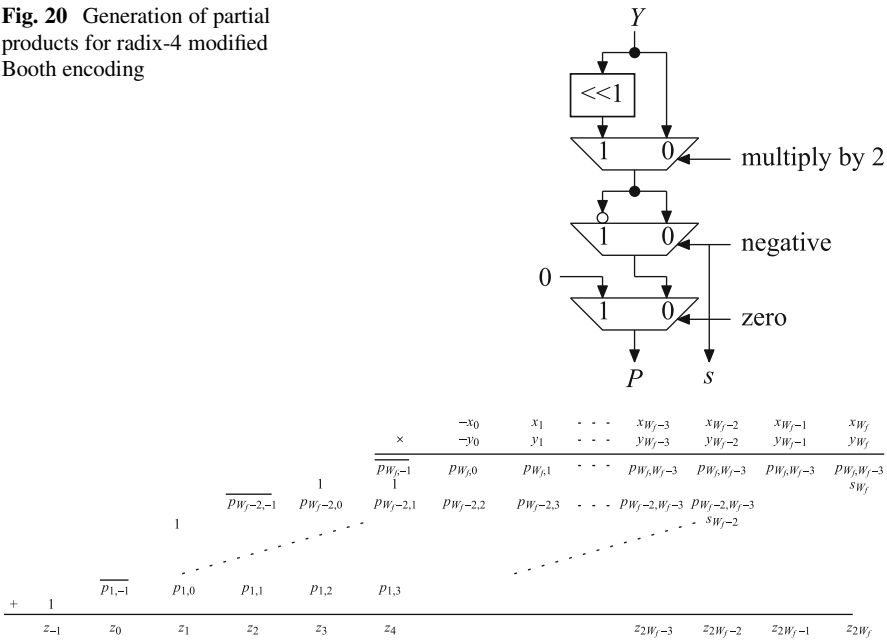


Fig. 20 Generation of partial products for radix-4 modified Booth encoding

Fig. 21 Partial product array for radix-4 modified Booth encoded multiplier

It is possible to use the modified Booth encoding with higher radices than two. However, that requires the computations of non-trivial multiples such as 3 for radix-8 and 3, 5, and 7 for radix-16. The number of rows is reduced roughly by a factor of k for the radix-2^k modified Booth encoding.

3.1.3 Reducing the Number of Columns

It is common that the results after the multiplication are quantized to be represented with fewer bits than the original result. To reduce the complexity of the multiplication in these cases it has been proposed to perform the quantization at the partial product stage [29]. This is commonly referred to as fixed-width multiplication referring to the fact that (most of) the partial products rows have the same width.

Simply truncating the partial products will result in a rather large error. Several methods have, therefore, been proposed to compensate for the introduced error [43].

3.2 Summation Structures

The problem of summing up the partial products can be solved in three general ways; sequential accumulation where a subset of the partial products are accumulated in each cycle, array accumulation which gives a regular structure, and tree accumulation which gives the smallest logic depth but in general an irregular structure.

3.2.1 Sequential Accumulation

In so-called add-and-shift multipliers, the partial bit-products are generated sequentially and successively accumulated as generated. Therefore, this type of multiplier is slow as it requires multiple cycles, but the required chip area is small. The accumulation can be done using any of the bit-parallel adders discussed above or using digit-serial or bit-serial accumulators. A major advantage of bit-serial over bit-parallel arithmetic is that it significantly reduces chip area. This is done in two ways. First, it eliminates wide buses and simplifies the wire routing. Second, by using small processing elements, the chip itself will be smaller and will require shorter wiring. A small chip can support higher clock frequencies and is, therefore, faster. Two's complement representation is suitable for DSP algorithms implemented with bit-serial arithmetic, since the bit-serial operations then can be done without knowing the sign of the numbers involved. Figure 22 shows a 5-bit serial/parallel multiplier, where the bit-products are generated row-wise. In a serial/parallel multiplier, the multiplicand X arrive bit-serially while the multiplier a is applied in a bit-parallel format. Many different schemes for bit-serial multipliers have been proposed. They differ mainly in which order bit-products are generated and accumulated and in the way subtraction is handled.

Addition of the first set of partial bit-products starts with the products corresponding to the LSB of X. Thus, in the first time slot, at bit x_{W_f}, we simply add, $a \times x_{W_f}$ to the initially cleared accumulator. Next, the D flip-flops are clocked

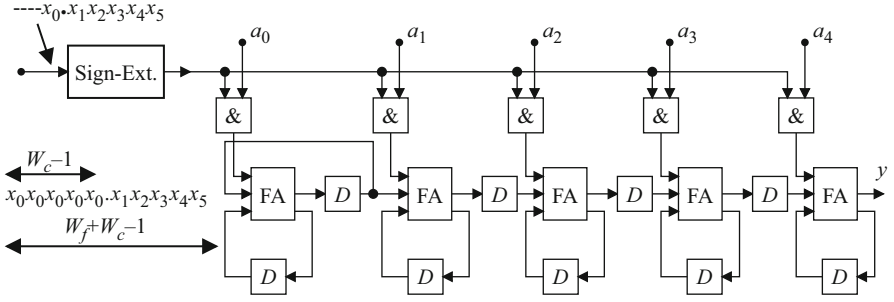


Fig. 22 Serial/parallel multiplier based on carry−save adders

and the sum-bits from the FAs are shifted one bit to the right, each carry-bit is saved and added to the full-adder in the same stage, the sign-bit is copied, and one bit of the product is produced at the output of the accumulator. These operations correspond to multiplying the accumulator contents by 2^{-1}. In the following clock cycle, the next bit of X is used to form the next set of bit-products, which are added to the value in the accumulator, and the value in the accumulator is again divided by 2. This process continues for W_f clock cycles, until the sign bit x_0 is reached, whereupon a subtraction must be done instead of an addition. During the first W_f clock cycles, the least significant part of the product is computed and the most significant is stored in the D flip-flops. In the next W_f clock cycles, zeros are, therefore, applied to the input so that the most significant part of the product is shifted out of the multiplier. Note that the accumulation of the bit-products is performed using a redundant representation, which is converted to a non-redundant representation in the last stage of the multiplier. A digit-serial multiplier, which accumulate several bits in each stage, can be obtained either via unfolding of a bit-serial multiplier or via folding of a bit-parallel multiplier.

3.2.2 Array Accumulation

Array multipliers use an array of almost identical cells for generation and accumulation of the bit-products. Figure 23 shows a realization of the Baugh-Wooley multiplier [2] with the multiplication time proportional to $2W_f$.

3.2.3 Tree Accumulation

The array structure provides a regular structure, but at the same time the delay grows linearly with the word length. Considering Figs. 19 and 21, they both provide a number of partial products that should be accumulated. As mentioned earlier, it is common to accumulate the partial products such that there are at most two partial products of each weight and then use a fast carry-propagation adder to perform the final step.

In Sect. 2.4, the problem of adding a number of bits was considered. Here, we will focus on structures using full and half adders (or 3:2 and 2:2 counters), although there are other structures proposed using different types of counters and compressors.

The first approach is to add as many full adders as possible to reduce as many partial products as possible. Then, we add as many half adders as possible to minimize the number of levels and try to shorten the word length for the vector merging adder. This approach is roughly the Wallace tree proposed in [54]. The main drawback of this approach is an excessive use of half adders. Dadda [8] instead proposed that full and half adders should only be used if required to obtain a number of partial products equal to a value in the Dadda series. The value of position n in the Dadda series is the maximum number of partial products that can be reduced

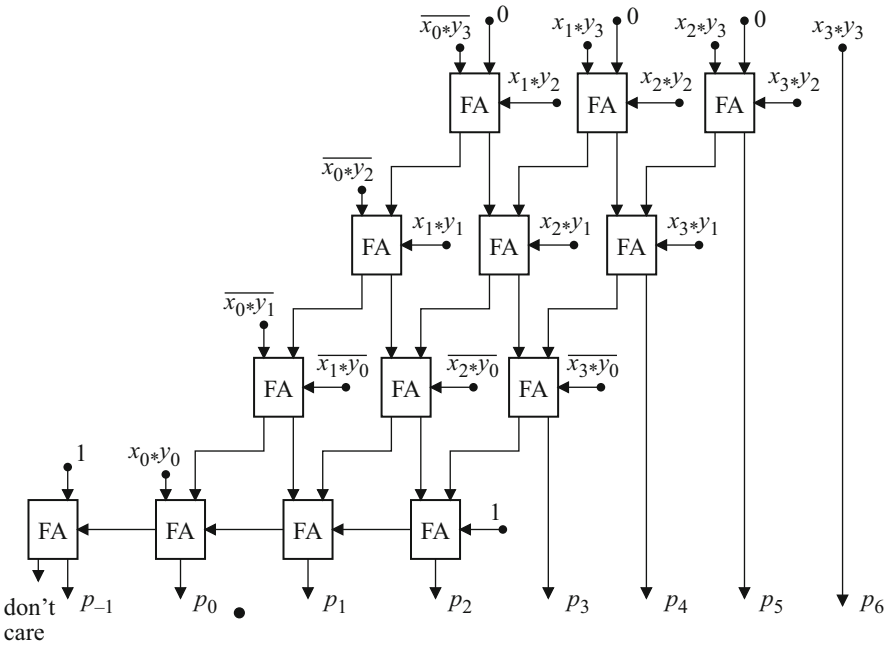


Fig. 23 A Baugh-Wooley multiplier

using n levels of full adders. The Dadda series starts $\{3, 4, 6, 9, 13, 19, \ldots\}$. The benefit of this is that the number of half adders is significantly reduced while still obtaining a minimum number of levels. However, the length of the vector merging adder increases. A compromise between these two approaches is the Reduced Area heuristic [3], where similarly to the Wallace tree, as many full adders as possible are introduced in each level. Half adders are on the other hand only introduced if required to reach a number of partial products corresponding to a value in the Dadda series or if the least significant weight with more than one partial products is represented with exactly two partial products. In this way, a minimum number of stages is obtained, while at the same time both the length of the vector merging adder and the number of half adders is kept small.

To illustrate the operation of the reduction tree approaches we use dot diagrams, where each dot corresponds to a bit (partial product) to be added. Bits with the same weight are in the same column and bits in adjacent columns have one position higher or lower weight, with higher weights to the left. The bits are manipulated by either full or half adders. The operation of these are illustrated in Fig. 24.

The reduction schemes are exemplified based on an unsigned 6×6-bits multiplication in Fig. 25. The complexity results are summarized in Table 3. It should be noted that the positioning of the results in the next level is done based on ease of illustration. From a functional point of view this step is arbitrary, but it is possible to optimize the timing by carefully utilizing different delays of the sum and

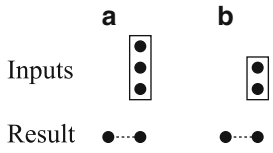


Fig. 24 Operation on bits in a dot diagram with (**a**) full adder and (**b**) half adder

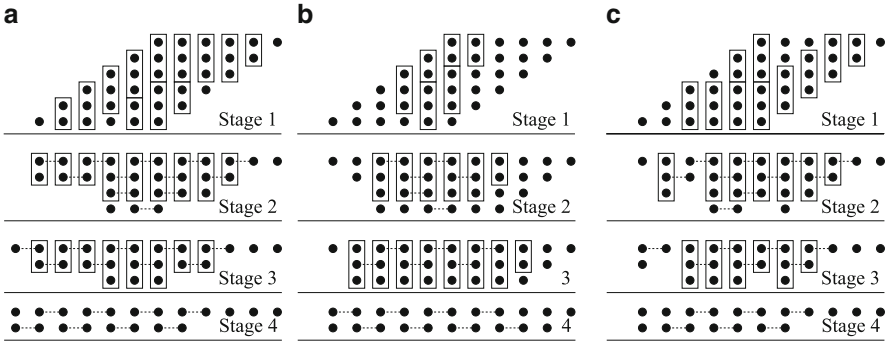


Fig. 25 Reduction trees for a 6 × 6-bits multiplier: (**a**) Wallace [54], (**b**) Dadda [8], and (**c**) Reduced area [3]

Table 3 Complexity of the three reduction trees in Fig. 25

Tree structure	Full adders	Half adders	VMA length
Wallace [54]	16	13	8
Dadda [8]	15	5	10
Reduced area [3]	18	5	7

carry outputs of the adder cells [39]. Furthermore, it is possible to reduce the power consumption by optimizing the interconnect ordering [41].

The reduction trees in Fig. 25 does not provide any regularity. This means that the routing is complicated and may become the limiting factor in an implementation. Reduction structures that provide a more regular routing, but still a small number of stages, include the Overturned stairs reduction tree [35] and the HPM tree [13].

3.3 Vector Merging Adder

The role of the vector merging adder is to add the outputs of the reduction tree. In general, any carry-propagation adder can be used, e.g., those presented in Sect. 2. However, the different input signals to the adders will typically be available at different delays from the multiplier input values. Therefore, it is possible to derive carry-propagation adders that utilize the different signal arrival times to optimize the adder delay [49].

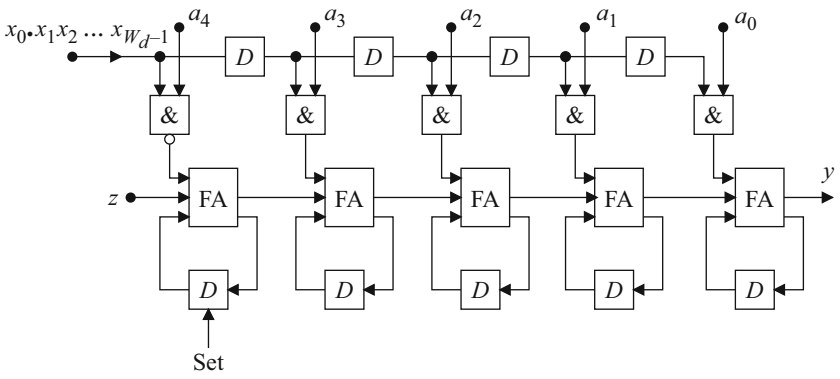


Fig. 26 Serial/parallel multiplier-accumulator

3.4 Multiply-Accumulate

In many DSP algorithms, computations of the form $Z = XY + A$ are common. These can be efficiently implemented by simply adding another row corresponding to A in the partial product array. In many cases, this will not increase the number of levels required.

For sequential operation, the modification of the first stage of the serial/parallel multiplier as shown in Fig. 26, makes it possible to add an input Z to be added to the product at the same level of significance as X.

3.5 Multiplication by Constants

When the multiplier coefficient is known, it is possible to reduce the complexity of the corresponding circuit. First of all, no circuitry is required to generate the partial products. Second, there will in general be fewer partial products to add. This can easily be realized considering the partial product array in Fig. 17. For the coefficient bits that are zero, all the corresponding partial product bits will also be zero, and, hence, these are not required to be added. To obtain more zero positions, the use of a minimum signed digit representation such as CSD is useful.

It is also possible to utilize potential redundancy in the computations to further reduce the complexity. How this is done in detail depends on which type of addition is assumed to the basic operation. As both addition and subtraction have the same complexity, we will refer to both as the addition. In the following, we will assume carry-propagation addition, i.e., two input and one output, realized in any way discussed in Sect. 2. Furthermore, for ease of exposition, we will assume that the standard sign-extension is used. For carry-save addition we refer to [19].

Consider a signed-digit representation of a multiplier coefficient X such as shown in (1) with $x_i \in \{-1, 0, 1\}$. Each non-zero position will produce a partial result row

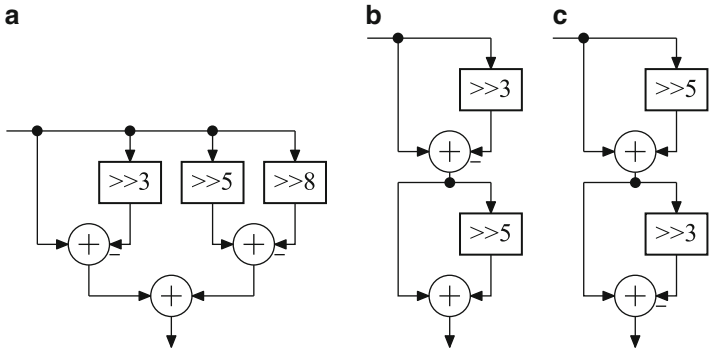


Fig. 27 Constant multiplication with $231/256 = 1.00\bar{1}0100\bar{1}$ using (**a**) no sharing, (**b**) sharing of the subexpression $100\bar{1}$, and (**c**) sharing of the subexpression 10001

and these partial result rows can be added in an arbitrary order. Now, if the same pattern of non-zero positions, called subexpression, occurs in more than one position of the representation, we only need to compute the corresponding partial result once and use it for all the instances where it is required. Figure 27a, b show examples of multiplication with the constant $231/256 = 1.00\bar{1}0100\bar{1}$ with and without utilizing redundancy, respectively. In this case, the subexpression $100\bar{1}$ is extracted, but we might just as well have chosen 10001 and subtracted one of the subexpressions as shown in Fig. 27c.

This can be performed is a systematic way as described below. However, first we note that if we multiply the same data with several constant coefficients, as in a transposed direct form FIR filter [57], the different coefficients can share subexpressions. Hence, the systematic way is as follows [21, 44]:

1. Represent the coefficients in a given representation.
2. For each coefficient find and count possible subexpressions. A subexpression is characterized by the difference in non-zero position and if the non-zeros have the same or opposite signs.
3. If there are common subexpressions, select one to replace and replace instances of it by introducing a new symbol in place of the subexpression. The most common approach is to select the most frequent subexpression, thus, applying a greedy optimization approach, and replace all instances of it. However, it should be noted that from a global optimality point of view this is not always the best.
4. If there were subexpressions replaced, go to Step 2 otherwise the algorithm is done.

There is a number of sources of possible sub-optimality in the procedure described above. The first is that the results are representation dependent, and, hence, it will in general reduce the complexity trying several different representations. It may seem to make sense to use an MSD representation as it originally have few non-zero positions. However, the more non-zeros the more likely is it that

common subexpressions will exist. For the single constant multiplication case this has been utilized for a systematic algorithm that searches all representations with the minimum number of non-zero digits plus k additional non-zero digits [17]. The second source is the selection of subexpression to replace. It is common to select the most frequent one, applying a greedy strategy, and replace all instances. However, it has been shown that from a global optimization point of view, it is not always beneficial to replace all subexpressions. Another issue is which subexpression to choose if there are more than one that are as frequent.

For single constant coefficients, an optimal approach based on searching all the possible interconnections of a given number of adders is presented in [17]. It is shown that multiplication with all coefficients up to 19 bits can be realized using at most five additions, compared to up to nine additions using a straightforward CSD realization without sharing. The optimal approach avoids the issue of representation dependence by only considering the decimal value at each addition, independent of underlying representation. For the multiple constant multiplication case several effective algorithms have been proposed over the years that avoids the problem of representation dependence [15, 53]. Theoretical lower bounds for related problems have been presented in [16].

3.6 Distributed Arithmetic

Distributed arithmetic is an efficient scheme for computing inner products of a fixed and a variable data vector

$$Y = a^T X = \sum_{i=1}^{N} a_i X_i. \tag{24}$$

The basic principle is owed to Croisier et al. [7]. The inner product can be rewritten using two's complement representation

$$Y = \sum_{i=1}^{N} a_i \left[-x_{i0} + \sum_{k=1}^{Wf} x_{ik} 2^{-k} \right], \tag{25}$$

where x_{ik} is the kth bit in x_i. By interchanging the order of the two summations we get

$$Y = -\sum_{i=1}^{N} a_i x_{i0} + \sum_{k=1}^{Wf} \left[\sum_{i=1}^{N} a_i x_{ik} \right] 2^{-k} \tag{26}$$

Table 4 Distributed arithmetic look-up table for $a_1 = (0.0100001)_{2C}$, $a_2 = (0.1010101)_{2C}$, and $a_3 = (1.1110101)_{2C}$

x_1	x_2	x_3	F_k	F_k
0	0	0	0	$(0.0000000)_{2C}$
0	0	1	a_3	$(1.1110101)_{2C}$
0	1	0	a_2	$(0.1010101)_{2C}$
0	1	1	$a_2 + a_3$	$(0.1001010)_{2C}$
1	0	0	a_1	$(0.0100001)_{2C}$
1	0	1	$a_1 + a_3$	$(0.0010110)_{2C}$
1	1	0	$a_1 + a_2$	$(0.1110110)_{2C}$
1	1	1	$a_1 + a_2 + a_3$	$(0.1101011)_{2C}$

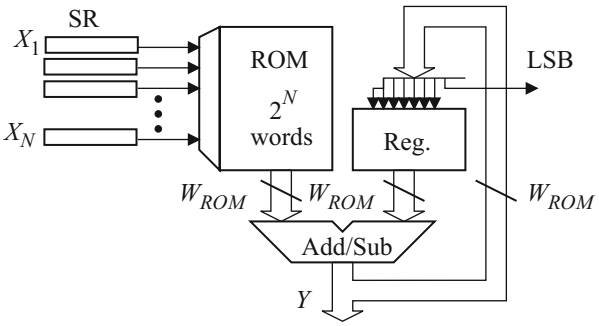


Fig. 28 Block diagram for distributed arithmetic

$$= -F_0(x_{10}, x_{20}, \ldots\ldots, x_{N0}) + \sum_{k=1}^{Wf} F_k(x_{1k}, x_{2k}, \ldots, x_{Nk}) 2^{-k}, \qquad (27)$$

where

$$F_k(x_{1k}, x_{2k}, \ldots, x_{Nk}) = \sum_{i=1}^{N} a_i x_{ik}. \qquad (28)$$

F_k is a function of N binary variables, ith variable being the kth bit in x_i. Since F_k can take on only 2^N values, it can be precomputed and stored in a look-up table. For example, consider the inner product $Y = a_1X_1 + a_2X_2 + a_3X_3$ where $a_1 = (0.0100001)_{2C}$, $a_2 = (0.1010101)_{2C}$, and $a_3 = (1.1110101)_{2C}$. Table 4 shows the function F_k and the corresponding addresses.

Figure 28 shows a realization of (27) by Horner's method

$$y = \left(\left(\ldots\left(\left(0 + F_{Wf}\right)2^{-1} + \ldots + F_2\right)2^{-1} + F_1\right)2^{-1} - F_0\right). \qquad (29)$$

The inputs, $X_1, X_2, \ldots\ldots X_N$, are shifted bit-serially out from the shift registers with the least-significant bit first. Bits x_{ik} are used to address the look-up table.

Since, the output is divided by 2, by the inherent shift, the circuit is called a shift-accumulator [57]. Computation of the inner product requires $W_f + 1$ clock cycles. In the last cycle, F_0 is subtracted from the accumulator register. Notice the resemblance with a shift-and-add implementation of a real multiplication.

A more parallel form of distributed arithmetic can also be realized by allocating several tables. The tables, which are identical, may be addressed in parallel and their appropriately shifted values.

3.6.1 Reducing the Memory Size

The memory requirement becomes very large for long inner products. There are mainly two ways to reduce the memory requirements. One of several possible ways to reduce the overall memory requirement is to partition the memory into smaller pieces that are added before the shift-accumulator, as shown in Fig. 29. The amount of memory is in this case reduced from 2^N words to $2 \times 2^{N/2}$ words. For example, for $N = 10$ we get $2^{10} = 1024$ words, which is reduced to only $2 \times 2^5 = 64$ words at the expense of an additional adders.

Memory size can be halved by using the ingenious scheme[7] based on the identity $X = \frac{1}{2}[X - (-X)]$, which can be rewritten

$$X = -(x_0 - \bar{x}_0)\, 2^{-1} + \sum_{k=1}^{Wf} (x_k - \bar{x}_k)\, 2^{-k-1} - 2^{-(W_f+1)}. \tag{30}$$

Note that $(x_k - \bar{x}_k)$ can only take on the values -1 or $+1$. Inserting this expression into (24) yields

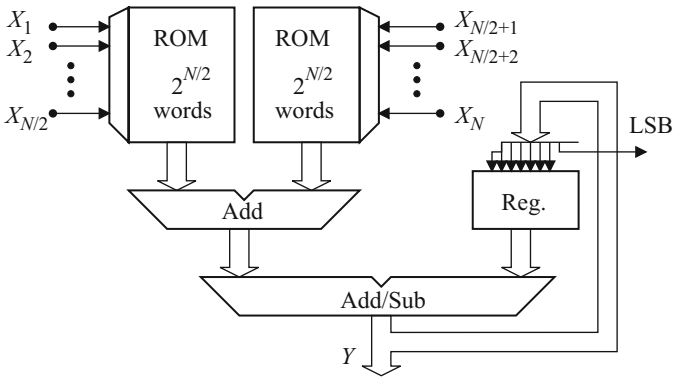


Fig. 29 Reducing the memory by partitioning

Table 5 Look-up table contents using half-sized memory

x_1	x_2	x_3	F_k	u_1	u_2	A/S
0	0	0	$-a_1 - a_2 - a_3$	0	0	A
0	0	1	$-a_1 - a_2 + a_3$	0	1	A
0	1	0	$-a_1 + a_2 - a_3$	1	0	A
0	1	1	$-a_1 + a_2 + a_3$	1	1	A
1	0	0	$+a_1 - a_2 - a_3$	1	1	S
1	0	1	$+a_1 - a_2 + a_3$	1	0	S
1	1	0	$+a_1 + a_2 - a_3$	0	1	S
1	1	1	$+a_1 + a_2 + a_3$	0	0	S

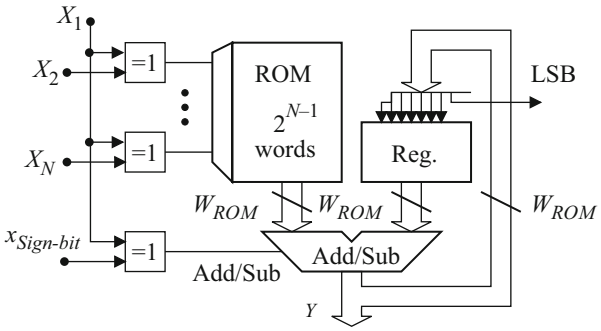


Fig. 30 Distributed arithmetic with half-sized memory

$$Y = \sum_{k=1}^{Wf} F_k(x_{1k}, \ldots, x_{Nk}) 2^{-k-1} - F_0(x_{10}, \ldots, x_N) 2^{-1} + F(0, \ldots, 0) 2^{-(W_f+1)}, \tag{31}$$

where

$$F_k(x_{1k}, x_{2k} \ldots\ldots., x_{Nk}) = \sum_{i=1}^{N} a_i \, (x_{ik} - \bar{x}_{ik}) \, . \tag{32}$$

The function F_k is shown in Table 5 for $N = 3$. Notice that only half the values are needed, since the other half can be obtained by changing the signs. To explore this redundancy we make the following address modification $u_1 = x_1 \oplus x_2$, $u_2 = x_1 \oplus x_3$, and $A/S = x_1 \oplus s_{\text{sign}-\text{bit}}$ where X_1 has been selected as control variable [57]. The control signal $x_{\text{sign}-\text{bit}}$ is zero at all times except when the sign bit of the inputs arrives. Figure 30 shows the resulting realization with halved look-up table. The XOR-gates used for halving the memory can be merged with the XOR-gates that are needed for inverting F_k.

3.6.2 Complex Multipliers

A complex multiplication requires three or four real multiplications and some additions but only two distributed arithmetic units, which from area, speed, and power consumption points of view are comparable to the real multiplier. Let $X = A + jB$ and $K = c + jd$ where K is the fixed coefficient and X is a variable. Now, the product of the two complex numbers can be written as

$$\begin{aligned}
KX &= (cA - dB) + j(dA + cB) \\
&= \left\{ -c(a_0 - \bar{a}_0)2^{-1} + \sum_{k=1}^{W_f} c(a_k - \bar{a}_k)2^{-k-1} - c2^{-(W_f+1)} \right\} \\
&\quad - \left\{ -d(b_0 - \bar{b}_0)2^{-1} + \sum_{k=1}^{W_f} d(b_k - \bar{b}_k)2^{-k-1} - d2^{-(W_f+1)} \right\} \\
&\quad + j \left\{ -d(a_0 - \bar{a}_0)2^{-1} \sum_{k=1}^{W_f} d(a_k - \bar{a}_k)2^{-k-1} - d2^{-(W_f+1)} \right\} \\
&\quad + j \left\{ -c(b_k - \bar{b}_k)2^{-1} + \sum_{k=1}^{W_f} c(b_k - \bar{b}_k)2^{-k-1} - c2^{-(W_f+1)} \right\} \\
&= F_1(a_0, b_0)2^{-1} + \sum_{k=1}^{W_f} F_1(a_k, b_k)2^{-k-1} + F_1(0,0)2^{-(W_f+1)} \\
&\quad + j \left\{ F_2(a_0, b_0)2^{-1} + \sum_{k=1}^{W_f} F_2(a_k, b_k)2^{-k-1} + F_2(0,0)2^{-(W_f+1)} \right\}.
\end{aligned}$$

Hence, the real and imaginary parts of the product can be computed using just two distributed arithmetic units. The content of the look-up table that stores F_1 and F_2 is shown in Table 6.

Obviously only two coefficients are needed, $(c + d)$ and $(c - d)$. If $a_j \oplus b_j = 1$, the F coefficients values are applied directly to the accumulators, and if $a_j \oplus b_j = 0$,

Table 6 ROM contents for a complex multiplier based on distributed arithmetic

a_i	b_i	F_1	F_2
0	0	$-(c-d)$	$-(c+d)$
0	1	$-(c+d)$	$(c-d)$
1	0	$(c+d)$	$-(c-d)$
1	1	$(c-d)$	$(c+d)$

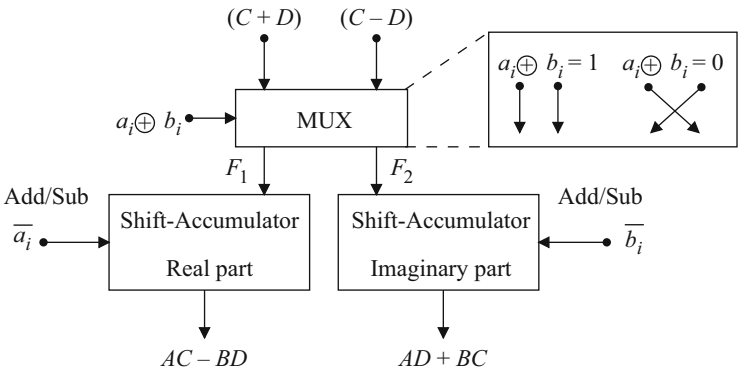


Fig. 31 Block diagram for a complex multiplier based on distributed arithmetic

the F coefficients values are interchanged and added or subtracted depending on the data bits a_k and b_k. The realization is shown in Fig. 31.

4 Division

Of the four basic arithmetic operations, the division is the most complex to compute. Furthermore, the result of a division consists of two components, the quotient, Z, and the remainder, R, such that

$$X = ZD + R, \tag{33}$$

where X is the dividend, $D \neq 0$ is the divisor, and $|R| < D$. By definition the sign of the remainder should be the same as that of the dividend. For the result to be a fractional number we must have $\left|\frac{X}{D}\right| \leq 1$. This can always be obtained by shifting the dividend and/or divisor. For ease of exposition we will initially start with unsigned data, but eventually introduce signed data. For further information on the methods presented here and others, we refer to [11].

4.1 Restoring and Nonrestoring Division

The simplest way to perform a division is to sequentially shift the dividend one position (multiply by two) and then check if the divisor has larger magnitude than the dividend. If so, the corresponding magnitude bit of the quotient is one and we subtract the divisor from the dividend. Conceptually, the comparison can be made by first subtracting the divisor from the dividend and then check if the result is positive

(quotient bit is one) or negative (quotient bit is zero). If the result is negative we need to add the divisor again, which gives the name *restoring division*.

The computation in step i can be written as

$$r_i = 2r_{i-1} - z_i D, \tag{34}$$

where $r_0 = X$. Therefore, if $2r_{i-1} - D$ is positive, we set $z_i = 1$, otherwise $z_i = 0$ and $r_i = 2r_{i-1}$. r_i is the remainder after iteration i and considering (33) we have $R = r_i 2^{-1}$. To compute a quotient with W_f fractional bits obviously W_f iterations of (34) are required.

Instead of restoring the remainder by adding the divisor, we can assign a negative quotient digit. This gives the *nonrestoring division* selection rule of the quotient digits, z_i, in (34) as

$$z_i = \begin{cases} 1, & r_{i-1} D \geq 0 \text{ i.e. same sign} \\ -1, & r_{i-1} D < 0 \text{ i.e. different signs.} \end{cases} \tag{35}$$

Note that with this definition of the selection rules the remainder will sometimes be positive, sometimes negative. Hence, division with a signed dividend and/or divisor is well covered within this approach. This also gives that the final remainder does not always have the same sign as the dividend. Hence, in that case we must compensate by adding or subtracting D to R and consequently subtracting or adding one LSB to Z.

The result from the nonrestoring division will be represented using a representation with $q_i \in \{-1, 1\}$. This representation is sometimes called *nega-binary* and is in fact not a redundant representation. The result should in most cases be converted into a two's complement representation. Naturally, one can convert this by forming a word with positive bits and one with negative bits and subtract the negative bits from the positive bits. However, for this all bits must be computed before the conversion can be done. Instead, it is possible to use the on-the-fly conversion technique in [10] to convert the digits into bits once they are computed.

Another consequence of the nega-binary representation is that if a zero remainder is obtained, this will not remain zero in the succeeding stages. Hence, a zero remainder should be detected and either the iterations stopped or corrected for at the end.

4.2 SRT Division

The *SRT* division scheme extends the nonrestoring scheme by allowing 0 as a quotient digit. Furthermore, by restricting the dividend to be in the range $1/2 \leq D < 1$, which can be obtained by shifting, the selection rule for the quotient digit in (34) can be defined as

$$z_i = \begin{cases} -1, & 2r_{i-1} < -1/2 \\ 0, & -1/2 \leq 2r_{i-1} < 1/2 \\ 1, & 1/2 \leq 2r_{i-1} \end{cases} \tag{36}$$

for the binary case. This has two main advantages: firstly, when $z_i = 0$ there is no need to add or subtract; secondly, comparing with $1/2$ or $-1/2$ only requires three bits of $2r_{i-1}$. There exists slightly improved selection rules that further reduce the number of additions/subtractions. However, the number of iterations are still W_f for a quotient with W_f fractional bits.

4.3 Speeding Up Division

While the number of additions/subtractions are reduced in the SRT scheme it would require an asynchronous circuit to improve the speed. In many situations, this is not wanted. Instead, to reduce the number of cycles one can use division with a higher radix. Using radix $b = 2^m$ reduces the number of iterations to $\left\lceil \frac{W_f}{m} \right\rceil$. The iteration is now

$$r_i = br_{i-1} - z_i D, \tag{37}$$

where $z_i \in \{0, 1, \ldots, b-1\}$ for restoring division. For SRT division $z_i \in \{-a, -a+1, \ldots, -1, 0, 1, \ldots, a\}$, where $\lceil (b-1)/2 \rceil \leq a \leq (b-1)$. The selection rules can be defined in several different ways similar to radix-2 discussed earlier. We can guarantee convergence by selecting the quotient digit such that $|r_i| < D$, which typically implies maximizing the magnitude of the quotient digit.

For SRT division it is possible to select the redundancy of the representation based on a. Higher redundancy leads to a larger overlap in the regions where one can select any of two different quotient digits. Having an overlap means that one can select the breakpoint such that few bits of r_i and D need to be compared. However, a higher redundancy means that there are more multiples of D that needs to be computed for the comparison. Hence, there is a trade-off between the number of bits that needs to be compared and the precomputations for the comparisons.

Even though higher-radix division reduces the number of iteration, each iteration still needs to be performed sequentially. In each step, the current remainder must be known and the quotient digit selected before it is possible to start a new step. There are two different ways to overcome this limitation. First, it is possible to overlap the complete computation of the partial remainder in step i and the selection of the quotient digit in step $i+1$. This is possible since not all bits of the remainder must be known to select the next quotient digit. Second, the remainder can be computed in a redundant number system.

4.4 Square Root Extraction

Computing the square root is in some ways a similar operation to division as one can sequentially iterate the remainder, initialized to the radicand, $r_0 = X$, with the partially computed square root $Z_i = \sum_{k=1}^{i} z_k 2^{-k}$ in a similar way as for division. More precisely, the iteration for square root extraction is

$$r_i = 2_r i - 1 - z_i \left(2z_i + z_i 2^{-i}\right), \tag{38}$$

where $Z_0 = 0$.

Schemes similar restoring, nonrestoring, and SRT division can be defined. For the quotient digit selection scheme similar to SRT division the square root is restricted to $1/2 \leq Z < 1$, which corresponds to $1/4 \leq X < 1$. The selection rule is then

$$z_i = \begin{cases} 1 & 1/2 \leq r_{i-1} < 2 \\ 0 & -1/2 < r_{i-1} < 1/2 \\ -1 & -2 \leq r_{i-1} \leq -2. \end{cases} \tag{39}$$

5 Floating-Point Representation

Floating-point numbers consists of two parts, the mantissa (or significand), M, and the exponent (or characteristic), E, with a number, X, represented as

$$X = Mb^E, \tag{40}$$

where b is the base of the exponent. For ease of exposition we assume $b = 2$. With floating-point numbers we obtain a larger dynamic range, but at the same time a lower precision compared to a fixed-point number system using the same number of bits.

Both the exponent and the mantissa are typically signed integer or fractional numbers. However, their representation are often not two's complement. For the mantissa it is common to use a sign-magnitude representation, i.e., use a separate sign-bit, S, and represent an unsigned mantissa magnitude with the remaining bits. For the exponent it is common to use excess-k, i.e., add k to the exponent to obtain an unsigned number.

5.1 Normalized Representations

A general floating-point representation is redundant since

$$M2^E = \frac{M}{2}2^{E+1}. \tag{41}$$

However, to use as much as possible of the dynamic range provided by the mantissa, we would like to use the representation without any leading zeros. This representation is called the *normalized* form.

Another benefit of normalized representations is that comparison is simpler. It is possible to just compare the exponents, and only if the exponents are the same, the mantissas must be compared. Also, as it is known there are no leading zeros, the first one in the representation is made explicit, and, hence, effectively add a bit to the representation.

5.2 IEEE Standard for Floating-Point Arithmetic, IEEE 754

Before the emergence of the IEEE 754 floating-point standard, typically different computer systems had different floating-point standards making the transportation of binary data between different systems difficult. Nowadays, while some computer systems have their own floating-point representations, most have converged to the IEEE 754 standard. The most recent installment was released in August 2008 [1].

The IEEE 754-2008 standard defines three binary and two decimal basic interchange formats, where we will focus on the 32-bit binary format, called binary32.

The binary32 format has a sign bit, eight exponent bits using excess-127 representation, and 23 bits for the mantissa plus a hidden leading one. The representation can be visualized as

$$\underbrace{s}_{\text{Sign}}\ \underbrace{e_{-7}e_{-6}e_{-5}e_{-4}e_{-3}e_{-2}e_{-1}e_0}_{E,\ 8-\text{bit biased exponent}}\ \underbrace{f_1 f_2 f_3 f_4 f_5 f_6 \ldots f_{22} f_{23}}_{F,\ 23-\text{bit unsigned fraction}}.$$

The value of the floating-point number is given by

$$X = (-1)^s\ 1.F\ 2^{E-127}. \tag{42}$$

Note the hidden one due to the normalized number system, so $M = 1.F$. This means that the actual mantissa value will be in the range $1 \leq M < 2 - 2^{-23}$. Out of the 256 possible values for the exponent, two have special meanings to deal with zero value, $\pm\infty$, and undefined results (Not-a-Number, NaN). This is outlined in Table 7.

Table 7 Special cases for the exponent in binary32

	$F = 0$	$F \neq 0$
$E = 0$	0	Denormalized
$E = 255$	$\pm\infty$	NaN

Table 8 The four smallest binary formats in IEEE 754-2008

Property	binary16	binary32	binary64	binary128
Total bits	16	32	64	128
Mantissa bits	10 + 1	23 + 1	52 + 1	112 + 1
Exponent bits	5	8	11	15
Bias	15	127	1023	16,383

The denormalized numbers are used to extend the dynamic range as the hidden one otherwise limits the smallest positive number to $2^{1-127} = 2^{-126}$. A denormalized number has a value of

$$X = (-1)^s \, 0.F \, 2^{-126}. \tag{43}$$

Using denormalized numbers it is possible to represent $2^{-23}2^{-126} = 2^{-149}$. However, the implementation cost of denormalized numbers are high, and, hence, are not always included.

There is also an extended format defined that is used for intermediate results in certain complex functions. The extended binary32 format uses 11 bits for the exponent and at least 32 bits for the mantissa (now without a hidden bit).

In Table 8, the main parameters of the binary floating-point formats up to 128 bits are outlined. binary32, binary64, and binary128 are the three basic binary formats. A conforming implementation must fully implement as least one of the basic formats.

5.3 Addition and Subtraction

Adding and subtracting floating-point values require that both operands have the same exponent. Hence, we have to shift the mantissa of the smaller operand as in (41) such that both exponents are the same. Then, assuming binary32 and $E_X \geq E_Y$, it is possible to factor out the exponent term as

$$Z = (-1)^{s_Z} M_Z 2^{E_Z - 127} = X \pm Y = \left((-1)^{s_X} \; M_X \pm (-1)^{s_Y} \; M_Y \, 2^{-(E_X - E_Y)} \right) 2^{E_X - 127}, \tag{44}$$

where we can identify

$$(-1)^{s_Z} \; \hat{M}_Z = (-1)^{s_X} \; M_X \pm (-1)^{s_Y} \; M_Y \, 2^{-(E_X - E_Y)} \tag{45}$$

and

$$\hat{E}_Z = E_X. \tag{46}$$

Depending on the operation required and the sign of the two operand, either a subtraction or an addition of the mantissas are performed. If the effective operation is an addition, we have $1 \leq \hat{M}_Z < 4$, which means that we may need to right-shift once to obtain the normalized mantissa, M_Z, and at the same time increase $\hat{E}_Z$ by one to obtain E_Z. If the effective operation is a subtraction, the result is $0 \leq |\hat{M}_Z| < 2$. For this case we might have to right-shift to obtain the normalized number, M_Z, and correspondingly decrease the exponent to obtain E_Z.

It should be noted that adding or subtracting sign-magnitude numbers is more complex compared to adding or subtracting two's complement numbers as one will have to make decisions based on the sign and the magnitude of the operators to determine which the effective operation to be performed is. In addition, in the case of subtraction, one needs to either determine which is the largest magnitude and subtract the smaller from the larger or negate the result in the case it is negative.

5.4 Multiplication

The multiplication of two floating-point numbers (assumed to be in IEEE 754 binary32 format) is computed as

$$Z = (-1)^{s_Z}\, M_Z\, 2^{E_Z-127} = XY = (-1)^{s_X}\, M_X\, 2^{E_X-127} (-1)^{s_Y}\, M_Y\, 2^{E_Y-127}, \tag{47}$$

where we see that

$$s_Z = s_X \oplus s_Y \tag{48}$$

$$\hat{M}_Z = M_X M_Y \tag{49}$$

$$\hat{E}_Z = E_X + E_Y - 127. \tag{50}$$

As we have $1 \leq M_X, M_Y < 2$ for normalized numbers, we get $1 \leq \hat{M}_Z < 4$. Hence, it may be required to shift $\hat{M}_Z$ one position to the right to obtain the normalized value M_Z, which can be seen by comparing with (41). If this happens one will also need to add 1 to $\hat{E}_Z$ to obtain E_Z.

This gives that the multiplication of two floating-point numbers corresponds to one fixed-point multiplication, one fixed-point addition, and a simple normalizing step after the operations.

For multiply-accumulate it is possible to use a *fused* architecture with the benefit that the alignment of the operand to be added can be done concurrently with the multiplication. In this way, it is possible to reduce the delay for the total MAC

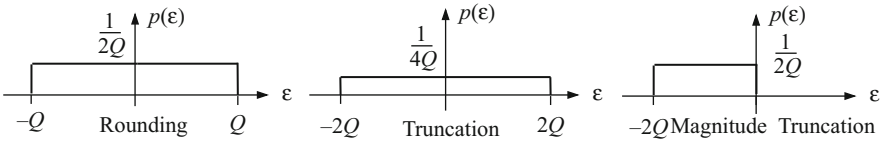


Fig. 32 Error distributions for floating-point arithmetic

operation compared to using separate multiplication and addition. Furthermore, rounding is performed only for the final output.

5.5 Quantization Error

The quantization error in the mantissa of a floating-point number is

$$X_Q = (1 + \epsilon)X. \tag{51}$$

Hence, the error is signal dependent and the analysis becomes very complicated [32, 46, 58]. Figure 32 shows the error distributions of floating-point arithmetic. Also, the quantization procedure needed to suppress parasitic oscillation in wave digital filters is more complicated for floating-point arithmetic.

6 Computation of Elementary Functions

The need of computing non-linear functions arises in many different algorithms. The straightforward method of approximating an elementary function is of course to just store the function values in a look-up table. However, this will typically lead to large tables, even though the resulting area from standard cell synthesis grows slower than the number of memory bits [18]. Instead it is of interest to find ways to approximate elementary functions using a trade-off between arithmetic operations and look-up tables. In this section, we briefly look at three different classes of algorithms. For a more thorough explanation of these and other methods we refer to [36].

6.1 CORDIC

The coordinate rotation digital computer (CORDIC) algorithm is a recursive algorithm to calculate elementary functions such as the trigonometric and hyperbolic (and their inverses) functions as well as magnitude and phase of complex vectors and was introduced by Volder [51] and generalized by Walther [55]. A summary of

the development of CORDIC can be found in [34, 52, 56]. It revolves around the idea of rotating the phase of a complex number by multiplying it by a succession of constant values. However, these multiplications can all be made as powers of 2 and hence, in binary arithmetic they can be done using just shifts and adds. Hence, CORDIC is in general a very attractive approach when a hardware multiplier is not available.

A rotation of a complex number $X + jY$ by an angle θ can be written as

$$\begin{bmatrix} X_r \\ Y_r \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}. \tag{52}$$

The idea of the CORDIC is to decompose the rotation by θ in several steps such that each rotation is a simple operation. In the straightforward CORDIC algorithm, we have

$$\theta = \sum_{k=0}^{\infty} d_k w_k, \;\; d_k = \pm 1, \;\; w_k = \arctan(2^{-k}). \tag{53}$$

Considering rotation k we get

$$\begin{aligned} \begin{bmatrix} X_{k+1} \\ Y_{k+1} \end{bmatrix} &= \begin{bmatrix} \cos(d_k w_k) & -\sin(d_k w_k) \\ \sin(d_k w_k) & \cos(d_k w_k) \end{bmatrix} \begin{bmatrix} X_k \\ Y_k \end{bmatrix} \\ &= \cos(w_k) \begin{bmatrix} 1 & -d_k 2^{-k} \\ d_k 2^{-k} & 1 \end{bmatrix} \begin{bmatrix} X_k \\ Y_k \end{bmatrix}. \end{aligned} \tag{54}$$

Now, neglecting the $\cos(w_k)$ term, we get a basic iteration which is a multiplication with 2^{-k} and an addition or subtraction. The sign of the rotation (d_n) is determined by comparing the required rotation angle θ with the currently rotated angle. This is typically done by using a third variable, Z_k, where $Z_0 = \theta$ and $Z_{k+1} = Z_k + d_k w_k$. Then

$$d_k = \begin{cases} 1 & Z_k \geq 0 \\ -1 & Z_k < 0. \end{cases} \tag{55}$$

The effect of neglecting the $\cos(w_k)$ in (54) is that the rotation is in fact not a proper rotation but instead a similarity [36]. Furthermore, as illustrated in Fig. 33, the magnitude of the vector is increased. The gain of the rotations depends on the number of iterations and can be written as

$$G(k) = \prod_{i=0}^{k} \sqrt{1 + 2^{-2i}}. \tag{56}$$

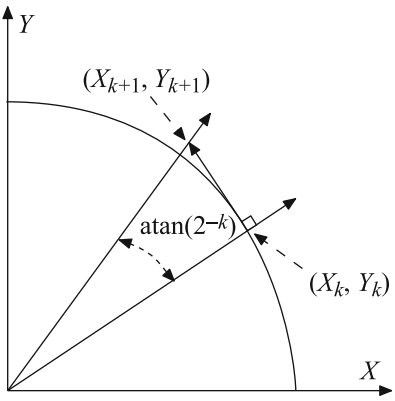


Fig. 33 Similarity (rotation) in the CORDIC algorithm

For $k \to \infty$, $G \approx 1.6468$. Several schemes to compensate for the gain has been proposed and a survey can be found in [50].

The above application of the CORDIC algorithm is usually referred to as rotation mode and can be used to compute $\sin(\theta)$ and $\cos(\theta)$ or perform rotations of complex vectors. There are also a vectoring mode, where the rotation is performed such that the imaginary part, Y_k, becomes zero.

The generalized CORDIC iterations can be written as

$$\begin{aligned} X_{k+1} &= X_k - m d_k Y_k 2^{-\sigma(k)} \\ Y_{k+1} &= Y_k + d_k X_k 2^{-\sigma(k)} \\ Z_{k+1} &= Z_k - d_k w_{\sigma(k)}. \end{aligned} \tag{57}$$

With an appropriate choice of m, d_k, w_k, and $\sigma(k)$ the CORDIC algorithm can perform a wide number of functions. These are summarized in Table 9, where three different types of CORDIC algorithms are introduced; Circular for computing trigonometric expressions, Linear for linear relationships, and Hyperbolic for hyperbolic computation. The scaling factor $\hat{G}_k$ for hyperbolic computations is

$$\hat{G}_k = \prod_{i=1}^{k} \sqrt{1 - 2^{-2(i-h_i)}} \tag{58}$$

and the factor h_k is defined as the largest integer such that $3^{h_k+1} + 2h_k - 1 \leq 2n$. In practice, this leads to that certain iteration angles, such that $k = \left(3^{i+1} - 1\right)/2$, are used twice to obtain convergence in the hyperbolic case.

The CORDIC computations can be performed in an iterative manner as in (57), but naturally also be unfolded. There has also been proposed radix-4 CORDIC algorithms, performing two iterations in each step, as well as different approaches using redundant arithmetic to speed up each iteration.

Table 9 Variable selection for generalized CORDIC

Type	Parameters	Rot. mode, $d_k = \text{sign}(z_k)$	Vec. mode, $d_k = -\text{sign}(y_k)$
Circular	$m = 1$	$X_k \rightarrow G_k(X_0 \cos(Z_0) - Y_0 \sin(Z_0))$	$X_n \rightarrow G_k\sqrt{X_0^2 + Y_0^2}$
	$w_k = \arctan(2^{-k})$	$Y_k \rightarrow G_k(Y_0 \cos(Z_0) + X_0 \sin(Z_0))$	$Y_n \rightarrow 0$
	$\sigma(k) = k$	$Z_k \rightarrow 0$	$Z_n \rightarrow Z_0 + \arctan\left(\frac{Y_0}{X_0}\right)$
Linear	$m = 0$	$X_k \rightarrow X_0$	$X_n \rightarrow X_0$
	$w_k = 2^{-k}$	$Y_k \rightarrow Y_0 + X_0 Z_0$	$Y_n \rightarrow 0$
	$\sigma(k) = k$	$Z_k \rightarrow 0$	$Z_n \rightarrow Z_0 + \frac{Y_0}{X_0}$
Hyperbolic	$m = -1$	$X_k \rightarrow \hat{G}_k(X_1 \cosh(Z_1) - Y_1 \sin(Z_1))$	$X_n \rightarrow \hat{G}_k\sqrt{X_1^2 - Y_1^2}$
	$w_k = \tanh^{-1}(2^{-k})$	$Y_k \rightarrow \hat{G}_k(Y_1 \cosh(Z_1) + X_1 \sinh(Z_1))$	$Y_n \rightarrow 0$
	$\sigma(k) = k - h_k$	$Z_k \rightarrow 0$	$Z_n \rightarrow Z_1 + \tanh^{-1}\left(\frac{Y_1}{X_1}\right)$

6.2 Polynomial and Piecewise Polynomial Approximations

It is possible to derive a polynomial $p(X)$ that approximates a function $f(X)$ by performing a Taylor expansion for a given point a such as

$$p(X) = \sum_{i=0}^{\infty} \frac{f^{(i)}(a)}{i!}(X - a)^i. \tag{59}$$

When the polynomial is restricted to a certain number of terms it is often better to optimize the polynomial coefficients as there are some accuracy to be gained. To determine the best coefficients is an approximation problems where typically there are more constraints (number of points for the approximation) than variables (polynomial order). This problem can be solved for a minimax solution using, e.g., Remez' exchange algorithm or linear programming. For a least square solution, the standard methods to solve over-determined systems can be applied. If fixed-point coefficients are required, the problem becomes much harder.

The polynomial approximations can be efficiently and accurately evaluated using Horner's method. This says that a polynomial

$$p(X) = b_0 + b_1 X + b_2 X^2 + \cdots + b_{n-1} X^{n-1} + b_n X^n \tag{60}$$

is to be evaluated as

$$p(X) = ((\ldots((b_n X + b_{n-1})X + b_{n-1})X + \cdots + b_2)\, X + b_1)\, X + b_0. \tag{61}$$

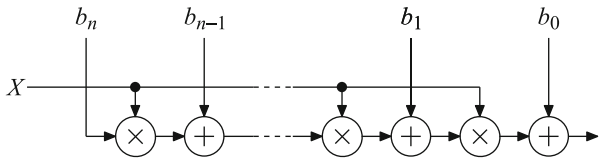


Fig. 34 Block diagram for Horner's method used for polynomial evaluation

Hence, there no need to compute any powers of x explicitly and a minimum number of arithmetic operations is used. Polynomial evaluation using Horner's method maps nicely to MAC-operations. The resulting scheme is illustrated in Fig. 34.

The drawback of Horner's method is that the computation is inherently sequential. An alternative is to use Estrin's method [36], which by explicitly computing the terms X^{2^i} rearranges the computation in a tree structure, increasing the parallelism and reducing the longest computational path. Estrin's method for polynomial evaluation can be written as

$$p(X) = (b_3X + b_2)\,X^2 + (b_1X + b_0) \tag{62}$$

for a third-order polynomial. For a seventh-order polynomial it becomes

$$p(X) = \Big((b_7X + b_6)\,X^2 + (b_5X + b_4)\Big)\,X^4 + \Big((b_3X + b_2)\,X^2 + (b_1X + b_0)\Big). \tag{63}$$

As can be seen, Estrin's method also maps well to MAC-operations.

The required polynomial order depends very much on the actual function that is approximated [36]. An approach to obtain a higher resolution despite using a lower polynomial order is to use different polynomials for different ranges. This is referred to as piecewise polynomials. A j segment n:th-order piecewise polynomial with segment breakpoints $x_k,\ k = 1, 2, \ldots, j+1$ can be written as

$$p(X) = \sum_{i=0}^{n} b_{i,j}\left(X - x_j\right)^i,\ x_j \le X < x_{j+1}. \tag{64}$$

From an implementation point of view it is often practical to have 2^k uniform segments and let the k most significant bits determine the segmentation. However, it can be shown that in general the total complexity is reduced for non-uniform segments. An illustration of a piecewise polynomial approximation is shown in Fig. 35.

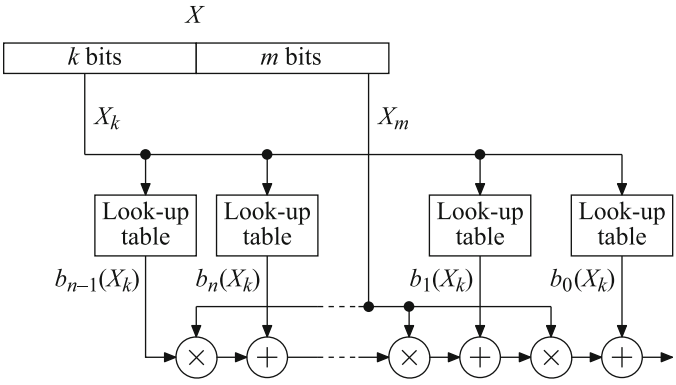


Fig. 35 Piecewise polynomial approximation using uniform segmentation based on the most significant bits

6.3 Table-Based Methods

The bipartite table method is based on splitting the input word, X, in three different subwords, X_0, X_1, and X_2. For ease of exposition we will assume that the length of these are identical, W_s and $W_f = 3W_s$, but in general it is possible to find a lower complexity realization by selecting non-uniform word lengths. Hence, we have

$$X = X_0 + 2^{-W_s} X_1 + 2^{-2W_s} X_2. \tag{65}$$

Now taking the first-order Taylor expansion of f at $X_0 + 2^{-W_s} X_1$ we get

$$f(X) \approx f\left(X_0 + 2^{-W_s} X_1\right) + 2^{-2W_s} X_2 f'\left(X_0 + 2^{-W_s} X_1\right). \tag{66}$$

Again, we take the Taylor expansion, this time a zeroth-order expansion of $f'(X_0 + 2^{-W_s} X_1)$ at X_0 as

$$f'\left(X_0 + 2^{-W_s} X_1\right) \approx f'(X_0) \tag{67}$$

This gives the bipartite approximation as

$$f(x) \approx T_1(X_0, X_1) + T_2(X_0, X_2) \tag{68}$$

where

$$\begin{aligned} T_1(X_0, X_1) &= f\left(X_0 + 2^{-W_s} X_1\right) \\ T_2(X_0, X_2) &= 2^{-2W_s} X_2 f'(X_0). \end{aligned} \tag{69}$$

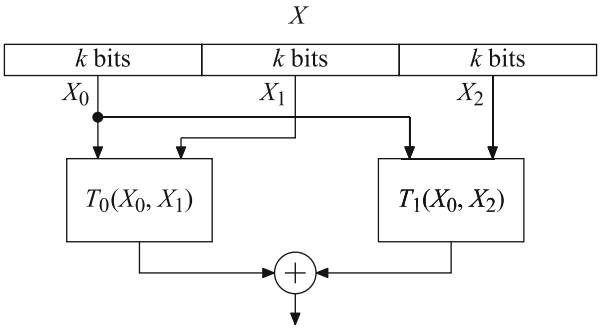


Fig. 36 Bipartite table approximation structure

The functions T_1 and T_2 are tabulated and the results are added. The resulting structure is shown in Fig. 36.

The bipartite approximation can be seen as a piecewise linear approximation where the same slope tables are used in several intervals. Here, T_1 contains the offset values, and T_2 contains tabulated lines with slope $f'(X_0)$.

The accuracy of the bipartite approximation can be improved by instead performing the first Taylor expansion at $X_0 + 2^{-W_s} X_1 + 2^{-2W_s-1}$ and the second at $X_0 + 2^{-W_s-1}$ [48]. It is also possible to split the input word into more subwords yielding a multipartite table approximation [9].

7 Further Reading

Several books have been published on related subjects. For general digital arithmetic we refer to [12, 25, 26, 42]. For the specific cases of approximation of elementary functions and floating-point arithmetic, [36] and [37] provide both broad overviews and in-depth knowledge, respectively.

References

1. IEEE standard for floating-point arithmetic (2008)
2. Baugh, C.R., Wooley, B.A.: A two's complement parallel array multiplication algorithm **C-22**(12), 1045–1047 (1973)
3. Bickerstaff, K.C., Schulte, M.J., Swartzlander Earl E., J.: Parallel reduced area multipliers. J. Signal Process. Syst. **9**(3), 181 (1995)
4. Brent, R.P., Kung, H.T.: A regular layout for parallel adders **C-31**(3), 260–264 (1982)
5. Chan, S.C., Yiu, P.M.: An efficient multiplierless approximation of the fast Fourier transform using sum-of-powers-of-two (SOPOT) coefficients **9**(10), 322–325 (2002)
6. Claasen, T., Mecklenbrauker, W., Peek, J.: Effects of quantization and overflow in recursive digital filters **24**(6), 517–529 (1976)

7. Croisier, A., Esteban, D., Levilion, M., Riso, V.: Digital filter for PCM encoded signals (1973). US Patent 3,777,130
8. Dadda, L.: Some schemes for parallel multipliers. Alta Frequenza **34**(5), 349–356 (1965)
9. de Dinechin, F., Tisserand, A.: Multipartite table methods **54**(3), 319–330 (2005)
10. Ercegovac, M.D., Lang, T.: On-the-fly conversion of redundant into conventional representations **C-36**(7), 895–897 (1987)
11. Ercegovac, M.D., Lang, T.: Division and square root: digit-recurrence algorithms and implementations. Kluwer Academic Publishers (1994)
12. Ercegovac, M.D., Lang, T.: Digital arithmetic. Elsevier (2004)
13. Eriksson, H., Larsson-Edefors, P., Sheeran, M., Sjalander, M., Johansson, D., Scholin, M.: Multiplier reduction tree with logarithmic logic depth and regular connectivity. In: Proc. IEEE Int. Symp. Circuits Syst., pp. 4–8 (2006)
14. Fettweis, A., Meerkotter, K.: On parasitic oscillations in digital filters under looped conditions **24**(9), 475–481 (1977)
15. Gustafsson, O.: A difference based adder graph heuristic for multiple constant multiplication problems. In: Proc. IEEE Int. Symp. Circuits Syst., pp. 1097–1100 (2007)
16. Gustafsson, O.: Lower bounds for constant multiplication problems **54**(11), 974–978 (2007)
17. Gustafsson, O., Dempster, A.G., Johansson, K., Macleod, M.D., Wanhammar, L.: Simplified design of constant coefficient multipliers. Circuits Syst. Signal Process. **25**(2), 225–251 (2006)
18. Gustafsson, O., Johansson, K.: An empirical study on standard cell synthesis of elementary function lookup tables. In: Proc. Asilomar Conf. Signals Syst. Comput., pp. 1810–1813 (2008)
19. Gustafsson, O., Wanhammar, L.: Low-complexity and high-speed constant multiplications for digital filters using carry-save arithmetic. In: Digital Filters. InTech (2011)
20. Harris, D.: A taxonomy of parallel prefix networks. In: Proc. Asilomar Conf. Signals Syst. Comput., vol. 2, pp. 2213–2217 Vol.2 (2003)
21. Hartley, R.I.: Subexpression sharing in filters using canonic signed digit multipliers **43**(10), 677–688 (1996)
22. Johansson, K., Gustafsson, O., Wanhammar, L.: Power estimation for ripple-carry adders with correlated input data. Proc. Int. Workshop Power Timing Modeling Optimization Simulation (2004)
23. Knowles, S.: A family of adders. In: Proc. IEEE Symp. Comput. Arithmetic, pp. 277–281 (2001)
24. Kogge, P.M., Stone, H.S.: A parallel algorithm for the efficient solution of a general class of recurrence equations **C-22**(8), 786–793 (1973)
25. Koren, I.: Computer arithmetic algorithms. Universities Press (2002)
26. Kornerup, P., Matula, D.W.: Finite precision number systems and arithmetic, vol. 133. Cambridge University Press (2010)
27. Ladner, R.E., Fischer, M.J.: Parallel prefix computation. J. ACM **27**(4), 831–838 (1980)
28. Liang, J., Tran, T.D.: Fast multiplierless approximations of the DCT with the lifting scheme **49**(12), 3032–3044 (2001)
29. Lim, Y.C.: Single-precision multiplier with reduced circuit complexity for signal processing applications **41**(10), 1333–1336 (1992)
30. Lim, Y.C., Yang, R., Li, D., Song, J.: Signed power-of-two term allocation scheme for the design of digital filters **46**(5), 577–584 (1999)
31. Liu, B.: Effect of finite word length on the accuracy of digital filters–a review **18**(6), 670–677 (1971)
32. Liu, B., Kaneko, T.: Error analysis of digital filters realized with floating-point arithmetic **57**(10), 1735–1747 (1969)
33. Macsorley, O.L.: High-speed arithmetic in binary computers. Proc. IRE **49**(1), 67–91 (1961)
34. Meher, P.K., Valls, J., Juang, T.B., Sridharan, K., Maharatna, K.: 50 years of CORDIC: Algorithms, architectures, and applications **56**(9), 1893–1907 (2009)
35. Mou, Z.J., Jutand, F.: 'overturned-stairs' adder trees and multiplier design **41**(8), 940–948 (1992)
36. Muller, J.M.: Elementary functions. Springer (2006)

37. Muller, J.M., Brisebarre, N., De Dinechin, F., Jeannerod, C.P., Lefevre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of floating-point arithmetic. Springer Science & Business Media (2009)
38. Noll, T.G.: Carry-save architectures for high-speed digital signal processing. J. Signal Process. Syst. **3**(1-2), 121 (1991)
39. Oklobdzija, V.G., Villeger, D., Liu, S.S.: A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach **45**(3), 294–306 (1996)
40. Omondi, A., Premkumar, B.: Residue number systems: theory and implementation. World Scientific (2007)
41. Oskuii, S.T., Kjeldsberg, P.G., Gustafsson, O.: Power optimized partial product reduction interconnect ordering in parallel multipliers. In: Proc. Norchip 2007, pp. 1–6 (2007)
42. Parhami, B.: Computer arithmetic, vol. 20. Oxford university press (2010)
43. Petra, N., Caro, D.D., Garofalo, V., Napoli, E., Strollo, A.G.M.: Truncated binary multipliers with variable correction and minimum mean square error **57**(6), 1312–1325 (2010)
44. Potkonjak, M., Srivastava, M.B., Chandrakasan, A.P.: Multiple constant multiplications: efficient and versatile framework and algorithms for exploring common subexpression elimination **15**(2), 151–165 (1996)
45. Puschel, M., Moura, J.M.F., Johnson, J.R., Padua, D., Veloso, M.M., Singer, B.W., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: Spiral: Code generation for DSP transforms **93**(2), 232–275 (2005)
46. Rao, B.D.: Floating point arithmetic and digital filters **40**(1), 85–95 (1992)
47. Samueli, H., Willson, A.: Nonperiodic forced overflow oscillations in digital filters **30**(10), 709–722 (1983)
48. Schulte, M.J., Stine, J.E.: Approximating elementary functions with symmetric bipartite tables **48**(8), 842–847 (1999)
49. Stelling, P.F., Oklobdzija, V.G.: Design strategies for optimal hybrid final adders in a parallel multiplier. J. Signal Process. Syst. **14**(3), 321 (1996)
50. Timmermann, D., Hahn, H., Hosticka, B., Rix, B.: A new addition scheme and fast scaling factor compensation methods for CORDIC algorithms. Integration, the VLSI J. **11**(1), 85–100 (1991)
51. Volder, J.E.: The CORDIC trigonometric computing technique. IRE Trans. Electron. Comput. **EC-8**(3), 330–334 (1959)
52. Volder, J.E.: The birth of CORDIC. J. Signal Process. Syst. **25**(2), 101 (2000)
53. Voronenko, Y., Püschel, M.: Multiplierless multiple constant multiplication. ACM Trans. Algorithms **3**(2), 11 (2007)
54. Wallace, C.S.: A suggestion for a fast multiplier **EC-13**(1), 14–17 (1964)
55. Walther, J.S.: A unified algorithm for elementary functions. In: Proc. Spring Joint Computer Conf., pp. 379–385. ACM (1971)
56. Walther, J.S.: The story of unified CORDIC. J. Signal Process. Syst. **25**(2), 107–112 (2000)
57. Wanhammar, L.: DSP integrated circuits. Academic press (1999)
58. Zeng, B., Neuvo, Y.: Analysis of floating point roundoff errors using dummy multiplier coefficient sensitivities **38**(6), 590–601 (1991)
59. Zimmermann, R.: Binary adder architectures for cell-based VLSI and their synthesis. Hartung-Gorre (1998)

Coarse-Grained Reconfigurable Array Architectures

Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts

Abstract Coarse-Grained Reconfigurable Array (CGRA) architectures accelerate the same inner loops that benefit from the high instruction-level parallelism (ILP) support in very long instruction word (VLIW) architectures. Unlike VLIWs, CGRAs are designed to execute only the loops, which they can hence do more efficiently. This chapter discusses the basic principles of CGRAs and the wide range of design options available to a CGRA designer, covering a large number of existing CGRA designs. The impact of different options on flexibility, performance, and power-efficiency is discussed, as well as the need for compiler support. The ADRES CGRA design template is studied in more detail as a use case to illustrate the need for design space exploration, for compiler support, and for the manual fine-tuning of source code.

1 Application Domain of Coarse-Grained Reconfigurable Arrays

Many embedded applications require high throughput. At the same time, the power consumption of battery-operated devices needs to be minimized to increase their autonomy. In general, the performance obtained on a programmable processor for a certain application can be defined as the reciprocal of the application execution time. Considering that most programs consist of a series P of consecutive phases with different characteristics, performance can be defined in terms of the operating frequencies f_p, the instructions executed per cycle IPC_p and instruction count IC_p

B. De Sutter (✉)
Ghent University, Gent, Belgium
e-mail: bjorn.desutter@ugent.be

P. Raghavan · A. Lambrechts
imec, Heverlee, Belgium
e-mail: ragha@imec.be; lambreca@imec.be

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_12

of each phase, and in terms of the time overhead involved in switching between the phases $t_{p \to p+1}$ as follows:

$$\frac{1}{\text{performance}} = \text{execution time} = \sum_{p \in P} \frac{IC_p}{IPC_p \cdot f_p} + t_{p \to p+1}. \quad (1)$$

The operating frequencies f_p cannot be increased infinitely because of power-efficiency reasons. Alternatively, a designer can increase the performance by designing or selecting a system that can execute code at higher IPCs. In a power-efficient architecture, a high IPC is reached for the most important phases $l \in L \subset P$, with L typically consisting of the compute-intensive inner loops, while limiting their instruction count IC_l and reaching a sufficiently high, but still power-efficient frequency f_l. Furthermore, the time overhead $t_{p \to p+1}$ as well as the corresponding energy overhead of switching between the execution modes of consecutive phases should be minimized if such switching happens frequently. Note that such switching only happens on hardware that supports multiple execution modes in support of phases with different characteristics.

Course-Grained Reconfigurable Array (CGRA) accelerators aim for these goals for the inner loops found in many digital signal processing (DSP) domains. Such applications have traditionally employed Very Long Instruction Word (VLIW) architectures such as the TriMedia 3270 [112] and the TI C64 [106], Application-Specific Integrated Circuits (ASICs), and Application-Specific Instruction Processors (ASIPs). To a large degree, the reasons for running these applications on VLIW processors also apply for CGRAs. First of all, a large fraction of the computation time is spent in manifest nested loops that perform computations on arrays of data and that can, possibly through compiler transformations, provide a lot of Instruction-Level Parallelism (ILP). Secondly, most of those inner loops are relatively simple. When the loops include conditional statements, those can be implemented by means of predication [70] instead of control flow. Furthermore, none or very few loops contain multiple exits or continuation points in the form of, e.g., `break` or `continue` statements. Moreover, after inlining the loops are free of function calls. Finally, the loops are not regular or homogeneous enough to benefit from vector computing, like on the EVP [8] or on Ardbeg [113]. When there is enough regularity and Data-Level Parallelism (DLP) in the loops of an application, vector computing can typically exploit it more efficiently than what can be achieved by converting the DLP into ILP and exploiting that on a CGRA. So in short, CGRAs are ideally suited for applications of which time-consuming parts have manifest behavior, large amounts of ILP and limited amounts of DLP.

Over the last decade, applications from many domains have been accelerated on CGRAs. These include video processing [7, 17, 18, 67, 71, 100], image processing [40], audio processing [103], linear [76] and non-linear [69, 92] algebras, software-defined radios [11, 12, 25, 104, 110], augmented reality [85], biomedical applications [44], and Map-Reduce algorithms [62]. In support of these applications, CGRAs have also been commercialized. The Samsung Reconfigurable Processor,

the commercialized version of the ADRES CGRA that Samsung and imec initially developed as a proof-of-concept, has been used in ultra-high definition televisions and in smartphones, amongst others.

In the remainder of this chapter, Sect. 2 presents the fundamental properties of CGRAs. Section 3 gives an overview of the design options for CGRAs. This overview help designers in evaluating whether or not CGRAs are suited for their applications and their design requirements, and if so, which CGRA designs are most suited. After the overview, Sect. 4 presents a case study on the ADRES CGRA architecture. This study serves two purposes. First, it illustrates the extent to which source code needs to be tuned to map well onto CGRA architectures. As we will show, this is an important aspect of using CGRAs, even when good compiler support is available and when a very flexible CGRA is targeted, i.e., one that puts very few restrictions on the loop bodies that it can accelerate. Secondly, our use case illustrates how Design Space Exploration is necessary to instantiate optimized designs from parameterizable and customizable architecture templates such as the ADRES architecture template. Some conclusions are drawn in Sect. 5.

2 CGRA Basics

CGRAs focus on the efficient execution of the type of loops discussed in the previous section. By neglecting non-loop code or outer-loop code that is assumed to be executed on other cores, CGRAs can take the VLIW principles for exploiting ILP in loops a step further to consume less energy and deliver higher performance, without compromising on available compiler support. Figures 1 and 2 illustrate this.

Higher performance for high-ILP loops is obtained through two main features that separate CGRA architectures from VLIW architectures. First, CGRA architectures typically provide more Issue Slots (ISs) than typical VLIWs do. In the CGRA literature, some other commonly used terms to denote CGRA ISs are Arithmetic-Logic Units (ALUs), Functional Units (FUs), or Processing Elements (PEs). Conceptually, these terms all denote the same: logic on which an instruction can be executed, typically one per cycle. For example, a typical ADRES CGRA [11–13, 24, 71, 73–75] consists of 16 issue slots, whereas the TI C64 features 8 slots, and

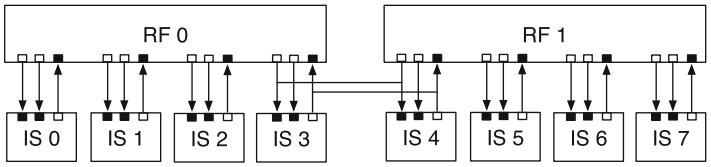


Fig. 1 An example clustered VLIW architecture with two RFs and eight ISs. Solid directed edges denote physical connections. Black and white small boxes denote input and output ports, respectively. There is a one-to-one mapping between input and output ports and physical connections

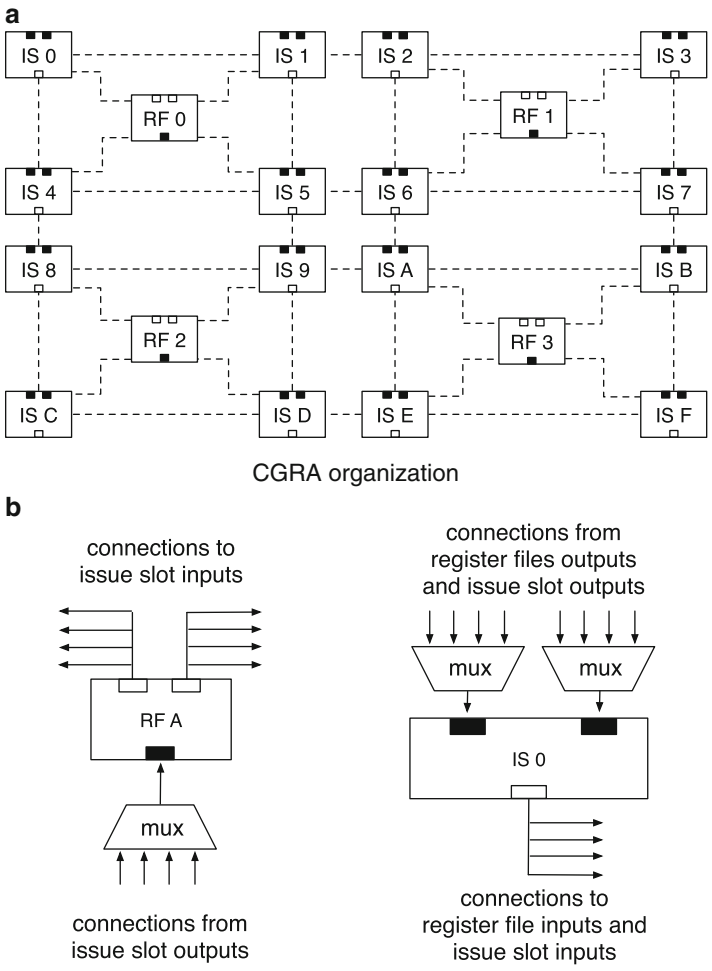


Fig. 2 Part (**a**) shows an example CGRA with 16 ISs and 4 RFs, in which dotted edges denote conceptual connections that are implemented by physical connections and muxes as in part (**b**)

the NXP TriMedia features only 5 slots. The higher number of ISs directly allows to reach higher IPCs, and hence higher performance, as indicated by Eq. (1). To support these higher IPCs, the bandwidth to memory is increased by having more load/store ISs than on a typical VLIW, and special memory hierarchies as found on ASIPs, ASICs, and other DSPs. These include FIFOs, stream buffers, scratch-pad memories, etc. Secondly, CGRA architectures typically provide a number of direct connections between the ISs that allow data to flow from one IS to another without

needing to pass data through a Register File (RF). As a result, less register copy operations need to be executed in the ISs, which reduces the IC factor in Eq. (1) and frees ISs for more useful computations.

Higher energy-efficiency is obtained through several features. Because of the direct connections between ISs, less data needs to be transferred into and out of RFs. This saves considerable energy. Also, because the ISs are arranged in a 2D matrix, small RFs with few ports can be distributed in between the ISs as depicted in Fig. 2. This contrasts with the many-ported RFs in (clustered) VLIW architectures, which basically feature a one-dimensional design as depicted in Fig. 1. The distributed CGRA RFs consume considerably less energy. Finally, by not supporting control flow, the instruction memory organization can be simplified. In statically reconfigurable CGRAs, this memory is nothing more than a set of configuration bits that remain fixed for the whole execution of a loop. Clearly this is very energy-efficient. Other CGRAs, called dynamically reconfigurable CGRAs, feature a form of distributed level-0 loop buffers [59] or other small controllers that fetch new configurations every cycle from simple configuration buffers. To support loops that include control flow and conditional operations, the compiler replaces that control flow by data flow by means of predication [70] or other mechanisms. In this way, CGRAs differ from VLIW processors that typically feature a power-hungry combination of an instruction cache, instruction decompression and decoding pipeline stages, and a non-trivial update mechanism of the program counter.

CGRA architectures have two main drawbacks. Firstly, because they only execute loops, they need to be coupled to other cores on which all other parts of the program are executed. This coupling can introduce run-time and design-time overhead. Secondly, as clearly visible in the example in Fig. 2, the interconnect structure of a CGRA is vastly more complex than that of a VLIW. On a VLIW, scheduling an instruction in some IS automatically implies the reservation of connections between the RF and the IS and of the corresponding ports. On CGRAs, this is not the case. Because there is no one-to-one mapping between connections and input/output ports of ISs and RFs, connections need to be reserved explicitly by the compiler or programmer together with ISs, and the data flow needs to be routed explicitly over the available connections. This can be done, for example, by programming switches and multiplexors (a.k.a. muxes) explicitly, like the ones depicted in Fig. 2b. Consequently more complex compiler technology than that of VLIW compilers [43] is needed to automate the mapping of code onto a CGRA. Moreover, writing assembly code for CGRAs ranges from being very difficult to virtually impossible, depending on the type of reconfigurability and on the form of processor control.

Having explained these fundamental concepts that differentiate CGRAs from VLIWs, we can now also differentiate them from Field-Programmable Gate Arrays (FPGAs), where the name CGRA actually comes from. Whereas FPGAs feature *bitwise* logic in the form of Look-Up Tables (LUTs) and switches, CGRAs feature more energy-efficient and area-conscious *word-wide* ISs, RFs, and interconnections. Hence the name coarse-grained array architecture. As there are much fewer ISs on a CGRA than there are LUTs on an FPGA, the number of bits required to configure

the CGRA ISs, muxes, and RF ports is typically orders of magnitude smaller than on FPGAs. If this number becomes small enough, dynamic reconfiguration can be possible every cycle. So in short, CGRAs can be seen as statically or dynamically reconfigurable coarse-grained FPGAs, or as 2D, highly-clustered loop-only VLIWs with direct interconnections between ISs that need to be programmed explicitly.

3 CGRA Design Space

The large design space of CGRA architectures features many design options. These include the way in which the CGRA is coupled to a main processor, the type of interconnections and computation resources used, the reconfigurability of the array, the way in which the execution of the array is controlled, support for different forms of parallelism, etc. This section discusses the most important design options and the influence of the different options on important aspects such as performance, power efficiency, compiler friendliness, and flexibility. In this context, higher flexibility equals placing fewer restrictions on loop bodies that can be mapped onto a CGRA.

Our overview of design options is not exhaustive. Its scope is limited to the most important features of CGRA architectures that feature a 2D array of ISs. However, the distinction between 1D VLIWs and 2D CGRAs is anything but well-defined. The reason is that this distinction is not simply a layout issue, but one that also concerns the topology of the interconnects. Interestingly, this topology is precisely one of the CGRA design options with a large design freedom.

3.1 Tight Versus Loose Coupling

Some CGRA designs are coupled loosely to main processors. For example, Fig. 3 depicts how the MorphoSys CGRA [60] is connected as an external accelerator to a TinyRISC Central Processing Unit (CPU) [1]. The CPU is responsible for executing non-loop code, for initiating DMA data transfers to and from the CGRA and the buffers, and for initiating the operation of the CGRA itself by means of special instructions added to the TinyRISC ISA.

This type of design offers the advantage that the CGRA and the main CPU can be designed independently, and that both can execute code concurrently, thus delivering higher parallelism and higher performance. For example, using the double frame buffers [60] depicted in Fig. 3, the MorphoSys CGRA can be operating on data in one buffer while the main CPU initiates the necessary DMA transfers to the other buffer for the next loop or for the next set of loop iterations. One drawback is that any data that needs to be transferred from non-loop code to loop code needs to be transferred by means of DMA transfers. This can result in a large overhead, e.g., when frequent switching between non-loop code and loops with few iterations occurs and when the loops consume scalar values computed by non-loop code.

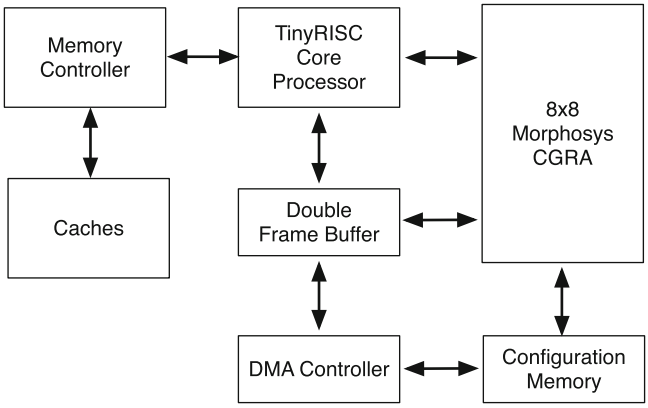


Fig. 3 A TinyRISC main processor loosely coupled to a MorphoSys CGRA array. Note that the main data memory (cache) is not shared and that no IS hardware or registers is are shared between the main processor and the CGRA. Thus, both can run concurrent threads

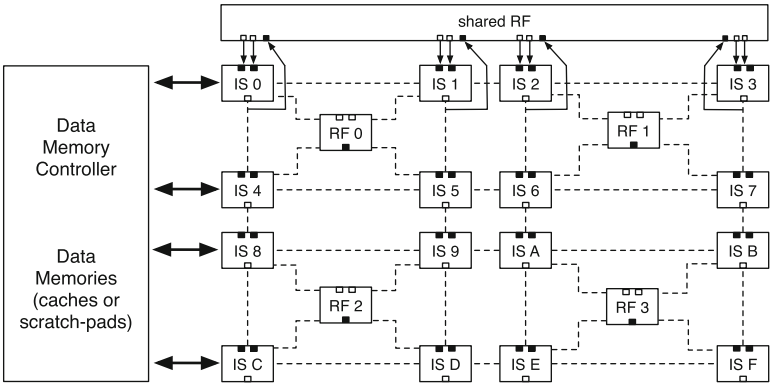


Fig. 4 A simplified picture of an ADRES architecture. In the main processor mode, the top row of ISs operates like a VLIW on the data in the shared RF and in the data memories, fetching instructions from an instruction cache. When the CGRA mode is initiated with a special instruction in the main VLIW ISA, the whole array starts operating on data in the distributed RFs, in the shared RF and in the data memories. The memory port in IS 0 is also shared between the two operating modes. Because of the resource sharing, only one mode can be active at any point in time

By contrast, an ADRES CGRA is coupled tightly to its main CPU. A simplified ADRES is depicted in Fig. 4. Its main CPU is a VLIW consisting of the shared RF and the top row of CGRA ISs. In the main CPU mode, this VLIW executes instructions that are fetched from a VLIW instruction cache and that operate on data in the shared RF. The idle parts of the CGRA are then disabled by clock-gating to save energy. By executing a `start_CGRA` instruction, the processor switches to CGRA mode in which the whole array, including the shared RF and the top row of ISs, executes a loop for which it gets its configuration bits from a configuration memory. This memory is omitted from the figure for the sake of simplicity.

The drawback of this tight coupling is that because the CGRA and the main processor mode share resources, they cannot execute code concurrently. However, this tight coupling also has advantages. Scalar values that have been computed in non-loop code, can be passed from the main CPU to the CGRA without any overhead because those values are already present in the shared RFs or in the shared memory banks. Furthermore, using shared memories and an execution model of exclusive execution in either main CPU or CGRA mode significantly eases the automated co-generation of main CPU code and of CGRA code in a compiler, and it avoids the run-time overhead of transferring data between memories. Finally, on the ADRES CGRA, switching between the two modes takes only two cycles. Thus, the run-time overhead is minimal. That overhead can still be considerable, however, in the case of nested loops, as inner loops are then entered and exited many times. Moreover, upon entry and exit of a software pipelined loop, resources are wasted as the software pipeline fills and drains in the so-called prologue and epilogue of the loop. This will be discussed in more detail in Sect. 4.1.1. Two design extensions have been proposed to reduce this overhead. First, instruction set extensions have been proposed to reduce the overhead that flattening of imperfectly nested loops introduces [57]. By flattening loop nests, less mode switches are necessary. Secondly, the Remus CGRA design for streaming data applications features an array in which the data flows in one direction, i.e., from one row to another, top to bottom [66, 117, 118]. The rows of the CGRA then operate as if they are a statically scheduled pipeline. During the epilogue of one loop, the rows gradually become unused by that loop, and hence they become available for the next loop to be executed. The next loop's prologue can hence start executing as soon as the current loop's epilogue has started. In many applications, this can save considerable execution time.

Silicon Hive CGRAs [14, 15] do not feature a clear separation between the CGRA accelerator and the main processor. Instead there is just a single processor that can be programmed at different levels of ILP, i.e., at different instruction word widths. This allows for a very simple programming model, with all the programming and performance advantages of the tight coupling of ADRES. Compared to ADRES, however, the lack of two distinctive modes makes it more difficult to implement coarse-grained clock-gating or power-gating, i.e., gating of whole sets of ISs combined instead of separate gating of individual ISs.

Somewhere in between loose and tight coupling is the PACT XPP design [79], in which the array consist of simpler ISs that can operate like a true CGRA, as well as of more complex ISs that are in fact full-featured small RISC processors that can run independent threads in parallel with the CGRA.

As a general rule, looser coupling potentially enables more Thread-Level Parallelism (TLP) and it allows for a larger design freedom. Tighter coupling can minimize the per-thread run-time overhead as well as the compile-time overhead. This is in fact no different from other multi-core or accelerator-based platforms.

3.2 CGRA Control

Many different mechanisms exist to control how code gets executed on CGRAs, i.e., to control which operation is issued on which IS at which time and how data values are transferred from producing operations to consuming ones. Two important aspects of CGRAs that drive different methods for control are reconfigurability and scheduling. Both can be static, dynamic, or a hybrid combination thereof.

3.2.1 Reconfigurability

Some CGRAs, like ADRES, Silicon Hive, and MorphoSys are fully dynamically reconfigurable: Exactly one full reconfiguration takes place for every execution cycle. Of course no reconfiguration takes places in cycles in which the whole array is stalled. Such stalls can happen, e.g., because memory accesses take longer than expected in the schedule as a result of a cache miss or a memory bank access conflict. This cycle-by-cycle reconfiguration is similar to the fetching of one VLIW instruction per cycle, but on these CGRAs the fetching is simpler as it only iterates through a loop body existing of straight-line CGRA configurations without control flow. Other CGRAs like the KressArray [37–39] are fully statically reconfigurable, meaning that the CGRA is configured before a loop is entered, and no reconfiguration takes place during the loop at all. Still other architectures feature a hybrid reconfigurability. The RaPiD [22, 27] architecture features partial dynamic reconfigurability, in which part of the bits are statically reconfigurable and another part is dynamically reconfigurable and controlled by a small sequencer. Yet another example is the PACT architecture, in which the CGRA itself can initiate events that invoke (partial) reconfiguration. This reconfiguration consumes a significant amount of time, however, so it is advised to avoid it if possible, and to use the CGRA as a statically reconfigurable CGRA.

In statically reconfigured CGRAs, each resource performs a single task for the whole duration of the loop. In that case, the mapping of software onto hardware becomes purely spatial, as illustrated in Fig. 5a. In other words, the mapping problem becomes one of placement and routing, in which instructions and data dependencies between instructions have to mapped on a 2D array of resources. For these CGRAs, compiler techniques similar to hardware synthesis techniques can be used, as those used in FPGA placement and routing [9].

By contrast, dynamic reconfigurability enables the programmer to use hardware resources for multiple different tasks during the execution of a loop or even during the execution of a single loop iteration. In that case, the software mapping problem becomes a spatial and temporal mapping problem, in which the operations and data transfers not only need to be placed and routed on and over the hardware resources, but in which they also need to be scheduled. A contrived example of a temporal mapping is depicted in Fig. 5b. Most compiler techniques [24, 26, 29, 73, 75, 78, 81, 82, 107] for these architectures also originate from the FPGA placement and routing

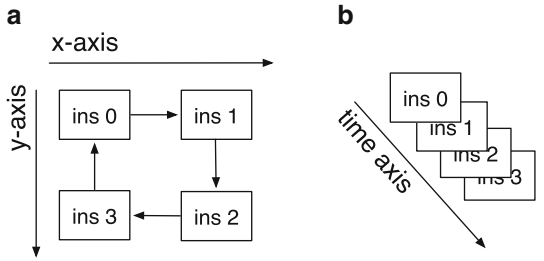


Fig. 5 Part (**a**) shows a spatial mapping of a sequence of four instructions on a statically reconfigurable 2 × 2 CGRA. Edges denote dependencies, with the edge from instruction 3 to instruction 0 denoting that instruction 0 from iteration i depends on instruction 3 from iteration $i - 1$. So only one out of four ISs is utilized per cycle. Part (**b**) shows a temporal mapping of the same code on a dynamically reconfigurable CGRA with only one IS. The utilization is higher here, at 100%

world. For CGRAs, the array of resources is not treated as a 2D spatial array, but as a 3D spatial-temporal array, in which the third dimension models time in the form of execution cycles. Scheduling in this dimension is often based on techniques that combine VLIW scheduling techniques such as modulo scheduling [43, 54, 93], with FPGA synthesis-based techniques [9]. Still other compiler techniques exist that are based on constraint solving [101], or on integer linear programming [2, 56, 127].

The most important advantage of static reconfigurability is the lack of reconfiguration overhead, in particular in terms of power consumption. For that reason, large arrays can be used that are still power-efficient. The disadvantage is that even in the large arrays the amount of resources constrains which loops can be mapped.

Dynamically reconfigurable CGRAs can overcome this problem by spreading the computations of a loop iteration over multiple configurations. Thus a small dynamically reconfigurable array can execute larger loops. The loop size is then not limited by the array size, but by the array size times the depth of the reconfiguration memories. For reasons of power efficiency, this depth is also limited, typically to tens or hundreds of configurations, which suffices for most if not all inner loops.

A potential disadvantage of dynamically reconfigurable CGRAs is the power consumption of the configuration memories, even for small arrays, and of the configuration fetching mechanism. The disadvantage can be tackled in different ways. ADRES and MorphoSys tackle it by not allowing control flow in the loop bodies, thus enabling the use of very simple, power-efficient configuration fetching techniques similar to level-0 loop buffering [59]. Whenever control flow is found in loop bodies, such as for conditional statements, this control flow then first needs to be converted into data flow, for example by means of predication and hyperblock formation [70]. While these techniques can introduce some initial overhead in the code, this overhead typically will be more than compensated by the fact that a more efficient CGRA design can be used.

The MorphoSys design takes this reduction of the reconfiguration fetching logic even further by limiting the supported code to Single Instruction Multiple Data

(SIMD) code. In the two supported SIMD modes, all ISs in a row or all ISs in a column perform identical operations. As such only one IS configuration needs to be fetched per row or column. As already mentioned, the RaPiD architecture limits the number of configuration bits to be fetched by making only a small part of the configuration dynamically reconfigurable. Kim et al. provide yet another solution in which the configuration bits of one column in one cycle are reused for the next column in the next cycle [52]. Furthermore, they also propose to reduce the power consumption in the configuration memories by compressing the configurations [53].

Still, dynamically reconfigurable designs exist that put no restrictions on the code to be executed, and that even allow control flow in the inner loops. The Silicon Hive design is one such design.

A general rule is that a limited reconfigurability puts more constraints on the types and sizes of loops that can be mapped. Which design provides the highest performance or the highest energy efficiency depends, amongst others, on the variation in loop complexity and loop size present in the applications to be mapped onto the CGRA. With large statically reconfigurable CGRAs, it is only possible to achieve high utilization for all loops in an application if all those loops have similar complexity and size, or if they can be made so with loop transformations, and if the iterations are not dependent on each other through long-latency dependency cycles (as was the case in Fig. 5). Dynamically reconfigurable CGRAs, by contrast, can also achieve high average utilization over loops of varying sizes and complexities, and with inter-iteration dependencies. That way dynamically reconfigurable CGRAs can achieve higher energy efficiency in the data path, at the expense of higher energy consumption in the control path. Which design option is the best depends also on the process technology used, and in particular on the ability to perform clock or power gating and on the ratio between active and passive power (a.k.a. leakage).

In that regard, it is interesting to note the recent research direction of so-called dual-V_{dd} and multi-V_{dd} CGRA designs [33, 115, 120] that goes beyond the binary approach of gating. In these designs, the supply voltage fed to different parts of a CGRA, which can be individual ISs or clusters thereof, can vary independently. This resembles dynamic voltage scaling as found on many modern multi-core CPUs, but in the case of CGRAs the supply voltages fed to a part of the CGRA is determined by the length of the critical path in the circuit that is triggered by the specific operations executing on that part of the array.

3.2.2 Scheduling and Issuing

Both with dynamic and with static reconfigurability, the execution of operations and of data transfers needs to be controlled. This can be done statically in a compiler, similar to the way in which operations from static code schedules are scheduled and issued on VLIW processors [28, 43], or dynamically, similar to the way in which out-of-order processors issue instructions when their operands become available [99]. Many possible combinations of static and dynamic reconfiguration and of static and dynamic scheduling exist.

A first class consists of dynamically scheduled, dynamically reconfigurable CGRAs like the TRIPS architecture [32, 95]. For this architecture, the compiler determines on which IS each operation is to be executed and over which connections data is to be transferred from one IS to another. So the compiler performs placement and routing. All scheduling (including the reconfiguration) is dynamic, however, as in regular out-of-order superscalar processors [99]. TRIPS mainly targets general-purpose applications, in which unpredictable control flow makes the generation of high-quality static schedules difficult if not impossible. Such applications most often provide relatively limited ILP, for which large arrays of computational resources are not efficient. So instead a small, dynamically reconfigurable array is used, for which the run-time cost of dynamic reconfiguration and scheduling is acceptable.

A second class of dynamically reconfigurable architectures avoids the overhead of dynamic scheduling by supporting VLIW-like static scheduling [28]. Instead of doing the scheduling in hardware where the scheduling logic then burns power, the scheduling for ADRES, MorphoSys and Silicon Hive architectures is done by a compiler. Compilers can do this efficiently for loops with regular, predictable behavior and high ILP, as found in many DSP applications. As is the case for VLIW architectures, software pipelining [43, 54, 93] is very important to expose the ILP in CGRA software kernels, so most compiler techniques [19, 24, 26, 29, 34, 35, 73, 75, 78, 81, 82, 107, 128] for statically scheduled CGRAs implement some form of software pipelining.

A third class of CGRAs are the statically reconfigurable, dynamically scheduled architectures, such as KressArray or PACT (neglecting the time-consuming partial reconfigurability of the PACT). The compiler performs placement and routing, and the code execution progress is guided by tokens or event signals that are passed along with data. Thus the control is dynamic, and it is distributed over the token or event path, similar to the way in which transport-triggered architectures [21] operate. These statically reconfigurable CGRAs do not require software pipelining techniques because there is no temporal mapping. Instead the spatial mapping and the control implemented in the tokens or event signals implement a hardware pipeline.

Hybrid designs exist as well. Park et al. use tokens not to trigger the execution of instructions, but to enable an opcode compression scheme without increasing decoder complexity with the end goal of reducing the power consumption [84]. In their statically scheduled CGRA, data-producing ISs send a token to the consuming ISs over a token datapath that complements the existing datapath. Based on the tokens that arrive, the consuming IS then already knows which type of operation it will need to execute, so less opcode bits need to be retrieved and decoded to program the IS. This way, they were able to obtain a 56% power reduction in the control path.

Another form of hybrid designs are the so-called triggered execution and dual-issue designs [36, 125, 126]. These are scheduled statically, but feature extensions to increase the resource utilization of loops bodies containing if-then-else structures. With standard predication techniques, the instructions of both the then and else branches occupy ISs. So in every iteration, the ISs used for the non-occupied branch are wasted. With the trigger-based and dual-issue extensions, two operations (one

from the then branch and one from the else branch) can be loaded together to configure the same IS, and additional predicate logic decides dynamically which of the operations is actually executed.

We can conclude by noting that, as in other architecture paradigms such as VLIW processing or superscalar out-of-order execution, dynamically scheduled CGRAs can deliver higher performance than statically scheduled ones for control-intensive code with unpredictable behavior. On dynamically scheduled CGRAs the code path that gets executed in an iteration determines the execution time of that iteration, whereas on statically scheduled CGRAs, the combination of all possible execution paths (including the slowest path which might be executed infrequently) determines the execution time. Thus, dynamically scheduled CGRAs can provide higher performance for some applications. However, the power-efficiency will then typically also be poor because more power will be consumed in the control path. Again, the application domain determines which design option is the most appropriate.

3.2.3 Thread-Level and Data-Level Parallelism

Another important aspect of control is the possibility to support different forms of parallelism. Obviously, loosely-coupled CGRAs can operate in parallel with the main CPU, but one can also try to use the CGRA resources to implement SIMD or to run multiple threads concurrently within the CGRA.

When dynamic scheduling is implemented via distributed event-based control, as in KressArray or PACT, implementing TLP is relatively simple and cheap. For small enough loops of which the combined resource use fits on the CGRA, it suffices to map independent thread controllers on different parts of the distributed control.

For architectures with centralized control, the only option to run threads in parallel is to provide additional controllers or to extend the central controller, for example to support parallel execution modes. While such extensions will increase the power consumption of the controller, the newly supported modes might suit certain code fragments better, thus saving in data path energy and configuration fetch energy.

The TRIPS controller supports four operation modes [95]. In the first mode, all ISs cooperate for executing one thread. In the second mode, the four rows execute four independent threads. In the third mode, fine-grained multi-threading [99] is supported by time-multiplexing all ISs over multiple threads. Finally, in the fourth mode each row executes the same operation on each of its ISs, thus implementing SIMD in a similar, fetch-power-efficient manner as is done in the two modes of the MorphoSys design. Thus, for each loop or combination of loops in an application, the TRIPS compiler can exploit the most suited form of parallelism.

The Raw architecture [105] is a hybrid between a many-core architecture and a CGRA architecture in the sense that it does not feature a 2D array of ISs, but rather a 2D array of tiles that each consist of a simple RISC processor. The tiles are connected to each other via a mesh interconnect, and transporting data over

this interconnect to neighboring tiles does not consume more time than retrieving data from the RF in the tile. Moreover, the control of the tiles is such that they can operate independently or synchronized in a lock-step mode. Thus, multiple tiles can cooperate to form a dynamically reconfigurable CGRA. A programmer can hence partition the 2D array of tiles into several, potentially differently sized, CGRAs that each run an independent thread. This provides very high flexibility to balance the available ILP inside threads with the TLP of the combined threads.

The Polymorphic Pipeline Array (PPA) [83] and similar designs [110] integrate multiple tightly-coupled ADRES-like CGRA cores into a larger array. Independent threads with limited amounts of ILP can run on the individual cores, but the resources of those individual cores can also be configured to form larger cores, on which threads with more ILP can then be executed. The utilization of the combined resources can be optimized dynamically by configuring the cores according to the available TLP and ILP at any point during the execution of a program.

Other architectures do not support (hardware) multi-threading within one CGRA core at all, like the Silicon Hive. The first solution to run multiple threads with these designs is to incorporate multiple CGRA accelerator cores in a System-on-Chip (SoC) [116]. The advantage is then that each accelerator can be customized for a certain class of loop kernels.

Alternatively, TLP can be converted into ILP and DLP by combining, at compile-time, kernels of multiple threads and by scheduling them together as one kernel, and by selecting the appropriate combination of scheduled kernels at run time [96].

3.3 Interconnects and Register Files

3.3.1 Connections

A wide range of connections can connect the ISs of a CGRA with each other, with the RFs, with other memories and with IO ports. Buses, point-to-point connections, and crossbars are all used in various combinations and in different topologies.

For example, some designs like MorphoSys and the most common ADRES and Silicon Hive designs feature a densely connected mesh-network of point-to-point interconnects in combination with sparser buses that connect ISs further apart. Thus the number of long power-hungry connections is limited. Multiple studies of point-to-point mesh-like interconnects as in Fig. 6 have been published in the past [13, 51, 55, 72]. Other designs like RaPiD feature a dense network of segmented buses. Typically the use of crossbars is limited to very small instances because large ones are too power-hungry. Fortunately, large crossbars are most often not needed, because many application kernels can be implemented as systolic algorithms, which map well onto mesh-like interconnects as found in systolic arrays [90].

Unlike crossbars and even busses, mesh-like networks of point-to-point connections scale better to large arrays without introducing too much delay or power consumption. For statically reconfigurable CGRAs, this is beneficial. Buses and

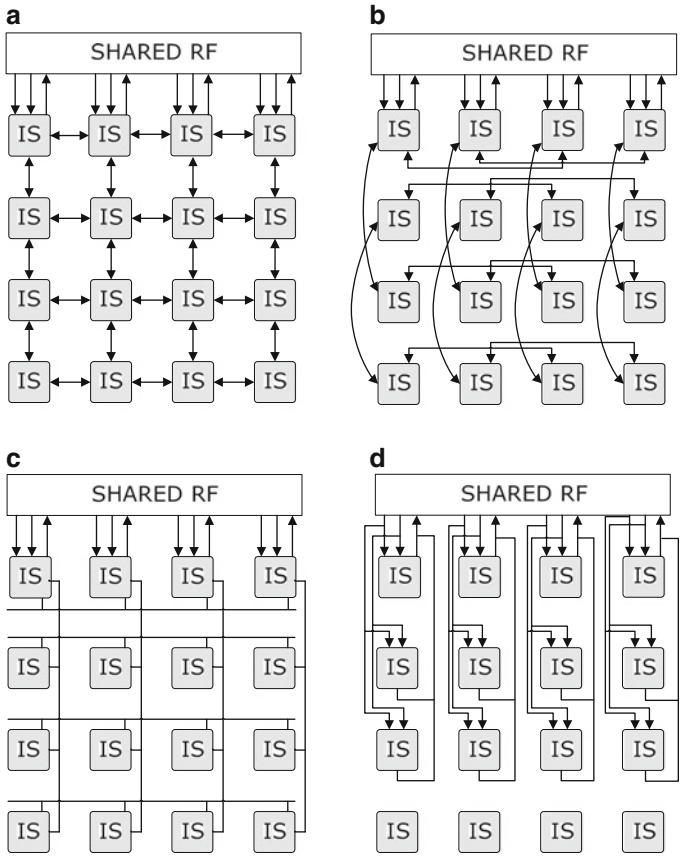


Fig. 6 Basic interconnects that can be combined. All bidirectional edges between two ISs denote that all outputs of one IS are connected to the inputs of the other IS and vice versa. Buses that connect all connected IS outputs to all connected IS inputs are shown as edges without arrows. (**a**) Nearest neighbor (nn), (**b**) next hop (nh), (**c**) buses (b), (**d**) extra (ex)

other long interconnects connect whole rows or columns to complement short-distance mesh-like interconnects. The negative effects that such long interconnects can have on power consumption or on obtainable clock frequency can be avoided by segmentation or by pipelining. In the latter case, pipelining latches are added along the connections or in between muxes and ISs. Our experience, as presented in Sect. 4.2.2 is that this pipelining will not necessarily lead to lower IPCs in CGRAs. This is different from out-of-order or VLIW architectures, where deeper pipelining increases the branch misprediction latency [99]. Instead at least some CGRA compilers succeed in exploiting the pipelining latches as temporary storage, rather than being hampered by them. This is the case in compiler techniques like [24, 73, 107] that are based on FPGA synthesis methods in which RFs and pipelining latches are treated as interconnection resources that span multiple cycles

instead of as explicit storage resources. This treatment naturally fits the 3D array modeling of resources along two spatial dimensions and one temporal dimension. Consequently, those compiler techniques can use pipelining latches for temporary storage as easily as they can exploit distributed RFs. This ability to use latches for temporary storage has been extended even beyond pipeline latches, for example to introduce retiming chains and shift registers in CGRA architectures [108].

As was already discussed in Sect. 3.1, the Remus architecture has an interconnect that lets data flow from top to bottom through an array. This fits streaming data applications, it simplifies the interconnect to potentially yield lower power consumption and higher clock speeds, and it enables to overlapping execution of one loop's epilogue with the next loop's prologue [125, 126].

3.3.2 Register Files

CGRA compilers place operations on ISs, thus also scheduling them, and route the data flow over the connections between the ISs. Those connections may be direct connections, or latched connections, or even connections that go through RFs. Therefore most CGRA compilers treat RFs not as temporary storage, but as interconnects that can span multiple cycles. Thus the RFs can be treated uniformly with the connections during routing. A direct consequence of this compiler approach is that the design space freedom of interconnects extends to the placement of RFs in between ISs. During the Design Space Exploration (DSE) for a specific CGRA instance in a CGRA design template such as the ADRES or Silicon Hive templates, both the real connections and the RFs have to be explored, and that has to be done together. Just like the number of real interconnect wires and their topology, the size of RFs, their location and their number of ports then contribute to the interconnectivity of the ISs. We refer to [13, 72] for DSEs that study both RFs and interconnects.

Besides their size and ports, another important aspect is that RFs can be rotating [94]. The power and delay overhead of rotation is very small in distributed RFs, simply because these RFs are small themselves. Still they can provide an important functionality. Consider a dynamically reconfigurable CGRA on which a loop is executed that iterates over x configurations, i.e., each iteration takes x cycles. That means that for a write port of an RF, every x cycles the same address bits get fetched from the configuration memory to configure the address set at that port. In other words, every x cycles a new value is being written into the register specified by that same address. This implies that values can stay in the same register for at most x cycles; then they are overwritten by a new value from the next iteration. In many loops, however, some values have a life time that spans more than x cycles, because it spans multiple loop iterations. To avoid having to insert additional data transfers in the loop schedules, rotating registers can be used. At the end of every iteration of the loop, all values in rotating registers rotate into another register to make sure that old values are copied to where they are not overwritten by newer values.

3.3.3 Predicates, Events and Tokens

To complete this overview on CGRA interconnects, we want to point out that it can be very useful to have interconnects of different widths. The data path width can be as small as 8 bits or as wide as 64 or 128 bits. The latter widths are typically used to pass SIMD data. However, as not all data is SIMD data, not all paths need to have the full width. Moreover, most CGRA designs and the code mapped onto them feature signals that are only one or a few bits wide, such as predicates or events or tokens. Using the full-width datapath for these narrow signals wastes resources. Hence it is often useful to add a second, narrow datapath for control signals like tokens or events and for predicates. How dense that narrow datapath has to be, depends on the type of loops one wants to run on the CGRA. For example, multimedia coding and decoding typically includes more conditional code than SDR baseband processing. Hence the design of, e.g., different ADRES architectures for multimedia and for SDR resulted in different predicate data paths being used, as illustrated in Sect. 4.2.1.

At this point, it should be noted that the use of predicates is fundamentally not that different from the use of events or tokens. In KressArray or PACT, events and tokens are used, amongst others, to determine at run time which data is selected to be used later in the loop. For example, for a C expression like `x + (a>b) ? y + z : y - z` one IS will first compute the addition `y+z`, one IS will compute the subtraction `y-z`, and one IS will compute the greater-than condition `a>b`. The result of the latter computation generates an event that will be fed to a multiplexor to select which of the two other computer values `y+z` and `y-z` is transferred to yet another IS on which the addition to `x` will be performed. Unlike the muxes in Fig. 2b that are controlled by bits fetched from the configuration memory, those event-controlled multiplexors are controlled by the data path.

In the ADRES architecture, the predicates guard the operations in ISs, and they serve as enable signals for RF write ports. Furthermore, they control special `select` operations that pass one of two input operands to the output port of an IS. Fundamentally, an event-controlled multiplexor performs exactly the same function as the `select` operation. So the difference between events or tokens and predicates is really only that the former term and implementation are used in dynamically scheduled designs, while the latter term is used in static schedules. As was already pointed out in Sect. 3.2.2, dual-issue and triggered instruction CGRAs combine the two forms to obtain higher resource utilization in the case of if-then-else structures.

3.4 Computational Resources

Issue slots are the computational resources of CGRAs. Over the last decade, numerous designs of such issue slots have been proposed, under different names, that include PEs, FUs, ALUs, and flexible computation components. Figure 7 depicts some of them. For all of the possible designs, it is important to know the

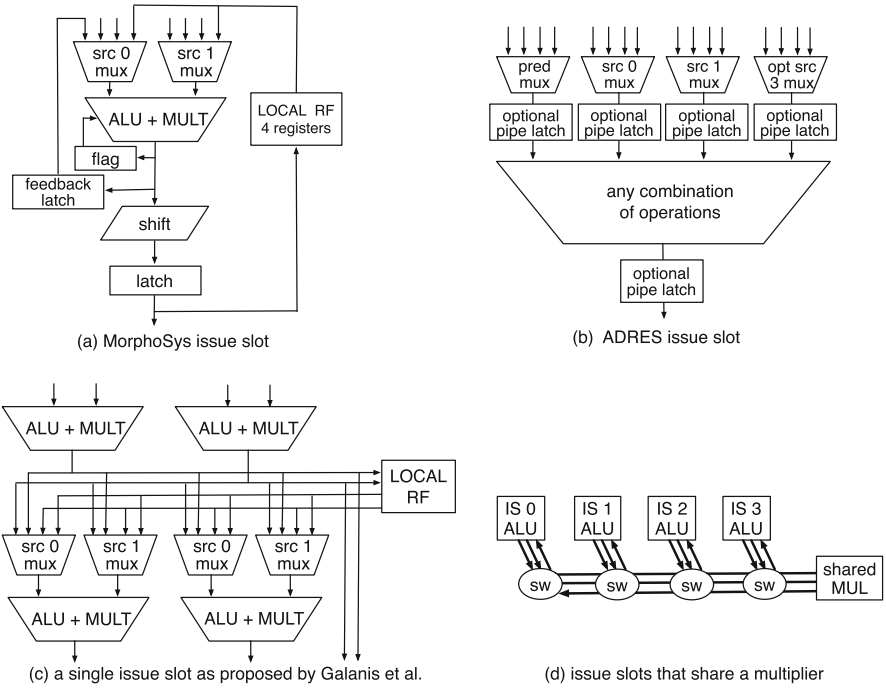


Fig. 7 Four different structures of ISs proposed in the literature. Part (**a**) displays a fixed MorphoSys IS, including its local RF. Part (**b**) displays the fully customizable ADRES IS, that can connect to shared or non-shared local RFs. Part (**c**) depicts the IS structure proposed by Galanis et al. [31], and (**d**) depicts a row of four RSPA ISs that share a multiplier [48]

context in which these ISs have to operate, such as the interconnects connecting them, the control type of the CGRA, etc.

Figure 7a depicts the IS of a MorphoSys CGRA. All 64 ISs in this homogeneous CGRA are identical and include their own local RF. This is no surprise, as the two MorphoSys SIMD modes (see Sect. 3.2.1) require that all ISs of a row or of a column execute the same instruction, which clearly implies homogeneous ISs.

In contrast, almost all features of an ADRES IS, as depicted in Fig. 7b, can be chosen at design time, and can be different for each IS in a CGRA that then becomes heterogeneous: the number of ports, whether or not there are latches between the multiplexors and the combinatorial logic that implements the operations, the set of operations supported by each IS, how the local registers file are connected to ISs and possibly shared between ISs, etc. As long as the design instantiates the ADRES template, the ADRES tool flow will be able to synthesize the architecture and to generate code for it. A similar design philosophy is followed by the Silicon Hive tools. Of course this requires more generic compiler techniques than those that generate code for the predetermined homogeneous ISs of, e.g., the MorphoSys CGRA. As we will discuss later in Sect. 3.6.2, this typically implies much longer

compilation times. Moreover, we need to note that while extensive specialization will typically benefit performance, it can also have negative effects, in particular on energy consumption [109].

Figure 7c depicts the IS proposed by Galanis et al. [31]. Again, all ISs are identical. In contrast to the MorphoSys design, however, these ISs consist of several ALUs and multipliers with direct connections between them and their local RFs. These direct connections within each IS can take care of a lot of data transfers, thus freeing time on the shared bus-based interconnect that connects all ISs. Thus, the local interconnect within each IS compensates for the lack of a scaling global interconnect. One advantage of this clustering approach is that the compiler can be tuned specifically for this combination of local and global connections and for the fact that it does not need to support heterogeneous ISs. Whether or not this type of design is more power-efficient than that of CGRAs with more design freedom and potentially more heterogeneity is unclear at this point in time. At least, we know of no studies from which, e.g., utilization numbers can be derived that allow us to compare the two approaches.

Some architectures combine the flexibility of heterogeneous ADRES ISs with clustering. For example, the CGRA Express [86] and the expression-grained reconfigurable array (EGRA) [3] feature heterogeneous clusters of relatively simple, fast ALUs. Within the clusters, those ALUs are chained by means of a limited number of latchless connections. Through careful design, the delay of those chains is comparable to the delay of other, more complex ISs on the CGRA that bound the clock frequency. So the chaining does not effect the clock frequency. It does allow, however, to execute multiple dependent operations within one clock cycle. It can therefore improve performance significantly. As the chains and clusters are composed of existing components such as ISs, buses, multiplexers and connections, these clustered designs do not really extend the design space of non-clustered CGRAs like ADRES. Still it can be useful to treat clusters as a separate design level in between the IS component level and the whole array architecture level, for example because it allows code generation algorithms in compilers to be tuned for there existence [86].

A specific type of clustering was proposed to handle floating-point arithmetic. While most CGRAs are limited to integer and fixed-point arithmetic, Lee at al. proposed to cluster two ISs to handle floating-point data [56]. In their design, both ISs in the cluster can operate independently on integer or fixed-point data, but they can also cooperate by means of a special direct interconnect between them. When they cooperate, one IS in the cluster consumes and handles the mantissas, while the other IS consumes and produces the exponents. As a single ISs can thus be used for both floating-point and integer computations, Lee et al. are able to achieve high utilization for integer applications, floating-point applications, as well as mixed applications.

Yet another type of clustering was proposed by Suh et al. [103]. They build a larger CGRA out of identical clusters, not to enable faster compilation or to obtain better performance, but to limit the time needed to perform design space explorations in order to reduce the time to market.

With respect to utilization, it is clear that the designs of Fig. 7a, b will only be utilized well if a lot of multiplications need to be performed. Otherwise, the area-consuming multipliers remain unused. To work around this problem, the sharing of large resources such as multipliers between ISs has been proposed in the RSPA CGRA design [48]. Figure 7d depicts one row of ISs that do not contain multipliers internally, but that are connected to a shared multiplier through switches and a shared bus. The advantage of this design, compared to an ADRES design in which each row features three pure ALU ISs and one ALU+MULT IS, is that this design allows the compiler to schedule multiplications in all ISs (albeit only one per cycle), whereas this scheduling freedom would be limited to one IS slot in the ADRES design. To allow this schedule freedom, however, a significant amount of resources in the form of switches and a special-purpose bus need to be added to the row. While we lack experimental data to back up this claim, we firmly believe that a similar increase in schedule freedom can be obtained in the aforementioned 3+1 ADRES design by simply extending an existing ADRES interconnect with a similar amount of additional resources. In the ADRES design, that extension would then also be beneficial to operations other than multiplications.

The optimal number of ISs for a CGRA depends on the application domain, on the reconfigurability, as well as on the IS functionality and on the DLP available in the form of subword parallelism. As illustrated in Sect. 4.2.2, a typical ADRES would consist of 4×4 ISs [12, 71]. TRIPS also features 4×4 ISs. MorphoSys provides 8×8 ISs, but that is because the DLP is implemented as SIMD over multiple ISs, rather than as subword parallelism within ISs.

In our experience, scaling dynamically reconfigurable CGRA architectures such as ADRES to very large arrays (8×8 or larger) is rarely useful, even with scalable interconnects like mesh or mesh-plus interconnects. Even in loops with high ILP, utilization drops significantly on such large arrays [77]. It is not clear what causes this lower utilization, and there might be several reasons. These include a lack of memory bandwidth, the possibility that the compiler techniques [24, 73] simply do not scale to such large arrays, or the fact that the relative connectivity in such large arrays is lower. Simply stated, when a mesh interconnects all ISs to their neighbors, each IS not on the side of the array is connected to 4 other ISs out of 16 in a 4×4 array, i.e., to 25% of all ISs, while it is connected to 4 out of 64 ISs on an 8×8 array, i.e., to 6.25% of all ISs.

Of course, large arrays can still be useful, e.g., if they can be partitioned in smaller arrays to run multiple threads in parallel, as discussed in Sect. 3.2.3. Also in CGRAs with limited connectivity, such as the Remus design introduced in Sect. 3.1, larger cores have proven useful.

3.5 Memory Hierarchies

CGRAs have a large number of ISs that need to be fed with data from the memory. Therefore the data memory sub-system is a crucial part of the CGRA design. Many

reconfigurable architectures feature multiple independent memory banks or blocks to achieve high data bandwidth.

The RAW architecture features an independent memory block in each tile for which Barua developed a method called modulo unrolling to disambiguate and assign data to different banks [5]. However, this technique can only handle array references through affine index expression on loop induction variables.

MorphoSys has a 256-bit wide frame buffer between the main memory and a reconfigurable array to feed data to the ISs operating in SIMD mode [60]. The efficient use of such a wide memory depends by and large on manual data placement and operation scheduling. Similar techniques for wide loads and stores have also been proposed in regular VLIW architectures for reducing power [91]. Exploiting that hardware requires manual data layout optimizations as well.

Both Silicon Hive and PACT feature distributed memory blocks without a crossbar. A Silicon Hive programmer has to specify the allocation of data to the memory for the compiler to bind the appropriate load/store operations to the corresponding memories. Silicon Hive also supports the possibility of interfacing the memory or system bus using FIFO interfaces. This is efficient for streaming processing but is difficult to interface when the data needs to be buffered on in case of data reuse.

The ADRES architecture template provides a parameterizable Data Memory Queue (DMQ) interface to each of the different single-ported, interleaved level-1 scratch-pad memory banks [23]. The DMQ interface is responsible for resolving bank access conflicts, i.e., when multiple load/store ISs would want to access the same bank at the same time. Connecting all load/store ISs to all banks through a conflict resolution mechanism allows maximal freedom for data access patterns and also maximal freedom on the data layout in memory. The potential disadvantage of such conflict resolution is that it increases the latency of load operations. In software pipelined code, however, increasing the individual latency of instructions most often does not have a negative effect on the schedule quality, because the compiler can hide those latencies in the software pipeline. In the main processor VLIW mode of an ADRES, the same memories are accessed in code that is not software-pipelined. So in that mode, the conflict resolution is disabled to obtain shorter access latencies.

Alternatively, a data cache can be added to the memory hierarchy to complement the scratch-pad memories. By letting the compiler partition the data over the scratch-pad memories and the data cache in an appropriate manner, high throughput can be obtained in the CGRA mode, as well as low latency in the VLIW mode [41, 45]. On a SoC with multiple CGRAs, the caches can be shared, and cache partitioning can be used to ensure that each CGRA obtains high throughput [116].

Furthermore, small local memories can be added exclusively to the CGRA to store data temporarily to lower the pressure on register files [124]. This way, memory hierarchies in CGRAs show many similarities to those found in modern Graphics Processing Units (GPUs). This should not be surprising. Samsung has already hinted that they plan to start using their Samsung Reconfigurable Processor designs in their future generations of GPUs [61]. Next to those GPU-like features, other features are adopted from high-level CPU designs. For example, data prefetch-

ing mechanisms have been proposed based on the history of the loop nests executed in CGRA mode [119].

3.6 Compiler Support

Two lines of research have to be discussed with respect to compiler support. The oldest one concerns the scheduler in the compiler back-end. This scheduler is responsible for determining where and when the operations of a loop body will be executed, and how data will flow through the interconnect from one IS to another. In some cases, it is also responsible for register allocation.

The other, more recent line of research concerns intermediate code generation and the optimization of intermediate code. This is the phase of the compiler that transforms the intermediate code to obtain loop bodies that are better suited to be mapped onto the targeted CGRAs, i.e., for which the scheduler can generate more efficient code.

3.6.1 Intermediate Code Generation and Optimization

In order to enable the back-end's scheduler to generate efficient code, i.e., code that utilizes the available resources of a CGRA well, some conditions need to be met: The loop bodies need to contain sufficient operations to utilize all the resources, the data dependencies between the operations need to enable high ILP, the memory access patterns should not create bottlenecks, as much as possible time has to be spent in inner loops, etc.

To obtain such loop bodies, compiler middle-ends apply loop transformations, such as flattening and unrolling [4]. For well-formed loop nests, such as affine ones, algebraic models are available, so-called polyhedral models [42], to reason about the degrees of freedom that a compiler has for reordering the operations in loop nests and to decide on the best transformation strategy for each loop. Such models have been used in parallelizing compilers of all kinds since about two decades [6]. The boundary conditions are somewhat different for CGRA compilers, however: entering and exiting CGRA mode results in considerably more overhead than doing so on general-purpose CPUs or VLIW processors and the number of available resources to be exploited through ILP is much higher. In Sect. 4.1, we will discuss these in more detail, when we discuss loop transformations for the ADRES CGRA template as a use case.

In CGRA programming environments that lack automated CGRA-specific loop optimization strategies, manual fine tuning of loops by rewriting their source code is therefore necessary to obtain acceptable code quality. Over the last couple of years, however, a range of automated loop optimization strategies has been developed that specifically target CGRAs and that can hence result in much more productive programming. Of those strategies, many rely on polyhedral models

and integer-linear programming for optimally merging affine or other perfect loop nests [64, 65, 68, 122, 123] and imperfectly nested loop nests [63, 121]. Others focus on determining the best loop unrolling parameters [98]. Whereas the aforementioned techniques focus on optimizing performance, some polyhedral techniques also consider battery conservation for mobile applications [88, 89].

3.6.2 CGRA Code Mapping and Scheduling Techniques

Apart from the specific algorithms used to schedule code, the major distinctions between CGRA schedulers relate to whether or not they support static scheduling, whether or not they support dynamic reconfiguration, whether or not they rely on special programming languages, and whether or not they are limited to specific hardware properties, or are instead flexible enough to support, e.g., very heterogeneous instances within an architecture template. Because most compiler research has been done to generate static schedules for CGRAs, we focus on those in this section. As already indicated in Sects. 3.2.1 and 3.2.2, many algorithms are based on FPGA placement and routing techniques [9] in combination with VLIW code generation techniques like modulo scheduling [54, 93] and hyperblock formation [70].

Whether or not compiler techniques rely on specific hardware properties is not always obvious in the literature, as not enough details are available in the descriptions of the techniques, and few techniques have been tried on a wide range of CGRA architectures. For that reason, it is very difficult to compare the efficiency (compilation time), the effectiveness (quality of generated code) and the flexibility (e.g., support for heterogeneity) of the different techniques.

The most widely applicable static scheduling techniques use different forms of Modulo Resource Routing Graphs (MRRGs). RRGs are time-space graphs, in which all resources (space dimension) are modeled with vertices. There is one such vertex per resource per cycle (time dimension) in the schedule being generated. Directed edges model the connections over which data values can flow from resource to resource. The schedule, placement, and routing problem then becomes a problem of mapping the Data Dependence Graph (DDG) of some loop body on the RRG. Scheduling refers to finding the right cycle to perform an operation (i.e., a DDG node) in the schedule, placement refers to finding the right IS (i.e., MRRG vertex) in that cycle, and routing refers to finding connections to transfer data from producing operations to consuming operations, i.e., to find a route in the MRRG for a DDG edge. In the case of a modulo scheduler, the modulo constraint is enforced by modeling all resource usage in the modulo time domain. This is done by modeling the appropriate modulo reservation tables [93] on top of the RRG, hence the name MRRG.

The granularity of its vertices depends on the precise compiler algorithm. One modulo graph embedding algorithm [81] for ADRES-like CGRAs models whole ISs or whole RFs with single vertices, whereas the simulated-annealing technique

in the DRESC [24, 73, 75] compiler that also targets ADRES instances models individual ports to ISs and RFs as separate vertices. Typically, fewer nodes that model larger components lead to faster compilation because the graph mapping problem operates on a smaller graph, but also to lower code quality because some combinations of resource usage cannot be modeled precisely. Moreover, models with fewer nodes also lack the flexibility to model a wide variation in resources, and hence can typically not model heterogeneous designs.

Several types of modulo schedulers for CGRAs exist. In the aforementioned DRESC, simulated annealing is used to explore different placement and routing options until a valid placement and routing of all operations and data dependencies is found. The cost function used during the simulated annealing is based on the total routing cost, i.e., the combined resource consumption of all placed operations and of all routed data dependencies. In this technique, a huge number of possible routes is evaluated, as a result of which the technique is very slow: Scheduling individual loops can take tens of minutes.

Later modulo scheduling techniques [29, 47, 78, 81, 82, 107] for ADRES-like CGRAs operate much more like (modulo) list schedulers [28]. These list-based CGRA schedulers still target MRRG representations of the hardware, and thus offer a large amount of flexibility in the architectures they support. Like DRESC, they rely heavily on routing costs. However, whereas DRESC first places DDG nodes in an MRRG and then tries to find good routes for the DDG edges connecting the nodes, these list schedulers work the opposite way. When one node of a DDG edge has already been placed (e.g., its sink node), a good place for the other node (the source node) is found by finding the cheapest possible path for the DDG edge in the MRRG, starting from the place of the already placed node. So in this case, the scheduler first identifies a good route for a DDG edge, and that route determines where its DDG node is placed. These schedulers are therefore called edge-centric schedulers. To find the best (i.e., cheapest) routes, they use a myriad of cost functions. These functions assign costs to nodes in the MRRG such that nodes that should not yet be occupied at a certain point during the iterative scheduling, e.g., because they model scarce resources that need to remain available for placing other DDG nodes later during the scheduling, are considered expensive and are hence avoided during the searches for cheapest routes. After every placement of a DDG node, the cost functions are updated in function of the next node to be placed, the nodes already placed and their places, the available resources, and the amounts and types of resources that will still be needed in the future. For some types of cost functions, these updates are simple, but for others they are very complex and computing them is time-consuming. The second [78] and third [107] generation edge-centric schedulers outperform the others in terms of generated code quality and compilation time because they offer a better balance between (1) cost function complexity; (2) priority functions, i.e., the order in which nodes are chosen to be placed onto the MRRG; and (3) their backtracing heuristics, i.e., the cases in which they unplace DDG nodes to try alternative places after a placement was found to block the generation of a valid, high quality schedule. The currently best scheduler even offers several modes of operation, in which fast, inaccurate cost functions are

tried first, and only if those fail, the slower, more accurate ones are used [107]. This delivers better code quality than DRESC can deliver, in particular for more heterogeneous CGRA designs, while requiring about 2 orders of magnitude less compilation time.

Several other graph-based modulo schedulers have been proposed that build on heavily simplified resource graphs to model the CGRA [19, 34, 35, 128]. Using different customized algorithms to find limited forms of sub-graph isomorphisms between a loop's DDG and the architecture resource graph, these schedulers can generate schedules very quickly. However, the limitation to certain forms of sub-graph isomorphisms can result in significantly lower code quality. Moreover, the simplified resource graphs cannot express many kinds of heterogeneity and features, such as varying places of latches in the CGRA. So these publications only consider rather homogeneous designs, in which only the supported instruction classes vary per IS. Some algorithms even seem to rely on the (in our view unrealistic) assumption that all operations have the same latency [34, 35].

Kim et al. presented a scheduler in which the generic NP-hard problem of modulo scheduling becomes tractable by imposing the constraint of following pre-calculated patternized rules [46]. As expected, the compilation times are improved by several orders of magnitude, at the cost of code quality (−30% compared to the already badly performing, first-generation edge-centric technique of [82]). Through its use of patternized rules, this scheduler is by construction limited to mostly homogeneous CGRAs. Lee et al. present an integer linear programming approach and a quantum-inspired evolutionary algorithm, both applied after an initial list scheduling [56]. Their mapping algorithms adopt high-level synthesis techniques combined with loop unrolling and software pipelining. They also target homogeneous targets.

MRRG-based compiler techniques are easily retargetable to a wide range of architectures, such as those of the ADRES template, and they can support many programming languages. Different architectures can simply be modeled with different MRRGs. It has even been demonstrated that by using the appropriate modulo constraints during the mapping of a DDG on a MRRG, compilers can generate a single code version that can be executed on CGRAs of different sizes [87]. This is particularly interesting for the PPA architecture that can switch dynamically between different array sizes [83] to support either a single big loop executing in a single threads or multiple smaller loops executing in parallel threads as discussed in Sect. 3.2.3. For CGRAs in which the hardware does not support parallel threads, the compiler can still merge the DDGs of multiple loops, and schedule them together, onto subpartitions of the CGRA [80]. That way, software-controlled multi-threading can still be achieved.

The aforementioned algorithms have been extended to not only consider the costs of utilized resources inside the CGRA during scheduling, but to also consider bank conflicts that may occur because of multiple memory accesses being scheduled in the same cycle [49, 50].

Many other CGRA compiler techniques have been proposed, most of which are restricted to specific architectures. Static reconfigurable architectures like RaPiD

and PACT have been targeted by compiler algorithms [16, 26, 114] based on placement and routing techniques that also map DDGs on RRGs. These techniques support subsets of the C programming language (no pointers, no structs, ...) and require the use of special C functions to program the IO in the loop bodies to be mapped onto the CGRA. The latter requirement follows from the specific IO support in the architectures and the modeling thereof in the RRGs.

For the MorphoSys architecture, with its emphasis on SIMD across ISs, compiler techniques have been developed for the SA-C language [111]. In this language the supported types of available parallelism are specified by means of loop language constructs. These constructs are translated into control code for the CGRA, which are mapped onto the ISs together with the DDGs of the loop bodies.

CGRA code generation techniques based on integer-linear programming have been proposed for the several architectures, both for spatial [2] and for temporal mapping [56, 127]. Basically, the ILP formulation consists of all the requirements or constraints that must be met by a valid schedule. This formulation is built from a DDG and a hardware description, and can hence be used to compile many source languages. It is unclear, however, to what extent the ILP formulation and its solution rely on specific architecture features, and hence to which extent it would be possible to retarget the ILP-formulation to different CGRA designs. A similar situation occurs for the constraint-based compilation method developed for the Silicon Hive architecture template [101], of which no detailed information is public. Furthermore, ILP-based compilation is known to be unreasonably slow. So in practice it can only be used for small loop kernels.

Code generation techniques for CGRAs based on instruction-selection pattern matching and list-scheduling techniques have also been proposed [30, 31]. It is unclear to what extent these techniques rely on a specific architecture because we know of no trial to use them for different CGRAs, but these techniques seem to rely heavily on the existence of a single shared-bus that connects ISs as depicted in Fig. 7c. Similarly, the static reconfiguration code generation technique by Lee et al. relies on CGRA rows consisting of identical ISs [58]. Because of this assumption, a two-step code generation approach can be used in which individual placements within rows are neglected in the first step, and only taken care of in the second step. The first step then instead focuses on optimizing the memory traffic.

Finally, compilation techniques have been developed that are really specialized for the TRIPS array layout and for its out-of-order execution [20].

4 Case Study: ADRES

This section presents a case study on one specific CGRA design template. The purpose of this study is to illustrate that it is non-trivial to compile and optimize code for CGRA targets, and to illustrate that within a design template, there is a need for hardware design exploration. This illustrates how both hardware and software

designers targeting CGRAs need a deep understanding of the interaction between the architecture features and the used compiler techniques.

ADRES [7, 11–13, 23, 24, 71, 73–75] is an architecture design template from which dynamically reconfigurable, statically scheduled CGRAs can be instantiated. In each instance, an ADRES CGRA is coupled tightly to a VLIW processor. This processor shares data and predicate RFs with the CGRA, as well as memory ports to a multi-banked scratch-pad memory as described in Sect. 3.1. The compiler-supported ISA of the design template provides instructions that are typically found in a load/store VLIW or RISC architecture, including arithmetic operations, logic operations, load/store operations, and predicate computing instructions. Additional domain-specific instructions, such as SIMD operations, are supported in the programming tools by means of intrinsics [102]. Local rotating and non-rotating, shared and private local RFs can be added to the CGRA as described in the previous sections, and connected through an interconnect consisting of muxes, buses and point-to-point connections that are specified completely by the designer. Thus, the ADRES architecture template is very flexible: it offers a high degree of design freedom, and it can be used to accelerate a wide range of loops.

4.1 Mapping Loops on ADRES CGRAs

The first part of this case study concerns the mapping of loops onto ADRES CGRAs, which are one of the most flexible CGRAs supporting a wide range of loops. This study illustrates that many loop transformations need to be applied carefully before mapping code onto ADRES CGRAs. We discuss the most important compiler transformations and, lacking a full-fledged loop-optimizing compiler, manual loop transformations that need to be applied to source code in order to obtain high performance and high efficiency. For other, less flexible CGRAs, the need for such transformations will even be higher because there will be more constraints on the loops to be mapped in the first place. Hence many of the discussed issues not only apply to ADRES CGRAs, but also to other CGRA architectures. We will conclude from this study that programming CGRAs with the existing compiler technology is not compatible with high programmer productivity.

4.1.1 Modulo Scheduling Algorithms for CGRAs

To exploit ILP in inner loops on VLIW architectures, compilers typically apply software pipelining by means of modulo scheduling [54, 93]. This is no different for ADRES CGRAs. In this section, we will not discuss the inner working of modulo scheduling algorithms. What we do discuss, are the consequences of using that technique for programming ADRES CGRAs.

After a loop has been modulo-scheduled, it consists of three phases: the prologue, the kernel and the epilogue. During the prologue, stages of the software-pipelined

loop gradually become active. Then the loop executes the kernel in a steady-state mode in which all software pipeline stages are active, and afterwards the stages are gradually disabled during the epilogue. In the steady-state mode, a new iteration is started after every II cycles, which stands for Initiation Interval. Fundamentally, every software pipeline stage is II cycles long. The total cycle count of a loop with *iter* iterations that is scheduled over ps software pipeline stages is then given by

$$cycles_{prologue} + II \cdot (iter - (ps - 1)) + cycles_{epilogue}. \quad (2)$$

In this formula, we neglect processor stalls because of, e.g., memory access conflicts or cache misses.

For loops with a high number of iterations, the term $II \cdot iter$ dominates this cycle count, and that is why modulo scheduling algorithms try to minimize II, thus increasing the IPC terms in Eq. (1).

The minimal II that modulo scheduling algorithms can reach is bound by $minII = \max(RecMII, ResMII)$. The first term, called resource-minimal II ($ResMII$) is determined by the resources required by a loop and by the resources provided by the architecture. For example, if a loop body contains nine multiplications, and there are only two ISs that can execute multiplications, then at least $\lceil 9/2 \rceil = 5$ cycles will be needed per iteration. The second term, called recurrence-minimal II ($RecMII$) depends on recurrent data dependencies in a loop and on instruction latencies. Fundamentally, if an iteration of a loop depends on the previous iteration through a dependency chain with accumulated latency $RecMII$, it is impossible to start that iteration before at least $RecMII$ cycles of the previous iteration have been executed.

The next section uses this knowledge to apply transformations that optimize performance according to Eq. (1). To do so successfully, it is important to know that ADRES CGRAs support only one thread, for which the processor has to switch from a non-CGRA operating mode to CGRA mode and back for each inner loop. So besides minimizing the cycle count of Eq. (2) to obtain higher IPCs in Eq. (1), it is also important to consider the terms $t_{p \to p+1}$ in Eq. (1).

4.1.2 Loop Transformations

Loop Unrolling

Loop unrolling and the induction variable optimizations that it enables can be used to minimize the number of iterations of a loop. When a loop body is unrolled x times, $iter$ decreases with a factor x, and $ResMII$ typically grows with a factor slightly less than x because of the induction variable optimizations and because of the ceiling operation in the computation of $ResMII$. By contrast, $RecMII$ typically remains unchanged or increases only a little bit as a result of the induction variable optimizations that are enabled after loop unrolling.

In resource-bound loops, $ResMII > RecMII$. Unrolling will then typically have little impact on the dominating term $II \cdot iter$ in Eq. (2). However, the prologue and the epilogue will typically become longer because of loop unrolling. Moreover, an unrolled loop will consume more space in the instruction memory, which might also have a negative impact on the total execution time of the whole application. So in general, unrolling resource-bound loops is unlikely to be very effective.

In recurrence-bound loops, $RecMII \cdot iter > ResMII \cdot iter$. The right hand side of this inequality will not increase by unrolling, while the left hand side will be divided by the unrolling factor x. As this improvement typically compensates for the longer prologue and epilogue, we can conclude that unrolling can be an effective optimization technique for recurrence-bound loops if the recurrences can be optimized with induction variable optimizations. This is no different for CGRAs than it is for VLIWs. However, for CGRAs with their larger number of ISs, it is more important because more loops are recurrence-bound.

Loop Fusion, Loop Interchange, Loop Combination and Data Context Switching

Fusing adjacent loops with the same number of iterations into one loop can also be useful, because fusing multiple recurrence-bound loops can result in one resource-bound loop, which will result in a lower overall execution time. Furthermore, less switching between operating modes takes place with fused loops, and hence the terms $t_{p \rightarrow p+1}$ are minimized. Furthermore, less prologues and epilogues need to be executed, which might also improve performance. This improvement will usually be limited, however, because the fused prologues and epilogues will rarely be much shorter than the sum of the original ones. Moreover, loop fusion does result in a loop that is bigger than any of the original loops, so it can only be applied if the configuration memory is big enough to fit the fused loop. If this is the case, less loop configurations need to be stored and possibly reloaded into the memory.

Interchanging an inner and outer loop serves largely the same purpose as loop fusion. As loop interchange does not necessarily result in larger prologues and epilogues, it can be even more useful, as can be the combining of nested loops into a single loop. Data-context switching [10] is a very similar technique that serves the same purpose. That technique has been used by Lee et al. for statically reconfigurable CGRAs as well [58], and in fact most of the loop transformations mentioned in this section can be used to target such CGRAs, as well as any other type of CGRA.

Live-In Variables

In our experience, there is only one caveat with the above transformations. The reason to be careful when applying them is that they can increase the number of live-in variables. A live-in variable is a variable that gets assigned a value before the

loop, which is consequently used in the loop. Live-in variables can be manifest in the original source code, but they can also result from compiler optimizations that are enabled by the above loop transformations, such as induction variable optimizations and loop-invariant code motion. When the number of live-in variables increases, more data needs to be passed from the non-loop code to the loop code, which might have a negative effect on $t_{p \rightarrow p+1}$. The existence and the scale of this effect will usually depend on the hardware mechanism that couples the CGRA accelerator to the main core. Possible such mechanisms are discussed in Sect. 3.1. In tightly-coupled designs like that of ADRES or Silicon Hive, passing a limited amount of values from the main CPU mode to the CGRA mode does not involve any overhead: the values are already present in the shared RF. However, if their number grows too big, there will not be enough room in the shared RF, which will result in much less efficient passing of data through memory. We have experienced this several times with loops in multimedia and SDR applications that were mapped onto our ADRES designs. So, even for tightly-coupled CGRA designs, the above loop transformations and the enabled optimizations need to be applied with great care.

Predication

The "basic" modulo scheduling techniques for CGRAs [24, 26, 29, 73, 75, 78, 81, 82, 107] only schedule loops that are free of control flow transfers. Hence any loop body that contains conditional statements first needs to be if-converted into hyperblocks by means of predication [70]. For this reason, many CGRAs, including ADRES CGRAs, support predication.

Hyperblock formation can result in very inefficient code if a loop body contains code paths that are executed rarely. All those paths contribute to $ResMII$ and potentially to $RecMII$. Hence even paths that get executed very infrequently can slow down a whole modulo-scheduled loop. Such loops can be detected with profiling, and if the data dependencies allow this, it can be useful to split these loops into multiple loops. For example, a first loop can contain the code of the frequently executed paths only, with a lower II than the original loop. If it turns out during the execution of this loop that in some iteration the infrequently executed code needs to be executed, the first loop is exited, and for the remaining iterations a second loop is entered that includes both the frequently and the infrequently executed code paths.

Alternatively, for some loops it is beneficial to have a so-called inspector loop with very small II to perform only the checks for all iterations. If none of the checks are positive, a second so-called executor loop is executed that includes all the computations except the checks and the infrequently executed paths. If some checks were positive, the original loop is executed.

One caveat with this loop splitting is that it causes code size expansion in the CGRA instruction memories. For power consumption reasons, these memories are kept as small as possible. This means that the local improvements obtained with the loop splitting need to be balanced with the total code size of all loops that need to share these memories.

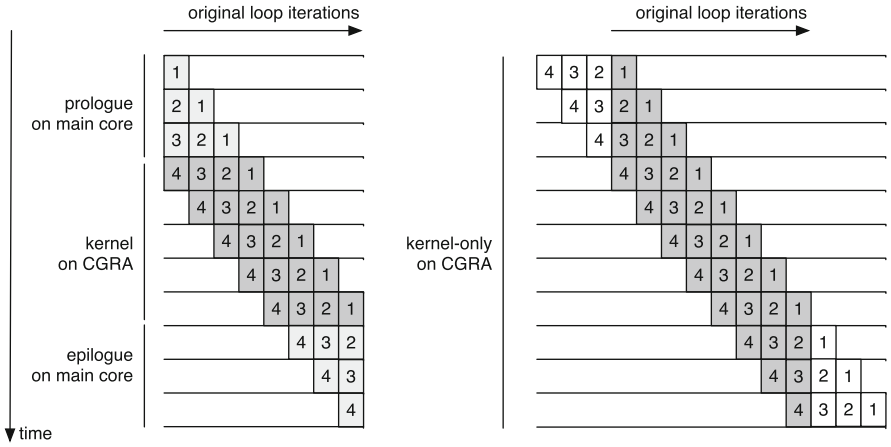


Fig. 8 On the left a traditional modulo-scheduled loop, on the right a kernel-only one. Each numbered box denotes one of four software pipeline stages, and each row denotes the concurrent execution of different stages of different iterations. Grayed boxes denote stages that actually get executed. On the left, the dark grayed boxes get executed on the CGRA accelerator, in which exactly the same code is executed every II cycles. The light grayed boxes are pipeline stages that get executed outside of the loop, in separate code that runs on the main processor. On the right, kernel-only code is shown. Again, the dark grey boxes are executed on the CGRA accelerator. So are the white boxes, but these get deactivated during the prologue and epilogue by means of predication

Kernel-Only Loops

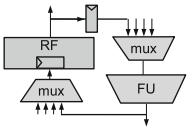
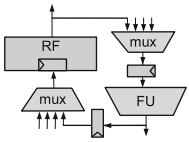
Predication can also be used to generate so-called kernel-only loop code. This is loop code that does not have separate prologue and epilogue code fragments. Instead the prologues and epilogues are included in the kernel itself, where predication is now used to guard whole software pipeline stages and to ensure that only the appropriate software pipeline stages are activated at each point in time. A traditional loop with a separate prologue and epilogue is compared to a kernel-only loop in Fig. 8. Three observations need to be made here.

The first observation is that kernel-only code is usually faster because the pipeline stages of the prologue and epilogue now get executed on the CGRA accelerator, which typically can do so at much higher IPCs than the main core. This is a major difference between (ADRES) CGRAs and VLIWs. On the latter, kernel-only loops are much less useful because all code runs on the same number of ISs anyway.

Secondly, while kernel-only code will be faster on CGRAs, more time is spent in the CGRA mode, as can be seen in Fig. 8. During the epilogue and prologue, the whole CGRA is active and thus consuming energy, but many ISs are not performing useful computations because they execute operations from inactive pipeline stages. Thus, kernel-only is not necessarily optimal in terms of energy consumption.

The third observation is that for loops where predication is used heavily to create hyperblocks, the use of predicates to support kernel-only code might over-stress

Table 1 Main differences between two studied ADRES CGRAs

	multimedia CGRA	**SDR CGRA**
# issue slots (FUs)	4x4	4x4
# load/store units	4	4
ld/st/mul latency	6/6/2 cycles	7/7/3 cycles
# local data RFs	12 (8 single-ported) of size 8	12 (8 single-ported) of size 4
data width	32	64
config. word width	896 bits	736 bits
ISA extensions	2-way SIMD, clipping, min/max	4-way SIMD, saturating arithm.
interconnect	Nearest Neighbor (NN) + 8 predicate buses	NN + next-hop + 8 data buses
pipelining		
power, clock, and area	91 mW at 300 MHz for 4mm^2	310mW at 400 MHz for 5.79mm^2

Power, clock and area include the CGRA and its configuration memory, the VLIW processor for non-loop code, including its 32K L1 I-cache, and the 32K 4-bank L1 data memory. These numbers are gate-level estimates

the predication support of the CGRA. In domains such as SDR, where the loops typically have no or very little conditional statements, this poses no problems. For applications that feature more complex loops, such as in many multimedia applications, this might create a bottleneck even when predicate speculation [97] is used. This is where the ADRES template proves to be very useful, as it allowed us to instantiate specialized CGRAs with varying predicate data paths, as can be seen in Table 1.

4.1.3 Data Flow Manipulations

The need for fine-tuning source code is well known in the embedded world. In practice, each compiler can handle some loop forms better than other forms. So when one is using a specific compiler for some specific VLIW architecture, it can be very beneficial to bring loops in the appropriate shape or form. This is no different when one is programming for CGRAs, including ADRES CGRAs.

Apart from the above transformations that relate to the modulo scheduling of loops, there are important transformations that can increase the "data flow" character of a loop, and thus contribute to the efficiency of a loop. Three C implementations of a Finite Impulse Response (FIR) filter in Fig. 9 provide an excellent example.

Figure 9a depicts a FIR implementation that is efficient for architectures with few registers. For architectures with more registers, the implementation depicted in Fig. 9b will usually be more efficient, as many memory accesses have been

a

```
const short c[15] = {-32, ..., 1216};
for (i = 0; i < nr; i++) {
  for(value = 0, j = 0; j < 15; j++)
    value += x[i+j]*c[j];
  r[i] = value;
}
```

b

```
const short c00 = -32, ..., c14 = 1216;
for (i = 0; i < nr; i++)
  r[i] = x[i+0]*c00 + x[i+1]*c01 + ... + x[i+14]*c14;
```

c

```
int i, value, d0, ..., d14;
const short c00 = -32, ..., c14 = 1216;
for (i = 0; i < nr+15; i++) {
  d0 = d1;  d1 = d2;  ... ; d13 = d14; d14 = x[i];
  value = c00 * d0 +  c01 * d1 + ... + c14 * d14;
  if (i >= 14) r[i - 14 ] = value;
}
```

Fig. 9 Three C versions of a FIR filter. (**a**) Original 15-tap FIR filter, (**b**) filter after loop unrolling, with hard-coded constants, (**c**) after redundant memory accesses are eliminated

Table 2 Number of execution cycles and memory accesses (obtained through simulation) for the FIR-filter versions compiled for the multimedia CGRA, and for the TI C64+ DSP

	Cycle count		Memory accesses	
Program	CGRA	TI C64+	CGRA	TI C64+
FIR (a)	11,828	1054	6221	1618
FIR (b)	1247	1638	3203	2799
FIR (c)	664	10,062	422	416

eliminated. Finally, the equivalent code in Fig. 9c contains only one load per outer loop iteration. To remove the redundant memory accesses, a lot of temporary variables had to be inserted, together with a lot of copy operations that implement a delay line. On regular VLIW architectures, this version would result in high register pressure and many copy operations to implement the data flow of those copy operations. Table 2 presents the compilation results for a 16-issue CGRA and for an 8-issue clustered TI C64+ VLIW. From the results, it is clear that the TI compiler could not handle the latter code version: its software-pipelining fails completely due to the high register pressure. When comparing the minimal cycle times obtained for the TI C64+ with those obtained for the CGRA, please note that the TI compiler applied SIMDization as much as it could, which is fairly orthogonal to scheduling and register allocation, but which the experimental CGRA compiler used for this experiment did not yet perform. By contrast, the CGRA compiler could optimize the code of Fig. 9c by routing the data of the copy operations over direct connections

between the CGRA ISs. As a result, the CGRA implementation becomes both fast and power-efficient at the same time.

This is a clear illustration of the fact that, lacking fully automated compiler optimizations, heavy performance-tuning of the source code can be necessary. The fact that writing efficient source code requires a deep understanding of the compiler internals and of the underlying architecture, and the fact that it frequently includes experimentation with various loop shapes, severely limits the programming productivity. This has to be considered a severe drawback of CGRAs architectures.

Moreover, as the FIR filter shows, the optimal source code for a CGRA target can be radically different than that for, e.g., a VLIW target. Consequently, the cost of porting code from other targets to CGRAs or vice versa, or of maintaining code versions for different targets (such as the main processor and the CGRA accelerator), can be high. This puts an additional limitation on programmer productivity.

4.2 ADRES Design Space Exploration

In this part of our case study, we discuss the importance and the opportunities for DSE within the ADRES template. First, we discuss some concrete ADRES instances that have been used for extensive experimentation, including the fabrication of working silicon samples. These examples demonstrate that very power-efficient CGRAs can be designed for specific application domains.

Afterwards, we will show some examples of DSE results with respect to some of the specific design options that were discussed in Sect. 3.

4.2.1 Example ADRES Instances

During the development of the ADRES tool chain and design, two main ADRES instances have been worked out. One was designed for multimedia applications [7, 71] and one for SDR baseband processing [11, 12]. Their main differences are presented in Table 1. Both architectures have a 64-entry data RF (half rotating, half non-rotating) that is shared with a unified three-issue VLIW processor that executes non-loop code. Thus this shared RF has six read ports and three write ports. Both CGRAs feature 16 FUs, of which four can access the memory (that consists of four single-ported banks) through a queue mechanism that can resolve bank conflicts. Most operations have latency one, with the exception of loads, stores, and multiplications. One important difference between the two CGRAs relates to their pipeline schemes, as depicted for a single IS (local RF and FU) in Table 1. As the local RFs are only buffered at their input, pipelining registers need to be inserted in the paths to and from the FUs in order to obtain the desired frequency targets as indicated in the table. The pipeline latches shown in Table 1 hence directly contribute in the maximization of the factor f_p in Eq. (1). Because the instruction sets and the target frequencies are different in both application domains, the SDR

CGRA has one more pipeline register than the multimedia CGRA, and they are located at different places in the design.

Traditionally, in VLIWs or in out-of-order superscalar processors, deeper pipelining results in higher frequencies but also in lower IPCs because of larger branch misprediction penalties. Following Eq. (1), this can result in lower performance. In CGRAs, however, this is not necessarily the case, as explained in Sect. 3.3.1. To illustrate this, Table 3 includes IPCs obtained when generating code for both CGRAs with and without the pipelining latches.

The benchmarks mapped onto the multimedia ADRES CGRA are a H.264AVC video decoder, a wavelet-based video decoder, an MPEG4 video coder, a black-and-white TIFF image filter, and a SHA-2 encryption algorithm. For each application at most the 10 hottest inner loops are included in the table. For the SDR ADRES CGRA, we selected two baseband modem benchmarks: one WLAN MIMO Channel Estimation and one that implements the remainder of a WLAN SISO receiver. All applications are implemented in standard ANSI C using all language features such as pointers, structures, different loop constructs (while, for, do-while), but not using dynamic memory management functions like `malloc` or `free`.

The general conclusions to be taken from the mapping results in Table 3 are as follows. (1) Very high IPCs are obtained at low power consumption levels of 91 and 310 mW and at relatively high frequencies of 300 and 400 MHz, given the standard cell 90 nm design. (2) Pipelining seems to be bad for performance only where the initiation interval is bound by $RecMII$, which changes with pipelining. (3) In some cases pipelining even improves the IPC.

Synthesizable VHDL is generated for both processors by a VHDL generator that generates VHDL code starting from the same XML architecture specification used to retarget the ANSI C compiler to different CGRA instances. A TSMC 90 nm standard cell GP CMOS (i.e. the General-Purpose technology version that is optimized for performance and active power, not for leakage power) technology was used to obtain the gate-level post-layout estimates for frequency, power and area in Table 1. More detailed results of these experiments are available in the literature for this SDR ADRES instance [11, 12], as well as for the multimedia instance [7, 71]. The SDR ADRES instance has also been produced in silicon in samples of a full SoC SDR chip [25]. The two ADRES cores on this SoC proved to be fully functional at 400 MHz, and the power consumption estimates have been validated.

One of the most interesting results is depicted in Fig. 10, which displays the average power consumption distribution over the ADRES SDR CGRA when the CGRA mode is active in the above SDR applications. Compared to VLIW processor designs, a much larger fraction of the power is consumed in the interconnects and in the FUs, while the configuration memory (which corresponds to an L1 VLIW instruction cache), the RFs and the data memory consume relatively little energy. This is particularly the case for the local RFs. This clearly illustrates that by focusing on regular loops and their specific properties, CGRAs can achieve higher performance and a higher power-efficiency than VLIWs. On the CGRA, most of the power is spent in the FUs and in the interconnects, i.e., on the actual computations and on the transfers of values from computation to computation. The latter two

Table 3 Results for the benchmark loops

Benchmark	CGRA	Loop	#ops	ResMII	Pipelined			Non-pipelined		
					RecMII	II	IPC	RecMII	II	IPC
AVC decoder	Multimedia	MBFilter1	70	5	2	6	11.7	1	6	11.7
		MBFilter2	89	6	7	9	9.9	6	8	11.1
		MBFilter3	40	3	3	4	10.0	2	3	13.3
		MBFilter4	105	7	2	9	11.7	1	9	11.7
		MotionComp	109	7	3	10	10.9	2	10	10.9
		FindFrameEnd	27	4	7	7	3.9	6	6	4.5
		IDCT1	60	4	2	5	12.0	1	5	12.0
		MBFilter5	87	6	3	7	12.4	2	7	12.4
		Memset	10	2	2	2	5.0	1	2	5.0
		IDCT2	38	3	2	3	12.7	1	3	12.7
		Average					10.0			10.5
Wavelet	Multimedia	Forward1	67	5	5	6	11.2	5	5	13.4
		Forward2	77	5	5	6	12.8	5	6	12.8
		Reverse1	73	5	2	6	12.2	1	6	12.2
		Reverse2	37	3	2	3	12.3	1	3	12.3
		Average					12.1			12.7
MPEG-4 encoder	Multimedia	MotionEst1	75	5	2	6	12.5	1	6	12.5
		MotionEst2	72	5	3	6	12.0	2	6	12.0
		TextureCod1	73	5	7	7	10.4	6	6	12.2
		CalcMBSAD	60	4	2	5	12.0	1	5	12.0
		TextureCod2	9	1	2	2	4.5	1	2	4.5
		TextureCod3	91	6	2	7	13.0	1	7	13.0
		TextureCod4	91	6	2	7	13.0	1	7	13.0
		TextureCod5	82	6	2	6	13.7	1	6	13.7
		TextureCod6	91	6	2	7	13.0	1	7	13.0
		MotionEst3	52	4	3	4	13.0	2	5	10.4
		Average					11.7			11.6
Tiff2BW	Multimedia	Main loop	35	3	2	3	11.7	1	3	11.7
SHA-2	Multimedia	Main loop	111	7	8	9	12.3	8	9	12.3
MIMO	SDR	Channel2	166	11	3	14	11.9	1	14	10.4
		Channel1	83	6	3	8	10.4	1	8	10.7
		SNR	75	5	4	6	12.5	2	6	12.5
		Average					11.6			11.2
WLAN	SDR	DemapQAM64	55	4	3	6	9.2	1	6	9.2
		64-point FFT	123	8	4	10	12.3	2	12	10.3
		Radix8 FFT	122	8	3	10	12.2	1	12	10.2
		Compensate	54	4	4	5	10.8	2	5	10.8
		DataShuffle	153	14	3	14	10.9	1	16	9.6
		Average					11.1			10.0

First, the target-version-independent number of operations (#ops) and the ResMII. Then for each target version the RecMII, the actually achieved II and IPC (counting SIMD operations as only one operation), and the compile time

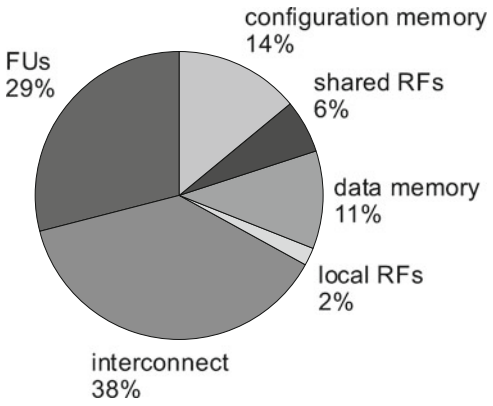


Fig. 10 Average power consumption distribution of the ADRES SDR CGRA in CGRA mode

aspects are really the fundamental parts of the computation to be performed, unlike the fetching of data or the fetching of code, which are merely side-effects of the fact that processors consist of control paths, data paths, and memories.

4.2.2 Design Space Exploration Example

Many DSEs have been performed within the ADRES template [7, 13, 18, 55, 71, 77]. We present one experimental result [55] here, not to present absolute numbers but to demonstrate the large impact on performance and on energy consumption that some design choices can have. In this experiment, a number of different interconnects have been explored for four microbenchmarks (each consisting of several inner loops): a MIMO SDR channel estimation, a Viterbi decoder, an Advanced Video Codec (AVC) motion estimation, and an AVC half-pixel interpolation filter. All of them have been compiled with the DRESC compiler for different architectures of which the interconnects are combinations of the four basic interconnects of Fig. 6, in which distributed RFs have been omitted for the sake of clarity.

Figure 11 depicts the relative performance and (estimated) energy consumption for different combinations of these basic interconnects. The names of the different architectures indicate which basic interconnects are included in its interconnect. For example, the architecture `b_nn_ex` includes the buses, nearest neighbor interconnects and extra connections to the shared RF. The lines connecting architectures in the charts of Fig. 11 connect the architectures on the Pareto fronts: these are the architectures that have an optimal combination of cycle count and energy consumption. Depending on the trade-off made by a designer between performance and energy consumption, he will select one architecture on that Pareto front.

The lesson to learn from these Pareto fronts is that relatively small architectural changes, in this case involving only the interconnect but not the ISs or the distributed RFs, can span a wide range of architectures in terms of performance and energy-efficiency. When designing a new CGRA or choosing for an existing one, it is hence

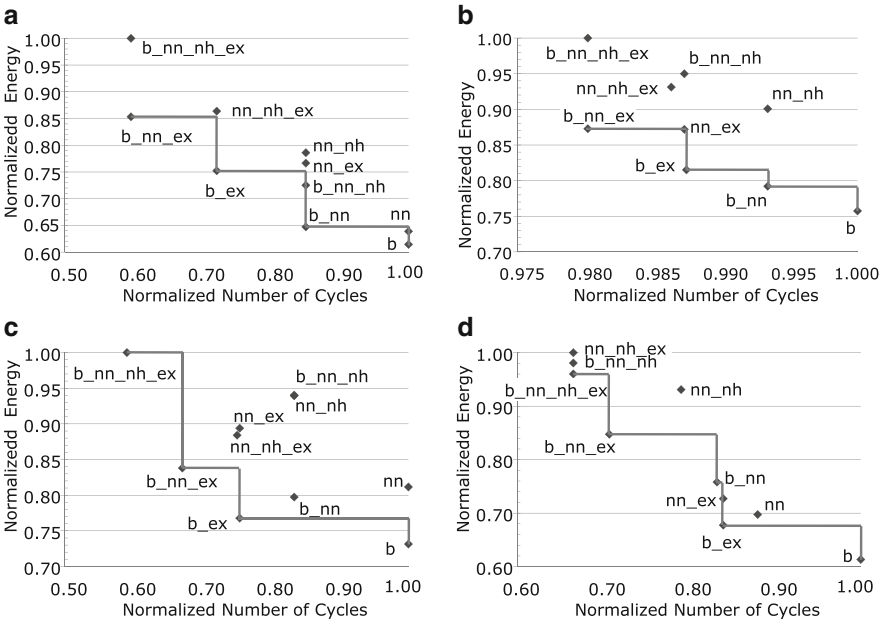


Fig. 11 DSE results for four microbenchmarks on 4 × 4 CGRAs with fixed ISs and fixed RFs, but with varying interconnects. (**a**) MIMO, (**b**) AVC interpolation, (**c**) Viterbi, (**d**) AVC motion estimation

absolutely necessary to perform a good DSE that covers ISA, ISs, interconnect and RFs. Because of the large design space, this is far from trivial.

5 Conclusions

This chapter on CGRA architectures presented a discussion of the CGRA processor design space as an accelerator for inner loops of DSP-like applications such as software-defined radios and multimedia processing. A range of options for many design features and design parameters has been related to power consumption, performance, and flexibility. In a use case, the need for design space exploration and for advanced compiler support and manual high-level code tuning have been demonstrated. The above discussions and demonstration support the following main conclusions. Firstly, CGRAs can provide an excellent alternative for VLIWs, providing better performance and better energy efficiency. Secondly, design space exploration is needed to achieve those goals. Finally, existing compiler support needs to be improved, and until that happens, programmers need to have a deep understanding of the targeted CGRA architectures and their compilers in order to manually tune their source code. This can significantly limit programmer productivity.

6 Further Reading

For further reading, the historic development of the ADRES architecture is interesting, from the first academic conception of the architecture and its initial compiler support [73–75], over the first fabricated prototypes [12, 25], to their commercial derivatives [44]. The historic development of appropriate compiler models [24] and scheduling techniques [78, 81, 82, 107] to achieve both high code quality and fast compilation is interesting as well.

Some of the more interesting recent research directions include power optimization by means of adaptive and multiple V_{dd}'s [33, 115, 120], architectural and compiler support for nested loops [57, 121–123], more dynamic control [126], and support for thread-level parallelism [80, 83, 110].

For pointers for further reading on other specific design aspects of CGRAs, we refer to the corresponding sections in this chapter, which include plenty of references.

References

1. Abnous, A., Christensen, C., Gray, J., Lenell, J., Naylor, A., Bagherzadeh, N.: Design and implementation of the "Tiny RISC" microprocessor. Microprocessors & Microsystems **16**(4), 187–193 (1992)
2. Ahn, M., Yoon, J.W., Paek, Y., Kim, Y., Kiemb, M., Choi, K.: A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In: DATE '06: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 363–368 (2006)
3. Ansaloni, G., Bonzini, P., Pozzi, L.: EGRA: A coarse grained reconfigurable architectural template. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **19**(6), 1062–1074 (2011)
4. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Comput. Surv. **26**(4), 345–420 (1994)
5. Barua, R.: Maps: a compiler-managed memory system for software-exposed architectures. Ph.D. thesis, Massachusetts Institute of Technology (2000)
6. Benabderrahmane, M.W., Pouchet, L.N., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction, CC'10/ETAPS'10, pp. 283–303. Springer-Verlag, Berlin, Heidelberg (2010)
7. Berekovic, M., Kanstein, A., Mei, B., De Sutter, B.: Mapping of nomadic multimedia applications on the ADRES reconfigurable array processor. Microprocessors & Microsystems **33**(4), 290–294 (2009)
8. van Berkel, k., Heinle F. amd Meuwissen, P., Moerman, K., Weiss, M.: Vector processing as an enabler for software-defined radio in handheld devices. EURASIP Journal on Applied Signal Processing **2005**(16), 2613–2625 (2005)
9. Betz, V., Rose, J., Marguardt, A.: Architecture and CAD for Deep-Submicron FPGAs. Kluwer Academic Publishers (1999)
10. Bondalapati, K.: Parallelizing DSP nested loops on reconfigurable architectures using data context switching. In: DAC '01: Proceedings of the 38th annual Design Automation Conference, pp. 273–276 (2001)

11. Bougard, B., De Sutter, B., Rabou, S., Novo, D., Allam, O., Dupont, S., Van der Perre, L.: A coarse-grained array based baseband processor for 100Mbps+ software defined radio. In: DATE '08: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 716–721 (2008)
12. Bougard, B., De Sutter, B., Verkest, D., Van der Perre, L., Lauwereins, R.: A coarse-grained array accelerator for software-defined radio baseband processing. IEEE Micro **28**(4), 41–50 (2008). http://doi.ieeecomputersociety.org/10.1109/MM.2008.49
13. Bouwens, F., Berekovic, M., Gaydadjiev, G., De Sutter, B.: Architecture enhancements for the ADRES coarse-grained reconfigurable array. In: HiPEAC '08: Proceedings of the International Conference on High-Performance Embedded Architectures and Compilers, pp. 66–81 (2008)
14. Burns, G., Gruijters, P.: Flexibility tradeoffs in SoC design for low-cost SDR. Proceedings of SDR Forum Technical Conference (2003)
15. Burns, G., Gruijters, P., Huiskens, J., van Wel, A.: Reconfigurable accelerators enabling efficient SDR for low cost consumer devices. Proceedings of SDR Forum Technical Conference (2003)
16. Cardoso, J.M.P., Weinhardt, M.: XPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture. In: FPL '02: Proceedings of the 12th International Conference on Field-Programmable Logic and Applications, pp. 864–874 (2002)
17. Cervero, T.: Analysis, implementation and architectural exploration of the H.264/AVC decoder onto a reconfigurable architecture. Master's thesis, Universidad de Los Palmas de Gran Canaria (2007)
18. Cervero, T., Kanstein, A., López, S., De Sutter, B., Sarmiento, R., Mignolet, J.Y.: Architectural exploration of the H.264/AVC decoder onto a coarse-grain reconfigurable architecture. In: Proceedings of the International Conference on Design of Circuits and Integrated Systems (2008)
19. Chen, L., Mitra, T.: Graph minor approach for application mapping on CGRAs. ACM Trans. on Reconf. Technol. and Systems **7**(3), 21 (2014)
20. Coons, K.E., Chen, X., Burger, D., McKinley, K.S., Kushwaha, S.K.: A spatial path scheduling algorithm for EDGE architectures. In: ASPLOS '06: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 129–148 (2006)
21. Corporaal, H.: Microprocessor Architectures from VLIW to TTA. John Wiley (1998)
22. Cronquist, D., Franklin, P., Fisher, C., Figueroa, M., Ebeling, C.: Architecture design of reconfigurable pipelined datapaths. In: Proceedings of the Twentieth Anniversary Conference on Advanced Research in VLSI (1999)
23. De Sutter, B., Allam, O., Raghavan, P., Vandebriel, R., Cappelle, H., Vander Aa, T., Mei, B.: An efficient memory organization for high-ILP inner modem baseband SDR processors. Journal of Signal Processing Systems **61**(2), 157–179 (2010)
24. De Sutter, B., Coene, P., Vander Aa, T., Mei, B.: Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In: LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 151–160 (2008)
25. Derudder, V., Bougard, B., Couvreur, A., Dewilde, A., Dupont, S., Folens, L., Hollevoet, L., Naessens, F., Novo, D., Raghavan, P., Schuster, T., Stinkens, K., Weijers, J.W., Van der Perre, L.: A 200Mbps+ 2.14nJ/b digital baseband multi processor system-on-chip for SDRs. In: Proceedings of the Symposium on VLSI Systems, pp. 292–293 (2009)
26. Ebeling, C.: Compiling for coarse-grained adaptable architectures. Tech. Rep. UW-CSE-02-06-01, University of Washington (2002)
27. Ebeling, C.: The general RaPiD architecture description. Tech. Rep. UW-CSE-02-06-02, University of Washington (2002)

28. Fisher, J., Faraboschi, P., Young, C.: Embedded Computing, A VLIW Approach to Architecture, Compilers and Tools. Morgan Kaufmann (2005)
29. Friedman, S., Carroll, A., Van Essen, B., Ylvisaker, B., Ebeling, C., Hauck, S.: SPR: an architecture-adaptive CGRA mapping tool. In: FPGA '09: Proceeding of the ACM/SIGDA International symposium on Field Programmable Gate Arrays, pp. 191–200. ACM, New York, NY, USA (2009)
30. Galanis, M.D., Milidonis, A., Theodoridis, G., Soudris, D., Goutis, C.E.: A method for partitioning applications in hybrid reconfigurable architectures. Design Automation for Embedded Systems **10**(1), 27–47 (2006)
31. Galanis, M.D., Theodoridis, G., Tragoudas, S., Goutis, C.E.: A reconfigurable coarse-grain data-path for accelerating computational intensive kernels. Journal of Circuits, Systems and Computers pp. 877–893 (2005)
32. Gebhart, M., Maher, B.A., Coons, K.E., Diamond, J., Gratz, P., Marino, M., Ranganathan, N., Robatmili, B., Smith, A., Burrill, J., Keckler, S.W., Burger, D., McKinley, K.S.: An evaluation of the TRIPS computer system. In: ASPLOS '09: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 1–12 (2009)
33. Gu, J., Yin, S., Liu, L., Wei, S.: Energy-aware loops mapping on multi-v_{dd} CGRAs without performance degradation. In: 22nd Asia and South Pacific Design Automation Conference, ASP-DAC 2017, Chiba, Japan, January 16–19, 2017, pp. 312–317 (2017)
34. Hamzeh, M., Shrivastava, A., Vrudhula, S.: EPIMap: using epimorphism to map applications on CGRAs. In: Proc. 49th Annual Design Automation Conf., pp. 1284–1291 (2012)
35. Hamzeh, M., Shrivastava, A., Vrudhula, S.B.K.: REGIMap: register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). In: Proc. Annual Design Automation Conf., pp. 1–10 (2013)
36. Hamzeh, M., Shrivastava, A., Vrudhula, S.B.K.: Branch-aware loop mapping on CGRAs. In: The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1–5, 2014, pp. 107:1–107:6 (2014)
37. Hartenstein, R., Herz, M., Hoffmann, T., Nageldinger, U.: Mapping applications onto reconfigurable KressArrays. In: Proceedings of the 9th International Workshop on Field Programmable Logic and Applications (1999)
38. Hartenstein, R., Herz, M., Hoffmann, T., Nageldinger, U.: Generation of design suggestions for coarse-grain reconfigurable architectures. In: FPL '00: Proceedings of the 10th International Workshop on Field Programmable Logic and Applications (2000)
39. Hartenstein, R., Hoffmann, T., Nageldinger, U.: Design-space exploration of low power coarse grained reconfigurable datapath array architectures. In: Proceedings of the International Workshop - Power and Timing Modeling, Optimization and Simulation (2000)
40. Hartmann, M., Pantazis, V., Vander Aa, T., Berekovic, M., Hochberger, C., De Sutter, B.: Still image processing on coarse-grained reconfigurable array architectures. In: Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia, pp. 67–72 (2007)
41. Jang, C., Kim, J., Lee, J., Kim, H.S., Yoo, D., Kim, S., Kim, H.S., Ryu, S.: An instruction-scheduling-aware data partitioning technique for coarse-grained reconfigurable architectures. In: Proc. ACM SIGPLAN/SIGBED Conf. Languages, compilers, and tools for embedded systems (LCTES), pp. 151–160 (2011)
42. Karp, R.M., Miller, R.E., Winograd, S.: The organization of computations for uniform recurrence equations. J. ACM **14**(3), 563–590 (1967)
43. Kessler, C.W.: Compiling for VLIW DSPs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)

44. Kim, C., Chung, M., Cho, Y., Konijnenburg, M., Ryu, S., Kim, J.: ULP-SRP: Ultra low power Samsung Reconfigurable Processor for biomedical applications. In: 2012 International Conference on Field-Programmable Technology, pp. 329–334 (2012). DOI 10.1109/FPT.2012.6412157
45. Kim, H.s., Yoo, D.h., Kim, J., Kim, S., Kim, H.s.: An instruction-scheduling-aware data partitioning technique for coarse-grained reconfigurable architectures. In: LCTES '11: Proceedings of the 2011 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems, pp. 151–160 (2011)
46. Kim, W., Choi, Y., Park, H.: Fast modulo scheduler utilizing patternized routes for coarse-grained reconfigurable architectures. ACM Trans. on Architec. and Code Optim. **10**(4), 1–24 (2013)
47. Kim, W., Yoo, D., Park, H., Ahn, M.: SCC based modulo scheduling for coarse-grained reconfigurable processors. In: Proc. Conf. on Field-Programmable Technology, pp. 321–328 (2012)
48. Kim, Y., Kiemb, M., Park, C., Jung, J., Choi, K.: Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In: DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 12–17 (2005)
49. Kim, Y., Lee, J., Shrivastava, A., Paek, Y.: Operation and data mapping for CGRAs with multi-bank memory. In: LCTES '10: Proceedings of the 2010 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 17–25 (2010)
50. Kim, Y., Lee, J., Shrivastava, A., Yoon, J., Paek, Y.: Memory-aware application mapping on coarse-grained reconfigurable arrays. In: HiPEAC '10: Proceedings of the 2010 International Conference on High Performance Embedded Architectures and Compilers, pp. 171–185 (2010)
51. Kim, Y., Mahapatra, R.: A new array fabric for coarse-grained reconfigurable architecture. In: Proceedings of the IEEE EuroMicro Conference on Digital System Design, pp. 584–591 (2008)
52. Kim, Y., Mahapatra, R., Park, I., Choi, K.: Low power reconfiguration technique for coarse-grained reconfigurable architecture. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **17**(5), 593–603 (2009)
53. Kim, Y., Mahapatra, R.N.: Dynamic Context Compression for Low-Power Coarse-Grained Reconfigurable Architecture. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **18**(1), 15–28 (2010)
54. Lam, M.S.: Software pipelining: an effective scheduling technique for VLIW machines. In: Proc. PLDI, pp. 318–327 (1988)
55. Lambrechts, A., Raghavan, P., Jayapala, M., Catthoor, F., Verkest, D.: Energy-aware interconnect optimization for a coarse grained reconfigurable processor. In: Proceedings of the International Conference on VLSI Design, pp. 201–207 (2008)
56. Lee, G., Choi, K., Dutt, N.: Mapping multi-domain applications onto coarse-grained reconfigurable architectures. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **30**(5), 637–650 (2011)
57. Lee, J., Seo, S., Lee, H., Sim, H.U.: Flattening-based mapping of imperfect loop nests for CGRAs. In: 2014 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2014, Uttar Pradesh, India, October 12–17, 2014, pp. 9:1–9:10 (2014)
58. Lee, J.e., Choi, K., Dutt, N.D.: An algorithm for mapping loops onto coarse-grained reconfigurable architectures. In: LCTES '03: Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 183–188 (2003)
59. Lee, L.H., Moyer, B., Arends, J.: Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In: ISLPED '99: Proceedings of the 1999 International symposium on Low power electronics and design, pp. 267–269. ACM, New York, NY, USA (1999)

60. Lee, M.H., Singh, H., Lu, G., Bagherzadeh, N., Kurdahi, F.J., Filho, E.M.C., Alves, V.C.: Design and implementation of the MorphoSys reconfigurable computing processor. J. VLSI Signal Process. Syst. **24**(2/3), 147–164 (2000)
61. Lee, W.J., Woo, S.O., Kwon, K.T., Son, S.J., Min, K.J., Jang, G.J., Lee, C.H., Jung, S.Y., Park, C.M., Lee, S.H.: A scalable GPU architecture based on dynamically reconfigurable embedded processor. In: Proc. ACM Conference on High-Performance Graphics (2011)
62. Liang, S., Yin, S., Liu, L., Guo, Y., Wei, S.: A coarse-grained reconfigurable architecture for compute-intensive MapReduce acceleration. Computer Architecture Letters **15**(2), 69–72 (2016)
63. Lin, X., Yin, S., Liu, L., Wei, S.: Exploiting parallelism of imperfect nested loops with sibling inner loops on coarse-grained reconfigurable architectures. In: 21st Asia and South Pacific Design Automation Conference, ASP-DAC 2016, Macao, January 25–28, 2016, pp. 456–461 (2016)
64. Liu, D., Yin, S., Liu, L., Wei, S.: Mapping multi-level loop nests onto CGRAs using polyhedral optimizations. IEICE Transactions **98-A**(7), 1419–1430 (2015)
65. Liu, D., Yin, S., Peng, Y., Liu, L., Wei, S.: Optimizing spatial mapping of nested loop for coarse-grained reconfigurable architectures. IEEE Trans. VLSI Syst. **23**(11), 2581–2594 (2015)
66. Liu, L., Deng, C., Wang, D., Zhu, M., Yin, S., Cao, P., Wei, S.: An energy-efficient coarse-grained dynamically reconfigurable fabric for multiple-standard video decoding applications. In: Proceedings of the IEEE 2013 Custom Integrated Circuits Conference, pp. 1–4 (2013). https://doi.org/10.1109/CICC.2013.6658434
67. Liu, L., Wang, D., Chen, Y., Zhu, M., Yin, S., Wei, S.: An implementation of multiple-standard video decoder on a mixed-grained reconfigurable computing platform. IEICE Transactions **99-D**(5), 1285–1295 (2016)
68. Madhu, K.T., Das, S., Nalesh, S., Nandy, S.K., Narayan, R.: Compiling HPC kernels for the REDEFINE CGRA. In: 17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24–26, 2015, pp. 405–410 (2015)
69. Mahadurkar, M., Merchant, F., Maity, A., Vatwani, K., Munje, I., Gopalan, N., Nandy, S.K., Narayan, R.: Co-exploration of NLA kernels and specification of compute elements in distributed memory CGRAs. In: XIVth International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2014, Agios Konstantinos, Samos, Greece, July 14–17, 2014, pp. 225–232 (2014)
70. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective compiler support for predicated execution using the hyperblock. In: MICRO 25: Proceedings of the 25th annual International symposium on Microarchitecture, pp. 45–54. IEEE Computer Society Press, Los Alamitos, CA, USA (1992)
71. Mei, B., De Sutter, B., Vander Aa, T., Wouters, M., Kanstein, A., Dupont, S.: Implementation of a coarse-grained reconfigurable media processor for AVC decoder. Journal of Signal Processing Systems **51**(3), 225–243 (2008)
72. Mei, B., Lambrechts, A., Verkest, D., Mignolet, J.Y., Lauwereins, R.: Architecture exploration for a reconfigurable architecture template. IEEE Design and Test of Computers **22**(2), 90–101 (2005)
73. Mei, B., Vernalde, S., Verkest, D., Lauwereins, R.: Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study. In: DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 1224–1229 (2004)
74. Mei, B., Vernalde, S., Verkest, D., Man, H.D., Lauwereins, R.: ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In: Proc. of Field-Programmable Logic and Applications, pp. 61–70 (2003)

75. Mei, B., Vernalde, S., Verkest, D., Man, H.D., Lauwereins, R.: Exploiting loop-level parallelism for coarse-grained reconfigurable architecture using modulo scheduling. IEE Proceedings: Computer and Digital Techniques **150**(5) (2003)
76. Merchant, F., Maity, A., Mahadurkar, M., Vatwani, K., Munje, I., Krishna, M., Nalesh, S., Gopalan, N., Raha, S., Nandy, S.K., Narayan, R.: Micro-architectural enhancements in distributed memory CGRAs for LU and QR factorizations. In: 28th International Conference on VLSI Design, VLSID 2015, Bangalore, India, January 3–7, 2015, pp. 153–158 (2015)
77. Novo, D., Schuster, T., Bougard, B., Lambrechts, A., Van der Perre, L., Catthoor, F.: Energy-performance exploration of a CGA-based SDR processor. Journal of Signal Processing Systems (2009)
78. Oh, T., Egger, B., Park, H., Mahlke, S.: Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In: LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 21–30 (2009)
79. PACT XPP Technologies: XPP-III Processor Overview White Paper (2006)
80. Pager, J., Jeyapaul, R., Shrivastava, A.: A software scheme for multithreading on CGRAs. ACM Trans. Embedded Comput. Syst. **14**(1), 19 (2015)
81. Park, H., Fan, K., Kudlur, M., Mahlke, S.: Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures. In: CASES '06: Proceedings of the 2006 International Conference on Compilers, architecture and synthesis for embedded systems, pp. 136–146 (2006)
82. Park, H., Fan, K., Mahlke, S.A., Oh, T., Kim, H., Kim, H.S.: Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In: PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 166–176 (2008)
83. Park, H., Park, Y., Mahlke, S.: Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In: MICRO '09: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 370–380 (2009)
84. Park, H., Park, Y., Mahlke, S.A.: A dataflow-centric approach to design low power control paths in CGRAs. In: Proc. IEEE Symp. on Application Specific Processors, pp. 15–20 (2009)
85. Park, J., Park, Y., Mahlke, S.A.: Efficient execution of augmented reality applications on mobile programmable accelerators. In: Proc. Conf. on Field-Programmable Technology, pp. 176–183 (2013)
86. Park, Y., Park, H., Mahlke, S.: CGRA express: accelerating execution using dynamic operation fusion. In: CASES '09: Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 271–280 (2009)
87. Park, Y., Park, H., Mahlke, S., Kim, S.: Resource recycling: putting idle resources to work on a composable accelerator. In: CASES '10: Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, pp. 21–30 (2010)
88. Peng, Y., Yin, S., Liu, L., Wei, S.: Battery-aware loop nests mapping for CGRAs. IEICE Transactions **98-D**(2), 230–242 (2015)
89. Peng, Y., Yin, S., Liu, L., Wei, S.: Battery-aware mapping optimization of loop nests for CGRAs. In: The 20th Asia and South Pacific Design Automation Conference, ASP-DAC 2015, Chiba, Japan, January 19–22, 2015, pp. 767–772 (2015)
90. Petkov, N.: Systolic Parallel Processing. North Holland Publishing (1992)
91. P. Raghavan, A. Lambrechts, M. Jayapala, F. Catthoor, D. Verkest, Corporaal, H.: Very wide register: An asymmetric register file organization for low power embedded processors. In: DATE '07: Proceedings of the Conference on Design, Automation and Test in Europe (2007)
92. Rákossy, Z.E., Merchant, F., Aponte, A.A., Nandy, S.K., Chattopadhyay, A.: Efficient and scalable CGRA-based implementation of column-wise Givens rotation. In: IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2014, Zurich, Switzerland, June 18–20, 2014, pp. 188–189 (2014)
93. Rau, B.R.: Iterative modulo scheduling. Tech. rep., Hewlett-Packard Lab: HPL-94-115 (1995)

94. Rau, B.R., Lee, M., Tirumalai, P.P., Schlansker, M.S.: Register allocation for software pipelined loops. In: PLDI '92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, pp. 283–299 (1992)
95. Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Burger, D., Keckler, S.W., Moore, C.R.: Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. SIGARCH Comput. Archit. News **31**(2), 422–433 (2003)
96. Scarpazza, D.P., Raghavan, P., Novo, D., Catthoor, F., Verkest, D.: Software simultaneous multi-threading, a technique to exploit task-level parallelism to improve instruction- and data-level parallelism. In: PATMOS '06: Proceedings of the 16th International Workshop on Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation, pp. 107–116 (2006)
97. Schlansker, M., Mahlke, S., Johnson, R.: Control CPR: a branch height reduction optimization for EPIC architectures. SIGPLAN Notices **34**(5), 155–168 (1999)
98. Shao, S., Yin, S., Liu, L., Wei, S.: Map-reduce inspired loop parallelization on CGRA. In: IEEE International Symposium on Circuits and Systems, ISCAS 2014, Melbourne, Victoria, Australia, June 1–5, 2014, pp. 1231–1234 (2014)
99. Shen, J., Lipasti, M.: Modern Processor Design: Fundamentals of Superscalar Processors. McGraw-Hill (2005)
100. Shi, R., Yin, S., Liu, L., Liu, Q., Liang, S., Wei, S.: The implementation of texture-based video up-scaling on coarse-grained reconfigurable architecture. IEICE Transactions **98-D**(2), 276–287 (2015)
101. Silicon Hive: HiveCC Databrief (2006)
102. Sudarsanam, A.: Code optimization libraries for retargetable compilation for embedded digital signal processors. Ph.D. thesis, Princeton University (1998)
103. Suh, D., Kwon, K., Kim, S., Ryu, S., Kim, J.: Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor. In: Proc. on Conf. Field-Programmable Technology, pp. 67–70 (2012)
104. Suzuki, T., Yamada, H., Yamagishi, T., Takeda, D., Horisaki, K., Vander Aa, T., Fujisawa, T., Van der Perre, L., Unekawa, Y.: High-throughput, low-power software-defined radio using reconfigurable processors. IEEE Micro **31**(6), 19–28 (2011)
105. Taylor, M., Kim, J., Miller, J., Wentzla, D., Ghodrat, F., Greenwald, B., Ho, H., Lee, M., Johnson, P., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Frank, V., Amarasinghe, S., Agarwal, A.: The Raw microprocessor: A computational fabric for software circuits and general purpose programs. IEEE Micro **22**(2), 25–35 (2002)
106. Texas Instruments: TMS320C64x Technical Overview (2001)
107. Theocharis, P., De Sutter, B.: A bimodal scheduler for coarse-grained reconfigurable arrays. ACM Trans. on Architecture and Code Optimization **13**(2), 15:1–15:26 (2016)
108. Van Essen, B., Panda, R., Wood, A., Ebeling, C., Hauck, S.: Managing short-lived and long-lived values in coarse-grained reconfigurable arrays. In: FPL '10: Proceedings of the 2010 International Conference on Field Programmable Logic and Applications, pp. 380–387 (2010)
109. Van Essen, B., Panda, R., Wood, A., Ebeling, C., Hauck, S.: Energy-Efficient Specialization of Functional Units in a Coarse-Grained Reconfigurable Array. In: FPGA '11: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 107–110 (2011)
110. Vander Aa, T., Palkovic, M., Hartmann, M., Raghavan, P., Dejonghe, A., Van der Perre, L.: A multi-threaded coarse-grained array processor for wireless baseband. In: Proc. 9th IEEE Symp. Application Specific Processors, pp. 102–107 (2011)
111. Venkataramani, G., Najjar, W., Kurdahi, F., Bagherzadeh, N., Bohm, W., Hammes, J.: Automatic compilation to a coarse-grained reconfigurable system-on-chip. ACM Trans. Embed. Comput. Syst. **2**(4), 560–589 (2003)
112. van de Waerdt, J.W., Vassiliadis, S., Das, S., Mirolo, S., Yen, C., Zhong, B., Basto, C., van Itegem, J.P., Amirtharaj, D., Kalra, K., Rodriguez, P., van Antwerpen, H.: The TM3270 media-processor. In: MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture, pp. 331–342. IEEE Computer Society, Washington, DC, USA (2005)

113. Woh, M., Lin, Y., Seo, S., Mahlke, S., Mudge, T., Chakrabarti, C., Bruce, R., Kershaw, D., Reid, A., Wilder, M., Flautner, K.: From SODA to scotch: The evolution of a wireless baseband processor. In: MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture, pp. 152–163. IEEE Computer Society, Washington, DC, USA (2008)
114. Programming XPP-III Processors White Paper (2006)
115. Xu, B., Yin, S., Liu, L., Wei, S.: Low-power loop parallelization onto CGRA utilizing variable dual v_{dd}. IEICE Transactions **98-D**(2), 243–251 (2015)
116. Yang, C., Liu, L., Luo, K., Yin, S., Wei, S.: CIACP: A correlation- and iteration- aware cache partitioning mechanism to improve performance of multiple coarse-grained reconfigurable arrays. IEEE Trans. Parallel Distrib. Syst. **28**(1), 29–43 (2017)
117. Yang, C., Liu, L., Wang, Y., Yin, S., Cao, P., Wei, S.: Configuration approaches to improve computing efficiency of coarse-grained reconfigurable multimedia processor. In: 24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2–4 September, 2014, pp. 1–4 (2014)
118. Yang, C., Liu, L., Wang, Y., Yin, S., Cao, P., Wei, S.: Configuration approaches to enhance computing efficiency of coarse-grained reconfigurable array. Journal of Circuits, Systems, and Computers **24**(3) (2015)
119. Yang, C., Liu, L., Yin, S., Wei, S.: Data cache prefetching via context directed pattern matching for coarse-grained reconfigurable arrays. In: Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5–9, 2016, pp. 64:1–64:6 (2016)
120. Yin, S., Gu, J., Liu, D., Liu, L., Wei, S.: Joint modulo scheduling and v_{dd} assignment for loop mapping on dual-v_{dd} CGRAs. IEEE Trans. on CAD of Integrated Circuits and Systems **35**(9), 1475–1488 (2016)
121. Yin, S., Lin, X., Liu, L., Wei, S.: Exploiting parallelism of imperfect nested loops on coarse-grained reconfigurable architectures. IEEE Trans. Parallel Distrib. Syst. **27**(11), 3199–3213 (2016)
122. Yin, S., Liu, D., Liu, L., Wei, S., Guo, Y.: Joint affine transformation and loop pipelining for mapping nested loop on CGRAs. In: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9–13, 2015, pp. 115–120 (2015)
123. Yin, S., Liu, D., Peng, Y., Liu, L., Wei, S.: Improving nested loop pipelining on coarse-grained reconfigurable architectures. IEEE Trans. VLSI Syst. **24**(2), 507–520 (2016)
124. Yin, S., Yao, X., Liu, D., Liu, L., Wei, S.: Memory-aware loop mapping on coarse-grained reconfigurable architectures. IEEE Trans. VLSI Syst. **24**(5), 1895–1908 (2016)
125. Yin, S., Zhou, P., Liu, L., Wei, S.: Acceleration of nested conditionals on CGRAs via trigger scheme. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, Austin, TX, USA, November 2–6, 2015, pp. 597–604 (2015)
126. Yin, S., Zhou, P., Liu, L., Wei, S.: Trigger-centric loop mapping on CGRAs. IEEE Trans. VLSI Syst. **24**(5), 1998–2002 (2016)
127. Yoon, J., Ahn, M., Paek, Y., Kim, Y., Choi, K.: Temporal mapping for loop pipelining on a MIMD-style coarse-grained reconfigurable architecture. In: Proceedings of the International SoC Design Conference (2006)
128. Yoon, J.W., Shrivastava, A., Park, S., Ahn, M., Jeyapaul, R., Paek, Y.: SPKM : A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In: Proc. 13th Asia South Pacific Design Automation Conf. (ASP-DAC), pp. 776–782 (2008)

High Performance Stream Processing on FPGA

John McAllister

Abstract Field Programmable Gate Array (FPGA) have plentiful computational, communication and member bandwidth resources which may be combined into high-performance, low-cost accelerators for computationally demanding operations. However, deriving efficient accelerators currently requires manual register transfer level design—a highly time-consuming and unproductive process. Software-programmable processors are a promising way to alleviate this design burden but are unable to support performance and cost comparable to hand-crafted custom circuits. A novel type of processor is described which overcomes this shortcoming for streaming operations. It employs a fine-grained processor with very high levels of customisability and advanced program control and memory addressing capabilities in very large-scale custom multicore networks to enable accelerators whose performance and cost match those of hand-crafted custom circuits and well beyond comparable soft processors.

1 Introduction

Field Programmable Gate Array (FPGA) technologies have long been recognised for their ability to enable very high-performance realisations of computationally demanding, highly parallel operations beyond the capability of other embedded processing technologies. Recent generations of FPGA have seen a rapid increase in this computational capacity and the emergence of System-on-Chip SoC-FPGA, incorporating heterogeneous multicore processors alongside FPGA programmable fabric. A key motivation for these hybrid architectures is the ability of FPGA to host performance-critical operations, offloaded from processors, as application-specific *accelerators* with any combination of high-performance, low cost or high energy efficiency.

J. McAllister (✉)
Institute of Electronics, Communications and Information Technology (ECIT), Queen's University Belfast, Belfast, UK
e-mail: jp.mcallister@ieee.org

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_13

The resources available with which accelerators may be built are enormous: the designer has, every second, access to trillions of multiply accumulate operations via on-chip DSP units [3, 30] and memory locations in Block RAM (BRAM) [3, 31], alongside the computationally powerful and highly flexible Look-Up Table (LUT) FPGA programmable logic [17]. For instance, the Virtex^6-7 family of Xilinx FPGAs offers up to 7×10^{12} multiply-accumulate (MAC) operations per second and 40×10^{12} bits/s memory access rates.

To combine these resources into accelerators of highest performance or lowest cost, though, requires manual design of custom circuit architectures at Register Transfer Level (RTL) in a hardware design language. This is a low level of design abstraction which imposes a heavy design burden, significantly more complicated than describing behaviour in a software programming language. Hence, for many years designers have sought a way to realise accelerators more rapidly without suffering critical performance or cost bottlenecks. Software-programmable 'soft' processors are one way to do so, but at present adopting such an approach demands substantial compromise on performance and cost. Soft processors allow their architecture to be tuned before synthesis to improve the performance and cost of the final result. Soft general-purpose processors such as MicroBlaze [32] and Nios-II [2] are performance-limited and a series of approaches attempt to resolve this issue. One approach uses soft vector coprocessors [9, 24, 33, 34] employing either assembly-level [34] or mixed C-macro and inline assembly programming. These enable performance increases by orders of magnitude beyond Nios-II and MIPS [34], but performance and cost still lag custom circuits. An alternative approach is to redesign the architecture of the central processor architecture for performance/cost benefit, and approach adopted in the iDEA [8] processor. Multicore architectures incorporating up to 16 [12, 22, 25] or even 100 processors in [12] have also been proposed.

However, the cost of enabling software programmability in all of these approaches is a reduction in performance or efficiency in the resulting accelerators, relative to custom circuit solutions. The result is that the performance of these architectures is only marginally beyond that of software-programmable devices and there is no evidence these are competitive with custom circuits. It appears that if FPGA soft processors are to be a viable alternative to custom accelerators then performance and cost must improve radically.

2 The FPGA-Based Processing Element (FPE)

A unique, lean soft processor—the *FPGA Processing Element* (FPE)—is proposed to resolve this deficiency. The architecture of the FPE is shown in Fig. 1. It contains only the minimum set of resources required for programmability: the instructions pointed to by the *Program Counter* (PC) are loaded from *Program Memory* (PM) and decoded by the *Instruction Decoder* (ID). Data operands are read either from *Register File* (RF), or in the case of immediate data *Immediate Memory* (IMM) and

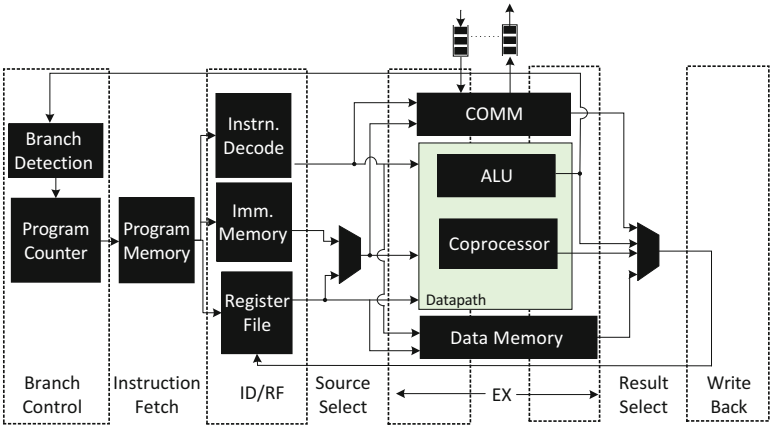


Fig. 1 The FPGA processing element

Table 1 FPE parameters and instructions

(a) FPE configuration parameters

Parameter	Meaning	Values
`DataWidth`	Data wordsize	16/32 bits
`DataType`	Type of data	Real/complex
`ALUWidth`	No. DSP48e slices	1–4
`PMDepth`	PM Capacity	Unlimited
`PMWidth`	PM Wordsize	Unlimited
`DMDepth`	DM/RF Capacity	Unlimited
`RFDepth`	No. RF locations	Unlimited
`TxCOMM`	No. Tx ports	$\leq$1024
`RxCOMM`	No. Rx ports	$\leq$1024
`IMMDepth`	IMM Capacity	Unlimited

(b) FPE instruction set

Instruction	Function
`LOOP`	Loop
`BEQ`/`BGT`/`BLT`	Branching
`GET`/`PUT`	FIFO get/put
`NOP`	No operation
`MUL`/**`ADD`**/**`SUB`**	Multiply/add/subtract
`MULADD(FWD)`	Multiply-add
`MULSUB(FWD)`	Multiply-subtract
`COPROC`	Coprocessor access
`LD`/**`ST`**	Load/store
`LDIMM`/`STIMM`	IMM load/store

processed by the ALU (implemented using a Xilinx DSP48e). In addition, a *Data Memory* (DM) is used for bulk data storage and a *Communication Adapter* (COMM) performs on/off-FPE communications.

The FPE is soft and hence *configurable* to allow its architecture to be customised pre-synthesis in terms of the aspects listed in Table 1(a). Beyond these, custom coprocessors can also be integrated alongside the ALU to accelerate specific custom instructions. Of course, the FPE is also *programmable*, with an instruction set described in Table 1(b).

When implemented on Xilinx Virtex 5 VLX110T FPGA, a 16 bit Real FPE costs 90 LUTs, 1 DSP48e and enables 483×10^6 multiply-add operations per second. This represents around 18% of the resource of a conventional MicroBlaze processor, whilst increasing performance by a factor 2.8.

The FPE's low cost allows it to be combined in very large numbers on a single FPGA, to realise operations via multicore architectures, with communication between FPEs via point-to-point queues. Hence the FPE may be viewed as a

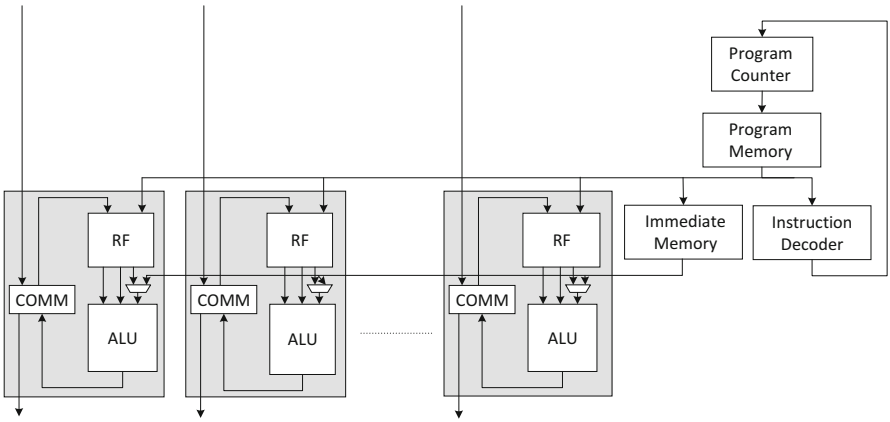


Fig. 2 SIMD processor architecture

fundamental building block for realising computationally demanding operations on FPGA.

To do so efficiently, the FPE should be able to exploit all the different types of parallelism in a program or application. Task parallelism is exploited in the multicore architectures proposed, but using these to realise data parallel operation is less than efficient, due to the duplication of control logic and data and memory resources. In this case each FPE will contain the same instructions in their PM, access RF in the same orders and execute the same programs. There is considerable overhead incurred when control resource is duplicated for each FPE. To avoid this occurring, the FPE is further extended into a configurable SIMD processor component, as illustrated in Fig. 2.

The width of the SIMD is configurable via a new parameter, `SIMDways`, which dictates the number of datapath lanes. All of the FPE instructions (except BEQ, BGT and BLT) can be used as SIMD instructions.

3 Case Study: Sphere Decoding for MIMO Communications

To illustrate the use of FPE-based multicores for FPGA accelerators, a case study—Sphere Decoding (SD) for Multiple-Input, Multiple-Output (MIMO) communications systems—is used. MIMO systems employ multiple transmit and multiple receive channels [26] to enable data rates of unprecedented capacity, prompting their adoption in standards such as 802.11n [14]. An M-element array of transmit antennas emit a vector $\mathbf{s} \in \mathbb{C}^M$ of QAM-modulated symbols. The vector of symbols $\mathbf{y} \in \mathbb{C}^N$ received at an N-element array of antennas is related to $\mathbf{s}$ by:

$$\mathbf{y} = \mathbf{H}\mathbf{s} + \mathbf{v}, \tag{1}$$

where $\mathbf{H} \in \mathbb{C}^{N \times M}$ represents the MIMO channel, used typically as a parallel set of flat-fading subchannels via Orthogonal Frequency Division Multiplexing (OFDM)

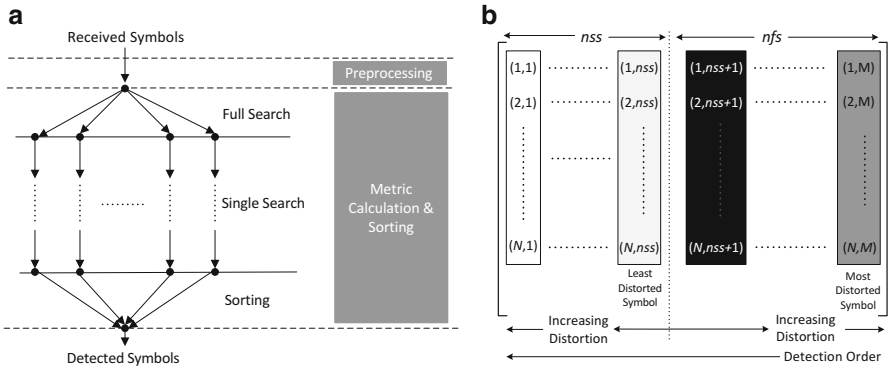


Fig. 3 FSD algorithm components. (**a**) FSD Tree Structure. (**b**) General Form of $\mathbf{H}^{\dagger}$

(108 in the case of 802.11n) and $\mathbf{v} \in \mathbb{C}^N$ additive noise. Sphere Decoding (SD) is used to derive an estimate $\hat{\mathbf{s}}$ of $\mathbf{s}$. It offers near that of the ideal ML detector, with significantly reduced complexity [20, 23]. The Fixed-Complexity SD (FSD) has a particularly low complexity, two-stage deterministic process which makes it ideal for efficient realisation via an FPGA accelerator [5]. FSD realises a two-stage detection process illustrated in Fig. 3a.

Algorithm 1 SQRD for FSD

input : $\mathbf{H}$, M
output: $\mathbf{Q}$, $\mathbf{R}$, **order**

Phase 1: *Initialization*
$\mathbf{Q} = \mathbf{H}$, $\mathbf{R} = \mathbf{0}_M$,
order $= [1, \cdots, M]$,
$nfs = \left\lceil \sqrt{M} - 1 \right\rceil$
for $i \leftarrow 1$ **to** M **do**
  $\mathbf{norm}_i = \left\| \mathbf{q}_i \right\|^2$
end
Phase 2: *SQRD ordering*
for $i \leftarrow 1$ **to** M **do**
  $k = min\,(nfs + 1, M - i + 1)$
  $k_i = \underset{j=i,\cdots,M}{\overset{k}{\arg\min}}\ \mathbf{norm}_j$
  Exchange columns i and k_i in $\mathbf{R}$, **order**, **norm** and $\mathbf{Q}$
  $r_{i,i} = \sqrt{\mathbf{norm}_i}$
  $\mathbf{q}_i = \mathbf{q}_i / r_{i,i}$
  for $l \leftarrow i + 1$ **to** M **do**
    $r_{i,l} = \mathbf{q}_i^H \cdot \mathbf{q}_l$
    $\mathbf{q}_l = \mathbf{q}_l - r_{i,l} \cdot \mathbf{q}_i$
    $\mathbf{norm}_l = \mathbf{norm}_l - r_{i,l}^2$
  end
end

Pre-Processing (PP) orders the symbols of $\mathbf{y}$ according to the perceived distortion experienced by each. This is achieved by reordering the columns of $\mathbf{H}$ to give $\mathbf{H}^{\dagger}$ (the general form of which is illustrated in Fig. 3b). Practically, this is achieved via an iterative Sorted QR Decomposition (SQRD) algorithm, described in Algorithm 1 [11].

SQRD-based PP ordering for FSD transforms the input channel matrix $\mathbf{H}$ to the product of a unitary matrix $\mathbf{Q}$ and an upper-triangular $\mathbf{R}$ via QR decomposition, whilst deriving **order**, the order of detection of the received symbols during MCS. It operates in two phases, as described in Algorithm 1. In Phase 1 $\mathbf{Q}$, $\mathbf{R}$, **order**, **norm** and nfs are initialized as shown in lines 2–5 of Algorithm 1, where $\mathbf{q}_i$ is the ith column of $\mathbf{Q}$. Phase 2 comprises M iterations, in each of which the kth lowest entry in **norm** is identified (lines 9 and 10) before the corresponding column of $\mathbf{R}$ and elements in **order** and **norm** are permuted with the ith (line 11) and orthogonalized (line 12–18). The resulting $\mathbf{Q}$, $\mathbf{R}$, and **order** are used for Metric Calculation and Sorting (MCS) as defined in (3) and (4).

Metric Calculation and Sorting uses an M-level decode tree to perform a Euclidean distance based statistical estimation of $\mathbf{s}$. Groups of M symbols undergo detection via a tree-search structure illustrated in Fig. 3a.

The number of nodes at each tree level is given by $\mathbf{n}_S = (n_1, n_2, \ldots, n_M)^T$. The first *nfs* levels process the symbols from the worst distorted paths by *Full Search* (FS) enumeration of all elements of the search space. This results in P child nodes at level $i+1$ per node at level i, where P is the number of QAM constellation points. For full diversity, *nfs* is given by

$$nfs = \lceil \sqrt{M} - 1 \rceil. \tag{2}$$

The remaining *nss* ($nss = M - nfs$) levels undergo *Single Search* (SS) where only a single candidate detected symbol is maintained between layers. At each MCS tree level, (3) and (4) are performed.

$$\tilde{s}_i = \hat{s}_{ZF,i} - \sum_{j=i+1}^{M_t} \frac{r_{ij}}{r_{ii}} \left(\hat{s}_{ZF,j} - \hat{s}_j \right) \tag{3}$$

$$d_i = \sum_{j=i}^{M_t} r_{ij}^2 \left\| \hat{s}_{ZF,j} - \hat{s}_j \right\|^2, \; D_i = d_i + D_{i+1} \tag{4}$$

In (3) and (4), r_{ij} refers to an entry in $\mathbf{R}$, derived by QR decomposition of $\mathbf{H}$ during PP, $\hat{s}_{ZF}$ is the center of the FSD sphere and $\tilde{s}_j$ is the jth detected data, which is sliced to $\hat{s}_j$ in subsequent iterations of the detection process [13]. Since D_{i+1} can be considered as the Accumulated Partial Euclidean Distance (APED) at level $j = i + 1$ of the MCS tree and d_i as the PED in level i, the APED can be obtained by recursively applying (4) from level $i = M$ to $i = 1$. The resulting candidate symbols are sorted based on their Euclidean distance measurements, and the final result produced post-sorting.

This behaviour is duplicated across all OFDM subcarriers, of which there are 108 in 4×4 16-QAM 802.11n MIMO. For real-time processing this behaviour is repeated independently for all 108 subcarriers and must occur within 4 μs and at a rate of 480 Mbps for real-time performance. These are challenging requirements which has seen detection using custom circuit accelerators become a well-studied real-time implementation problem [4, 7, 15, 16, 21, 27]. It is notable that none of these uses software-programmable accelerator components. This section considers the use of the FPE to realise such a solution.

4 FPE-Based Pre-processing Using SQRD

The SQRD preprocessing technique is low-complexity relative to other, ideal preprocessing approaches. It is also numerically stable and lends itself well to fixed-point implementation, hence making it suitable for realisation on FPGA, as a result of its reliance on QRD. However, there are two major issues that must be resolved to enable FPE-based SQRD PP for 4×4 802.11n. It computational complexity remains high as outlined in Table 2; given the capabilities of a single FPE, it appears that a large-scale multi-FPE architecture is required to enable SQRD for 4 × 4 802.11n. Its reliance on square root and division operations also present a challenge, since these operations are not native to the DSP48e components used as the datapaths for the FPE and will have low performance when realised thereon [19].

To avoid this performance bottleneck, datapath coprocessors are considered to enable real-time division and square-root operations.

4.1 FPE Coprocessors for Arithmetic Acceleration

Non-restoring 16-bit division [19] requires 312 cycles when implemented using only the DSP48e in an *16R* FPE. This equates to approximately 1.2×10^6 div/s (divisions per second). Hence, around 100 FPEs would be required to realise the 120×10^6 divisions required per second (MDiv/s) for 4 × 4 SQRD for 802.11n. The high resource cost this would entail can be alleviated by adding radix-2 or radix-4 non-restoring division coprocessors [19] alongside the DSP48e in the FPE ALU (Fig. 4).

The performance, cost and efficiency (in terms of throughput per LUT, or TP/LUT) of the programmed FPE when division is realised using a programmed approach and the DSP48e only, (*FPE-P*) and when radix-2 or radix-4 coprocessors

Table 2 4 × 4 SQRD operational complexity

Operation	+/−	×	÷	√
op/second ($\times 10^9$)	3.24	12.72	0.12	0.12

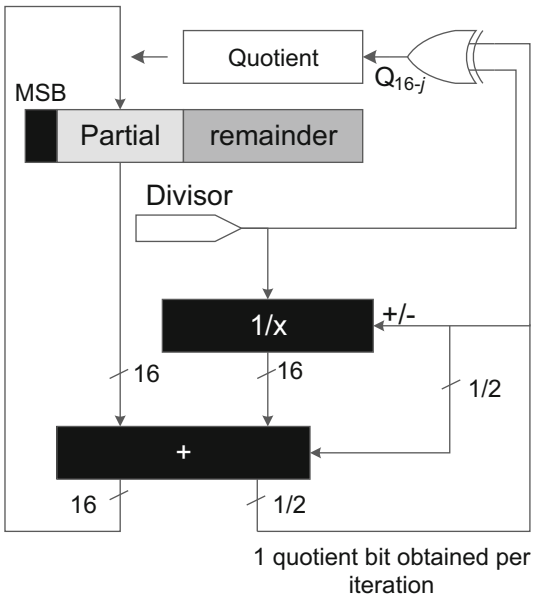


Fig. 4 FPE division coprocessor

Table 3 SQRD division implementations

Solution	Resource			Throughput
	FPEs	DSP48es	LUTs	(MDiv/s)
FPE-P	100	100	13,600	120
FPE-R_2	5	5	900	120
FPE-R_4	4	4	944	144

are added alongside the DSP48e (*FPE-*R_2, *FPE-*R_4 respectively) on Virtex 5 FPGA is described in Table 3. The *FPE-*R_2 and *FPE-*R_4 solutions both increase throughput, by factors of 8.9 and 13.3 respectively and hence increase hardware efficiency by respective factors of 9.4 and 10.7 as compared to *FPE-P*. Since 4×4 802.11n MIMO requires 120 MDiv/s for SQRD-based preprocessing, the implied cost and performance metrics of each option are summarised in Table 3. According to these estimates, *FPE-*R_2 represents the lowest cost real-time solution, enabling a 93.4% reduction in resource cost relative to *FPE-P*. This approach is adopted in the FPE-based SQRD implementation.

To realise the 120×10^6 square root operations required per second (MSQRT/s), performance and cost estimates for software-based execution on the FPE using the pencil-and-paper method [19] (*FPE-P*), or by adding a CORDIC coprocessor [28] (*FPE-C*) are compared in Table 4(a). The coprocessor-based *FPE-C* solution at once increases throughput and efficiency by factors of 23 and 10 respectively as compared to *FPE-P*, implying the resources required to realise real-time square-root for SQRD-based detection of 4×4 802.11n MIMO can be estimated as in Table 4(b). As this shows, *FPE-C* enables real-time performance using only 11% of the resource required by *FPE-P*, and is adopted for realising FPE-based square root operations.

Table 4 FPE square root options

(a) 16-Bit PSQRT, CSQRT			(b) 802.11n SQRD		
	FPE-P	*FPE-C*		*FPE-P*	*FPE-C*
PM/RF locations	29/14	8/1	FPEs	63	3
LUTs	142	330	LUTs	8946	990
DSP48es	1	0	DSP48es	63	3
Clock (MHz)	367.7	350	T (MSQRT/s)	121.6	130.8
Latency (cycles)	191	8			
T (MSQRT/s)	1.93	43.6			
T/LUT ($\times 10^{-3}$)	13.6	132.1			

4.2 SQRD Using FPGA

Integrating these components into a coherent processing architecture to perform SQRD, and replicating that behaviour to provide PP for the 108 subcarriers of 802.11n MIMO is a large scale accelerator design challenge. Figure 5 describes the SQRD algorithm as a, iterative four-task (T_1, $T_{2.1}$–$T_{2.3}$) process. The first task, T_1, conducts channel norm ordering, and computes the diagonal elements of **R** (lines 11–13 in Algorithm 1). This is followed by $T_{2.1}$–$T_{2.3}$, which are independent and permute and update **Q**, **R** and **norm** respectively (lines 14–18 in Algorithm 1).

This process is realised using a 4-FPE Multiple Instruction, Multiple Data (MIMD) architecture, shown in Fig. 6, is used. All FPEs employ 16-bit datapaths and are otherwise configured as described in Table 5(a). FPE_1–FPE_3 permute **Q**, **R** and **norm** and iteratively update ($T_{2.1}$–$T_{2.3}$ in Fig. 5). FPE_4 calculates the diagonal elements of **R** (T_1). The SQRD process executes in three phases. Initially, **H** and the calculation of **norm** are distributed amongst the FPEs, with the separate parts of **norm** gathered by FPE_4 to undergo ordering, division and square root. The results are distributed to the outer FPEs for permutation and update of **Q**, **R** and **norm**. Inter-FPE communication occurs via point-to-point FIFO links, chosen due to their relatively low cost on FPGA and implicit ability to synchronize the multi-FPE architecture in a data-driven manner whilst avoiding data access conflicts.

The performance and cost of the 4-FPE grouping is given in Table 5(b). According to these metrics, the throughput of each 4-FPE group is sufficient to support SQRD-based PP of 3 802.11n subcarriers. To process all 108 subcarriers, the architecture is replicated 36 times, as shown in Fig. 6. The mapping of subcarriers to groups is as described in Fig. 6.

On Xilinx Virtex 5 VSX240T FPGA, the cost and performance of this architecture is described in Table 5(b). As this describes, 32.5 MSQRD/s are achieved, in excess of the 30 MSQRD/s required for 4 × 4 802.11n MIMO.

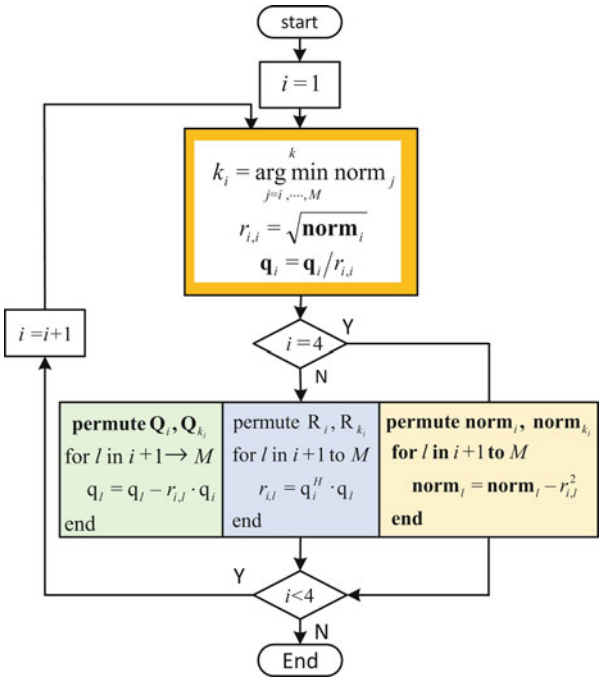


Fig. 5 4 × 4 SQRD

5 FSD Tree-Search for 802.11n

Computing MCS for FSD in 4 × 4 16 QAM 802.11n is even more computationally demanding than SQRD-based preprocessing. The operational complexity is described in Table 6(a). When a single 4 × 4 16-QAM FSD MCS is implemented on a *16R* FPE, the performance and cost are as reported as *16R-MCS* in Table 6(b).

To scale this performance to support all 108 subcarriers for 4 × 4 16-QAM 802.11n MIMO, a large-scale architecture is required. Two important observations of the application's behaviour help guide the choice of multiprocessing architecture:

1. THE FSD MCS tree exhibits strong SIMD-like behaviour, where each branch (Fig. 3a), performs an identical sequence of operations on data-parallel samples.
2. The number of FPEs required to implement MCS for all 108 OFDM subcarriers on a single, very wide SIMD processor implies limitations on the achievable clock rate as a result of high signal fan-outs to broadcast instructions from a central PM to a very large number of ALUs, restricting performance [10]. Hence, a collection of smaller SIMDs is used.

As described in Table 6(b), the cost of *16R-MCS* as compared to the basic 16-bit FPE described in Sect. 2 (from 90 LUTs to 2530 approximately) is significantly higher. This large increase is due to the large PM required to house the 4591

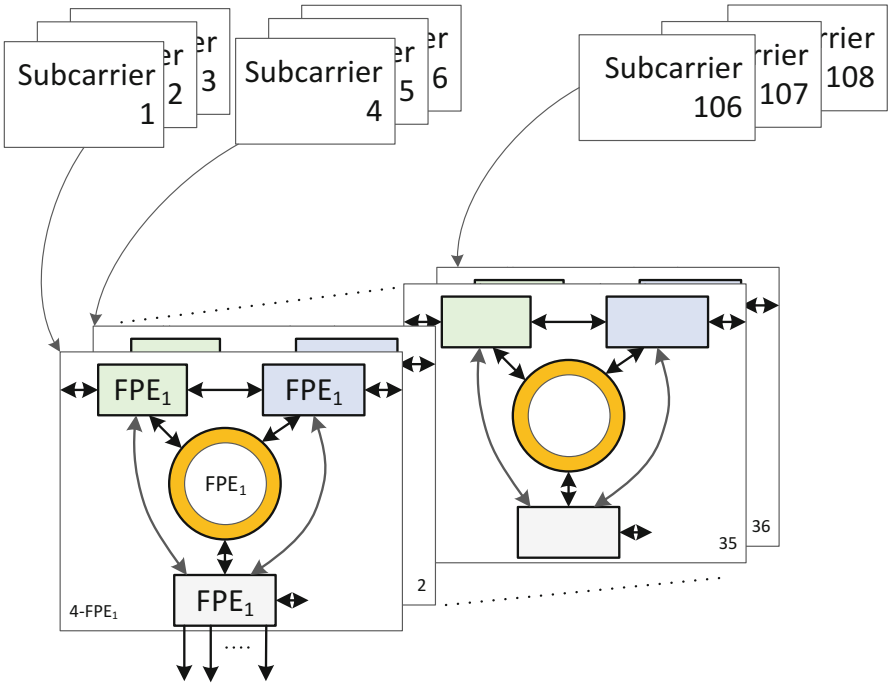


Fig. 6 4 × 4 SQRD mapping

Table 5 4-FPE-based SQRD

(a) FPE configuration		(b) FPE-SQRD metrics		
Parameter	Value		4-FPE$_1$	FPE-SQRD
PMDepth	350	LUTs	2109	70,560
RFDepth	32	DSP48es	4	144
IMMDepth	32	Clock (MHz)	315	265
DMDepth	64	T (MSQRD/s)	1.07	32.5
TxComm	32	Latency (μS)	0.9	1.1
RxComm	32			

instructions. A significant factor in this large number of instructions are the comparison operations required for slicing (Eq. (3)) and sorting the PED metrics, which require branch instructions, which have associated NOP operations due to the deep FPE pipeline and the lack of forwarding logic [10]. These represent wasted cycles and dramatically increase cost and reduce throughput—branch and NOP instructions represent 50.7% of the total number of instructions. Optimising the FPE to reduce the impact of these branch instructions could have a significant impact on the MCS cost/performance.

Table 6 802.11n MCS complexity

(a) Operational complexity		(b) MCS implementation options		
Operation	op/s ($\times 10^9$)		*16R-MCS*	16R
$+/-$	32.37	LUTs	2520	805
$\times$	19.20	DSP48es	1	0
		Clock (MHz)	367.7	350
		L (Cycles)	3281	1420
		T (MOP/s)	1.9	4.5

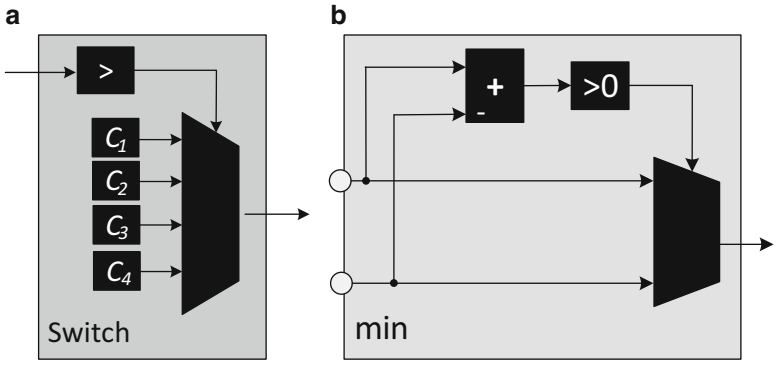


Fig. 7 (**a**) Switch coprocessor (**b**) Min coprocessor

5.1 FPE Coprocessors for Data Dependent Operations

Employing ALU coprocessors can significantly reduce these penalties. A switch coprocessor compares the input to each of four constants, determined pre-synthesis (a logical depiction of behaviour is shown in Fig. 7a), selecting the closest. This increases the efficiency of slicing by comparing an input operand to one of a number of pre-defined values. Similarly, a MIN coprocessor (Fig. 7b) can be used to accelerate sorting.

Each of these coprocessors occupy around 20 LUTs, but their ability to eliminate wasted instructions can significantly reduce the PM size. This can enable significant reductions in overall cost and increases in performance as described in column 3 of Table 6(b). Including these components results in a 68% reduction in resource cost and a factor 2.3 increase in throughput. The resulting component is capable of realising FSD MCS for a single 802.11n subcarrier in real-time, providing a good foundation unit for implementing MCS for all 108 subcarriers.

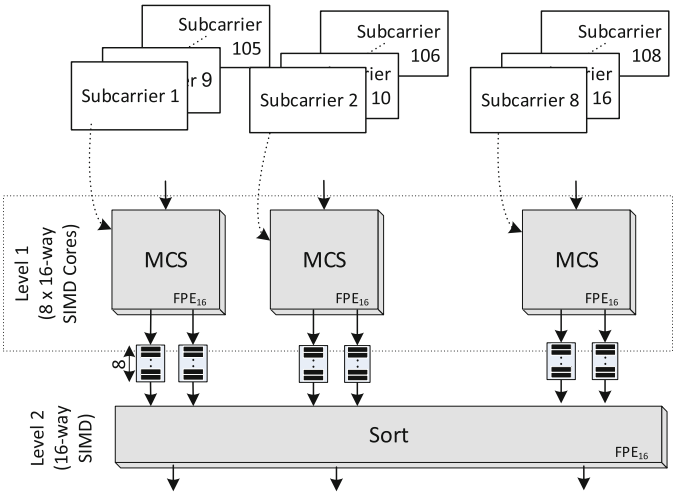


Fig. 8 802.11n OFDM MCS-SIMD mapping

5.2 SIMD Implementation of 802.11n FSD MCS

To scale the FPE to realise all 108 subcarriers, a range of architectures may be used. The data-parallel operation of the subcarriers suggests that a very wide single SIMD could be used, providing the most efficient realisation from the perspective of PM and control logic cost. However, as the width of an FPE SIMD unit increases beyond 16 lanes, the instruction broadcast from the single central PM limits the speedup which may be obtained by constraining the clock frequency. Hence, 16-way SIMDs are employed and FSD MCS for all 108 802.11n subcarriers is implemented on a dual-layer network of such processors, as illustrated in Fig. 8.

Level 1 consists of eight SIMDs. The 802.11n subcarriers are clustered into eight groups $\{G_i = \{j : (j-1) \mod 8 = i\}_{j=1}^{108}\}_{i=0}^{7}$, where j is the set of subcarriers processed by FPE i. The 16 branches of the MCS tree for each subcarrier are processed in parallel across the 16 ways of the Level 1 SIMD onto which they have been mapped. Sorting for the subcarriers implemented in each Level 1 SIMD is performed by adjacent pairs of ways in the Level 2 SIMD—hence given the 8 Level 1 SIMDs, the Level 2 SIMD is composed of 16 ways.

Each FPE is configured to exploit 16-bit real-valued arithmetic [6]. All processors exploit `PMDepth` = 128, `RFDepth` = 32 and `DMDepth` = 0, and communication between the two levels exploit 8-element FIFO queues. The Level 1 SIMDs incorporate SWITCH coprocessors to accelerate the slicing operation, whilst the Level 2 SIMDs support the MIN ALU extension to accelerate the sort operation.

The program flow for each Level 1 SIMD is as illustrated in Fig. 9a. Each FPE performs a single branch of the MCS tree, with the empty parts of the program flow—representing NOP instructions—used to properly synchronise movement of data into and out of memory.

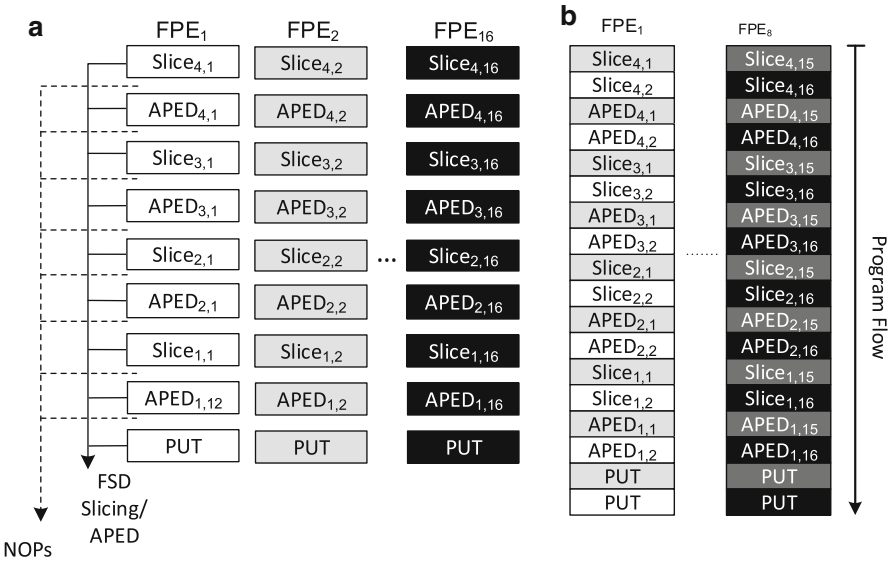


Fig. 9 FPE branch interleaving. (**a**) Original FSD threads. (**b**) Interleaved threads

Table 7 4×4 16-QAM FSD using FPE

	FPE-MCS	FPE-FSD
LUT	16,601	96,115
DSP48e	144	408
Clock (MHz)	296	189
T (Mbps)	502.5	483
L (μS)	0.9	2.3

The NOP cycles represent 29% of the total instruction count but since they represent ALU idle cycles they should preferably be eliminated. To do so, NOP cycles in one branch can be occupied by the useful, independent instructions from another, i.e. the branches may be interleaved as illustrated in Fig. 9.This interleaving occupies wasted NOP cycles, to the extent that when two branches are interleaved the proportion of wasted cycles is reduced to 4%.

On Xilinx Virtex 5 VSX240T FPGA, this multi-SIMD architecture enables FSD-MCS for 802.11n as reported Table 7. As this shows, it comfortably exceeds the real-time performance criteria of 802.11n.

Together with the results of the SQRD preprocessing accelerator, these MCS metrics show that the FPE can support accelerators for applications with demanding real-time requirements. By using massively parallel networks of simple processors (>140 in this case), FPGA can support real-time behaviour and can enable solutions with resource cost comparable to custom circuits. When the PP and MCS are combined to create a full FSD detector (*FPE-FSD* in Table 7) the resulting architecture is the only software-defined FPGA structure to enable real-time performance for 4×4 16 QAM 802.11n.

6 Stream Processing for FPGA Accelerators

The FPE is a load-store structure, supporting only register-register and immediate instructions. All non-constant operands and results access the ALU via Register File (RF). Consider the effect of this approach for a 256-point FFT (FFT_{256}) realised using two FPE configurations: an 8-way FPE SIMD (FPE_8) or a MIMD multi-FPE composed of 8 SISD FPEs (8-FPE). The FFT mappings and itemized ALU, communication (IPC), memory (MEM) and NOP instructions for each are shown in Fig. 10.

Figure 10 shows that the efficiency of each of these programs is low—only 52.5% and 31.8% of the respective cycles in 8-FPE_1 and FPE_8 are used for ALU instructions. The resulting effect on accelerator performance and cost is clear from Table 8, which compares 8-FPE_1 with the Xilinx Core Generator FFT [29] component. The FPE is not competitive with the custom circuit Xilinx FFT, which exhibits twice the performance at a fraction of the LUT cost.

These results follow from the restriction to register-register instructions. Each FFT_{256} stage consume 512 complex words. Since RF is the most resource-costly element of the FPE, buffering this volume of data requires BRAM Data Memory (DM); in order for these operands to be processed and results stored, a large number of loads (stores) are required between BRAM and RF, increasing PM cost. Given the simplicity of the FFT butterfly operation, the overhead imposed by these is significant. This is combined with the effect of the FPE's requirement to be standalone: since it must handle its own communication, further cycles are consumed transferring incoming and outgoing data between DM and COMM, reducing program efficiency still further. Finally, each of these transfers induces a latency between source and destination—as Fig. 11 illustrates, each FPE

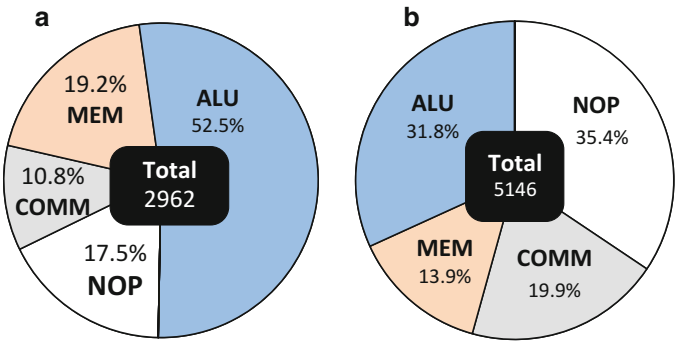


Fig. 10 FFT_{256}: FPE-based 256 Point FFT. (**a**) 8-FPE_1. (**b**) FPE_8

Table 8 256-Point FFT performance/cost comparison

	Cost		T	T/LUT
	LUTs	DSP48e	(MSamples/s)	($\times 10^3$)
8-FPE_1	2296	8	30.5	13.3
Xilinx	621	6	61.9	99.7

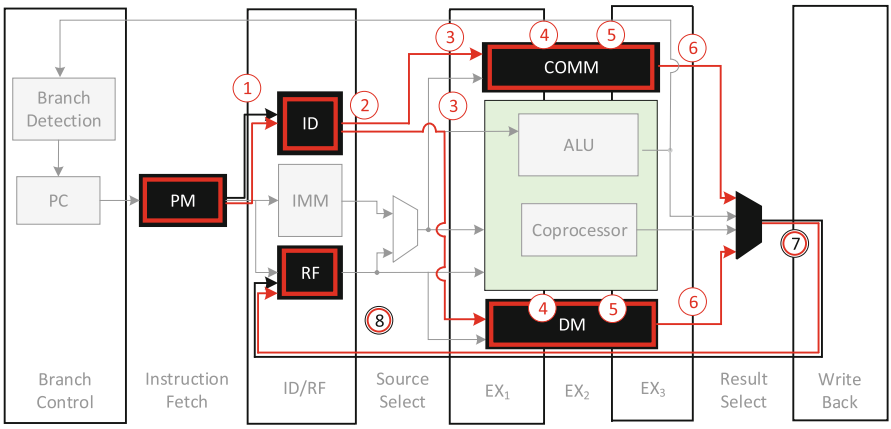


Fig. 11 Load-store paths in the FPE

DM-RF (black) and COMM-RF (red) transfer takes eight cycles, imposing the need for **NOP**s.

These factors combine to severely limit the efficiency of the FPE for applications such as FFT. Mitigating the effect of these overheads requires two features:

- Direct instruction access to any combination of RF, DM and COMM for either instruction source or destination.
- In cases where local buffering is not required, data streaming through the PE should be enabled, reducing load/store and communication cycle overhead.

6.1 Streaming Processing Elements

To support these features, a streaming FPE (sFPE) is proposed. The sFPE is still standalone, software-programmable and lean, but supports a processing approach—streaming—which diverges from the load-store FPE approach. Streaming means that focus is placed on ensuring that data can stream into and out of operation sources and destinations and through the ALU without the need for load and store cycles. This streaming takes two forms:

- Internal: between RF, DM, COMM and IMM without load-store cycles.
- External: from input FIFOs to output FIFOs via only ALU.

The architecture of a SISD $sFPE_1$ is illustrated in Fig. 12. There are three main architectural features of note.

- An entire pipeline stage is dedicated to instruction decode (ID)
- A *FlexData* data manager has been added which allows zero-latency access to any data source or sink.

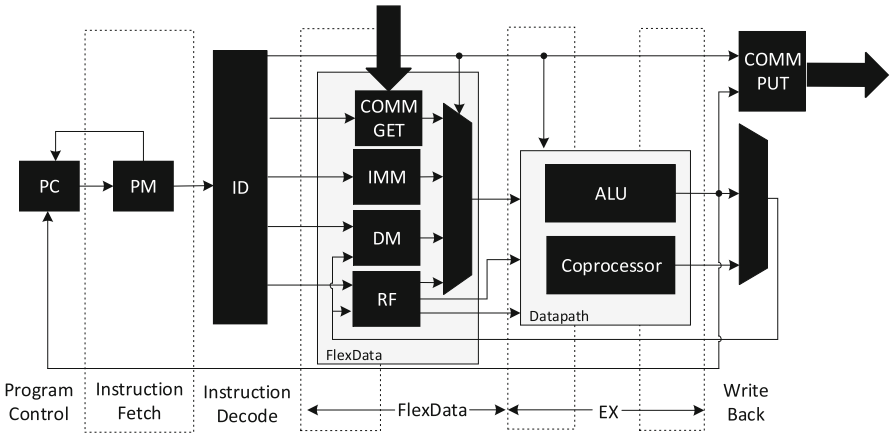


Fig. 12 SISD sFPE architecture

- Off-FPE communication has been decoupled into read (COMMGET) and write (COMMPUT) components

In the sFPE, ID and FlexData are assigned entire pipeline stages. The ID determines the source or destination of any instruction operand or result, with all of the potential sources or destinations of data incorporated in FlexData to allow each to be addressed with equal latency; this flat memory architecture is unique to the sFPE. This approach removes the load/store overhead of accessing, for example, data memory or off-FPE communication; all data operands and results may be sourced/produced to any of IMM, RF, DM or COMM with identical pipeline control and without the need for explicit load and store cycles or instructions for DM or COMM.

To allow unbuffered streaming from input FIFOs or output FIFOs via ALU, simultaneous read/write to external FIFOs is required, with direct access to ALU in both directions. Decoupling the off-FPE communication components into COMMGET and COMMPUT allow each to be accessed with zero-latency, from a single instruction—note that these both reside in the same pipeline stage and hence conform to the regular dataflow pipeline maintained across the remainder of FlexData. In addition, since all of COMMGET, COMMPUT, DM, RF and IMM access distinct memory resources (with separate memory banks employed within the sFPE and a FIFO employed per off-sFPE communication channel) there is no memory bandwidth bottleneck resulting from decoupling these accesses in this way—all could be accessed simultaneously if needed.

Table 9 ALU operand/destination instruction coding

Op	Source/sink	`x`
`Rx`	RF	Register location
`&x`	DM	DM address
`^x`	COMMGET/COMMPUT	IPC channel no.
`x`	IMM	Constant value

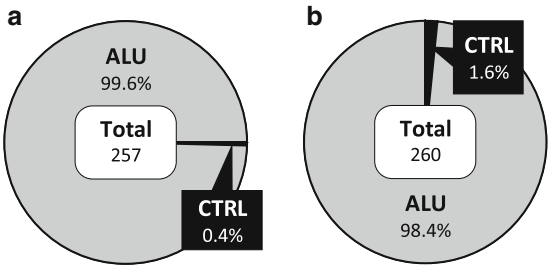


Fig. 13 FFT_{256}: sFPE implementations. (**a**) 8-$sFPE_1$. (**b**) $sFPE_8$

6.2 Instruction Coding

To support the increase level of specialisation of the operands in each instruction, however, operand addressing needs to become more complicated. Generally, sFPE ALU instructions take the form:

```
INSTR dest, opA, opB, opC
```

where `INSTR` is the instruction class, `dest` identifies the result destination and `opA`, `opB`, `opC` identify the source operands. The possible encodings of each of `dest`, `opA`, `opB`, `opC` and the destination are described in Table 9.

This encoding allows any of RF, DM, COMMGET and COMMPUT to be addressed directly from the absolute addresses quoted in the sFPE instruction. Constant operands are hard-coded into the instruction and IMM locations allocated by the assembler.

This architecture and data access strategy can lead to sFPE programs which are substantially more efficient that their FPE counterparts. Using the sFPE, the number of instructions needed for FFT_{256} in both the 8-sFPE and $sFPE_8$ variants are described in Fig. 13.

In MIMD 8-sFPE form, the total number of instructions required is 257, a decrease of around 91%. In addition, the efficiency of this realisation is now 99.6%, with only a single non-ALU instruction required for control. Similarly, $sFPE_8$ requires 95.9% fewer instructions and operates with an efficiency of 98.4%. Given these metrics it is reasonable to anticipate increases in throughput for 8-sFPE and $sFPE_8$ by factors of 20 and 30.

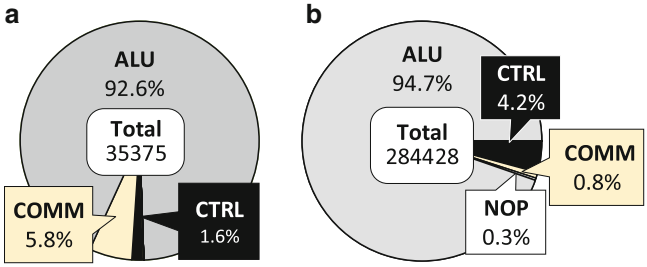


Fig. 14 Itemised sFPE matrix multiplication and ME operations. (**a**) Matrix multiplication. (**b**) Motion estimation

7 Streaming Block Processing

In many operations, however, addressing modes other than the simple direct approach used in the FPE are vital. An itemized instruction breakdown for multiplication of two 32×32 matrices and Full-Search ME (FS-ME) with a 16×16 macroblock on a 32×32 search window are quoted in Fig. 14.

A number of points are notable. Firstly, the programs are very efficient, verifying the techniques described in the previous section. However, the programs are extremely large—35,375 instructions for matrix multiplication (MM) and 284,428 for FS-ME. To store this number of instructions, a very large PM is required, requiring a lot of FPGA resources—for FS-ME, 241 BRAMs would be required for the PM alone. These demands are a direct result of the FPE's restriction to direct addressing. This is because, in a direct addressing scheme then every operation requires an instruction; for MM and ME, this translates a very large number of instructions.

However, both of these operations and their operand accesses are very regular and can be captured in programs with many fewer instructions than those quoted above. Both repeat the same operation many times on small subsets of the input data at regularly-spaced memory locations. For example, Bock-MM of two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$ when $m = n = p = 8$ via four 4×4 submatrices. Assuming that A and B are stored in contiguous memory locations in row-major order and that C is derived in row-major order, the operand memory access are as illustrated in Fig. 15.

To compute an element of a submatrix of C, the inner product of a four-element vector of contiguous locations in A (a row of the submatrix) and a four-element vector of elements spaced by 8 locations in B (a column of the submatrix) is formed. Afterwards either or both of the row of A or column of B are incremented to derive the next element of C, before operation proceeds to the next submatrix. The resulting memory accesses are highly predictable: a regular repeated increment along the rows of A and columns of B, periodic re-alignment to a new row of A and/or column of B, repeated multiple times before realigning for subsequent submatrices.

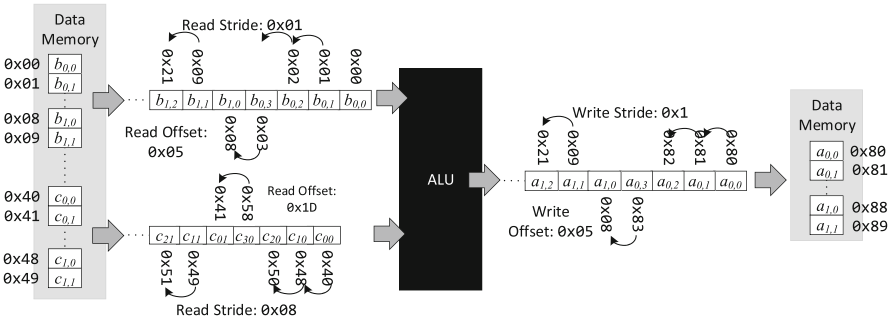


Fig. 15 sFPE block matrix multiply operand addressing

These patterns can be used to enable highly compact programs if two features are available—`repeat`-style behaviour with the ability for a single instruction to address blocks or memory are regularly-spaced locations when invoked multiple times by a `repeat`.

7.1 Loop Execution Without Overheads

To enable low-overhead loop operation, the sFPE is augmented with the ability to perform `repeat`-type behaviour. This means managing the PC such that when a `repeat` instruction is encountered, the body of the associated block of statements is executed a number of times. This task if fulfilled by a PC Manager (PCM), the behaviour of which is described in Fig. 16.

The PCM controls PC update given its previous value and the instruction referenced in PM given pieces of information—the start and end lines of the body statements to be repeated `S` and `E`, the number of repetitions `N`. These are encoded in a `RPT` instruction added to the sFPE instruction set. These instructions are encoded as:

```
RPT N S E
```

The behaviour of `RPT` is shown in Listing 1. This dictates five repetitions of lines 2–5. Any number of repeat instructions can be nested to allow efficient execution of loop nests with static and compile-time known loop bounds.

Listing 1 RPT Instruction Coding

```
RPT 5 2 4
  INSTR1...
  INSTR2...
  INSTR3...
```

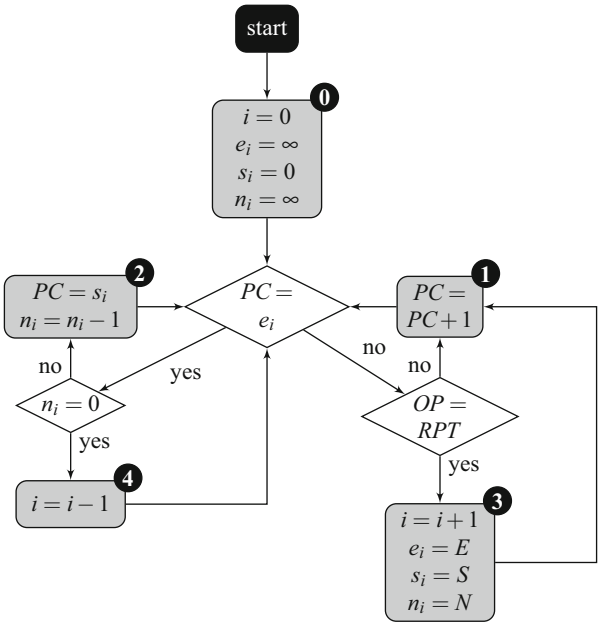


Fig. 16 sFPE PCM behaviour

The PCM arbitrates the PC to ensure that the body statements are repeated the correct number of times and support the construction of nested repeat operations. It enacts the flowchart in Fig. 16. For an n-level nest it maintains a $n + 1$-element lists of metrics, with an additional element added to support infinite repetition of the top-level program, considered to be an implicit infinite repeat instruction. For layer i of the loop nest, the start line, end line and number of repetitions are stored in element $i + 1$ of the lists s, e and n respectively. In all cases $s_0 = 0$, $e_0 = \infty$ and $n_0 = \infty$ to represent the start line, end line and number of repetitions of the top-level program (❶ in Fig. 16).[1] Every time a repeat instruction is encountered i, the current index into s, e and n is incremented and the values of the new element initialised using `S`, `E` and `N` from the decoded instruction in ❸. Regular PC updating then proceeds (❶) until either another repeat instruction is detected or until e_i is encountered. In the latter case, the number of iterations of the current statement is decremented (❷) or, if $n_i = 0$ all of the iterations of the current repeat statement have been completed and control of the loop nest reverts to the previous level (❹).

The PCM component requires 36 LUTs and hence imposes a relatively high resource cost as compared to the FPE. This can be controlled by compile-time customisation via the parameters listed in Table 10.

[1]Note that this assumes that the end line of the program is a `JMP` instruction with the start line as the target.

Table 10 PC configuration parameters

Parameter	Meaning	Values
`pcm_en`	Enable/disable PCM	Boolean
`pcm_depth`	Max. repeat nest depth	$\mathbb{N} \in [1, 2^{32} - 1]$

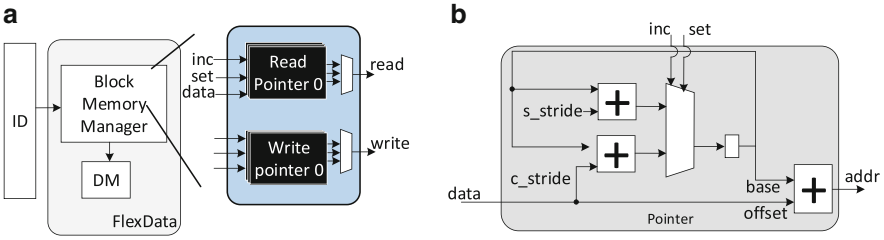


Fig. 17 sFPE block memory management elements. (**a**) sFPE FlexData. (**b**) Pointer Architecture

The `pcm_en` parameter is a Boolean which dictates whether the PCM is included or not. When it is, the maximum depth of loop nest is configurable via `pcm_en` which can take, hypothetically, any integer value. As such, the PCM may be included or excluded and hence imposes no cost when it is not required; further, when it is included its cost can be tuned to the application at hand by adjusting the maximum depth of loop nest.

7.2 Block Data Memory Access

Enabling block memory access requires three important capabilities:

- Auto-increment with any constant stride
- Manual increment with any stride
- Custom offset

The need for each of these is evident in MM: auto-increment traverses along rows and columns with a fixed memory stride—there are many such operations and so eliminating the need for an individual instruction for each reduce overall instruction count considerably. Manual increment is required for movement between rows/columns, whilst custom offset is used to identify the starting point for the increments, such as the first element of a submatrix.

A *Block Memory Manager* (BMM) is incorporated in the sFPE FlexData, as illustrated in Fig. 17a, to enable these properties. The BMM arbitrates access to DM via *Read Pointers* (RPs) and *Write Pointers* (WPs). The architecture of FlexData and a pointer is illustrated in Fig 17b.

Each pointer controls access to a subset (block) of the sFPE DM and addresses individual elements of that block via a combination of two subaddress elements: a

Table 11 BMM configuration parameters

Parameter	Meaning	Values
`mode`	Addressing mode	Direct, block
`n_rptrs` / `n_wptrs`	No. of read /write pointers	$\mathbb{N} \in [1, 2^{32}]$
`s_stride`	Constant stride	$\mathbb{N} \in [1, 2^{32}]$

Table 12 BMM instructions

Operand field	Meaning
`INC_RP` / `INC_WP`	Increment base of RP/WP `n` to `val`
`SET_RP` /`SET_WP`	Set offset of RP/WP `n` to `val`

Table 13 ALU block operand instruction coding

Operand field	`ofs`	`idx`	!
Meaning	Offset	Pointer reference	Autoincrement base

base and an **offset**. The offset selects the root block data element whilst the base iterates over elements relative to the offset.

Pointers operate in one of three modes. Either the base auto-increments, or it is incremented by explicit instruction, or the offset increments by explicit instruction. All three modes are supported under the control of the `set`, `inc` and `data` interfaces. The offset selects the root data element of the submatrices of A, B and C, with the base added to address elements relative to the offset. The base is updated via two mechanisms, under the control of `inc`. The first auto-increments by a value (`s_stride` in Fig.17b) set as a constant at synthesis time. Manually incrementing the base is achieved by `c_stride`, which is defined at run-time. Finally, when update of the offset is required, `data` is accepted on assertion of `set`. To allow absolute minimum cost for any operation, configuration parameters for the sFPE FlexData, BMM and pointer components are configurable by the parameters in Table 11.

It is notable that addressing mode is now a configuration parameter of the sFPE, with direct and block modes supported. In direct mode, the BMM is absent whilst it is included in the block mode. In that case, the cost can be minimised via control of the number of read and write pointers via `n_rptrs` and `n_rptrs`. Finally, the auto-increment stride `s_stride` for each pointer is fixed at the point of synthesis.

To support custom increment of the base and offset for each pointer, BMM instructions take the form

```
INSTR n val
```

where `n` specify the pointer. The permitted values of `INSTR` are given in Table 12.

ALU operands accessing DM have an encoding of the form `&<ofs><idx><!>`, elaborated in Table 13.

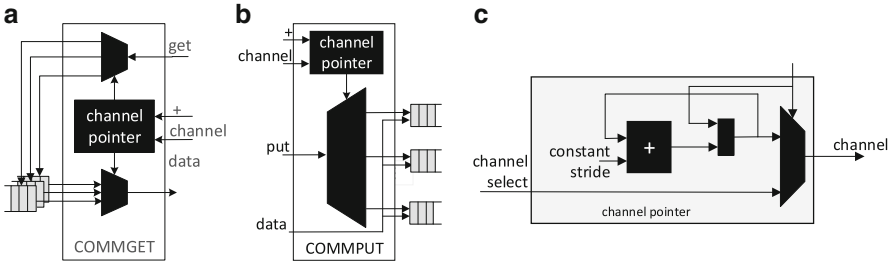


Fig. 18 sFPE COMM adapters. (**a**) COMMGET. (**b**) COMMPUT. (**c**) COMM pointer

Table 14 COMM configuration parameters

Parameter	Meaning	Values
`mode`	Addressing mode	Direct, block
`n_chan`	No. channels	$\mathbb{N} \in [1, 64)$
`s_stride`	Constant stride	$\mathbb{N} \in [1, 64)$

Table 15 sFPE-based MM and ME: itemized PM

	Matrix multiply			Motion estimation		
Class	sFPE	sFPE-B	δ (%)	sFPE	sFPE-B	δ (%)
ALU	32,768	32	−99.9	268353	26	−99.9
COMM	2048	6	−99.7	2467	14	−99.4
CTRL	559	4	−99.7	12582	12	−99.9
NOP	0	6		1026	6	−99.6
Total	35,375	54	−99.8	284428	58	−99.9

7.3 Off-sFPE Communications

The COMMGET and COMMPUT components, illustrated in Fig. 18 are also both configurable according to the parameters in Table 14.

Each of COMMGET and COMMPUT can operate under direct and block addressing modes. In direct mode, individual FIFO channels and be accessed via addresses encoded within the instruction. Instructions for either COMM unit are encoded as:

`^<p><ofs/idx><!>`

where `p` differentiates *peek* (read-without-destroying) and *get* (read-and-destroy) operations, `ofs` denotes the offset, `idx` the pointer reference and `!` autoincrement.

7.4 Stream Frame Processing Efficiency

The effect of these streaming and block addressing features can be profound. The number of instructions required by direct (sFPE) and block-based (sFPE-B) sFPE modes are quoted in Table 15. Very large reductions in program size have resulted

from the addition of block memory management—sFPE-B requires fewer than 1% of the number of instructions required by sFPE. Hence, the stream processing and advanced program and memory control features of the sFPE have a clear beneficial effect on program efficiency and scale. Section 8 compares sFPE-based accelerators for a number of typical signal and image processing operations against real-time performance criteria and custom circuit and soft processor alternatives.

8 Experiments

Accelerators were created using the sFPE for five typical operations:

- 512-point Fast Fourier Transform (FFT)
- 1024 × 1024 Matrix Multiplication
- Sobel Edge Detection (SED) on 1280 × 768 image frames.
- FS-ME: 16 × 16 macroblock, 32 × 32 search window on CIF 352 × 288 images.
- Variable Block Size ME (VBS-ME) with 16 × 16 macroblock, 32 × 32 search window on CIF 720 × 480 images.

The sFPF configurations used to realise each of these operations are described in Table 16. All accelerators target Xilinx Kintex®-7 XC7K70TFBG484 using Xilinx ISE 14.2.

These configurations expose the flexibility of the sFPE. One notable feature is the complete absence of RF in many components, such as MM, FS-ME and FFT. This is a very substantial resource saving which has been enabled as a result of the sFPE being able to stream data from and to COMM components and DM. This flexibility also enables a number of performance and cost advantages, as quoted in Fig. 19. Specifically, the FSME accelerator exhibits real-throughput for H.264; VBS-ME can support real-time processing of 480p video in H.264 Level 2.2. To the best of the authors' knowledge, these are the first time an FPGA-based software-programmable component has demonstrated this capability.

To compare the performance and cost of sFPE-based accelerators relative to custom circuits, sFPE FFTs for IEEE 802.11ac have been developed and compared

Table 16 sFPE-based accelerator configurations

	MM	FS-ME	SED	FFT
Config.	$sFPE_8$	$sFPE_{32}$	$3\text{-}sFPE_3$	5-sFPE
`data_ws`	32	16	16	16
`data_type`	Real	Real	Real	Complex
`dm_depth`	1024	1009	1800	[0,32,32,128,512]
`pm_depth`	64	64	113	[68,78,190,758,1949]
`rf_depth`	0	0	32	0
`n_rptrs`	2	2	1	1
`n_wptrs`	1	1	1	1

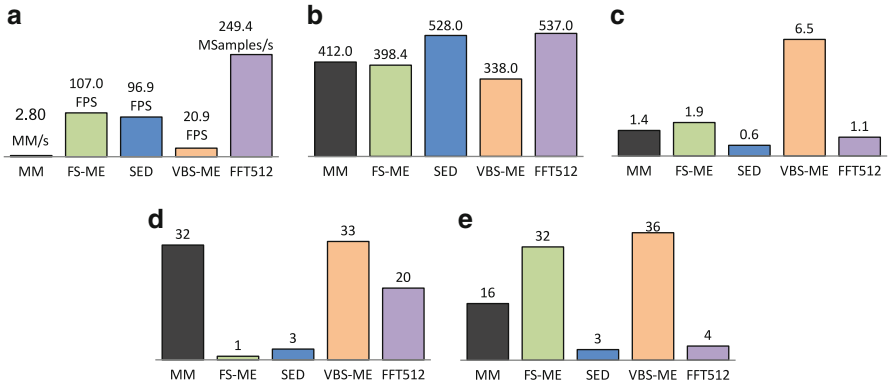


Fig. 19 sFPE accelerators. (**a**) T. (**b**) clk (MHz). (**c**) LUTs. (**d**) DSP48e. (**e**) BRAM

Table 17 802.11ac FFT characteristics

Frequency (MHz)	20	40	80	160
FFT	64	128	256	512
Throughput ($\times 10^6$ Samples/s)	160	320	640	1280

Table 18 sFPE FFT configurations

Parameter	FFT_{64}	FFT_{128}	FFT_{256}	FFT_{512}
Config.	1-$sFPE_3$	1-$sFPE_8$	3-$sFPE_8$	5-$sFPE_8$
`data_ws`	16			
`data_type`	Complex			
`dm_depth`	192	128	[32,256]	[0,32,32,128,512]
`pm_depth`	1184	902	[134,1852]	[68,78,190,758,1949]
`rf_depth`	0			
`sm_depths`	32	64	[32,128]	[0,32,32,64,256]

to both the Xilinx FFT and those generated by Spiral [18]. The IEEE 802.11ac standard [1] mandates 8-channel FFT operations on 20 MHz, 40 MHz, 80 MHz and 160 MHz frequency bands with FFT size and throughput requirements as outlined in Table 17.

These multi-sFPE accelerator configurations are summarised in Table 18—in the case where more than one sFPE is used, the configurations of each are presented in vector format.[2] The performance and cost of the resulting architectures are described in Fig. 20.

Figure 20 shows that the sFPE FFT accelerators for 802.11ac, supported by clock rates of 528 MHz (FFT_{64}, FFT_{128}), 506 MHz (FFT_{256}) and 512 MHz (FFT_{512}), the real-time throughput requirements listed in Table 17 are satisfied. In addition, performance and cost are highly competitive with the Xilinx and Spiral custom circuits. The LUT, DSP48e and BRAM costs are lower than the Xilinx FFT in 9 out

[2]Note that FFT_{512} takes a different configuration to the 512-point FFT previously addressed.

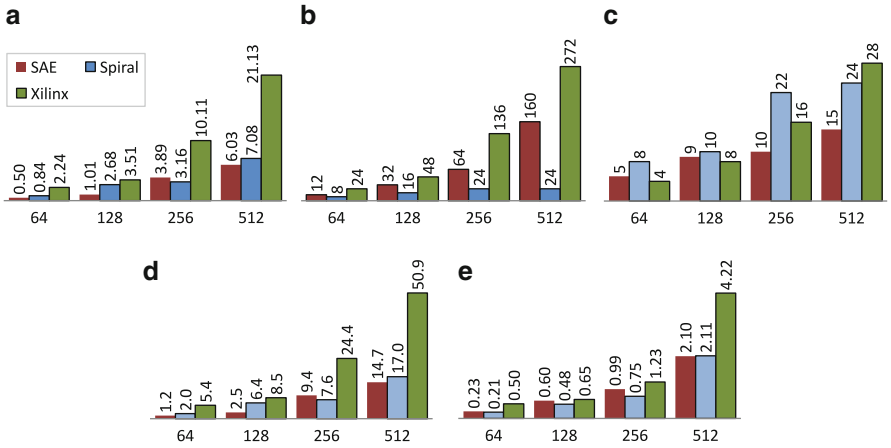


Fig. 20 FPGA-based FFT: performance and cost. (**a**) LUT cost ($\times 10^3$). (**b**) DSP48e cost. (**c**) BRAM cost. (**d**) % device occupied. (**e**) T ($\times 10^9$ Samples/s)

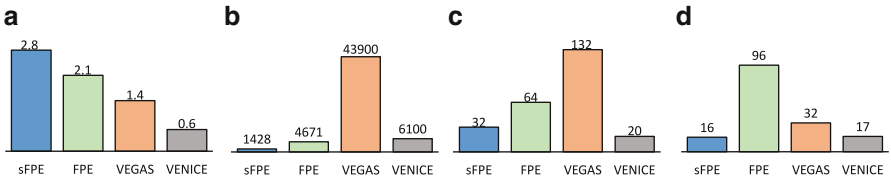


Fig. 21 Softcore matrix multiplication: performance and cost comparison. (**a**) T (MM/s). (**b**) LUTs. (**c**) DSP48e. (**d**) BRAM

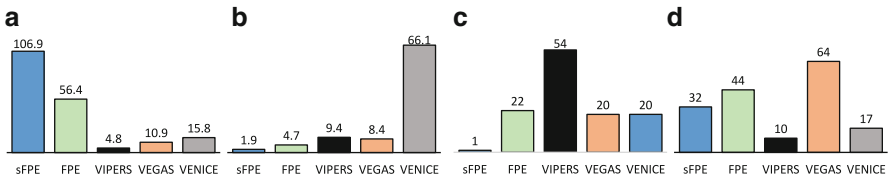


Fig. 22 Softcore FS-ME: performance and cost comparison. (**a**) T (FPS). (**b**) LUTs ($\times 10^3$). (**c**) DSP48e. (**d**) BRAM

of 12 cases, with savings of up to 69, 53 and 56%. Relative to the Spiral FFT, the performance and cost of the sFPE accelerators are similarly encouraging, enabling increased throughput in all but one case and reduced LUT and BRAM costs in 7 out of 8 cases; savings reaching 62.8% and 55% respectively. The Spiral FFTs have consistently lower DSP48e cost, however the total proportion of the device occupied by each, reported in Fig. 20d, remains in favour of the sFPE in all but one instance.

The performance and cost of sFPE-based MM and FS-ME is compared with other soft processors in Figs. 21 and 22.

When applied to MM, the performance and cost advantages relative to 32-way VEGAS (VEGAS_{32}) [9] and 4-way VENICE (VENICE_4) [24] are clear. Relative to

$VEGAS_{32}$, throughput is increased by a factor 2 despite requiring only 25% of the number of datapath lanes. As compared to $VENICE_4$, throughput is increased by a factor 4.7 whilst LUT and BRAM cost are reduced by 76% and 5% respectively.

sFPE-based ME is compared with $VIPERS_{16}$, $VEGAS_4$ and $VENICE_4$ and the FPE in Fig. 22. $sFPE_{32}$ is the only realisation capable of supporting the 30 FPS throughput requirement for standards such as H.264, with absolute throughput increased by factors of 22.3, 9.8 and 6.8 relative to $VIPERS_{16}$, $VEGAS_4$ and $VENICE_4$.

These results demonstrate the benefit of the sFPE relative to other soft processors—coupled performance/cost increases of up to three orders of magnitude. Of course, the softcores to which the sFPE is compared here are general purpose components and hence offer substantially greater run-time processing capability than the sFPE, which is highly tuned to the operation for which it was created. In that respect, the sFPE is more a component for constructing fixed-function accelerators than a general-purpose softcore. However, despite employing similar multi-lane processing approaches as VIPERS, VEGAS and VENICE the sFPE's focus on extreme efficiency, multicore processing, stream processing and novel block memory management have enabled very substantial performance and cost benefits.

9 Summary

Soft processors for FPGA suffer from substantial cost and performance penalties relative to custom circuits hand-crafted at register transfer level. Performance and resource overheads associated with the need for a host general purpose processor, load-store processing, loop handling, addressing mode restrictions and inefficient architectures combine to amplify cost and limit performance.

This paper describes the first approach which challenges this convention. The sFPE presented realises accelerators using multicore networks of fine-grained, high performance and standalone processors. The sFPE enables performance and cost unprecedented amongst soft processors by adopting a streaming operation model to ensure high efficiency. combined with advanced loop handling and addressing constructs for very compact and high performance operation on large data sets. These enable efficiency routinely in excess of 90% and performance and cost which are comparable to custom circuit accelerators and well in advance of existing soft processors.

Specifically, real-time accelerators for 802.11ac FFT and H.264 FS-ME VBS-ME are described; the former of these exhibits performance and cost which are highly competitive with custom circuits. In addition, it is shown how sFPE-based MM and ME accelerators offer improvements in resource/cost by up to three orders of magnitude. To the best of the authors' knowledge, these capabilities are unique, not only for FPGA, but for any semiconductor technology.

This work lays a promising foundation for the construction of complete FPGA accelerators, but in addition may be used to further ease the design process. For

example, in the case where off-chip memory access is required, the programmable nature of the SAE means that it may also be used as a memory controller to execute custom memory access schedules and highly efficient block access. However, resolving this and other accelerator peripheral functions is left as future work.

References

1. 802.11 Working Group: IEEE P802.11ac/D2.2 Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 4: Enhancements for Very High Throughput for Operation in Bands below 6 GHz (2012)
2. Altera Inc.: Nios II Processor Reference Handbook (2014)
3. Altera Inc.: Stratix V Device Handbook (2014)
4. Antikainen, J., Salmela, P., Silven, O., Juntti, M., Takala, J., Myllyla, M.: Application-Specific Instruction Set Processor Implementation of List Sphere Detector. In: Conf. Record of the Forty-First Asilomar Conf. on Signals, Systems and Computers, 2007, pp. 943–947 (2007). https://doi.org/10.1109/ACSSC.2007.4487358
5. Barbero, L., Thompson, J.: Fixing the Complexity of the Sphere Decoder for MIMO Detection. IEEE Trans. Wireless Communications pp. 2131–2142 (2008). https://doi.org/10.1109/TWC.2008.060378
6. Barbero, L.G., Thompson, J.S.: Rapid Prototyping of a Fixed-Throughput Sphere Decoder for MIMO Systems. In: IEEE Intl. Conf. on Communications, pp. 3082–3087 (2006). https://doi.org/10.1109/ICC.2006.255278
7. Burg, A., Borgmann, M., Wenk, M., Zellweger, M., Fichtner, W., Bolcskei, H.: VLSI Implementation of MIMO Detection Using The Sphere Decoding Algorithm. IEEE Journal of Solid-State Circuits **40**(7), 1566–1577 (2005). https://doi.org/10.1109/JSSC.2005.847505
8. Cheah, H.Y., F., B., Fahmy, S., Maskell, D.L.: The iDEA DSP Block Based Soft Processor for FPGAs. ACM Trans. Reconfigurable Technol. Syst. **7**(1) (2014)
9. Chou, C.H., Severance, A., Brant, A.D., Liu, Z., Sant, S., Lemieux, G.G.: VEGAS: Soft Vector Processor with Scratchpad Memory. In: Proc. ACM/SIGDA Intl. Symp. Field Programmable Gate Arrays, FPGA '11, pp. 15–24. ACM, New York, NY, USA (2011). https://doi.org/10.1145/1950413.1950420. URL http://doi.acm.org/10.1145/1950413.1950420
10. Chu, X., McAllister, J.: FPGA Based Soft-core SIMD Processing: A MIMO-OFDM Fixed-Complexity Sphere Decoder Case Study. In: IEEE Int. Conf. on Field-Programmable Technology (FPT), pp. 479–484 (2010). https://doi.org/10.1109/FPT.2010.56814639
11. Chu, X., McAllister, J.: Software-Defined Sphere Decoding for FPGA-Based MIMO Detection. IEEE Transactions on Signal Processing **60**(11), 6017–6026 (2012). https://doi.org/10.1109/TSP.2012.2210951
12. Hannig, F., Lari, V., Boppu, S., Tanase, A., Reiche, O.: Invasive Tightly-Coupled Processor Arrays: A Domain-Specific Architecture/Compiler Co-Design Approach. ACM Trans. Embed. Comput. Syst. **13**(4s), 133:1–133:29 (2014). https://doi.org/10.1145/2584660
13. Hanzo, L., Webb, W., Keller, T.: Single and Multi-carrier Quadrature Amplitude Modulation: Principles and Applications for Personal Communications, WLANs and Broadcasting (2000)
14. IEEE802.11n: 802.11n-2009 IEEE Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput (2009). https://doi.org/10.1109/IEEESTD.2009.5307322
15. Janhunen, J., Silven, O., Juntti, M., Myllyla, M.: Software Defined Radio Implementation of K-best List Sphere Detector Algorithm. In: Intl. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), pp. 100–107 (2008). https://doi.org/10.1109/ICSAMOS.2008.4664852

16. Li, M., Bougard, B., Xu, W., Novo, D., Van Der Perre, L., Catthoor, F.: Optimizing Near-ML MIMO Detector for SDR Baseband on Parallel Programmable Architectures. Design, Automation and Test in Europe (DATE) pp. 444–449 (2008). https://doi.org/10.1109/DATE.2008.4484721
17. McAllister, J.: FPGA-based DSP. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, 2nd edn., pp. 363–392. Springer US (2010)
18. Milder, P., Franchetti, F., Hoe, J.C., Püschel, M.: Computer Generation of Hardware for Linear Digital Signal Processing Transforms. ACM Trans. Des. Autom. Electron. Syst. **17**(2), 15:1–15:33 (2012). https://doi.org/10.1145/2159542.2159547
19. Parhami, B.: Computer Arithmetic: Algorithms and Hardware Designs, 2nd edition edn. OUP USA (2010)
20. Pohst, M.: On The Computation of Lattice Vectors of Minimal Length, Successive Minima and Reduced Bases with Applications. SIGSAM Bull. **15**(1), 37–44 (1981). http://doi.acm.org/10.1145/1089242.1089247
21. Qi, Q., Chakrabarti, C.: Parallel High Throughput Soft-output Sphere Decoder. In: IEEE Workshop on Signal Processing Systems (SIPS), pp. 174–179 (2010). https://doi.org/10.1109/SIPS.2010.5624783
22. Ravindran, K., Satish, N., Jin, Y., Keutzer, K.: An FPGA-based soft multiprocessor system for IPv4 packet forwarding. In: Field Programmable Logic and Applications, 2005. International Conference on, pp. 487–492 (2005). https://doi.org/10.1109/FPL.2005.1515769
23. Schnorr, C.P., Euchner, M.: Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. Mathematical Programming **66**(1), 181–199 (1994)
24. Severance, A., Lemieux, G.: VENICE: A Compact Vector Processor for FPGA Applications. In: Field-Programmable Technology (FPT), 2012 Intl. Conf. on, pp. 261–268 (2012). https://doi.org/10.1109/FPT.2012.6412146
25. Unnikrishnan, D., Zhao, J., Tessier, R.: Application specific customization and scalability of soft multiprocessors. In: Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on, pp. 123–130 (2009). https://doi.org/10.1109/FCCM.2009.41
26. Wolniansky, P., Foschini, G., Golden, G., Valenzuela, R.: V-BLAST: An Architecture for Realizing Very High Data Rates Over The Rich-Scattering Wireless Channel. In: 1998 URSI Int. Symp. Signals, Systems, and Electronics, pp. 295–300 (1998). https://doi.org/10.1109/ISSSE.1998.738086
27. Wu, B., Masera, G.: A Novel VLSI Architecture of Fixed-Complexity Sphere Decoder. In: 13th Euromicro Conf. on Digital System Design: Architectures, Methods and Tools, pp. 737–744 (2010). https://doi.org/10.1109/DSD.2010.10
28. Xilinx Inc.: LogiCORE IP CORDIC v4.0 (2011)
29. Xilinx Inc.: LogiCORE IP Fast Fourier Transform v7.1 (2011)
30. Xilinx Inc.: 7 Series DSP48E1 Slice User Guide (2013)
31. Xilinx Inc.: 7 Series FPGAs Memory Resources User Guide (2014)
32. Xilinx Inc.: MicroBlaze Processor Reference Guide (2014)
33. Yiannacouras, P., Steffan, J., Rose, J.: Portable, Flexible, and Scalable Soft Vector Processors. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **20**(8), 1429–1442 (2012). https://doi.org/10.1109/TVLSI.2011.2160463
34. Yu, J., Eagleston, C., Chou, C.H., Perreault, M., Lemieux, G.: Vector Processing as a Soft Processor Accelerator. ACM Trans. Reconfigurable Technology and Systems **2**(2) (2009)

Application-Specific Accelerators for Communications

Chance Tarver, Yang Sun, Kiarash Amiri, Michael Brogioli, and Joseph R. Cavallaro

Abstract For computation-intensive digital signal processing algorithms, complexity is exceeding the processing capabilities of general-purpose digital signal processors (DSPs). In some of these applications, DSP hardware accelerators have been widely used to off-load a variety of algorithms from the main DSP host, including the fast Fourier transform, digital filters, multiple-input multiple-output detectors, and error correction codes (Viterbi, turbo, low-density parity-check) decoders. Given power and cost considerations, simply implementing these computationally complex parallel algorithms with high-speed general-purpose DSP processor is not very efficient. However, not all DSP algorithms are appropriate for off-loading to a hardware accelerator. First, these algorithms should have data-parallel computations and repeated operations that are amenable to hardware implementation. Second, these algorithms should have a deterministic dataflow graph that maps to parallel datapaths. In this chapter, we focus on some of the basic and advanced digital signal processing algorithms for communications and cover major examples of DSP accelerators for communications.

1 Introduction

In current fourth-generation (4G) wireless systems and emerging fifth-generation (5G), the signal processing algorithm complexity has far exceeded the processing capabilities of general-purpose digital signal processors (DSPs). With the inclusion of multiple-input multiple-output (MIMO) technology and advanced forward error correction coding in many wireless systems, it becomes increasingly critical to develop area and power efficient designs. One can not simply implement computation intensive DSP algorithms with gigahertz DSPs. Besides, it is also critical to reduce base station power consumption by utilizing optimized hardware accelerator

C. Tarver (✉) · Y. Sun · K. Amiri · M. Brogioli · J. R. Cavallaro
Rice University, Houston, TX, USA
e-mail: tarver@rice.edu; kiaa@alumni.rice.edu; brogioli@alumni.rice.edu; cavallar@rice.edu

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_14

design. In 5G this is even more true with the addition of massive MIMO where hundreds of antennas will simultaneously serve tens of users at high data rates for greater throughput and spectral efficiency [6, 35].

In this chapter, we will describe a few computationally complex DSP algorithms in a wireless system that are likely to be offloaded to a specialized accelerator yielding high performance. These algorithms include turbo decoding, low-density parity-check (LDPC) decoding, MIMO detection, channel equalization, fast Fourier transform (FFT), inverse fast Fourier transform (IFFT), and digital predistortion (DPD). Often these hardware accelerators are integrated into the same die with DSP processors. In addition, it is also possible to leverage a field-programmable gate array (FPGA) or a graphics processing unit (GPU) to provide reconfigurable massive computation capabilities. This is described for FPGAs in another chapter of this handbook [45].

DSP workloads are typically numerically intensive with large amounts of both instruction and data level parallelism. To exploit this parallelism with a programmable processor, most DSP systems utilize very long instruction word or VLIW architectures. VLIW architectures typically include one or more register files on the processor die versus a single monolithic register file as is often the case in general-purpose computing. Examples of such architectures are the NXP StarCore processor [48], the Texas Instruments TMS320C6x series DSPs [75] as well as SHARC DSPs from Analog Devices [5], to name a few. A comprehensive overview of the general-purpose DSP processors is given in other chapters of this handbook such as [51] and [34].

In some cases, due to the idiosyncratic nature of many DSPs and the implementation of some of the more powerful instructions in the DSP core, an optimizing compiler cannot always target core functionality in a perfect manner. Examples of this include high-performance fractional arithmetic instructions, for example, which may perform highly SIMD functionality which the compiler cannot always deem safe at compile time.

While the aforementioned VLIW based DSP architectures provide increased parallelism and higher numerical throughput performance, this comes at a cost of ease in programmability. Typically such machines are dependent on advanced optimizing compilers that are capable of aggressively analyzing the instruction and data level parallelism in the target workloads, and mapping it onto the parallel hardware. Due to a large number of parallel functional units and deep pipeline depths, modern DSPs are often difficult to hand program at the assembly level while achieving optimal results. As such, one technique used by the optimizing compiler is to vectorize much of the data level parallelism often found in DSP workloads. In doing this, the compiler can often fully exploit the single instruction multiple data, or SIMD functionality found in modern DSP instruction sets.

Despite such highly parallel programmable processor cores and advanced compiler technology, however, it is quite often the case that the amount of available instruction and data level parallelism in modern signal processing workloads far exceeds the limited resources available in a VLIW based programmable processor core. For example, the implementation complexity for a 40 Kbps DS-CDMA system

would be 41.8 Gflops/s for 60 users [78], not to mention 100 Mbps+ 3GPP LTE system and tens of Gbps in 5G [6]. This complexity largely exceeds the capability of modern DSP processors which typically can provide under 10 Gflops/s performance per core, such as 9.6 Gflops/s TI 6652 DSP processor and 3 Gflops ADI TigerSHARC processor. In other cases, the functionality required by the workload is not efficiently supported by more general-purpose instruction sets typically found in embedded systems. As such the need for acceleration at both the fine-grain and coarse-grain levels is often required; the former for instruction set architecture (ISA) like optimization, and the latter for task like optimization [69].

Additionally, wireless system designers often desire the programmability offered by software running on a DSP core versus a hardware-based accelerator, to allow flexibility in various proprietary algorithms. Examples of this can be functionality such as channel estimation in baseband processing, for which a given vendor may want to use their algorithm to handle various users in varying system conditions versus a pre-packaged solution. Typically these demands result in a heterogeneous system which may include one or more of the following: software programmable DSP cores for data processing, hardware-based accelerator engines for data processing, and in some instances general-purpose processors, GPUs, or micro-controller type solutions for control processing.

The motivations for heterogeneous DSP system solutions including hardware acceleration stem from the tradeoffs between software programmability versus the performance gains of custom hardware acceleration in its various forms. There are a number of heterogeneous accelerator based architectures currently available today, as well as various offerings and design solutions being offered by the research community.

There are a number of DSP architectures which include true hardware-based accelerators which are not programmable by the end user. One example of this is the Texas Instrument's TCI66x series of DSPs which include hardware-based Viterbi or turbo decoder accelerators for acceleration of wireless channel decoding [74].

A recent progression in accelerators is the use of GPUs for general purpose computation including signal processing. This is often referred to as GPGPU which stands for General-Purpose computation on Graphics Processing Units. Their high-performance, many-core architecture is well suited for problems that fit a SIMD computational style. Moreover, because they are software based, there is more flexibility and portability. Some examples of GPU based accelerators include a Massive MIMO detector in [40] and an accelerator for LTE-A turbo decoding in [84]. GPU technology and tools are rapidly progressing and can offer more parallelism and faster performance for less power. They are also now standard in most mobile system-on-chips such as the Adreno GPU on Snapdragon chips by Qualcomm. Considering their prevalence and performance, GPUs are a resource that should be exploited on modern, heterogeneous compute systems.

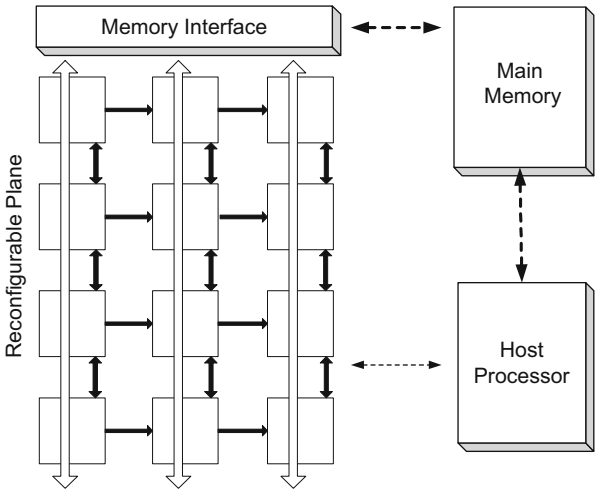


Fig. 1 Traditional coarse grained accelerator architecture [10]

1.1 Coarse Grain Versus Fine Grain Accelerator Architectures

Coarse-grain accelerator based DSP systems entail a co-processor type design whereby larger amounts of work are run on the sometimes configurable co-processor device. Current technologies being offered in this area support offloading of functionality such as FFT and various matrix-like computations to the accelerator versus executing in software on the programmable DSP core. For examples of architectures for accelerating FFT computations, see [23] in this handbook.

As shown in Fig. 1, coarse-grained heterogeneous architectures typically include a loosely coupled computational grid attached to the host processor. These types of architectures are sometimes built using an FPGA, ASIC, or a vendor programmable acceleration engine for portions of the system. Tightly coupled loop nests or kernels are then offloaded from executing in software on the host processor to executing in hardware on the loosely coupled grid.

Fine-grain accelerator based architectures are the flip-side to the coarse-grained accelerator mindset. Typically, ISAs provide primitives that allow low-cost, low-complexity implementations while still maintaining high performance for a broad range of input applications. In certain cases, however, it is often advantageous to offer instructions specialized to the computational needs of the application. Adding new instructions to the ISA, however, is a difficult decision to make. On the one hand, they may provide significant performance increases for certain subsets of applications, but they must still be general enough such that they are useful across a much wider range of applications. Additionally, such instructions may become obsolete as software evolves and may complicate future hardware implementations of the ISA [86]. Vendors such as Cadence, however, offer toolsets to produce

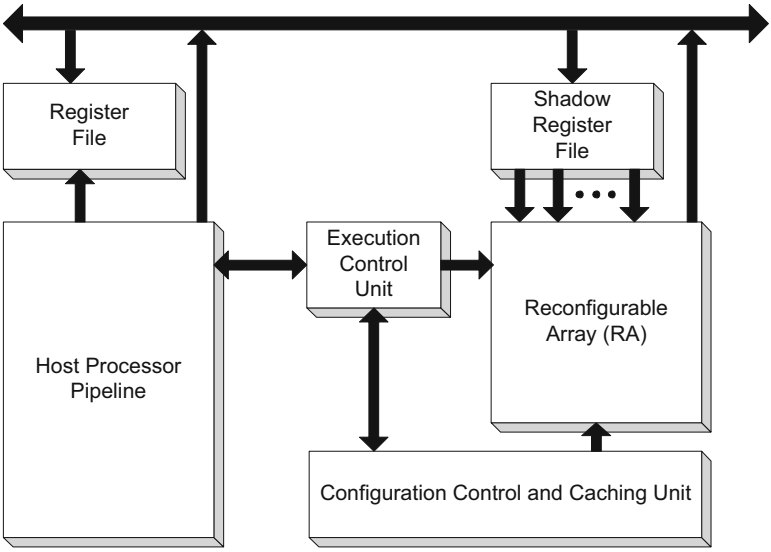


Fig. 2 Example fine-grained reconfigurable architecture with customizable ALU for ISA extensions [10]

configurable, extensible processor architectures typically targeted at the embedded community [14]. These types of products typically allow the user to configure a predefined subset of processor components to fit the specific demands of the input application. Figure 2 shows the layout of a typical fine-grained reconfigurable architecture whereby a custom ALU is coupled with the host processors pipeline.

In summary, both fine and coarse-grained acceleration can be beneficial to the computational demands of DSP applications. Depending on the overall design constraints of the system, designers may choose a heterogeneous coarse-grained acceleration system or a strict software programmable DSP core system.

1.2 Hardware/Software Workload Partition Criteria

In partitioning any workload across a heterogeneous system comprised of reconfigurable computational accelerators, programmable DSPs or programmable host processors, and varied memory hierarchy, a number of criteria must be evaluated in addition to application profile information to determine whether a given task should execute in software (on the host processor or GPU) or in hardware (on FPGA or ASIC), as well where in the overall system topology each task should be mapped. It is these sets of criteria that typically mandate the software partitioning, and ultimately determine the topology and partitioning of the given system.

Spatial locality of data is one concern in partitioning a given task. In a typical software implementation running on a host processor, the ability to access data in a particular order efficiently is of great importance to performance. Issues such as latency to memory, data bus contention, data transfer times to local compute element such as accelerator local memory, as well as type and location of memory all need to be taken into consideration. In cases where data is misaligned, or not contiguous or uniformly strided in memory, additional overhead may be needed to arrange data before block DMA transfers can take place or data can efficiently be computed on. In cases where data is not aligned properly in memory, significant performance degradations can be seen due to decreased memory bandwidth when performing unaligned memory accesses on some architectures. When data is not uniformly strided, it may be difficult to burst transfer even single dimensional strips of memory via DMA engines. Consequently, with non-uniformly strided data it may be necessary to perform data transfers into local accelerator memory for computation via programmed I/O on the part of the host DSP. Inefficiencies in such methods of data transfer can easily overshadow any computational benefits achieved by compute acceleration of the FPGA. The finer the granularity of computation offloaded for acceleration in terms of compute time, quite often the more pronounced the side effects of data memory transfer to local accelerator memory.

Data level parallelism is another important criteria in determining the partitioning for a given application. Many applications targeted at VLIW-like architectures, especially signal processing applications, exhibit a large amount of both instruction and data level parallelism [31]. Many signal processing applications often contain enough data level parallelism to exceed the available functional units of a given architecture. FPGA fabrics, GPUs, and highly parallel ASIC implementations can exploit these computational bottlenecks in the input application by providing not only large numbers of functional units but also large amounts of local block data RAM to support very high levels of instruction and data parallelism, far beyond that of what a typical VLIW signal processing architecture can afford in terms of register file real estate. Furthermore, depending on the instruction set architecture of the host processor or DSP, performing sub-word or multiword operations may not be feasible given the host machine architecture. Most modern DSP architectures have fairly robust instruction sets that support fine-grained multiword SIMD acceleration to a certain extent. It is often challenging, however, to efficiently load data from memory into the register files of a programmable SIMD style processor to be able to efficiently or optimally utilize the SIMD ISA in some cases.

Computational complexity of the application often bounds the programmable DSP core, creating a compute bottleneck in the system. Algorithms that are implemented in FPGA are often computationally intensive, exploiting greater amounts of instruction and data level parallelism than the host processor can afford, given the functional unit limitations and pipeline depth. By mapping computationally intense bottlenecks in the application from software implementation executing on host processor to hardware implementation in FPGA, one can effectively alleviate bottlenecks on the host processor and permit extra cycles for additional computation or algorithms to execute in parallel.

Task-level parallelism in a portion of the application can play a role in the ideal partitioning as well. Quite often, embedded applications contain multiple tasks that can execute concurrently, but have a limited amount of instruction or data level parallelism within each unique task [79]. Applications in the networking space, and baseband processing at layers above the data plane typically need to deal with processing packets and traversing packet headers, data descriptors and multiple task queues. If the given task contains enough instruction and data level parallelism to exhaust the available host processor compute resources, it can be considered for partitioning to an accelerator. In many cases, it is possible to concurrently execute multiple of these tasks in parallel either across multiple host processors or across both host processor and FPGA compute engine depending on data access patterns and cross task data dependencies. There are a number of architectures which have accelerated tasks in the control plane, versus data plane, in hardware. One example of this is the NXP Semiconductor QorIQ platform [49] which provides hardware acceleration for frame managers, queue managers, and buffer managers. In doing this, the architecture effectively frees the programmable processor cores from dealing with control plane management.

2 Hardware Accelerators for Communications

Processors for wireless cellular systems beyond the second-generation systems typically require high speed, throughput, and flexibility. In addition to this, computationally intensive algorithms are used to remove often high levels of multiuser interference especially in the presence of multiple transmit and receive antenna MIMO systems. Time-varying wireless channel environments can also dramatically deteriorate the performance of the transmission, further requiring powerful channel equalization, detection, and decoding algorithms for different fading conditions at the mobile handset. In these types of environments, it is often the case that the amount of available parallel computation in a given application or kernel far exceeds the available functional units in the target processor. Even with modern VLIW style DSPs, the number of available functional units in a given clock cycle is limited and prevents full parallelization of the application for maximum performance. Further, the area and power constraints of mobile handsets make a software-only solution difficult to realize.

Figure 3 depicts a typical MIMO receiver model. Three major blocks, MIMO channel estimator and equalizer, MIMO detector, and channel decoder, determine the computation requirements of a MIMO receiver. Thus, it is natural to offload these very computationally intensive tasks to hardware accelerators to support high data rate applications. Example include 3GPP LTE-Advanced with 3 Gbps downlink peak data rate, and future standards such as 5G targeting 10 Gbps speeds.

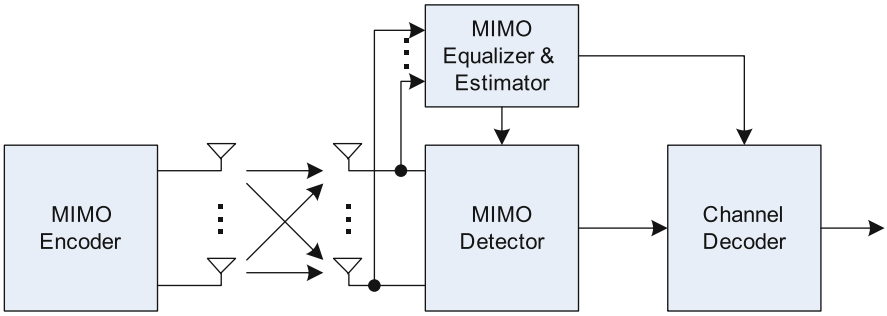


Fig. 3 Basic structure of a MIMO receiver

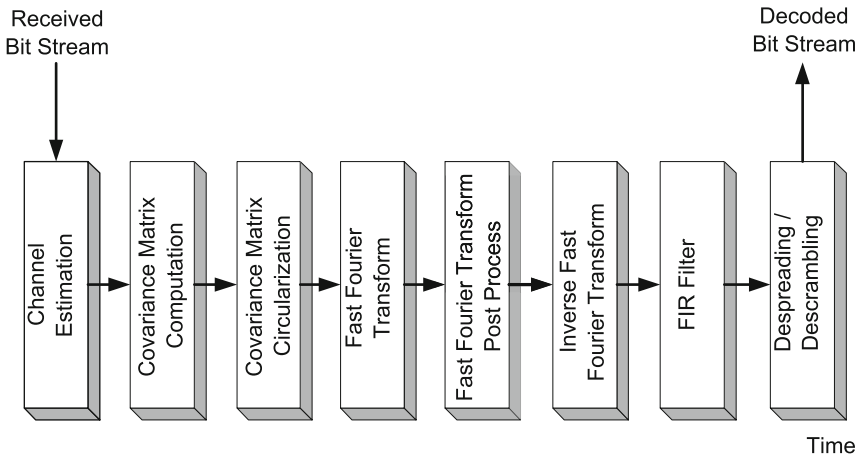


Fig. 4 Workload partition for a channel equalizer

2.1 MIMO Channel Equalization Accelerator

The total workload for a given channel equalizer performed as a baseband processing part on the mobile receiver can be decomposed into multiple tasks as depicted in Fig. 4. This block diagram shows the various software processing blocks, also known as kernels, that make up the channel equalizer firmware executing on the DSP of the mobile receiver. The tasks are channel estimation based on known pilot sequence, covariance computation (first row or column) and circularization, FFT/IFFT post-processing for updating equalization coefficients, finite-impulse response (FIR) filtering applied on the received samples (received frame), and user detection (despreading-descrambling) for recovering the user information bits. The computed data is shared between the various tasks in a pipeline fashion, in that the output of covariance computation is used as the input to the matrix circularization algorithm.

The computational complexity of the various components of the workload vary with the number of users in the system, the number of users entering and leaving the cell, and the channel conditions. Regardless of this variance in the system conditions at runtime, the dominant portions of the workload are the channel estimation, FFT, IFFT, FIR filtering, and despreading-descrambling.

As an example, using the workload partition criteria for partitioning functionality between a programmable DSP core and system containing multiple hardware for a 3.5G HSDPA system, it has been shown that impressive performance results can be obtained. In studying the bottlenecks of such systems when implemented on a programmable DSP core in software, it has been found the key bottlenecks in the system to be the channel estimation, FFT, IFFT, FIR filter, and to a lesser extent despreading-descrambling as illustrated in Fig. 4 [11]. By migrating the 3.5G implementation from a solely software based implementation executing on a TMS320C64x based programmable DSP core to a heterogeneous system containing not only programmable DSP cores but also distinct hardware acceleration for the various bottlenecks, the authors achieve almost an 11.2× speedup in the system [11]. Figure 5 illustrates the system partitioning between programmable DSP core and hardware (e.g. FPGA or ASIC) accelerator that resulted in load balancing the aforementioned bottlenecks.

The arrows in the diagram illustrate the data flow between local programmable DSP core on-chip data caches and the local RAM arrays. In the case of channel estimation, the work is performed in parallel between the programmable DSP core and hardware acceleration. Various other portions of the workload are offloaded to hardware-based accelerators while the programmable DSP core performs the lighter-weight signal-processing code and bookkeeping.

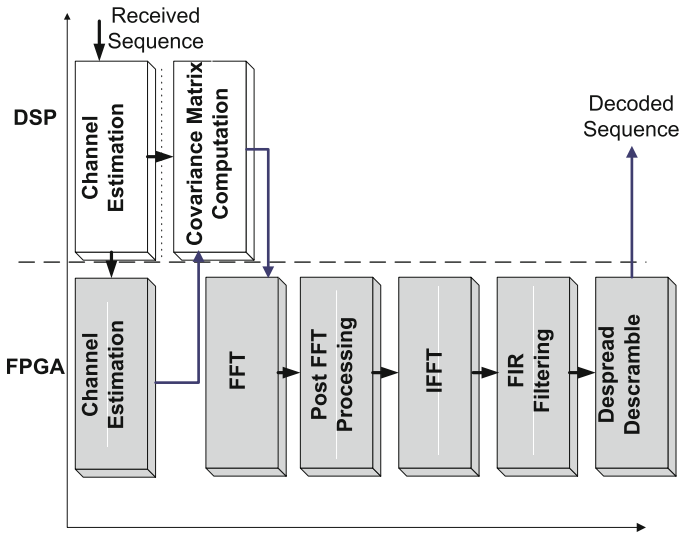


Fig. 5 Channel equalizer DSP/hardware accelerator partitioning

Fig. 6 MIMO transmitter and receiver

Despite the ability to achieve over 11× speedup in performance, it is important to note that the experimental setup used in these studies was purposely pessimistic. The various FFT, IFFT, etc. compute blocks in these studies were offloaded to discrete FPGA/ASIC accelerators. As such, data had to be transferred, for example, from local IFFT RAM cells to FIR filter RAM cells. This is pessimistic in terms of data communication time. In most cases the number of gates required for a given accelerator implemented in FPGA/ASIC was low enough that multiple accelerators could be implemented within a single FPGA/ASIC drastically reducing chip-to-chip communication time.

2.2 MIMO Detection Accelerators

MIMO systems, Fig. 6, have been shown to be able to greatly increase the reliability and data rate for point-to-point wireless communication [35, 72]. Multiple-antenna systems can be used to improve the reliability and diversity in the receiver by providing the receiver with multiple copies of the transmitted information. This diversity gain is obtained by employing different kinds of space-time block codes (STBC) [3, 70, 71]. In such cases, for a system with M transmit antennas and N receive antennas and over a time span of T time symbols, the system can be modeled as

$$\mathbf{Y} = \mathbf{HX} + \mathbf{N}, \tag{1}$$

where $\mathbf{H}$ is the $N \times M$ channel matrix. Moreover, $\mathbf{X}$ is the $M \times T$ space-time code matrix where its x_{ij} element is chosen from a complex-valued constellation Ω of the order $w = |\Omega|$ and corresponds to the complex symbol transmitted from the i-th antenna at the j-th time. The $\mathbf{Y}$ matrix is the received $N \times T$ matrix where y_{ij} is the perturbed received element at the i-th receive antenna at the j-th time. Finally, $\mathbf{N}$ is the additive white Gaussian noise matrix on the receive antennas at different time slots.

MIMO systems could also be used to further expand the transmit data rate using other space-time coding techniques, particularly layered space-time (LST) codes [19]. One of the most prominent examples of such space-time codes is Vertical Bell Laboratories Layered Space-Time (V-BLAST) [25], otherwise known as spatial multiplexing (SM). In the spatial multiplexing scheme, independent symbols are transmitted from different antennas at different time slots; hence, supporting even higher data rates compared to space-time block codes of lower data rate [3, 70]. The

spatial multiplexing MIMO system can be modeled similarly to Eq. (1) with $T = 1$ since there is no coding across the time domain:

$$\mathbf{y} = \mathbf{H}\mathbf{x} + \mathbf{n}, \tag{2}$$

where $\mathbf{H}$ is the $N \times M$ channel matrix, $\mathbf{x}$ is the M-element column vector where its x_i-th element corresponds to the complex symbol transmitted from the i-th antenna, and $\mathbf{y}$ is the received N-th element column vector where y_i is the perturbed received element at the i-th receive antenna. The additive white Gaussian noise vector on the receive antennas is denoted by $\mathbf{n}$.

While spatial multiplexing can support very high data rates, the complexity of the maximum-likelihood detector in the receiver increases exponentially with the number of transmit antennas. Thus, unlike the case in Eq. (1), the maximum-likelihood detector for Eq. (2) requires a complex architecture and can be very costly. In order to address this challenge, a range of detectors and solutions have been studied and implemented. In this section, we discuss some of the main algorithmic and architectural features of such detectors for spatial multiplexing MIMO systems.

2.2.1 Maximum-Likelihood (ML) Detection

The maximum likelihood (ML) or optimal detection of MIMO signals is known to be an NP-complete problem. The ML detector for Eq. (2) is found by minimizing the

$$\left|\left|\mathbf{y} - \mathbf{H}\mathbf{x}\right|\right|_2^2 \tag{3}$$

norm over all the possible choices of $\mathbf{x} \in \Omega^M$. This brute-force search can be a very complicated task, and as already discussed, incurs an exponential complexity in the number of antennas. In fact for M transmit antennas and modulation order of $w = |\Omega|$, the number of possible $\mathbf{x}$ vectors is w^M. Thus, unless for small dimension problems, it would be infeasible to implement it within a reasonable area-time constraint [12, 22].

2.2.2 Sphere Detection

Sphere detection can be used to achieve ML (or close-to-ML) detection. In fact, while the norm minimization of Eq. (3) is exponential complexity, it has been shown that using the sphere detection method, the ML solution can be obtained with much lower complexity [18, 29, 30, 80].

In order to avoid the significant overhead of the ML detection, the distance norm can be simplified [17] as follows:

$$D(\mathbf{s}) = \| \mathbf{y} - \mathbf{Hs} \|^2$$
$$= \| \mathbf{Q^H y} - \mathbf{Rs} \|^2 = \sum_{i=M}^{1} |y_i{}' - \sum_{j=i}^{M} R_{i,j} s_j|^2, \tag{4}$$

where $\mathbf{H} = \mathbf{QR}$ represents the channel matrix QR decomposition, $\mathbf{R}$ is an upper triangular matrix, $\mathbf{QQ^H} = \mathbf{I}$ and $\mathbf{y}' = \mathbf{Q^H y}$.

The norm in Eq. (4) can be computed in M iterations starting with $i = M$. When $i = M$, i.e. the first iteration, the initial partial norm is set to zero, $T_{M+1}(\mathbf{s}^{(M+1)}) = 0$. Using the notation of [13], at each iteration the partial Euclidean distances (PEDs) at the next levels are given by

$$T_i(\mathbf{s}^{(i)}) = T_{i+1}(\mathbf{s}^{(i+1)}) + |e_i(\mathbf{s}^{(i)})|^2 \tag{5}$$

with $\mathbf{s}^{(i)} = [s_i, s_{i+1}, \ldots, s_M]^T$, and $i = M, M-1, \ldots, 1$, where

$$|e_i(\mathbf{s}^{(i)})|^2 = |y_i{}' - R_{i,i} s_i - \sum_{j=i+1}^{M} R_{i,j} s_j|^2. \tag{6}$$

One can envision this iterative algorithm as a tree traversal with each level of the tree corresponding to one i value or transmit antenna, and each node having w' children based on the modulation chosen.

The norm in Eq. (6) can be computed in M iterations starting with $i = M$, where M is the number of transmit antennas. At each iteration, partial (Euclidian) distances, $PD_i = |y_i{}' - \sum_{j=i}^{M} R_{i,j} s_j|^2$ corresponding to the i-th level, are calculated and added to the partial norm of the respective parent node in the $(i-1)$-th level, $PN_i = PN_{i-1} + PD_i$. When $i = M$, i.e. the first iteration, the initial partial norm is set to zero, $PN_{M+1} = 0$. Finishing the iterations gives the final value of the norm. As shown in Fig. 7, one can envision this iterative algorithm as a tree traversal problem where each level of the tree represents one i value, each node has its own PN, and w children, where w is the QAM modulation size. In order to reduce the search complexity, a threshold, C, can be set to discard the nodes with $PN > C$. Therefore, whenever a node k with a $PN_k > C$ is reached, any of its children will have $PN \geq PN_k > C$. Hence, not only the k-th node, but also its children, and all nodes lying beneath the children in the tree, can be pruned out.

There are different approaches to search the entire tree, mainly classified as depth-first search (DFS) approach and K-best approach, where the latter is based on breadth-first search (BFS) strategy. In DFS, the tree is traversed vertically [4, 13]; while in BFS [27, 82], the nodes are visited horizontally, i.e. level by level.

In the DFS approach, starting from the top level, one node is selected, the PNs of its children are calculated, and among those new computed PNs, one of them, e.g. the one with the least PN, is chosen, and that becomes the parent node for the next iteration. The PNs of its children are calculated, and the same procedure continues

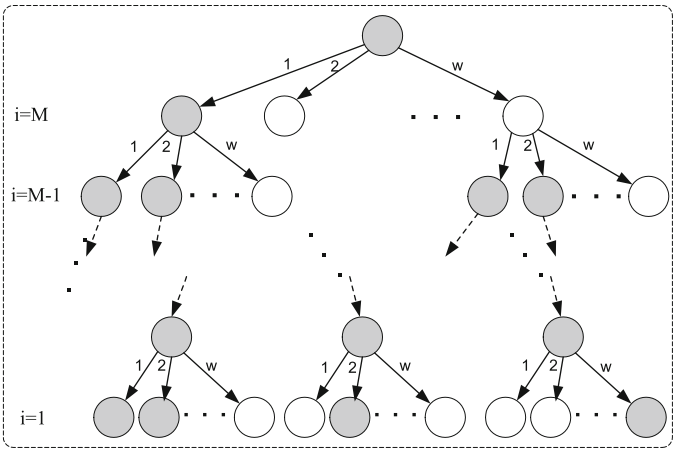


Fig. 7 Calculating the distances using a tree. Partial norms, PNs, of dark nodes are less than the threshold. White nodes are pruned out

until a leaf is reached. At this point, the value of the global threshold is updated with the PN of the recently visited leaf. Then, the search continues with another node at a higher level, and the search controller traverses the tree down to another leaf. If a node is reached with a PN larger than the radius, i.e. the global threshold, then that node, along with all nodes lying beneath that, are pruned out, and the search continues with another node.

The tree traversal can be performed in a breadth-first manner. At each level, only the best K nodes, i.e. the K nodes with the smallest T_i, are chosen for expansion. This type of detector is generally known as the K-best detector. Note that such a detector requires sorting a list of size $K \times w'$ to find the best K candidates. For instance, for a 16-QAM system with $K = 10$, this requires sorting a list of size $K \times w' = 10 \times 4 = 40$ at most of the tree levels.

2.2.3 Computational Complexity of Sphere Detection

In this section, we derive and compare the complexity of the proposed techniques. The complexity in terms of number of arithmetic operations of a sphere detection operation is given by

$$J_{SD}(M, w) = \sum_{i=M}^{1} J_i E\{D_i\}, \tag{7}$$

where J_i is the number of operations per node in the i-th level. In order to compute J_i, we refer to the VLSI implementation of [13], and note that, for each node, one needs to compute the $R_{i,j}s_j$, multiplications, where, except for the diagonal

element, $R_{i,i}$, the rest of the multiplications are complex valued. The expansion procedure, Eq. (4), requires computing $R_{i,j}s_j$ for $j = i+1, \ldots, M$, which would require $(M-i)$ complex multiplications, and also computing $R_{i,i}s_i$ for all the possible choices of $s_j \in \Omega$. Even though, there are w different s_js, there are only $(\frac{\sqrt{w}}{2} - 1)$ different multiplications required for QAM modulations. For instance, for a 16-QAM with $\{\pm 3 \pm 3j, \pm 1 \pm 1j, \pm 3 \pm 1j, \pm 1 \pm 3j\}$, computing only $(R_{i,j} \times 3)$ would be sufficient for all the choices of modulation points. Finally, computing the $\| \, . \, \|^2$ requires a squarer or a multiplier, depending on the architecture and hardware availabilities.

In order to compute the number of adders for each norm expansion in (4), we note that there are $(M-i)$ complex valued adders required for ${y_i}' - \sum_{j=i+1}^{M} R_{i,j}s_j$, and w more complex adders to add the newly computed $R_{i,i}s_i$ values. Once the w different norms, $|{y_i}' - \sum_{j=i}^{M} R_{i,j}s_j|^2$, are computed, they need to be added to the partial distance coming from the higher level, which requires w more addition procedures. Finally, unless the search is happening at the end of the tree, the norms need to be sorted, which assuming a simple sorter, requires $w(w+1)/2$ compare-select operations.

Therefore, keeping in mind that each complex multiplier corresponds to four real-valued multipliers and two real-valued adders, and that every complex adder corresponds to two real-valued adders, J_i is calculated by

$$
\begin{aligned}
J_i(M, w) &= J_{mult} + J_{add}(M, w) \\
J_{mult}(M, w) &= ((\frac{\sqrt{w}}{2} - 1) + 4(M-i) + 1) \\
J_{add}(M, w) &= (2(M-i) + 2w + w) + (w(w+1)/2) \cdot sign(i-1),
\end{aligned}
$$

where $sign(i-1)$ is used to ensure sorting is counted only when the search has not reached the end of the tree, and is equal to:

$$
sign(t) = \begin{cases} 1 & t \geq 1 \\ 0 & otherwise \end{cases}. \tag{8}
$$

Moreover, we use θ, β and γ to represent the hardware-oriented costs for one adder, one compare-select unit, and one multiplication operation, respectively.

Figure 8 shows the number of addition and multiplication operations needed for a 16-QAM system with different number of antennas.

2.2.4 Depth-First Sphere Detector Architecture

The depth-first sphere detection algorithm [13, 18, 22, 30] traverses the tree in a depth-first manner: the detector visits the children of each node before visiting its siblings. A constraint, referred to as radius, is often set on the PED for each level

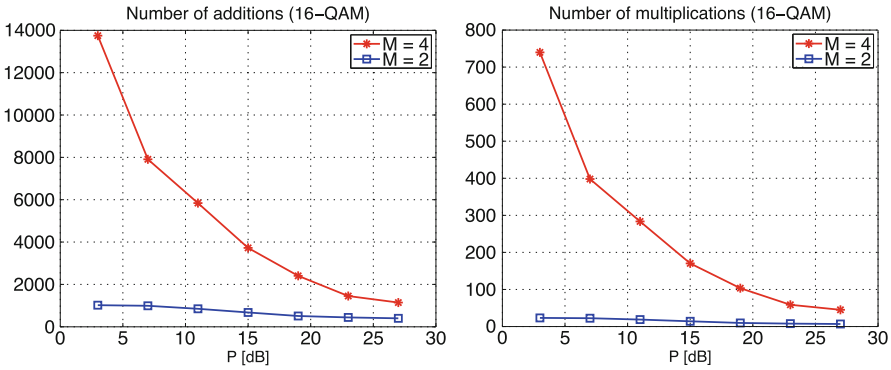


Fig. 8 Number of addition and multiplications operations for 16-QAM with different number of antennas, M

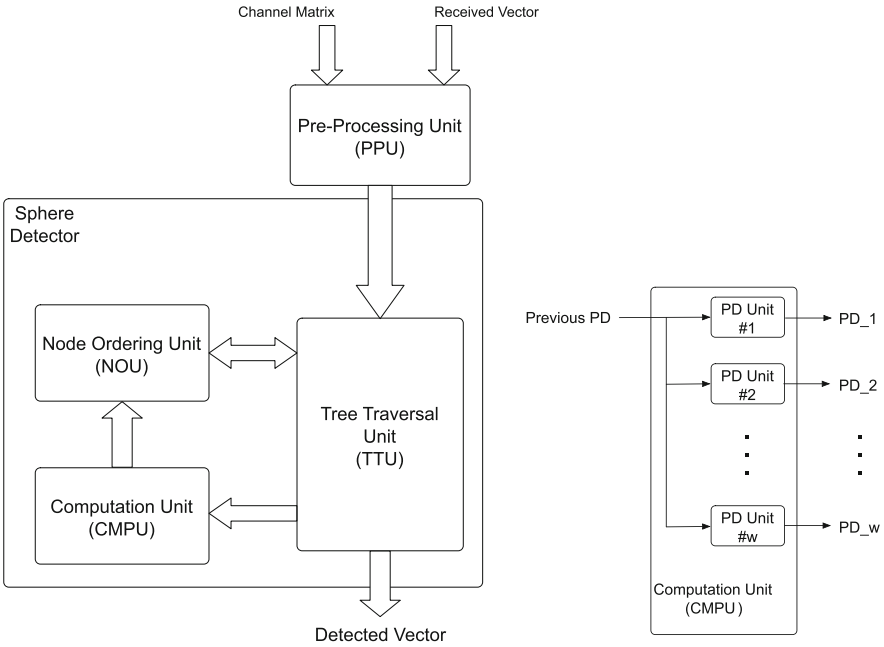


Fig. 9 Sphere detector architecture with multiple PED function units

of the tree. A generic depth-first sphere detector architecture is shown in Fig. 9. The pre-processing unit (PPU) is used to compute the QR decomposition of the channel matrix as well as calculate $\mathbf{Q}^H\mathbf{y}$. The tree traversal unit (TTU) is the controlling unit which decides in which direction and with which node to continue. The computation unit (CMPU) computes the partial distances, based on Eq. (4), for w different s_j. Each PD unit computes $|y_i' - \sum_{j=i}^{M} R_{i,j}s_j|^2$ for each of the w children of a node.

Table 1 FPGA resource utilization for sphere detector

Device	Xilinx Virtex-4 xc4vfx100-12ff1517
Number of slices	4065/42176 (9%)
Number of FFs	3344/84352 (3%)
Number of look-up tables	6457/84352 (7%)
Number of RAMB16	3/376 (1%)
Number of DSP48s	32/160 (20%)
Max. Freq.	125.7 MHz

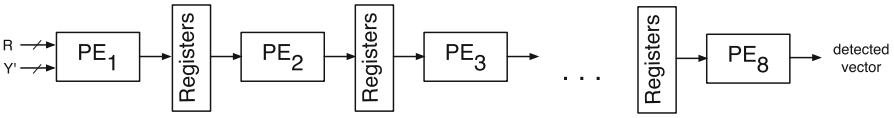


Fig. 10 The K-best MIMO detector architecture: the intermediate register banks contain the sorting information as well as the other values, i.e. **R** matrix

Finally, the node ordering unit (NOU) is for finding the minimum and saving other legitimate candidates, i.e. those inside R_i, in the memory.

As an example to show the algorithm complexity, an FPGA implementation synthesis result for a 50 Mbps 4×4 16-QAM depth-first sphere detector is summarized in Table 1 [4].

2.2.5 K-Best Detector Architecture

K-best is another popular algorithm for implementing close-to-ML MIMO detection [16, 27, 82]. The performance of this scheme is suboptimal compared to ML and sphere detection. However, it has a fixed complexity and relatively straightforward architecture. In this section, we briefly introduce the architecture [27] to implement the K-best MIMO detector. As illustrated in Fig. 10, the PE elements at each stage compute the Euclidean norms of Eq. (6), and find the best K candidates, i.e. the K candidates with the smallest norms, and pass them as the surviving candidates to the next level. It should be pointed out that Eq. (2) can be decomposed into separate real and imaginary parts [27], which would double the size of the matrices. While such decomposition reduces the complex-valued operations of nodes into real-valued operations, it doubles the number of levels of the tree. Therefore, as shown in Fig. 10, there are 8 K-best detection levels for the 4-antenna system. By selecting the proper K value, the real-value decomposition MIMO detection will not cause performance degradation compared to the complex-value MIMO detection [47].

In summary, both depth-first and K-best detectors have a regular and parallel data flow that can be efficiently mapped to hardware. The large amount of required multiplications makes the algorithm very difficult to be realized in a DSP processor. As the main task of the MIMO detector is to search for the best candidate in a very

short time period, it would be more efficient to be mapped on a parallel hardware searcher with multiple processing elements. Thus, to sustain the high throughput MIMO detection, a MIMO hardware accelerator is necessary.

2.3 Channel Decoding Accelerators

Error correcting codes are widely used in digital transmission, especially in wireless communications to combat the harsh wireless transmission medium. To achieve high throughput, researchers are investigating advanced error correction codes that approach the capacity of a channel. The most commonly used error correcting codes in modern systems are convolutional codes, turbo codes, and low-density parity-check (LDPC) codes. As a core technology in wireless communications, forward error correction (FEC) coding has migrated from the basic 2G convolutional/block codes to more powerful 3G turbo codes, LDPC codes for 4G and 802.11ac systems, and potentially a new class of codes called polar codes for 5G.

As codes become more complicated, the implementation complexity, especially the decoder complexity, increases dramatically which largely exceeds the capability of the general-purpose DSP processor. Even the most capable DSPs today would need some types of acceleration coprocessor to offload the computation-intensive error correcting tasks. Moreover, it would be much more efficient to implement these decoding algorithms on dedicated hardware because typical error correction algorithms use special arithmetic and therefore are more suitable for ASICs or FPGAs. Bitwise operations, linear feedback shift registers, and complex look-up tables can be very efficiently realized with ASICs/FPGAs.

In this section, we will present some important error correction algorithms and their efficient hardware architectures. We will cover major error correction codes used in the current and next generation communication standards, such as 3GPP LTE, IEEE 802.11ac Wireless LAN, IEEE 802.16e WiMax, etc.

2.3.1 Turbo Decoder Accelerator Architecture

Turbo codes are a class of high-performance capacity-approaching error-correcting codes [8]. As a break-through in coding theory, turbo codes are widely used in many 3G/4G wireless standards such as CDMA2000, WCDMA/UMTS, 3GPP LTE, and IEEE 802.16e WiMax. However, the inherently large decoding latency and complex iterative decoding algorithm have made it rarely being implemented in a general-purpose DSP. For example, Texas Instruments' latest multi-core DSP processor TI C6614 employs a turbo decoder accelerator to support 365 Mbps LTE turbo codes for the base station [73]. The decoding throughput requirement for 3GPP LTE turbo codes is 80 Mbps in the uplink and 320 Mbps in the downlink. Because the turbo codes used in many standards are very similar, e.g. the encoding polynomials are same for WCDMA/UMTS/LTE, the turbo decoder is often accelerated by reconfigurable hardware.

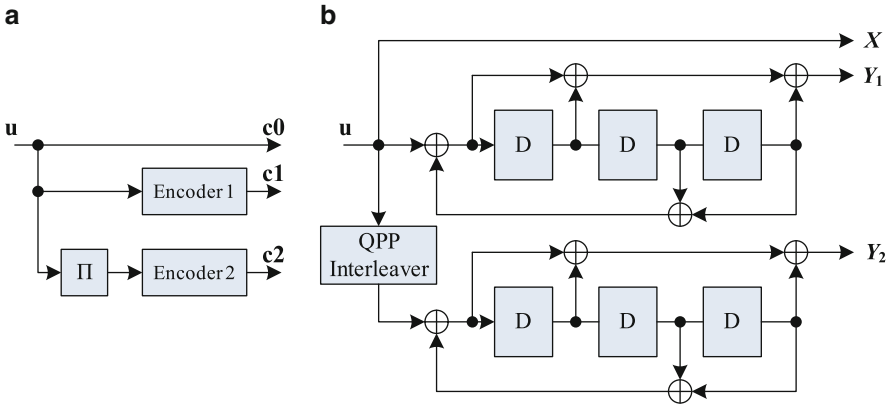


Fig. 11 Turbo encoder structure. (**a**) Basic structure. (**b**) Structure of turbo encoder in 3GPP LTE

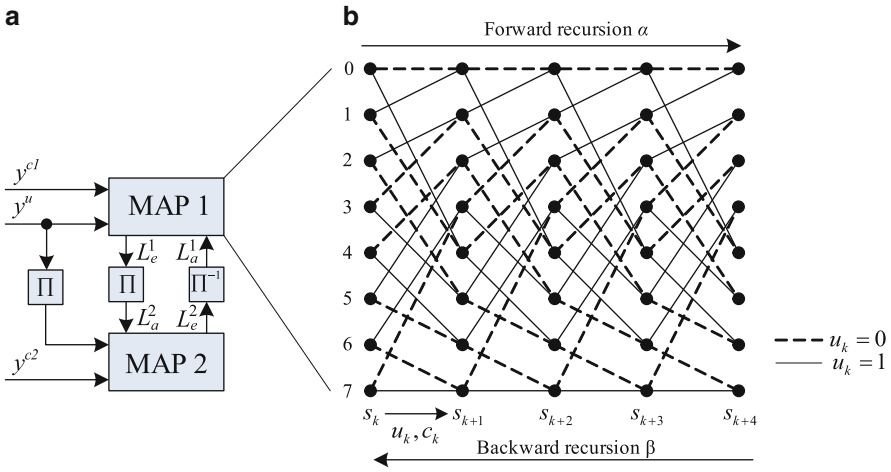


Fig. 12 Basic structure of an iterative turbo decoder. (**a**) Iterative decoding based on two MAP decoders. (**b**) Forward/backward recursion on trellis diagram

A classic turbo encoder structure is depicted in Fig. 11. The basic encoder consists of two systematic convolutional encoders and an interleaver. The information sequence **u** is encoded into three streams: systematic, parity 1, and parity 2. Here the interleaver is used to permute the information sequence into a second different sequence for encoder 2. The performance of a turbo code depends critically on the interleaver structure [55].

The BCJR algorithm [7], also called forward-backward algorithm or Maximum *a posteriori* (MAP) algorithm, is the main component in the turbo decoding process. The basic structure of turbo decoding is functionally illustrated in Fig. 12. The decoding is based on the MAP algorithm. During the decoding process, each

MAP decoder receives the channel data and *a priori* information from the other constituent MAP decoder through interleaving (π) or deinterleaving (π^{-1}), and produces extrinsic information at its output. The MAP algorithm is an optimal symbol decoding algorithm that minimizes the probability of a symbol error. It computes the *a posteriori* probabilities (APPs) of the information bits as follows:

$$\Lambda(\hat{u}_k) = \overset{*}{\underset{\mathbf{u}:u_k=1}{\max}} \Big\{\alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k) + \beta_k(s_k))\Big\} \tag{9}$$

$$- \overset{*}{\underset{\mathbf{u}:u_k=0}{\max}} \Big\{\alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k) + \beta_k(s_k))\Big\}, \tag{10}$$

where α_k and β_k denote the forward and backward state metrics, and are calculated as follows:

$$\alpha_k(s_k) = \overset{*}{\underset{s_{k-1}}{\max}}\{\alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k)\}, \tag{11}$$

$$\beta_k(s_k) = \overset{*}{\underset{s_{k+1}}{\max}}\{\beta_{k+1}(s_{k+1}) + \gamma_k(s_k, s_{k+1})\}. \tag{12}$$

The γ_k term above is the branch transition probability that depends on the trellis diagram, and is usually referred to as a branch metric. The max*{.} operator employed in the above descriptions is the core arithmetic computation that is required by the MAP decoding. It is defined as:

$$\overset{*}{\max}(a, b) = \log(e^a + e^b) = \max(a, b) + \log(1 + e^{-|a-b|}). \tag{13}$$

A basic add-compare-select-add unit is shown in Fig. 13. This circuit can process one step of the trellis per cycle and is often referred to as Radix-2 ACSA unit. To increase the processing speed, the trellis can be transformed by merging every two stages into one radix-4 stage as shown in Fig. 14. Thus, the throughput can be

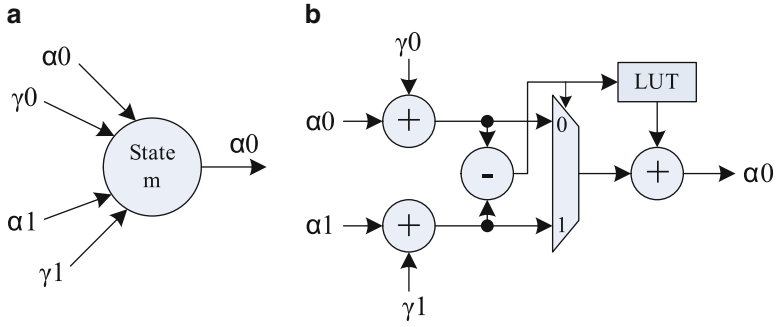


Fig. 13 ACSA structure. (**a**) Flow of state metric update. (**b**) Circuit implementation of an ACSA unit

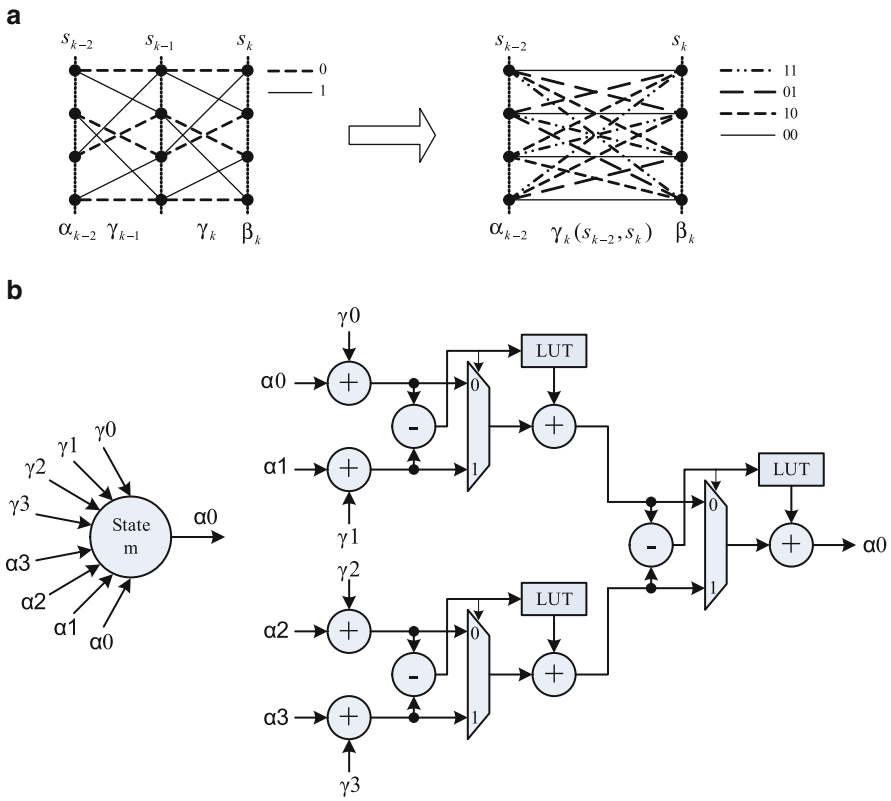


Fig. 14 (**a**) An example of radix-4 trellis. (**b**) Radix-4 ACSA circuit implementation

doubled by applying this transform. For an N state turbo code, N such ACSA unit would be required in each step of the trellis processing. To maximize the decoding throughput, a parallel implementation is usually employed to compute all the N state metrics simultaneously.

In the original MAP algorithm, the entire set of forward metrics needs to be computed before the first soft log-likelihood ratio (LLR) output can be generated. This results in a large storage of K metrics for all N states, where K is the block length and N is the number of states in the trellis diagram. Similar to the Viterbi algorithm, a sliding window algorithm is often applied to the MAP algorithm to reduce the decoding latency and memory storage requirement. By selecting a proper length of the sliding window, e.g. 32 for a rate 1/3 code, there is nearly no bit error rate (BER) performance degradation. Figure 15a shows an example of the sliding window algorithm, where a dummy reverse metric calculation (RMC) is used to get the initial values for β metrics. The sliding window hardware architecture is shown in Fig. 15b. The decoding operation is based on three recursion units, two used for the reverse (or backward) recursions (dummy RMC 1 and effective RMC 2), and one

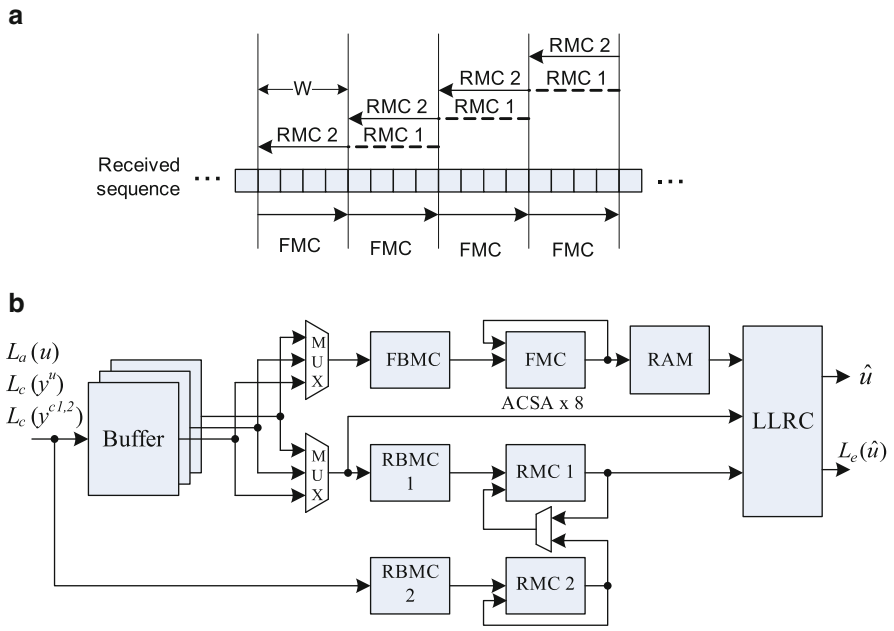


Fig. 15 Sliding window MAP decoder. (**a**) An example of sliding window MAP algorithm, where a dummy RMC is performed to achieve the initial β metrics. (**b**) MAP decoder hardware architecture

for forward metric calculation (FMC). Each recursion unit contains parallel ACSA units. After a fixed latency, the decoder produces the soft LLR outputs on every clock cycle. To further increase the throughput, a parallel sliding window scheme [15, 37, 41, 44, 58, 64, 67, 81, 83] is often applied as shown in Fig. 16.

Another key component of turbo decoders is the interleaver. Generally, the interleaver is a device that takes its input bit sequence and produces an output sequence that is as uncorrelated as possible. Theoretically a random interleaver would have the best performance, but it is difficult to implement a random interleaver in hardware. Thus, researchers are investigating pseudo-random interleavers such as the row-column permutation interleaver for 3G Rel-99 turbo coding as well as the new QPP interleaver [59] for LTE turbo coding. The main differences between these two types of pseudo-random interleavers is the capability to support parallel turbo decoding. The drawback of the row-column permutation interleaver is that memory conflicts will occur when employing multiple MAP decoders for parallel decoding. Extra buffers are necessary to solve the memory conflicts caused by the row-column permutation interleaver [56]. Given an information block length N, the x-th QPP interleaved output position is given by

$$\Pi(x) = (f_2 x^2 + f_1 x) \bmod N, 0 \leq x, f_1, f_2 < N. \tag{14}$$

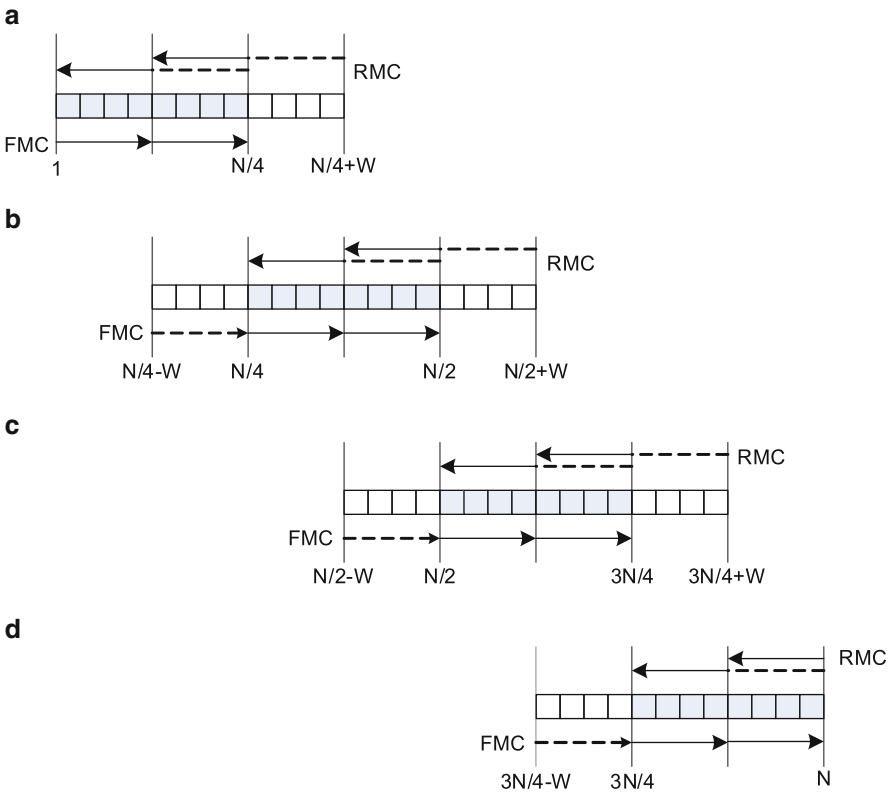


Fig. 16 An example of parallel sliding window decoding, where a decode block is sliced into four sections. The sub-blocks are overlapped by one sliding window length W in order to get the initial value for the boundary states

It has been shown in [59] that the QPP interleaver will not cause memory conflicts as long as the parallelism level is a factor of N. The simplest approach to implement an interleaver is to store all the interleaving patterns in non-violating memory such as ROM. However, this approach can become very expensive because it is necessary to store a large number of interleaving patterns to support decoding of multiple block size turbo codes such as 3GPP LTE turbo codes. Fortunately, there usually exists an efficient hardware implementation for the interleaver. For example, Fig. 17 shows a circuit implementation for the QPP interleaver in 3GPP LTE standard [67].

A basic turbo accelerator architecture is shown in Fig. 18. The main difference between the Viterbi decoder and the turbo decoder is that the turbo decoder is based on the iterative message passing algorithms. Thus, a turbo accelerator may need more communication and control coordination with the DSP host processor. For example, the interleaving addresses can be generated by the DSP processor and passed to the turbo accelerator. The DSP can monitor the decoding process to decide when to terminate the decoding if there are no more decoding gains. Alternately, the

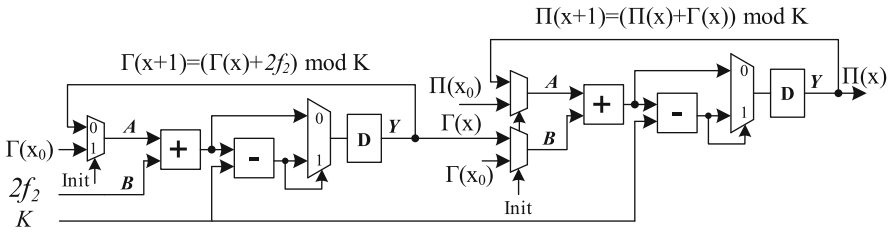


Fig. 17 An circuit implementation for the QPP interleaver $\pi(x) = (f_2x^2 + f_1x) \bmod K$ [67]

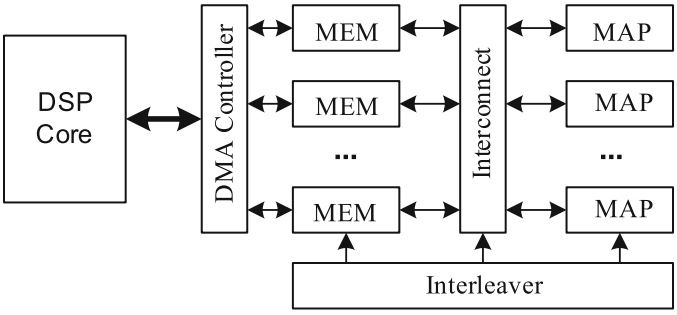


Fig. 18 Turbo decoder accelerator architecture. Multiple MAP decoders are used to support high throughput decoding of turbo codes. Special function units such as interleavers are also implemented in hardware

turbo accelerator can be configured to operate without DSP intervention. To support this feature, some special hardware such as interleavers have to be configurable via DSP control registers. To decrease the required bus bandwidth, intermediate results should not be passed back to the DSP processor. Only the successfully decoded bits need to be passed back to the DSP processor, e.g. via the DSP DMA controller. Further, to support multiple turbo codes in different communication systems, a flexible MAP decoder is necessary. In fact, many standards employ similar turbo code structures. For instance, CDMA, WCDMA, UMTS, and 3GPP LTE all use an eight-state binary turbo code with polynomial (13, 15, 17). Although IEEE 802.16e WiMax and DVB-RCS standards use a different eight-state double binary turbo code, the trellis structures of these turbo codes are very similar as illustrated in Fig. 19. Thus, it is possible design multi-standard turbo decoders based on flexible MAP decoder datapaths [43, 57, 67]. It has been shown in [67] that the area overhead to support multi-codes is only about 7%. In addition, when the throughput requirement is high, e.g. more than 20 Mbps, multiple MAP decoders can be activated to increase the throughput performance.

In summary, due to the iterative structures, a turbo decoder needs more Gflops than what is available in a general-purpose DSP processor. For this reason, Texas Instruments' latest C66x DSP processor integrates a 282 Mbps LTE turbo decoder accelerator in the same die [73]. Because of the parallel and recursive algorithms and special logarithmic arithmetics, it is more cost effective to realize a turbo decoder in hardware.

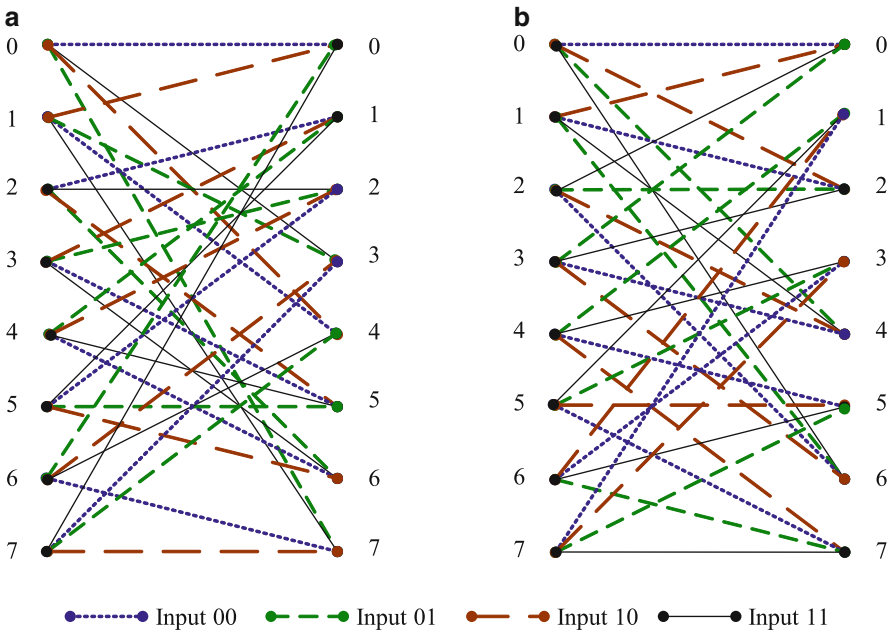


Fig. 19 Radix-4 trellis structures of (**a**) CDMA/WCDMA/UMTS/LTE turbo codes and (**b**) WiMax/DVB-RCS turbo codes

2.3.2 LDPC Decoder Accelerator Architecture

A low-density parity-check (LDPC) code [21] is another important error correcting code that is the among one of the most efficient coding schemes. The remarkable error correction capabilities of LDPC codes have led to their adoption in many standards, such as IEEE 802.11ac, IEEE 802.16e, and IEEE 802 10GBase-T. The huge computation and high throughput requirements make it very difficult to implement a high throughput LDPC decoder on a general-purpose DSP. For example, a 5.4 Mbps LDPC decoder was implemented on TMS320C64xx DSP running at 600 MHz [36]. This throughput performance is not enough to support high data rates defined in new wireless standards. Thus, it is important to develop area and power efficient hardware LDPC decoding accelerators.

A binary LDPC code is a linear block code specified by a very sparse binary $M \times N$ parity check matrix: $\mathbf{H} \cdot \mathbf{x}^T = 0$, where $\mathbf{x}$ is a codeword and $\mathbf{H}$ can be viewed as a bipartite graph where each column and row in $\mathbf{H}$ represent a variable node and a check node, respectively.

The decoding algorithm is based on the iterative message passing algorithm (also called belief propagation algorithm), which exchanges the messages between the variable nodes and check nodes on graph. The hardware implementation of LDPC decoders can be serial, semi-parallel, and fully-parallel as shown in Fig. 20. Fully-parallel implementation has the maximum processing elements to achieve very high

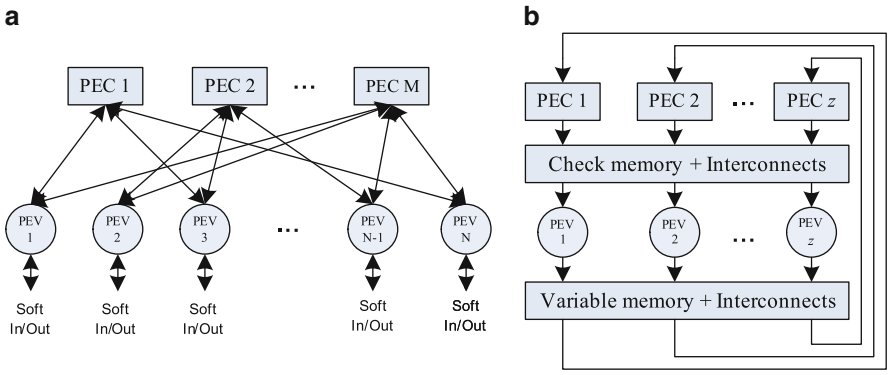


Fig. 20 Implementation of LDPC decoders, where PEC denotes processing element for check node and PEV denotes processing element for variable node: (**a**) fully-parallel and (**b**) semi-parallel

throughput. Semi-parallel implementation, on the other hand, has a lesser number of processing elements that can be re-used, e.g. z number of processing elements are employed in Fig. 20b. In a semi-parallel implementation, memories are usually required to store the temporary results. In many practical systems, semi-parallel implementations are often used to achieve 100 Mbps to 1 Gbps throughput with reasonable complexity [9, 26, 32, 54, 60–63, 65, 66, 87].

In LDPC decoding, the main complexity comes from the check node processing. Each check node receives a set of variable node messages denoted as $\mathcal{N}_m$. Based on these data, check node messages are computed as

$$\Lambda_{mn} = \sum_{j \in \mathcal{N}_m \setminus n} \boxplus \lambda_{mj} = \Big(\sum_{j \in \mathcal{N}_m} \boxplus \lambda_{mj} \Big) \boxminus \lambda_{mn},$$

where Λ_{mn} and λ_{mn} denote the check node message and the variable node message, respectively. The special arithmetic operators $\boxplus$ and $\boxminus$ are defined as follows:

$$\begin{aligned} a \boxplus b &\triangleq f(a,b) = \log \frac{1 + e^a e^b}{e^a + e^b} \\ &= \operatorname{sign}(a) \operatorname{sign}(b) \Big(\min(|a|, |b|) + \log(1 + e^{-(|a|+|b|)}) - \log(1 + e^{-\left||a|-|b|\right|}) \Big), \\ a \boxminus b &\triangleq g(a,b) = \log \frac{1 - e^a e^b}{e^a - e^b} \\ &= \operatorname{sign}(a) \operatorname{sign}(b) \Big(\min(|a|, |b|) + \log(1 - e^{-(|a|+|b|)}) - \log(1 - e^{-\left||a|-|b|\right|}) \Big). \end{aligned}$$

Figure 21 shows a hardware implementation from [61] to compute check node message Λ_{mn} for one check row m. Because multiple check rows can be processed

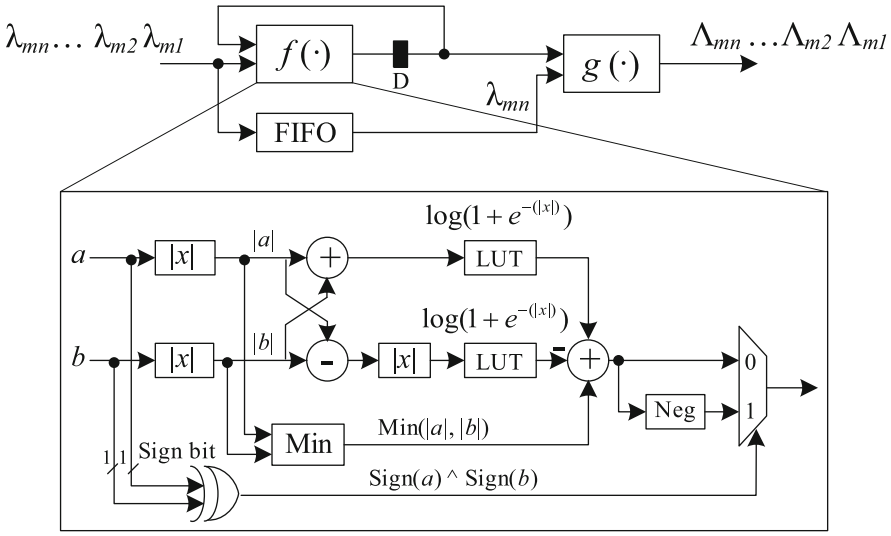


Fig. 21 Recursive architecture to compute check node messages [61]

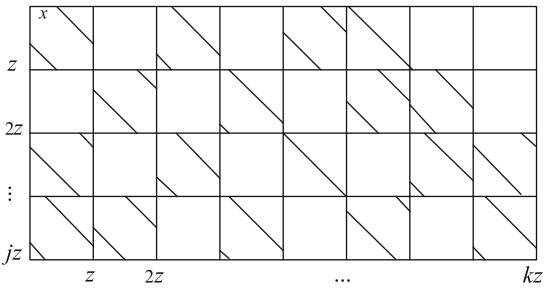


Fig. 22 Structured LDPC parity check matrix with j block rows and k block columns. Each sub-matrix is a $z \times z$ identity shifted matrix

simultaneously in the LDPC decoding algorithm, multiple such check node units can be used to increase decoding speed. As the number of ALU units in a general-purpose DSP processor is limited, it is difficult to achieve more than 10 Mbps throughput in a DSP implementation.

Given a random LDPC code, the main complexity comes not only from the complex check node processing, but also from the interconnection network between check nodes and variable nodes. To simplify the routing of the interconnection network, many practical standards usually employ structured LDPC codes, or quasi-cyclic LDPC (QC-LDPC) codes. The parity check matrix of a QC-LDPC code is shown in Fig. 22. Table 2 summaries the design parameters of the QC-LDPC codes for IEEE 802.11n WLAN and IEEE 802.16e WiMax wireless standards. As can be seen, many design parameters are in the same range for these two applications, thus it is possible to design a reconfigurable hardware to support multiple standards [61].

As an example, a multi-standard semi-parallel LDPC decoder accelerator architecture is shown in Fig. 23 [61]. In order to support several hundreds Mbps data rate,

Table 2 Design parameters for **H** in standardized LDPC codes

	z	j	k	Check node degree	Variable node degree	Max. throughput
WLAN 802.11n	27–81	4–12	24	7–22	2–12	600 Mbps
WiMax 802.16e	24–96	4–12	24	6–20	2–6	144 Mbps

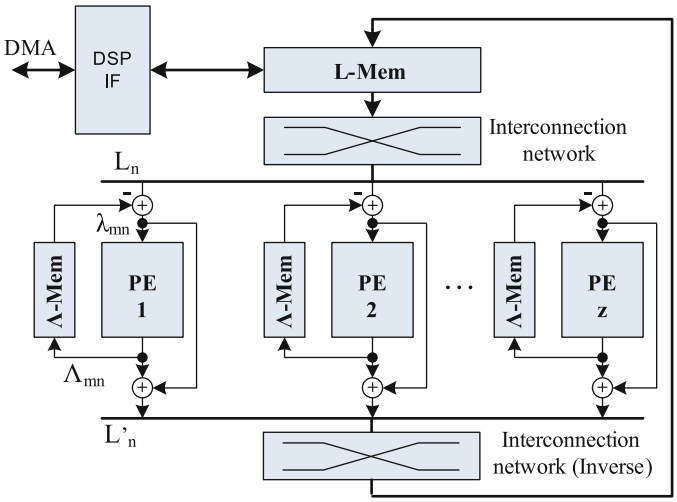


Fig. 23 Semi-parallel LDPC decoder accelerator architecture. Multiple PEs (number of z) are used to increase decoding speed. Variable messages are stored in L-memory and check messages are stored in Λ-memory. An interconnection network along with an inverse interconnection network are used to route data

multiple PEs are used to process multiple check rows simultaneously. As with turbo decoding, LDPC decoding is also based on an iterative decoding algorithm. The iterative decoding flow is as follows: at each iteration, $1 \times z$ APP messages, denoted as L_n are fetched from the L-memory and passed through a permuter (e.g. barrel shifter) to be routed to z PEs (z is the parallelism level). The soft input information λ_{mn} is formed by subtracting the old extrinsic message Λ_{mn} from the APP message L_n. Then the PEs generate new extrinsic messages Λ_{mn} and APP messages L_n, and store them back to memory. The operation mode of the LDPC accelerator needs to be configured in the beginning of the decoding. After that, it should work without DSP intervention. Once it has finished decoding, the decoded bits are passed back to the DSP processor. Figure 24 shows the ASIC implementation result of this decoder (VLSI layout view) and its power consumption for different block sizes. As the block size increases, the number of active PEs increases, thus more power is consumed.

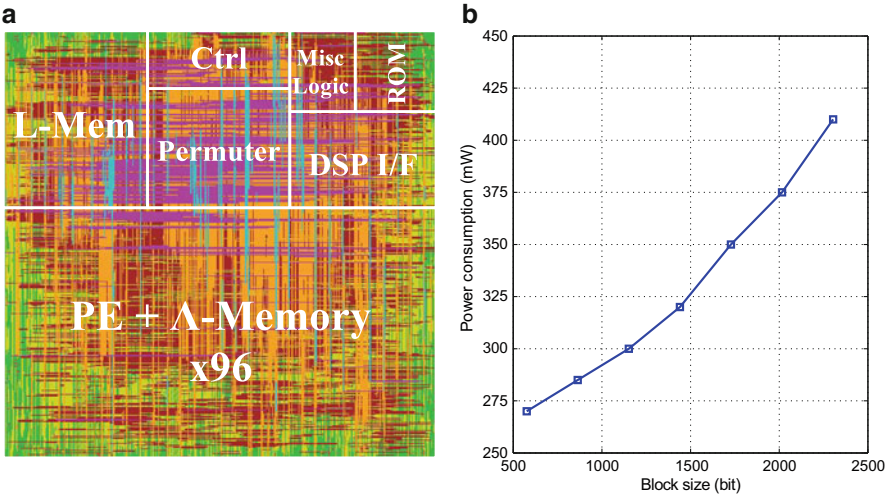


Fig. 24 An example of a LDPC decoder hardware accelerator [61]. (**a**) VLSI layout view (3.5 mm^2 area, 90 nm technology). (**b**) Power consumptions for different block sizes

2.4 Digital Predistortion

In communications, the power amplifier (PA) is a critical component in the radio frontend that gives a signal enough power to have sufficient range. Unfortunately, it is inherently a nonlinear device, and moreover, there is an inverse relationship between its power efficiency and its nonlinearity [24]. It is desirable in many applications to have a power efficient PA, especially considering that the PA consumes most of the power in an RF system [33]. However, the nonlinearities are an undesirable tradeoff. They cause spectral regrowth around the main carrier, intermodulation distortions (IMDs), and other in-band distortions which negatively impacts BERs. An example of the spectral regrowth and IMDs are shown in Fig. 25. Here, an uplink LTE-Advanced signal through a nonlinear PA model with memory effects.

Most PAs are more nonlinear at high power levels as the device approaches saturation and are more linear at lower power levels. To reduce distortions, it is often necessary to reduce the operating power to be in the linear region of operation. However this maximum power backoff, as it's known in the 3GPP literature, causes a reduction in range and power efficiency [1]. This is especially problematic for modern signals. Multicarrier signals such as orthogonal frequency division multiplexing (OFDM) are valued for their spectral efficiency, but they have high peak-to-average power ratios (PAPR) meaning they have large fluctuations in their power level. To keep the device operation in the linear region when there are these large, rapid changes in power, the user must operate with even more backoff than would be necessary for constant power envelope signals. Hence, PA linearization has received substantial attention in recent years.

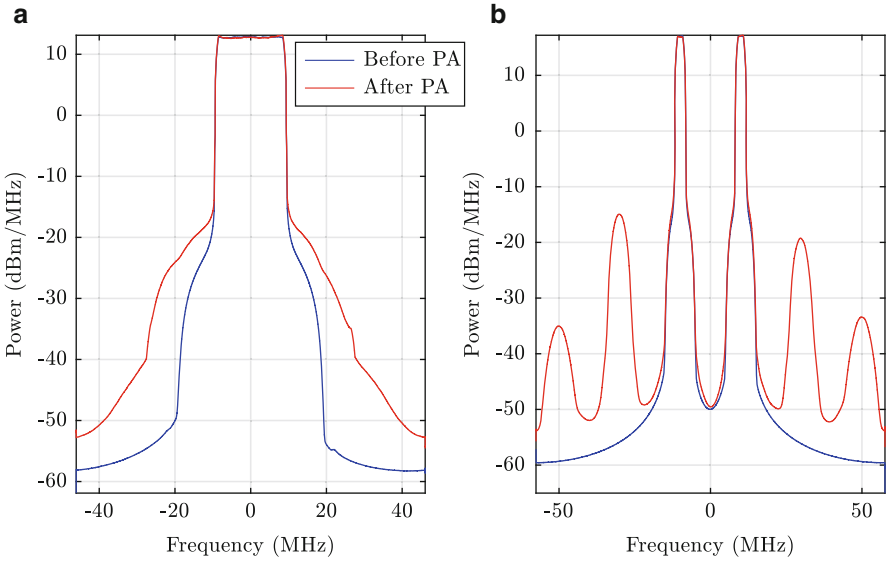


Fig. 25 The effect of a nonlinear PA on an input signal. (**a**) A 20 MHz LTE-Advanced uplink signal is broadcast and there is significant spectral regrowth around the main carrier. (**b**) Two 3 MHz LTE-Advanced uplink signals are broadcast noncontiguously with severe IMD spurious emissions in the nearby spectrum

PA linearization was first considered in the 1920s with analog, feedforward circuitry. Since the '80s, a technique for linearization called predistortion has been dominant. Now, most of these predistorters are digital, and they are used heavily in satellite and telecommunications systems. For example, consider the following commercial solutions. Xilinx has a DPD intellectual property (IP) block that can linearize up to a 100 MHz bandwidth on their FPGAs [85]. Alternatively, the TI GC5322 is a dedicated IC for performing linearization. DPD coefficients are computed on a DSP. The IC can take an input with up to a 40 MHz signal bandwidth, and it linearizes up to a 140 MHz bandwidth [76].

However, many of these available solutions are becoming inadequate for 4G and 5G technologies. As spectrum becomes more scarce and data rates increase, communications standards are necessarily becoming more frequency agile. LTE-Advanced achieves this through a technology called carrier aggregation (CA) where multiple component carriers (CCs) are used simultaneously to achieve a larger, virtual bandwidth. These CCs may be adjacent or noncontiguous in the same LTE band or may be placed in different LTE bands. The largest CC bandwidth in the standard is 20 MHz. With CA, up to five of these can be used simultaneously on the downlink to achieve a virtual carrier bandwidth of 100 MHz [77]. Modern LTE modems have quickly adopted the technology. For example, the Snapdragon 835 supports four downlink and two uplink carriers [52].

As these bandwidths increase, the necessary feedback sampling rates and the DPD complexity rate dramatically grow. Moreover, as the bandwidths increase, the number of DPD parameters needed to be estimated and applied also grows as memory effects of the PA become more pronounced [33]. Hence, there is a need for novel algorithms and implementations in this area. The data-parallelism in the predistortion algorithms makes it a good candidate for acceleration on the various technologies previously discussed. Recently, there has also been interest in implementing DPD on the mobile devices. Computational complexity has been a concern that has limited DPDs adaption in this area, but recent developments in mobile processioning power have led to new implementations targeted for the mobile users. In the following sections, we examine a GPU and FPGA implementation targeted to this.

2.4.1 Full-Band DPD Mobile GPU Accelerator Architecture

There has been a substantial increase in the available computing power on mobile devices over the last 10 years. Modern system-on-chips (SoCs) often integrate multicore CPUs, GPUs, and DSPs to make a tightly integrated, heterogeneous, compute system. Multicore CPUs and GPUs have toolchains and languages that have rapidly matured such as OpenMP, CUDA, and OpenCL which lead to a rapid implementation of a powerful design with throughputs that rival FPGAs and ASICs.

In [38], the first CUDA-based GPU implementation of DPD was done. The work was improved upon in [39]. In these works, the implementation is tested on a Jetson embedded development board with a mobile GPU. This is connected to the wireless open access research platform (WARP) v3 software-defined radio (SDR) board [42].

It is the goal of the predistorter to distort the input signal with the inverse of the distortion that the PA will introduce. The modeling of a PA and its corresponding predistorter can be done to various degrees of precision. Often as a more complete and precise model is used the complexity of the predistortion increases. Using the most general form of modeling, a Volterra series could be used. This model includes memory effects for each nonlinearities in a way that each memory tap could have a different nonlinear model. Hence, there are many parameters in this model. A simplification that is commonly used is to separate the memory effects and nonlinearities. One such model is an augmented parallel Hammerstein (APH) structure. This is shown in Fig. 26a. The input samples pass through a nonlinear function ψ and then a memory system realized as an FIR filter, H. The branches of the parallel structure are combined to form the predistorter output.

This particular implementation also includes correction for other imperfections in the TX RF hardware including I/Q mismatch compensation and local oscillator (LO) leakage correction. This is realized by the "conjugate branch" with $\bar{\psi}$ and $\bar{H}$ in the APH structure and the addition of a constant, c, respectively. The learning is performed offline using the widely adopted indirect learning architecture shown in Fig. 26b.

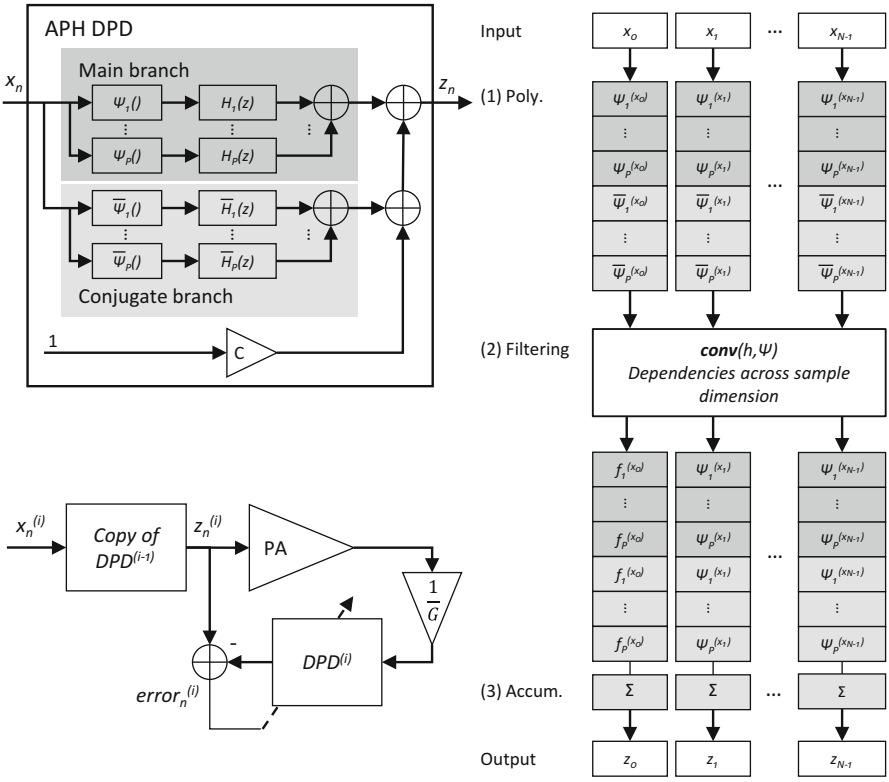


Fig. 26 (**a**) APH DPD structure. (**b**) Indirect learning architecture. (**c**) Data flow and parallelism [39]

The GPU implementation performs instructions in parallel based on a single instruction multiple threads (SIMT) paradigm. Three kernels are run with a large number of parallel threads. The three kernels are polynomial computation, filtering computation, and accumulation shown in Fig. 26c. The authors are able to support throughputs of 221.8 Msamples per second on a Maxwell GPU with over 10 dB of IMD suppression.

2.4.2 Sub-band FPGA Accelerator Architecture

In [2], the authors focus on the case of DPD for mobile users with non-contiguous transmissions. With non-contiguous transmissions, such as intra-band non-contiguous CA in LTE-A, the necessary sampling rate to linearize the spurious emissions created from the IMD rapidly grows as the CC spacing grows. DPD quickly becomes more costly as a fast ADC is required and a corresponding fast throughput is maintained in the DPD computations. Instead, a sub-band technique

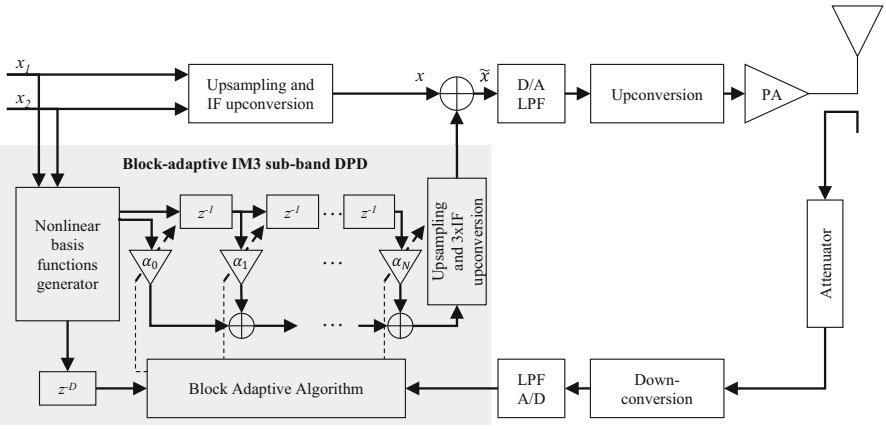


Fig. 27 Block-adaptive decorrelation-based sub-band DPD system architecture for third-order spurious intermodulation reduction in a noncontiguous transmitter [2]

can be used where one linearizes individual spurious emissions that are in violation of the spurious emission masks. The idea is to inject a spur before the PA with the opposite phase of the natural IMD spur so that they cancel out. By targeting individual spurs which in many scenarios are the limiting factor for transmitter emission violations, the complexity is significantly reduced when compared to other full-band methods.

A block diagram of the sub-band DPD system architecture is shown in Fig. 27 where two CCs, x_1 and x_2 are used. A block-adaptive least-mean-squares decorrelation learning algorithm is used for coefficient training. The observed time-domain signal of a spur, $e(n)$ is correlated with the expected third-order signal at the spur, $u(n)$, which is predicted based off the PA modeling such as in Eq. (17). At each iteration, the DPD coefficient α moves in the opposite direction of the correlation so that when the DPD injection signal is combined with the main signal, $x(n)$, and goes through the PA, the output $y(n)$ sees a reduction in the spur. The algorithm is iterated until the error signal is completely decorrelated with basis function.

An important concern in a design with feedback is the loop delay of the system. For example, when a change to the DPD coefficient is made, there will be some time before the DPD learning algorithm actually observes the change since it must propagate through the system. If this was not accounted for then the device would be learning on stale data for a short time which could lead to oscillations, overshoot, or other instabilities in the DPD coefficient convergence. This can be remedied by using a block-adaptive technique where learning is done on a block of many samples, then learning pauses for another block so that the updates have time to propagate.

The analysis is shown below for a noncontiguous signal being broadcast through a third-order, parallel Hammerstein (PH) PA model at the baseband equivalent level. The two CCs, $x_1(n)$ and $x_2(n)$, are assumed to be separated by Δf. The PA input and output signals, $x(n)$ and $y(n)$, read

$$x(n) = x_1(n)e^{j2\pi\frac{\Delta f}{2f_s}n} + x_2(n)e^{-j2\pi\frac{\Delta f}{2f_s}n} \tag{15}$$

$$y(n) = f_{1,n} \star x(n) + f_{3,n} \star |x(n)|^2 x(n), \tag{16}$$

where $f_{1,n}$ and $f_{3,n}$ are the filters in the main and third order PH branches, respectively, which model the memory effects and $\star$ is the convolution operator. Through substitution of Eq. (15) into Eq. (16), output spurious emissions can be recovered. For example, the positive IM3 term is found to be

$$y_{IM3_+}(n) = f_{3,n}^{3+} \star (x_2^*(n)x_1^2(n)). \tag{17}$$

Here, $f_{3,n}^{3+}$ is the baseband equivalent response of $f_{3,n}$ at the positive $IM3$ sub-band around $(f_c + \Delta f/2)$, where f_c denotes the carrier frequency. Stemming from the signal structure in Eq. (17), a natural injection signal is a filtered version of the basis function $x_2^*(n)x_1^2(n)$ using a filter α_n with memory depth N. Incorporating such DPD processing, the composite baseband equivalent PA input $\tilde{x}(n)$ signal reads

$$\tilde{x}(n) = x(n) + \left[\alpha_n^* \star (x_2^*(n)x_1^2(n))\right] e^{j2\pi\frac{3\Delta f}{2f_s}n}. \tag{18}$$

Substituting now $\tilde{x}(n)$ in (16), the positive IM3 sub-band signal at the PA output becomes

$$\begin{aligned}\tilde{y}_{IM3_+}(n) \approx\ & (f_{3,n}^{3+} + f_{1,n}^{3+} \star \alpha_n^*) \star x_2^*(n)x_1^2(n) \\ & + 2f_{3,n}^{3+} \star \left[(|x_1(n)|^2 + |x_2(n)|^2)(\alpha_n^* \star x_2^*(n)x_1^2(n))\right],\end{aligned} \tag{19}$$

Based on the DPD architecture in Fig. 27 and the block-based learning while assuming an estimation block size of M samples and $N+1$ DPD filter coefficients, the DPD parameter learning algorithm becomes

$$\boldsymbol{\alpha}(n+1) = \boldsymbol{\alpha}(n) - \frac{\mu}{||\mathbf{U}(n)||^2 + C}\left[\mathbf{e}^H(n)\mathbf{U}(m)\right]^T, \tag{20}$$

where

$$e(n) = \tilde{y}_{IM3_+}(n) \tag{21}$$

$$\mathbf{e}(m) = [e(n_m)\ \ e(n_m+1)\ \ \ldots\ \ e(n_m+M-1)]^T \tag{22}$$

$$u(n) = x_2^*(n)x_1^2(n) \tag{23}$$

$$\mathbf{u}(n_m) = [u(n_m)\ \ u(n_m+1)\ \ \ldots\ \ u(n_m+M-1)]^T \tag{24}$$

$$\mathbf{U}(m) = [\mathbf{u}(n_m)\ \ \mathbf{u}(n_m-1)\ \ \ldots\ \ \mathbf{u}(n_m-N)] \tag{25}$$

$$\boldsymbol{\alpha}(m) = [\alpha_0(m)\ \ \alpha_1(m)\ \ \ldots\ \ \alpha_N(m)]^T. \tag{26}$$

The running complexity for linearizing a single sub-band with this method consists generating a basis function and, in the case of a third-order memoryless DPD, multiplying it by a DPD coefficient α. This consists of a total of 3 complex multiplications which can be implemented with a total of 18 operations per sample. The minimum sampling rate to linearize a third order term needs to be three times the bandwidth of the widest CC. For example, with a 5 MHz LTE-A signal, a 15 MHz sample rate can be used for a DPD application complexity of 0.27 GFLOPS.

A dedicated DPD accelerator can be used for both the learning and application of the DPD so that it can be done in real-time. The authors of [2] do this on the Virtex 6 FPGA of a WARPv3 SDR board for real-time DPD learning and suppression. The generation of the basis function, the multiplication of the coefficient α, and the addition of this to the original signal $x(n)$ shown in Eq. (18) is all done in a streaming, pipelined manner so that there only an overhead of an additional 13 clock cycles in the baseband PHY design. For the learning, the authors input the signal from the observed output of the PA through the analog-to-digital converter. The spurious emission is isolated by passing the signal through a low-pass, FIR filter. From here, the LMS learning step is implemented similarly to Eq. (20) in a fully pipelined manner so that learning is done quickly in a parallel, streaming architecture.

3 Summary

Digital signal processing complexity in high-speed wireless communications is driving a need for high performance heterogeneous DSP systems with real-time processing. Many wireless algorithms, such as channel decoding and MIMO detection, demonstrate significant data parallelism. For this class of data-parallel algorithms, application specific DSP accelerators are necessary to meet real-time requirements while minimizing power consumption. Spatial locality of data, data level parallelism, computational complexity, and task level parallelism are four major criteria to identify which DSP algorithm should be off-loaded to an accelerator. Additional cost incurred from the data movement between DSP and hardware accelerator must be also considered.

There are a number of DSP architectures which include true hardware based accelerators. Examples of these include the Texas Instruments' CI66x series of DSPs which include a 365 Mbps turbo decoding accelerator [73], and NXP Semiconductor's six core broadband wireless access DSP MSC8156 which includes a programmable 200 Mbps turbo decoding accelerator (6 iterations), a 115 Mbps Viterbi decoding accelerator (K = 9), an FFT/IFFT accelerator for sizes 128, 256, 512, 1024 or 2048 points at up to 350 million samples/s, and a DFT/IDFT for sizes up to 1536 points at up to 175 million samples/s [20].

Relying on a single DSP processor for all signal processing tasks would be a clean solution. As a practical matter, however, multiple DSP processors are necessary for implementing a next generation wireless handset or base station.

This means greater system cost, more board space, and more power consumption. Integrating hardware communication accelerators, such as MIMO detectors and channel decoders, into the DSP processor silicon can create an efficient System-on-Chip. This offers many advantages: the dedicated accelerators relieve the DSP processor of the parallel computation-intensive signal processing burden, freeing DSP processing capacity for other system control functions that more greatly benefit from programmability.

4 Further Reading

This chapter serves as a brief introduction to the application-specific accelerators for communications. For more detailed discussion on the VLSI signal processing system design and implementation, readers are encouraged to read the following book [50]. For more information on the software/hardware co-design as well as the hardware accelerators for 3G/4G wireless systems, one can read the following dissertations [10, 60]. Finally, major DSP processor vendors such as Texas Instruments, Analog Devices, and NXP Semiconductors provide many application notes about their DSP hardware accelerators [5, 48, 75].

Readers are also advised to look at several other chapters of this handbook. For example, [28] discusses the fundamental computer arithmetic, [51] talks about the general-purpose DSP processors, and [34] introduces the VLIW DSP processors. Wireless transceiver signal processing is also discussed in [53]. When making accelerators, usually we need to utilize a fixed word-length and fixed-point arithmetic. This is discussed in [28, 68], and [46].

References

1. LTE; Evolved Universal Terrestrial Radio Access (E-UTRA) User Equipment (UE) radio transmission and reception, 3GPP TS 36.101 V13.2.1 (Release 13) (May 2016)
2. Abdelaziz, M., Tarver, C., Li, K., Anttila, L., Martinez, R., Valkama, M., Cavallaro, J.R.: Sub-Band Digital Predistortion for Noncontiguous Transmissions: Algorithm Development and Real-Time Prototype Implementation. In: 2015 49th Asilomar Conference on Signals, Systems and Computers, pp. 1180–1186 (2015). https://doi.org/10.1109/ACSSC.2015.7421326
3. Alamouti, S.M.: A Simple Transmit Diversity Technique for Wireless Communications. IEEE Journal on Selected Areas in Communications **16**(8), 1451–1458 (1998)
4. Amiri, K., Cavallaro, J.R.: FPGA Implementation of Dynamic Threshold Sphere Detection for MIMO Systems. In: IEEE Asilomar Conf. on Signals, Syst. and Computers, pp. 94–98 (2006)
5. Analog Devices: The SHARC Processor Family. http://www.analog.com/en/products/processors-dsp/sharc.html (2016)
6. Andrews, J.G., Buzzi, S., Choi, W., Hanly, S.V., Lozano, A., Soong, A.C.K., Zhang, J.C.: What Will 5G Be? IEEE Journal on Selected Areas in Communications **32**(6), 1065–1082 (2014). https://doi.org/10.1109/JSAC.2014.2328098
7. Bahl, L., Cocke, J., Jelinek, F., Raviv, J.: Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate. IEEE Transactions on Information Theory **IT-20**, 284–287 (1974)

8. Berrou, C., Glavieux, A., Thitimajshima, P.: Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes. In: IEEE Int. Conf. on Commun., pp. 1064–1070 (1993)
9. Brack, T., Alles, M., Lehnigk-Emden, T., Kienle, F., Wehn, N., Lapos, Insalata, N., Rossi, F., Rovini, M., Fanucci, L.: Low Complexity LDPC Code Decoders for Next Generation Standards. In: Design, Automation, and Test in Europe, pp. 1–6 (2007)
10. Brogioli, M.: Reconfigurable Heterogeneous DSP/FPGA Based Embedded Architectures for Numerically Intensive Embedded Computing Workloads. Ph.D. thesis, Rice University, Houston, Texas, USA (2007)
11. Brogioli, M., Radosavljevic, P., Cavallaro, J.: A General Hardware/Software Codesign Methodology for Embedded Signal Processing and Multimedia Workloads. In: IEEE Asilomar Conf. on Signals, Syst., and Computers, pp. 1486–1490 (2006)
12. Burg, A.: VLSI Circuits for MIMO Communication Systems. Ph.D. thesis, Swiss Federal Institute Of Technology, Zurich, Switzerland (2006)
13. Burg, A., Borgmann, M., Wenk, M., Zellweger, M., Fichtner, W., Bolcskei, H.: VLSI Implementation of MIMO Detection using the Sphere Decoding Algorithm. IEEE Journal of Solid-State Circuits **40**(7), 1566–1577 (2005)
14. Cadence Design Systems: https://ip.cadence.com/ipportfolio/tensilica-ip (2017)
15. Cheng, C.C., Tsai, Y.M., Chen, L.G., Chandrakasan, A.: A 0.077 to 0.168 nJ/bit/iteration Scalable 3GPP LTE Turbo Decoder with an Adaptive Sub-Block Parallel Scheme and an Embedded DVFS Engine. In: IEEE Custom Integrated Circuits Conference, pp. 1–4 (2010)
16. Cupaiuolo, T., Siti, M., Tomasoni, A.: Low-Complexity High Throughput VLSI Architecture of Soft-Output ML MIMO Detector. In: Design, Automation and Test in Europe Conference and Exhibition, pp. 1396–1401 (2010)
17. Damen, M.O., Gamal, H.E., Caire, G.: On Maximum Likelihood Detection and the Search for the Closest Lattice Point. IEEE Transaction on Information Theory **49**(10), 2389–2402 (2003)
18. Fincke, U., Pohst, M.: Improved Methods for Calculating Vectors of Short Length in a Lattice, Including a Complexity Analysis. Mathematics of Computation **44**(170), 463–471 (1985)
19. Foschini, G.: Layered Space-Time Architecture for Wireless Communication in a Fading Environment when Using Multiple Antennas. Bell Labs. Tech. Journal **2**, 41–59 (1996)
20. Freescale Semiconductor: MSC8156 Six Core Broadband Wireless Access DSP. www.freescale.com/starcore (2009)
21. Gallager, R.: Low-Density Parity-Check Codes. IEEE Transactions on Information Theory **IT-8**, 21–28 (1962)
22. Garrett, D., Davis, L., ten Brink, S., Hochwald, B., Knagge, G.: Silicon Complexity for Maximum Likelihood MIMO Detection Using Spherical Decoding. IEEE Journal of Solid-State Circuits **39**(9), 1544–1552 (2004)
23. Garrido, M., Qureshi, F., Takala, J., Gustafsson, O.: Hardware architectures for the fast Fourier transform. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
24. Ghannouchi, F.M., Hammi, O.: Behavioral Modeling and Predistortion. IEEE Microwave Magazine **10**(7), 52–64 (2009). https://doi.org/10.1109/MMM.2009.934516
25. Golden, G., Foschini, G.J., Valenzuela, R.A., Wolniansky, P.W.: Detection Algorithms and Initial Laboratory Results Using V-BLAST Space-Time Communication Architecture. Electronics Letters **35**(1), 14–15 (1999)
26. Gunnam, K., Choi, G.S., Yeary, M.B., Atiquzzaman, M.: VLSI Architectures for Layered decoding for Irregular LDPC Codes of WiMax. In: IEEE International Conference on Communications, pp. 4542–4547 (2007)
27. Guo, Z., Nilsson, P.: Algorithm and Implementation of the K-best Sphere Decoding for MIMO Detection. IEEE Journal on Seleteced Areas in Communications **24**(3), 491–503 (2006)
28. Gustafsson, O., Wanhammar, L.: Arithmetic. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
29. Han, S., Tellambura, C.: A Complexity-Efficient Sphere Decoder for MIMO Systems. In: IEEE International Conference on Communications, pp. 1–5 (2011)

30. Hassibi, B., Vikalo, H.: On the Sphere-Decoding Algorithm I. Expected Complexity. IEEE Transaction On Signal Processing **53**(8), 2806–2818 (2005)
31. Hunter, H.C., Moreno, J.H.: A New Look at Exploiting Data Parallelism in Embedded Systems. In: Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, pp. 159–169 (2003)
32. Jin, J., Tsui, C.: Low-Complexity Switch Network for Reconfigurable LDPC Decoders. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **18**(8), 1185–1195 (2010)
33. Katz, A., Wood, J., Chokola, D.: The Evolution of PA Linearization: From Classic Feedforward and Feedback Through Analog and Digital Predistortion. IEEE Microwave Magazine **17**(2), 32–40 (2016). https://doi.org/10.1109/MMM.2015.2498079
34. Kessler, C.W.: Compiling for VLIW DSPs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
35. Larsson, E.G., Edfors, O., Tufvesson, F., Marzetta, T.L.: Massive MIMO for Next Generation Wireless Systems. IEEE Communications Magazine **52**(2), 186–195 (2014). https://doi.org/10.1109/MCOM.2014.6736761
36. Lechner, G., Sayir, J., Rupp, M.: Efficient DSP Implementation of an LDPC Decoder. In: IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, vol. 4, pp. 665–668 (2004)
37. Lee, S.J., Shanbhag, N.R., Singer, A.C.: Area-Efficient High-Throughput MAP Decoder Architectures. IEEE Transaction on VLSI Systems **13**(8), 921–933 (2005)
38. Li, K., Ghazi, A., Boutellier, J., Abdelaziz, M., Anttila, L., Juntti, M., Valkama, M., Cavallaro, J.R.: Mobile GPU Accelerated Digital Predistortion on a Software-Defined Mobile Transmitter. In: 2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP), pp. 756–760 (2015). https://doi.org/10.1109/GlobalSIP.2015.7418298
39. Li, K., Ghazi, A., Tarver, C., Boutellier, J., Abdelaziz, M., Anttila, L., Juntti, M.J., Valkama, M., Cavallaro, J.R.: Parallel Digital Predistortion Design on Mobile GPU and Embedded Multicore CPU for Mobile Transmitters. CoRR **abs/1612.09001** (2016). URL http://arxiv.org/abs/1612.09001
40. Li, K., Yin, B., Wu, M., Cavallaro, J.R., Studer, C.: Accelerating Massive MIMO Uplink Detection on GPU for SDR Systems. In: 2015 IEEE Dallas Circuits and Systems Conference (DCAS), pp. 1–4 (2015). https://doi.org/10.1109/DCAS.2015.7356600
41. Lin, C.H., Chen, C.Y., Wu, A.Y.: Area-Efficient Scalable MAP Processor Design for High-Throughput Multistandard Convolutional Turbo Decoding. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **19**(2), 305–318 (2011)
42. Mango: WARP Project. URL http://warpproject.org
43. Martina, M., Nicola, M., Masera, G.: A Flexible UMTS-WiMax Turbo Decoder Architecture. IEEE Transactions on Circuits and Systems II **55**(4), 369–273 (2008)
44. May, M., Ilnseher, T., Wehn, N., Raab, W.: A 150Mbit/s 3GPP LTE Turbo Code Decoder. In: IEEE Design, Automation & Test in Europe Conference & Exhibition, pp. 1420–1425 (2010)
45. McAllister, J.: High performance stream processing on FPGA. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
46. Menard, D., Caffarena, G., Lopez, J.A., Novo, D., Sentieys, O.: Analysis of finite word-length effects in fixed-point systems. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
47. Myllylä, M., Silvola, P., Juntti, M., Cavallaro, J.R.: Comparison of Two Novel List Sphere Detector Algorithms for MIMO-OFDM Systems. In: IEEE International Symposium on Personal Indoor and Mobile Radio Communications, pp. 1–5 (2006)
48. NXP Semiconductor: StarCore SC3900FP. http://www.nxp.com/assets/documents/data/en/brochures/BRSC3900DSPCORE.pdf (2013)
49. NXP Semiconductor: QorIQ Layerscape: A Converged Architecture Approach (2017)
50. Parhi, K.K.: VLSI Digital Signal Processing Systems Design and Implementation. Wiley (1999)
51. Pelcat, M.: Models of architecture for DSP systems. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)

52. Qualcomm: Snapdragon 835 Mobile Platform. online: https://www.qualcomm.com/products/snapdragon/processors/835 (2017)
53. Renfors, M., Juntti, M., Valkama, M.: Signal processing for wireless transceivers. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
54. Rovini, M., Gentile, G., Rossi, F., Fanucci, L.: A Scalable Decoder Architecture for IEEE 802.11n LDPC Codes. In: IEEE Global Telecommunications Conference, pp. 3270–3274 (2007)
55. Sadjadpour, H., Sloane, N., Salehi, M., Nebe, G.: Interleaver Design for Turbo Codes. IEEE Journal on Seleteced Areas in Communications **19**(5), 831–837 (2001)
56. Salmela, P., Gu, R., Bhattacharyya, S., Takala, J.: Efficient Parallel Memory Organization for Turbo Decoders. In: Proc. European Signal Processing Conf., pp. 831–835 (2007)
57. Shin, M.C., Park, I.C.: A Programmable Turbo Decoder for Multiple 3G Wireless Standards. In: IEEE Solid-State Circuits Conference, vol. 1, pp. 154–484 (2003)
58. Studer, C., Benkeser, C., Belfanti, S., Huang, Q.: Design and Implementation of a Parallel Turbo-Decoder ASIC for 3GPP-LTE. IEEE Journal of Solid-State Circuits **46**(1), 8–17 (2011)
59. Sun, J., Takeshita, O.: Interleavers for Turbo Codes Using Permutation Polynomials Over Integer Rings. IEEE Transaction on Information Theory **51**(1), 101–119 (2005)
60. Sun, Y.: Parallel VLSI Architectures for Multi-Gbps MIMO Communication Systems. Ph.D. thesis, Rice University, Houston, Texas, USA (2010)
61. Sun, Y., Cavallaro, J.R.: A Low-power 1-Gbps Reconfigurable LDPC Decoder Design for Multiple 4G Wireless Standards. In: IEEE International SOC Conference, pp. 367–370 (2008)
62. Sun, Y., Cavallaro, J.R.: Scalable and Low Power LDPC Decoder Design Using High Level Algorithmic Synthesis. In: IEEE International SOC Conference (SoCC), pp. 267–270 (2009)
63. Sun, Y., Cavallaro, J.R.: A Flexible LDPC/Turbo Decoder Architecture. Journal of Signal Processing System **64**(1), 1–16 (2011)
64. Sun, Y., Cavallaro, J.R.: Efficient Hardware Implementation of a Highly-Parallel 3GPP LTE, LTE-Advance Turbo Decoder. Integration, the VLSI Journal, Special Issue on Hardware Architectures for Algebra, Cryptology and Number Theory **44**(4), 305–315 (2011)
65. Sun, Y., Karkooti, M., Cavallaro, J.R.: VLSI Decoder Architecture for High Throughput, Variable Block-Size and Multi-Rate LDPC Codes. In: IEEE International Symposium on Circuits and Systems (ISCAS), pp. 2104–2107 (2007)
66. Sun, Y., Wang, G., Cavallaro, J.R.: Multi-Layer Parallel Decoding Algorithm and VLSI Architecture for Quasi-Cyclic LDPC Codes. In: IEEE International Symposium on Circuits and Systems, pp. 1776–1779 (2011)
67. Sun, Y., Zhu, Y., Goel, M., Cavallaro, J.R.: Configurable and Scalable High Throughput Turbo Decoder Architecture for Multiple 4G Wireless Standards. In: IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 209–214 (2008)
68. Sung, W.: Optimization of number representations. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
69. Sutter, B.D., Raghavan, P., Lambrechts, A.: Coarse grained reconfigurable array architectures. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
70. Tarokh, V., Jafarkhani, H., Calderbank, A.R.: Space-Time Block Codes from Orthogonal Designs. IEEE Transactions on Information Theory **45**(5), 1456–1467 (1999)
71. Tarokh, V., Jafarkhani, H., Calderbank, A.R.: Space Time Block Coding for Wireless Communications: Performance Results. IEEE Journal on Selected Areas in Communications **17**(3), 451–460 (1999)
72. Telatar, I.E.: Capacity of Multiantenna Gaussian Channels. European Transaction on Telecommunications **10**, 585–595 (1999)
73. Texas Instruments: TMS320TCI6614 Communications Infrastructure KeyStone SoC Data Manual. http://www.ti.com/lit/ds/symlink/tms320tci6614.pdf (2013)
74. Texas Instruments: Communications Processors Products. http://focus.ti.com/docs/prod/folders/print/tms320c6474.html (2016)

75. Texas Instruments: Digital Signal Processors. https://www.ti.com/lsds/ti/processors/dsp/overview.page (2017)
76. Texas Instruments: Wideband Transmit IC Solution with integrated Digital Predistortion, Digital Upconversion. online: http://www.ti.com/product/GC5322/description (2017)
77. Wannstrom, J.: Carrier Aggregation Explained. online: http://www.3gpp.org/technologies/keywords-acronyms/101-carrier-aggregation-explained (2013)
78. Wijting, C., Ojanpera, T., Juntti, M., Kansanen, K., Prasad, R.: Groupwise Serial Multiuser Detectors for Multirate DS-CDMA. In: IEEE Vehicular Technology Conference, vol. 1, pp. 836–840 (1999)
79. Willmann, P., Kim, H., Rixner, S., Pai, V.S.: An Efficient Programmable 10 Gigabit Ethernet Network Interface Card. In: ACM International Symposium on High-Performance Computer Architecture, pp. 85–86 (2006)
80. Witte, E., Borlenghi, F., Ascheid, G., Leupers, R., Meyr, H.: A Scalable VLSI Architecture for Soft-Input Soft-Output Single Tree-Search Sphere Decoding. IEEE Tran. on Circuits and Systems II: Express Briefs **57**(9), 706–710 (2010)
81. Wong, C.C., Chang, H.C.: Reconfigurable Turbo Decoder with Parallel Architecture for 3GPP LTE System. IEEE Tran. on Circuits and Systems II: Express Briefs **57**(7), 566–570 (2010)
82. Wong, K., Tsui, C., Cheng, R.S., Mow, W.: A VLSI Architecture of a K-best Lattice Decoding Algorithm for MIMO Channels. In: IEEE International Symposium on Circuits and Systems, vol. 3, pp. 273–276 (2002)
83. Wu, M., Sun, Y., Wang, G., Cavallaro, J.R.: Implementation of a High Throughput 3GPP Turbo Decoder on GPU. Journal of Signal Processing Systems **65**(2), 171 (2011). https://doi.org/10.1007/s11265-011-0617-7
84. Wu, M., Wang, G., Yin, B., Studer, C., Cavallaro, J.R.: LTE-A Turbo Decoder on GPU and Multicore CPU. In: 2013 Asilomar Conference on Signals, Systems and Computers, pp. 824–828 (2013). https://doi.org/10.1109/ACSSC.2013.6810402
85. Xilinx: Digital Pre-Distortion. online: https://www.xilinx.com/products/intellectual-property/ef-di-dpd.html (2017)
86. Ye, Z.A., Moshovos, A., Hauck, S., Banerjee, P.: CHIMAERA: A High Performance Architecture with a Tightly Coupled Reconfigurable Functional Unit. In: Proceedings of the 27th Annual International Symposium on Computer Architecture, pp. 225–235 (2000)
87. Zhong, H., Zhang, T.: Block-LDPC: A Practical LDPC Coding System Design Approach. IEEE Transactions on Circuits and Systems I **52**(4), 766–775 (2005)

System-on-Chip Architectures for Data Analytics

Gwo Giun (Chris) Lee, Chun-Fu Chen, and Tai-Ping Wang

Abstract Artificial Intelligence (AI) in Industry 4.0, intelligent transportation system, intelligent biomedical systems and healthcare, etc., plays an important role requiring complex algorithms. Deep learning in machine learning, for example, is a popular AI algorithm with high computational demands on EDGE platforms in Internet-of-Things applications. This chapter introduces the Algorithm/Architecture Co-Design system design methodology for concurrent design of an algorithm with highly efficient, flexible and low power architecture in constituting the Smart System-on-Chip design.

1 Introduction

In the 1960s, Marshall McLuhan published the book entitled, "The Extensions of Man" focusing primarily on television, an electronic media as being the outward extension of human nervous system, which from contemporary interpretation marks the previous stage of Big Data.

In concurrent Industry 4.0 ecosystem, Internet-of-Things (IoT) facilitate extra sensory perception in reaching out even farther via sensors interconnected through signals with information exchange. As such, innovations in intelligent surveillance and monitoring technologies has not only made possible advancements towards smart cities, intelligent transportation systems (ITS) including autonomous cars, intelligent home (iHome), and intelligent biomedical and healthcare systems, generation of even bigger data will inevitably be witnessed. Further inward extension of human information perception could also be experienced when observing genomic, neurological, and other physiological phenomena when going deeper inwards into

G. G. Lee (✉) · T.-P. Wang
Department of Electrical Engineering, National Cheng Kung University, Tainan City, Taiwan
e-mail: clee@mail.ncku.edu.tw

C.-F. Chen
IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_15

the human body, again with tremendously big data such as from the human brain and especially the human genome.

Ubiquitous Artificial Intelligence (AI), brought forth by wearable, mobile and other IoT devices, requires not only more complex algorithms, but also automated analytics algorithm for versatile applications which starting from science and engineering such as multimedia, communication, and biotechnology, will diversify towards other cross disciplinary domains. Machine learning algorithms such as deep learning which in addition to having self-learning capabilities also demand excessively high complexity in processing these big heterogeneous data.

With mathematical fundamentals as foundations for the analysis of corresponding dataflow models from algorithms, intelligent, flexible, and efficient, analytics architectures, including both software and hardware for VLSI, GPU, multicore, high performance computing, and reconfigurable computing systems, etc., this chapter innovates discussions on Smart System-on-Chip design, in expediting the field of signal and information processing systems into futuristic new era of the Internet-of-Things and high performance computing based on Algorithm/Architecture Co-design.

In Algorithm/Architecture Co-Design (AAC), in manners similar to Parhi et al. and Ha et al., [10, 22], algorithms will be modelled using dataflow graphs (DFG) which represent different realizations or implementations of an algorithm also referred to as architecture instantiation. Having information on both algorithmic behavior and architectural information including both software and hardware for implementation, the DFG so proposed provides a mathematical representation which better models the underlying computational platform for systematic analysis thus providing flexible and efficient management of the computational and storage resources. Through Eigen-analysis of the DFG, homogeneity and heterogeneity properties of parallel computing are introduced like the homogeneous systolic array as presented by Hu et al. [12]. By exploring the similarities in the DFG's of different algorithm, reconfigurable architectures at different level of granularities could be explored where Sutter et al. introduced reconfigurability at coarse granularity [28].

In this chapter, we shall also introduce the design methodology for video compression and MPEG reconfigurable video coding which was discussed in more depth by Chen et al. and Mattavelli et al. respectively. Furthermore, AAC is also applicable to the design of DSP processors [29] multi-core SoC [3], etc.

2 Algorithm/Architecture Co-design: Analytic Architecture for SMART SoC

NIKLAUS EMIL WIRTH introduced the innovative idea that *Programming = Algorithm + Data Structure*. Inspired by this, we advance the concept to the next level by stating that *Design = Algorithm + Architecture*. With concurrent exploration of algorithm and architecture entitled Algorithm/Architecture Co-design (AAC), this

methodology innovates a leading paradigm shift in advanced system design from System-on-a-Chip to IoT, and heterogeneous system.

As high performance computing becomes exceedingly demanding and IoT generated data becomes increasingly bigger, flexible parallel/reconfigurable processing are crucial in the design of efficient and flexible signal processing systems with low power consumption. Hence the analysis of algorithms for potential computing in parallel, efficient data storage and data transfer is crucial. In analogous to the analysis of speech and image data in machine learning, this section characterizes the analysis of dataflow models representing algorithms, for analytics architecture, a cross-level-of-abstraction system design methodology for SoC on versatile platforms [18].

2.1 Architectural Platform

Current intelligent algorithms such as those for big data analytics and machine learning are becoming ever more complex. Rapid and continuous enhancements in semiconductor and information communication technologies (ICT) with innovations in especially advanced systems and architectural platforms capable of accommodating these intelligent algorithms targeting versatile applications including ubiquitous AI are therefore in high demand. These broad application specific requirements such as for SMART SoC platforms necessitates trade off among efficiency represented by performance per unit of silicon area (performance/silicon area); flexibility of usage due to changes or updates in algorithm; and low power consumption.

Conventional implementations of algorithms were usually placed at two architectural extremes of either pure hardware or pure software. Although application-specific integrated circuit (ASIC) implementation of algorithms provides the highest speed or best performance, this is however achieved via tradeoff of platform flexibility. Pure software implementations on single-chip processors or CPUs are the most flexible, but require high power overhead and result in slower processing speed. Hence, several other classes of architectural platform, such as instruction set digital signal processors (DSP) and application specific instruction set processors (ASIP), have also been used as shown in Fig. 1.

It is thus crucial that system design methodologies, such as Smart SoC systems, emphasize on optimal trade-off among efficiency, flexibility, and low-power consumptions. Consequently, embedded multicore processors or SoCs and reconfigurable architectures may frequently be favored. Furthermore, heterogeneous data generated from versatile IoT devices have further escalated system design towards cloud and heterogeneous systems in the post Moore's Law era.

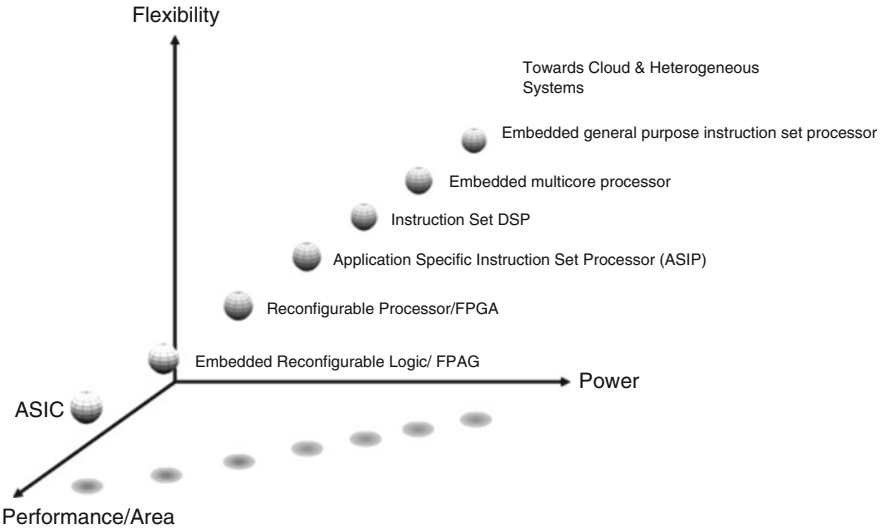


Fig. 1 Architectural platforms trading off performance/area, flexibility and power

2.2 Algorithm/Architecture Co-design: Abstraction at the System Level

As signal and information processing applications such as visual computing and communication become increasingly more complicated, corresponding increase in hardware complexity in SoC design has also required reciprocity in software design especially for embedded multicore processors and reconfigurable platforms. In coping with large systems, design details for specific applications are abstracted into several levels of abstraction.

In traditional ASIC design flow, physical characteristics are typically abstracted as timing delay at the RTL level. For Smart SoC with yet even higher complexity, abstraction has been elevated further to system level with algorithmic intrinsic complexity metrics intelligently extracted from dataflow models, featuring both hardware and software characteristics for subsequent cross level of abstraction design.

2.2.1 Levels of Abstraction

The design space for a specific application is composed of all the feasible software and hardware implementation solutions or instances and is therefore spanned by corresponding design attributes characterizing all abstraction levels [7].

In a top down manner, design process in this method proceeds from algorithm development to software and or hardware implementation. Abstracting unnecessary design details and separating the design flow into several hierarchies of abstraction

level as shown in Fig. 2 could efficiently enhance the design capability. For a specific application, the levels of abstraction include the algorithmic, architectural, register transfer, gate, and physical design levels. As shown in Fig. 2, more details are added as the design progresses to lower abstraction levels and hence with larger design space.

Figure 3 illustrates design details at every abstraction level of the design space. At the algorithmic level, functionalities are explored, and the characterizing time unit used is in order of seconds. Real-time processing, for example, is a common constraint for visual applications, and the temporal domain precision is measured in terms of frames per second (FPS).

At the architectural level, exploration focuses on data transaction features including data transfer, storage, and computation. These information subsequently facilitate design for hardware/software partition, memory configuration, bus protocol, and modules comprising the system. The time unit is in number of cycles.

At the silicon intellectual property (IP) or macro level, micro-architecture characteristics including the datapath and controller are considered, with the timing accuracy also counted in cycles. At the module level, features could for instance be various arithmetic units comprising the datapath. The gate level is characterized by logic operation for digital circuits. At the circuit level, voltage and current are notable and finally electrons are considered at the device level.

The discussions above reveal that higher levels of abstraction are characterized by coarser timing and physical scales and are finer at lower levels. In traditional ASIC design flow, efforts were focused primarily at the register transfer level (RTL), where physical circuit behaviors with parasitical capacitance and inductance are abstracted

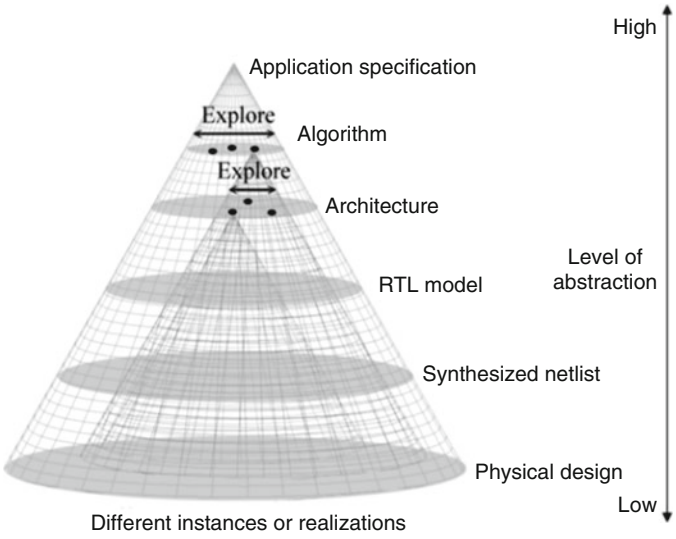


Fig. 2 Levels of abstraction

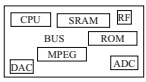
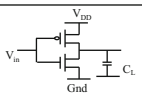
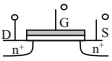
Levels	Symbols	Features	Time units
Algorithm		System functionality	Seconds
Architecture		System architecture	Number of cycles
IP (Macro)		IP functionality and micro-architecture	Number of cycles
Module		Arithmetic operation	Cycle
Gate		Logic operation	ns
Circuit		Voltage, current	ps
Device		Electron	ps

Fig. 3 Features at various levels of abstraction

within timing delay. In the currently proposed AAC design methodology, abstraction is further elevated to the system level where dataflow or transaction-level modeling bridges the cross algorithm and architecture levels design space.

2.2.2 Joint Exploration of Algorithms and Architecture

Traditional design methodologies are usually based on the execution of a series of sequential stages: the theoretical study of a fully specified algorithm, the mapping of the algorithm to a selected architecture, the evaluation of the performance, and the final implementation. However, these sequential design procedures are no longer adequate to cope with the increasing complexity demands of Smart SoC design challenges. Conventional sequential design flow yields independent design and development of the algorithm from the architecture. However, with ever increasing complexity of both algorithm and system platforms in each successive generation, such unidirectional steps in traditional designs will inevitably lead to the scenario that designers may either develop highly efficient but highly complex algorithms that cannot be implemented or else may offer platforms that are impractical for real world applications because the processing capabilities cannot be efficiently exploited by the newly developed algorithms. Hence, seamless weaving of the two previously autonomous algorithmic development and architecture development will unavoidably be observed.

As shown in Fig. 4, AAC facilitates the concurrent exploration of algorithm and architecture optimizations through the extraction of algorithmic intrinsic complexity

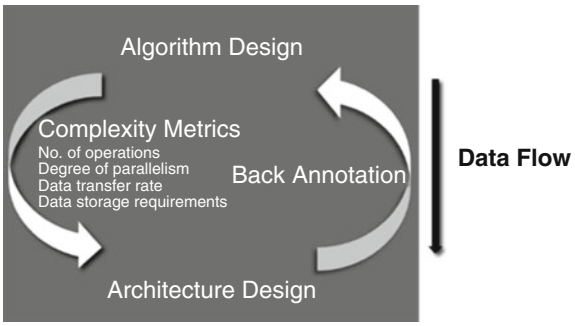


Fig. 4 Concept of algorithm/architecture co-exploration

measures from dataflow models. Serving as a bridge between algorithms containing behavioral information and architecture with design or implementation information, system level features including, number of operations, degree of parallelism, data transfer rate, and data storage requirements are extracted as quantitative complexity measures to provide early understanding and characterization of the system architecture in cross level designs.

As depicted in Fig. 2, the cost of design changes is high when designs have already progressed to the later stages at lower level of abstraction and frequently affects the success of the overall project. Hence it is crucial that these algorithmic intrinsic complexity measures provide early understanding of the architectural design and subsequent implementation requirements within the algorithm and architecture co-design space as shown in Fig. 5. This is in essence a systematic analytics architecture mechanism for the mapping of algorithms to platforms with optimal balancing of efficiency, flexibility, and power consumption via architectural space exploration before software/hardware partitioning.

In situations when the existing architectures or platforms are not able to accommodate the complexities as it is necessary to feedback or back annotate the complexity information to the algorithmic level for algorithm modification as depicted in Figs. 4 and 5.

Hence AAC provides a cross level methodology for smart system design by which abstraction of architecture features within complexity metrics has been further escalated to the system level! This is of course the same technique in traditional ASIC design flow with physical characteristics at physical layers being abstracted as timing parameters at the microarchitecture or RTL level.

2.3 Algorithmic Intrinsic Complexity Metrics and Assessment

Finding out intrinsic complexity metrics of algorithms providing important architectural information is critical for Algorithm/Architecture Co-Exploration (AAC) since the metrics is capable of being feedback- or back-annotated in early design stages to facilitate concurrent optimizations of both algorithm and architectures.

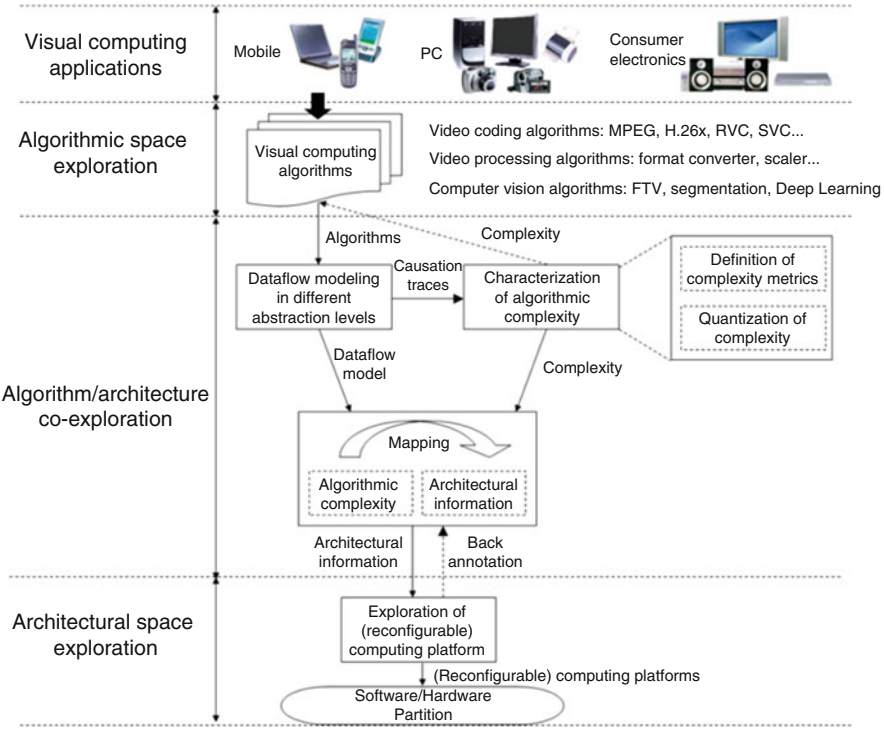


Fig. 5 Advanced visual system design methodology

The complexity metrics have to be intrinsic to the algorithm and hence are not biased toward either hardware or software. In other words, they should be platform independent so as to reveal the anticipated architectural features and electronic ingredients in the early design stages. In order to characterize the complexity of algorithms, this chapter introduces four essential algorithmic intrinsic complexity metrics, number of operations, degree of parallelism, data transfer rate, data storage requirement, and the corresponding quantification methods based on the metrics.

2.3.1 Number of Operations

The number of arithmetic and logic operations is one of the most intuitive metrics that can measure the intrinsic complexity of an algorithm during computation. An algorithm possessing more operations requires more computational power in either the software on processor-based platforms or the hardware on application-specific system platforms. Consequently, the number of operations in terms of these four arithmetic operators, including addition, subtraction, multiplication, and division and logic, operations can be used to characterize the complexity of the algorithm and

hence to provide insight into architectures such as number of processing elements (PE) needed and the corresponding operating clock rate for real-time applications.

Estimating the number of operations of an algorithm can provide designers with the intrinsic complexity that is independent of whether implementation is in software or hardware. The number of operations can exhibit the gate count estimation if implementation is intended in ASICs. Furthermore, extracting the common operations and the number of operations in an algorithm can help engineers figure out feasible field programmable gate array (FPGA) configurations. On the contrary, if an algorithm is mapped into software, one can know what kind of instruction set architecture is required in the general-purpose CPU or DSP coprocessors. Since this metrics can give designers insight into either software or hardware implementation in early design stages, it can effectively facilitate software/hardware partition and co-design.

To make this metric more accurate, the types of computational operations have to be particularly distinguished, since various operations have different costs in implementation. Among the four basic arithmetic operations, the complexity of addition and subtraction are similar and simplest, multiplication is so complex that it can be executed by a series of additions and shifts based on Booth's algorithm [2], and division is the most complicated, since it can be performed by shifts, subtractions, and comparisons. In CPU profiling, different types of operations spend distinct CPU cycles according to the instruction set architecture. In ASIC and FPGA designs, each basic mathematical operation and logic operation has different gate counts and number of configurable logic blocks (CLBs), respectively. Furthermore, other than gate count and the number of CLBs, one can estimate the average power consumption at algorithmic level according to the numbers of operation per second.

In addition to the types of operation, the precision of operand in terms of bit depth and type of operand (fixed point or floating point) can significantly influence the implementation cost and hence need to be especially specified. In general, the gate count of PE increases as the precision grows higher. Besides, the hardware propagation delay is affected by the precision as well. Hence, the precision is an important factor in determining the critical path length, maximum clock speed, and hence the throughput of electronic systems. If an algorithm is implemented on the processor-orientated platforms composed of general-purpose processors, single-instruction multiple data (SIMD) machines, or application-specific processors, the precision of operand will directly determine the number of instructions needed to complete an operation. Consequently, the operand precision is also a very important parameter as measuring the number of operations.

Furthermore, whether the input of an operator is variable or constant has to be differentiated, since a complicated constant-input operation can be executed via a few simple operations. For example, a constant-input multiplication can be implemented by fewer additions and shifts, where the shifts can be efficiently implemented by just wiring in hardware. In software, the constant operations can be executed by immediate-type instructions that need less access to registers. Hence,

the variable or constant-input operant is also a significant factor that should be considered.

The number of different types of operations can be easily quantified according to the algorithm descriptions. Horowitz et al. [11] introduced a complexity analysis methodology based on calculating the number of fundamental operations needed by each subfunction together with the function call frequency in statistics for different video contents. The worst-case and average-case computational complexity can then be estimated according to the experimental results. This method can efficiently estimate the number of operations for content-adaptive visual computing algorithms. Besides, Ravasi and Mattavelli presented a software instrumentation tool capable of automatically analyzing the high-level algorithmic complexity without rewriting program codes [26, 27]. This can be done by instrumentation of all the operations that take place as executing the program. These two techniques can dynamically quantify the relatively intrinsic algorithmic complexity on number of operations for ESL design.

2.3.2 Degree of Parallelism

The degree of parallelism is another metric characterizing the complexity of algorithms. Some partial operations within an algorithm are independent. These independent operations can be executed simultaneously and hence reveal the degree of parallelism. An algorithm whose degree of parallelism is higher has larger flexibility and scalability in architecture exploration. On the contrary, greater data dependence results in less parallelism, thereby giving a more complex algorithm. The degree of parallelism embedded within algorithms is one of the most essential complexity metrics capable of conveying architectural information for parallel and distributed systems at design stages as early as the algorithm development phase. This complexity metric is again transparent to either software or hardware. If an algorithmic function is implemented in hardware, this metric is capable of exhibiting the upper bound on the number of parallel PEs in datapath. If the function is intended in software, the degree of parallelism can provide insight and hence reveal information pertaining to parallel instruction set architecture in the processor. Furthermore, it can also facilitate the design and configurations of multicore platforms.

Amdahl's law introduced a theoretical maximum speed-up for parallelizing a software program [1]. The theoretical upper bound is determined by the ratio of sequential part within the program, since the sequential part cannot be paralleled due to the high data dependencies. Amdahl's law provided an initial idea in characterizing parallelism. In a similar manner, the instruction-level parallelism (ILP) that is more specific for processor-oriented platforms is quantified at a coarser data granularity based on the graph theory [8]. The parallelization potential defined based on the ratio between the computational complexity and the critical path length is also capable of estimating the degree of parallelism [24]. The computational complexity is measured by means of the total number of operations,

and the critical path length is then defined as the largest number of operations that have to be sequentially performed. The parallelization potential based on the number of operations reveals more intrinsic parallelism measurements at a finer data granularity as compared to Amdahl's law and the ILP method.

Kung's array processor design methodology [16] employed the dependency graph (DG) to lay out all basic operation to the finest details in one single step based on single assignment codes. Hence, DG is capable of explicitly exhibiting data dependencies between detailed operations of dataflow at the finest granularity. This design methodology provides more insight into the exploitation of algorithmic intrinsic parallelism. For instance, the systolic arrays architecture can efficiently implement algorithms possessing regular dependency dataflow graphs (DFGs), such as the full search motion estimation. As considering algorithms having irregular data dependencies, the outlines of causation trace graphs [14] generated by dataflow models were used by Janneck et al. in rendering a comparative characterization of parallelism. Similar to Parhi's folding and unfolding techniques [23], the thinner portion of a causation trace graph contains more sequential operations, while the wider portion of has relatively higher degree of parallelism.

One of the versatile parallelisms embedded within algorithms can be revealed as the independent operation sets that are independent of each other and hence can be executed in parallel without synchronization. However, the independent operation sets are composed of dependent operations that have to be sequentially performed. Hence, in a strict manner, the degree of parallelism embedded in an algorithm is equal to the number of the fully independent operation sets. To efficiently explore and quantify such parallelism, Lee et al. [20] proposed to represent the algorithm by a high-level dataflow model and analyze the corresponding DFG. The high-level dataflow model is capable of well depicting the interrelationships between computations and communications. The generated DFG can clearly reveal the data dependencies between the operations by vertexes and directed edges, where the vertexes denote the operations and the directed edges represent the sources and destinations of the data, which is similar to the DG used in Kung's array processor design methodology [16] and the causation trace graphs proposed by Janneck et al. [14].

Inspired by the principal component analysis in the information theory, Lee et al. [20] further employed the spectral graph theory [6] for systematically quantifying and analyzing the DFGs via Eigen-decomposition, since that the spectral graph theory can facilitate the analysis of data dependency and connectivity of the DFGs simplistically by means of linear algebra. Consequently, it is capable of quantifying the parallelism of the algorithm with robust mathematically and theoretical analysis applicable to a broad range of real-world scenarios.

Given a DFG G of an algorithm composed of n vertexes that represent operations and m edges that denote data dependency and flow of data, in which the vertex set of G is $V(G) = \{v_1, v_2, \ldots, v_n\}$ and the edge set of G is $E(G) = \{e_1, e_2, \ldots, e_m\}$. The spectral graph theory can study the properties of G such as connectivity by the analysis of the spectrum or eigenvalues and eigenvectors of the Laplacian matrix $\mathbf{L}$ representing G, which is defined as [6, 9]

$$\mathbf{L}(i,j) = \begin{cases} \text{degree}(v_i) & \text{, if } i = j, \\ -1 & \text{, if } v_i \text{ and } v_j \text{ are adjacent,} \\ 0 & \text{, otherwise.} \end{cases} \quad (1)$$

where degree(v_i) is the number of edges connected to the ith vertex v_i. In the Laplacian matrix, the ith diagonal element shows the number of operations that are connected to the ith operation and the off-diagonal element denotes whether two operations are connected. Hence, the Laplacian matrix can clearly express the DFG by a compact linear algebraic form.

Based on the following well-known properties of the spectral graph theory: (I) the smallest Laplacian eigenvalue of a connected graph equals 0 and the corresponding eigenvector = $[1, 1, \ldots, 1]^T$, (II) there exists exactly one eigenvalue = 0 for the Laplacian matrix of a connected graph, and (III) The number of connected components in the graph equals the number of eigenvalue = 0 of the Laplacian matrix, it is obvious that in a strict sense, the degree of the parallelism embedded within the algorithm is equal to the number of the eigenvalue = 0 of the Laplacian matrix of the DFG. Besides, based on the spectral graph theory, the independent operation sets can be identified according to the eigenvectors associated with the eigenvalues = 0. Furthermore, by comparing the eigenvalues and eigenvectors of each independent operation set, one can know whether the parallelism is homogeneous or heterogeneous, which is critical in selecting or designing the instruction set architecture.

This method can be easily extended to the analysis of versatile parallelisms at various data granularities, namely multigrain parallelism. These multigrain parallelisms will eventually be used for the exploration of multicore platforms and reconfigurable architectures or Instruction Set Architecture (ISA) with coarse and fine granularities, respectively. If the parallelism is homogeneous at fine data granularity, the SIMD architecture is preferable, since the instructions are identical. On the contrary, the very long instruction word (VLIW) architecture is favored for dealing with the heterogeneous parallelism composed of different types of operations. As the granularity goes coarser, the types of parallelism can help design the homogeneous or heterogeneous multicore platforms accordingly. In summary, this method can efficiently and exhaustively explore the possible parallelism embedded in algorithms with various granularities. The multigrain parallelism extracted can then facilitate the design space exploration for the advanced AAC.

By directly setting eigenvalues of $\mathbf{L} = 0$, it is easy to prove that the degree of parallelism is equal to the dimension of the null space of $\mathbf{L}$ and the eigenvectors are the basis spanning the null space. In general, the number of operations needed to derive the null space of a Laplacian matrix is proportional to the number of edges. Hence, this method provides an efficient approach to quantify the degree of parallelism and the independent operation sets. This method is applicable to large-scale problems by avoiding the computation-intensive procedures of solving traditional Eigen-decomposition problem. In addition, since the Laplacian matrix is

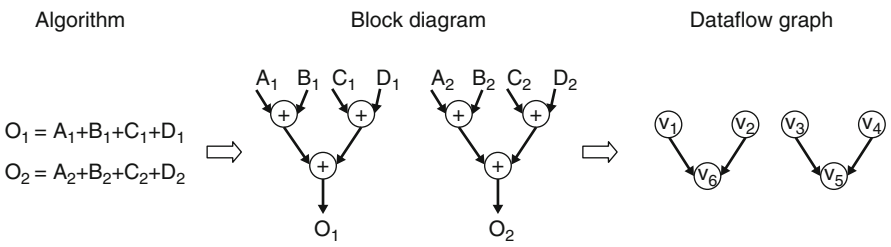


Fig. 6 An example for an illustration of quantifying the algorithmic degree of parallelism

sparse and symmetrical, it can be efficiently implemented and processed by linking list or compressed row storage (CRS) format.

Figure 6 displays a simple example to illustrate the quantification of the algorithmic intrinsic parallelism. The DFG composed of six operations represented by vertexes labeled with different numbers. The corresponding Laplacian matrix $\mathbf{L}$ of the DFG with the arbitrary label is

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & -1 & 2 & 0 \\ -1 & -1 & 0 & 0 & 0 & 2 \end{bmatrix} \tag{2}$$

The eigenvalues and the corresponding eigenvectors of $\mathbf{L}$ are

$$\begin{matrix} \lambda = 0 & 0 & 1 & 1 & 3 & 3 \end{matrix}$$

$$\mathbf{x} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ -2 \\ 0 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 2 \end{bmatrix}, \tag{3}$$

where λ and $\mathbf{x}$ are the eigenvalues and eigenvectors of $\mathbf{L}$, respectively. From the above result, we can know that the DFG is composed of two independent operation sets, since it has two Laplacian eigenvalues = 0. So, the degree of parallelism in this algorithm is two. Subsequently, by observing the first eigenvector associated with $\lambda = 0$, we can find that the values corresponding to v_1, v_2, and v_6 are nonzero, indicating that the three operations form a connected dataflow subgraph. In a similar manner, the other eigenvectors associated with $\lambda = 0$ can reveal the rest connected dataflow subgraph. Besides, one can find that the two independent operation sets

should be isomorphic, since their eigenvalues and eigenvectors are identical. Hence, the parallelism in this algorithm is homogeneous. This example precisely explains the parallelism extraction and analysis method based on the spectral graph theory.

The spectral parallelism quantification method has several advantages. First of all, it provides a theoretically robust method in quantifying the parallelism of algorithms, whereas the causation trace [16] provided only comparative information for the potentials of parallelisms. Besides, benefiting from dataflow modeling, this method is also applicable for characterizing algorithms with irregular data dependencies. In addition, as compared to the analysis based on the high-level programming model in [24] and the quantification of ILP in [8], the parallelism metric is more intrinsic and hence will not be specific only to processor-oriented platforms and is capable of mapping algorithms onto generic platforms and even those for distributed systems. However, the quantification of ILP [8] is used primarily for software implementations. Furthermore, the data structures in instruction-level programming models could influence the parallelism extracted in [24].

In traditional graph theory, connected components can be identified by the depth first search (DFS) or breadth first search (BFS). In general, the algorithmic complexity of the DFS and BFS in terms of the number of operations is linearly proportional to the number of edges plus the number of vertexes. However, the number of operations required by the spectral framework is just proportional to the number of edges when solving the null space of the Laplacian matrix. In addition, the multigrain spectral analysis is capable of systematically decomposing DFGs in a top-down manner, since the eigenvalues and eigenvectors of a graph is the union of those of its individual components. Besides, the spectral method can effectively tell whether the parallelism is either homogeneous or heterogeneous. Furthermore, the spectrum of Laplacian matrix is invariant of the graph matrix regardless of orders in which the vertices are labeled and the Laplacian matrix can be efficiently implemented in CRS format. These features make the handling of matrices representing DFGs efficient in computers and hence preferable for very efficient design automation.

2.3.3 Data Transfer Rate

Aside from the number of operations and degree of parallelism, the amount of data transfer is also an intrinsic complexity metric as executing an algorithm. Algorithms can be represented by natural languages, mathematical expressions, flowcharts, pseudo codes, high-level programming languages, and so on. In signal processing applications, mathematical expression is one of the most abstract, definite, and compact methods to represent an algorithm. The corresponding signal flow graphs and dataflow models [7, 25] can be then obtained based on mathematical representation [21]. The dataflow graph is capable of depicting the interrelationships between computations and communications.

To systematically extract the information embedded in graph, matrix representation is commonly used to represent a DFG. For instance, adjacent matrix

introduces the connections among vertices and Laplacian matrix also displays the connectivity embedded in graph. These matrix representations are usually in behalf of undirected graph; however, in the study of data transfer of visual signal processing, data causality is also a significant information that should be retained in matrix representation. Hence, a dependency matrix conveying data causality of a directed or undirected graph is required, and its mathematical expression is illustrated as (4).

$$\mathbf{M}(i,j) = \begin{cases} -1 & \text{, if vertex } v_j \text{ is the tail of edge } e_i \\ 1 & \text{, if vertex } v_j \text{ is the head of edge } e_i \\ 0 & \text{, otherwise} \end{cases} \tag{4}$$

To explore the method to quantify corresponding data storage requirement and data transfer rate via dependency matrix, edge cut is applied since edge cut is a cut that results in a connected DFG into several disconnected sub-DFGs by removing the edges in this cut. Therefore, the size of edge cut (or number of edges in this cut) could be used to estimate the amount of data would be transferred among sub-DFGs due to the fact that data should be sent or received (via edges) by tasks (vertices). On the other hand, the behavior of edge cut in DFG is equivalent to applying an indicator vector **x** that separates vertices in DFG into two sides for dependency matrix, **M**. For example, a simple DFG of an average filter is shown in Fig. 7, the indicator vector **x** of corresponding edge cut is $[1, -1, -1, 1,]^T$, then this edge cut separates v_1 and v_4 into one group and v_2 and v_3 belong to the other group. (The vertices at the side with more input data would be set as 1.) Furthermore, by computing **Mx**, the characteristics of edges in DFG would be revealed. In this example, **Mx** is $[2, 0, -2]^T$ and there are three type of edges that are introduced by **Mx**, including in-edge-cut (value in **Mx** is positive, e_1), out-edge-cut (value in **Mx** is negative, e_3), non-edge-cut (value in **Mx** is zero, e_2). According to **Mx**, the amount of data transfer was equal to the half of the summation of all absolute values in **Mx**. Corresponding dependency matrix (**M**), indicator vector (**x**), characteristics of edges (**Mx**), and amount of data transfer are depicted in (5). Therefore, **Mx** clearly presents the number of edges crossed by this edge cut and hence corresponding data transfer rate could be systematically quantified due to the fact that data transactions occurred on the edges in DFG. Consequently, the amount of data transfer of this edge cut is 2.

$$\mathbf{M} = \begin{array}{c} \\ e_1 \\ e_2 \\ e_3 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix} \end{array}, \quad \mathbf{x} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} \Rightarrow \mathbf{Mx} = \begin{bmatrix} 2 \\ 0 \\ -2 \end{bmatrix}$$

$$\textit{Amount of data transfer} = \frac{1}{2}\sum_{i=1}^{N} |\mathbf{Mx}(i)| = 2 \tag{5}$$

In general, DFG presents a process for computing one Data Granularity (DG) of an algorithm and then this DFG is applied iteratively until all to-be-computed DGs are accomplished; for example, we might build up a DFG for block-based

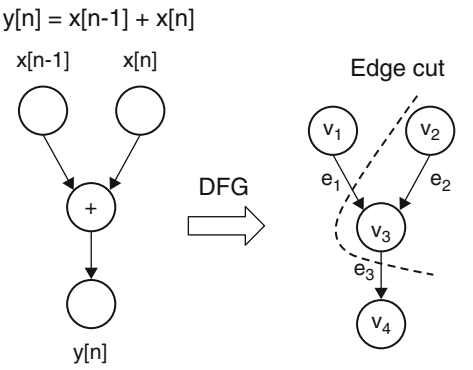


Fig. 7 A simple DFG and an edge cut separate vertices into two sides

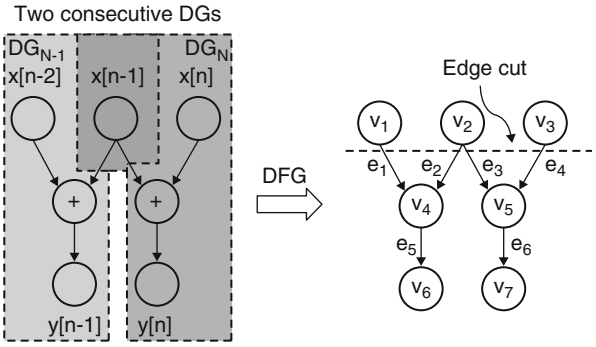


Fig. 8 A DFG composed of two consecutive DGs

Motion Estimation (ME) and hence one DG is one block; as a consequence, to achieve ME for one frame, this DFG is used for all DGs in one frame. Therefore, when we combine consecutive processes for different DGs into one DFG, there are some data could be concurrently used for two DGs, i.e., the data could be reused and the amount of data transfer would be reduced. For example, two consecutive processes of Fig. 7 is presented in Fig. 8, and another edge cut crosses all input data. Its corresponding dependency matrix **M**, indicator vector **x**, and characteristics of edges **Mx** are illustrated in (6). We could find out the corresponding amount of data transfer would be four when directly computing absolute summation over **Mx**. However, it is clear that if v_2 could be reused for both DG_{N-1} and DG_N, when DG_N denotes the DFG computes the n-th DG, the amount of data transfer would be reduced from four to three but one extra storage size is required. Here we present a systematic approach to indicate how many data could be reused and which data would be reused through the dependency matrix.

$$\mathbf{M} = \begin{array}{c} \\ e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 \end{bmatrix} \end{array}, \quad \mathbf{x} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \\ -1 \\ -1 \\ -1 \end{bmatrix} \Rightarrow \mathbf{Mx} = \begin{bmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 0 \\ 0 \end{bmatrix}$$

$$Amount\ of\ data\ transfer = \frac{1}{2}\sum_{i=1}^{N} |\mathbf{Mx}(i)| = 4 \tag{6}$$

Dependency matrix concurrently conveys the direction of data transaction and the dependency of tasks, so it can indicate that the location where data are transacted through the determined edge cut. To clearly explain the proposed method, here we define the symbols used hereafter; an edge characteristic vector **y**, equals to (1/2)**Mx** and one operator, ∩, an element-wise operation which reserves the elements with the same sign and set others as zero. Therefore, through vector **y** and operator ∩, the reusable data could be indicated. ∩ operator remains the elements that exchange data in this edge cut and hence we could create a new matrix **M'** whose i-th column is $\mathrm{col}_i(\mathbf{M}) \cap \mathbf{y}$, where $\mathrm{col}_i(\mathbf{M})$ is i-th column of matrix **M**. After that, as shown in (7), for each column in **M'**, a maximum operator would be performed on the elements with positive values to calculate maximum numbers of data should be sent from this vertex; on the other hand, for each column in **M'**, the remaining elements with negative values would be summed up to be the amount of output data. Hence, the amount of data transfer with data reuse could be quantified systematically. Furthermore, when we merge more DGs into one DFG, we have the potential to reduce more data transfer; however, storage requirement would also be increased if more DGs are considered at the same time. As a result, we have a systematic manner to explore design space in terms of amount of data transfer and storage requirement.

Data could be reused

$$\mathbf{y} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{M'} = \begin{array}{c} \\ e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$|1| + |1| + |1| + 0 + 0 + 0 + 0 = 3$$

$$Amount\ of\ data\ transfer = 3\ (v_2\ is\ reused) \tag{7}$$

2.3.4 Data Storage Requirement

A system is said to be memoryless if its output depends on only the input signals at the same time. However, in visual computing applications such as video coding and processing, some intermediate data have to be stored in memory depending on the dataflow of algorithms in higher abstraction levels. Consequently, in order to perform the appropriate algorithmic processing, data storage must be properly configured based on the dataflow scheduling of the intermediate data. Hence, the

algorithmic storage configuration is another essential intrinsic complexity metric in AAC design methodology, which is transparent to either software or hardware designs. For software applications, the algorithmic storage configuration helps design the memory modules such as cache or scratch-pad and the corresponding data arrangement schemes for the embedded CPU. In hardware design, the immediate data can be stored in local memory to satisfy the algorithmic scheduling based on this complexity metric. The minimum storage requirement of an algorithm is determined by the maximum amount of data that needed to be accessed at a time instance, which of course depends on the reuse rate of data.

To provide the better visual quality, more context information should be stored to exploit and hence the storage size requirement is intended to be increased. In usual, the picture data is stored in the external storage due to the large amount of data. Therefore, data transfer rate balance between internal and external storage is crucial. There are two extreme cases of this consideration. (I) All the needed data is stored in the internal storage that requires the minimum external data transfer rate and (II) all the required data is stored in the external storage that requires the maximum external data transfer rate since the needed data would be fetched when the algorithm demanded. An intuitive manner to allocate partial picture data in the internal storage and remaining data in the external storage. However, these two factors are usually inversely proportional. In the following subsections, a systematic manner to explore the balance between internal data storage and external data transfer rate through different executing orders and various data granularities. Hence, a feasible solution can be found during the design space exploration for the target application of multidimensional video signal processing.

The first factor, executing order in dataflow, affects internal storage size and external data transfer rate and the executing order is always restricted to the data causality of the algorithm. Figure 9 shows a dataflow dependency graph of a typical image/video processing algorithm exploiting the contextual information in the spatial domain. To a causal system, only upper and left contextual information can be referenced.

Three different executing order is illustrated in Fig. 10, including (a) the raster scan order, (b) diagonal scan order with two rows, and (c) diagonal scan order with three rows and the number labeled on the vertices denotes that the executing order of nodes. There are some assumptions are applied for discussing the effect of executing order on the internal data storage and the external data transfer rate. The contextual information at left side is stored in the internal storage and the data at upper line should be fetched from external storage. Thus, the internal storage size is counted according to the data size of left reference and average external data transfer rate is measured based on the amount of upper data reference should be fetched within one time unit.

By analyzing the dataflow illustrated in Fig. 10a, the required storage size is the one data unit and external data transfer rate is three data units. The dataflow depicted in Fig. 10b needed to store three data units and transfer three data units during processing every two data units. The last one dataflow illustrated in Fig. 10c

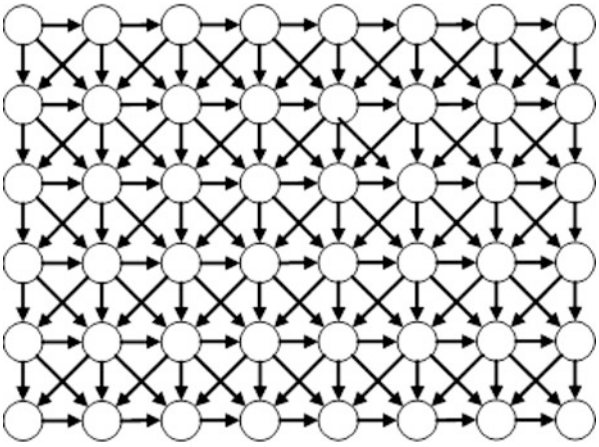


Fig. 9 Dataflow dependency graph of a typical image/video processing algorithm

stored five data units and three data units should be transferred when processing every three data units.

In summary, the first dataflow requires the smallest data storage requirement but the average data transfer rate is the largest among these three dataflow models due to the fact that the required data would be fetched from external data storage once requisition. On the other hand, the third dataflow possesses the largest internal storage size since more contextual information should be kept to process the data unit at distinct rows; however, the required average data transfer rate is the smallest one because most of data have been stored in the internal storage already.

The tradeoff between internal storage size and average data transfer rate is made in accordance with the distinct executing orders. Figure 11 showed the analyzed result from diagonal scan from one row to thirty-two rows. The normalized average data transfer rate is inverse proportional to the internal data storage size. Figure 11 shows that the reduction ratio of average data transfer rate could be achieved by adding some overhead on the internal storage size. The curve in Fig. 11 can facilitate the design space exploration in terms of the internal data storage and external data transfer rate based on AAC.

The second factor, data granularities in dataflow, affects internal storage size and external data transfer rate. For example, transformation from pixel-wise raster scan (Fig. 12a) to block-wise raster scan (Fig. 12b) is the concept to change the data granularity from fine data granularity to coarse data granularity; that is, design space is explored across various data granularities. For instance, filter processing exploits spatial information to determine the local feature and it usually needs to extend taps for filtering. Hence, the dataflow with coarser data granularity (Fig. 12b) possess higher possibility to reuse the data since there are data overlapped between two consecutive blocks. By analyzing the dataflow with coarser data granularity, the internal storage size can be characterized as (8):

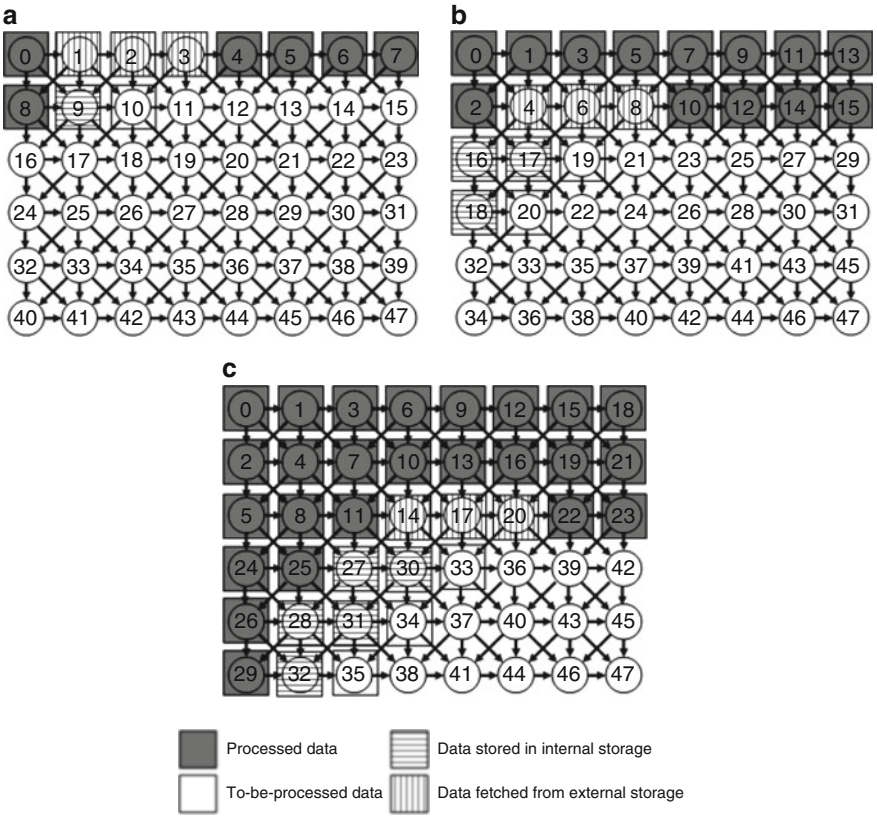


Fig. 10 Storage size comparison of various executing orders

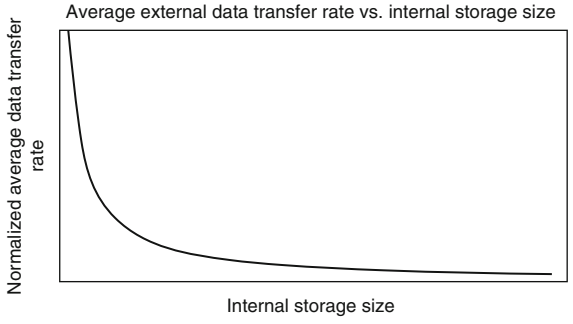


Fig. 11 Normalized average external data transfer rates versus internal storage sizes for various executing order

$$(B_H + N_H - 1) \times (B_V + N_V - 1) \tag{8}$$

where N_H and N_V are the extended taps required from the algorithm in the horizontal and vertical directions, respectively. B_H and B_V are the width and height of one data granularity. The amount of non-overlapped input data needed for the current processed granularity is expressed by:

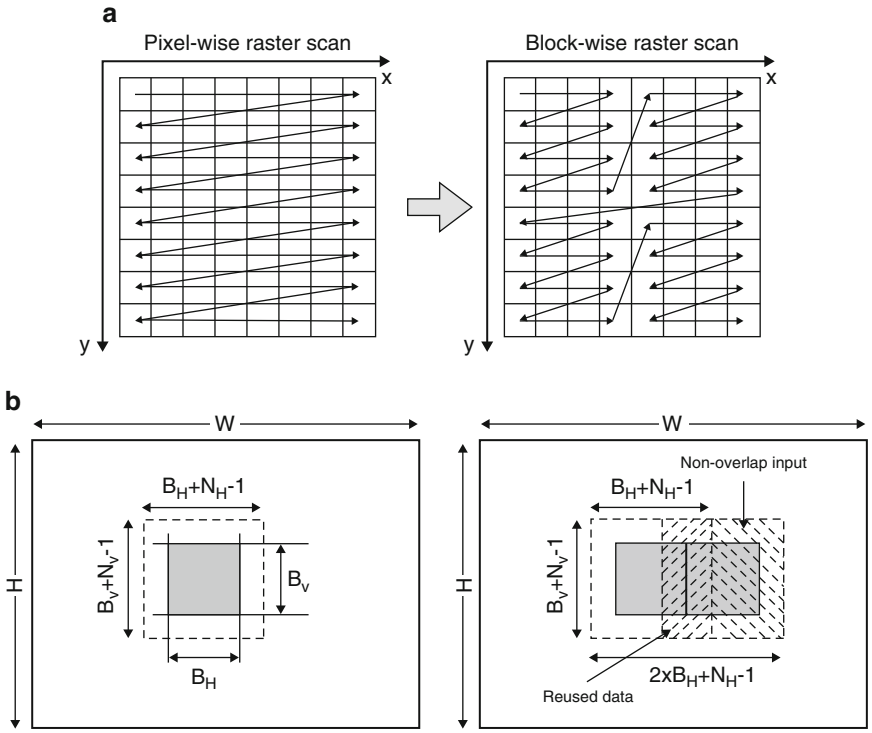


Fig. 12 Dataflow flow with coarse-granularity. (**a**) Pixel-wise raster scan vs. block-wise raster scan. (**b**) Overlap of input data

$$B_H \times (B_V + N_V - 1) \tag{9}$$

and the accompanying input data transfer rate per granularity is

$$(B_V + N_V - 1)/B_V \tag{10}$$

The amount of non-overlapped input data depended on B_H, B_V, and N_V. On the other hand, the external data transfer rate per granularity is only related to the B_V and N_V since that the parameter, B_H, is compensated by the raster scan processing order. Regarding to the vertical scan order, the results can be derived in the similar manner and the mathematic expressions are similar to (9) and (10). According to expression in (10), external data transfer rate can be adjusted by using different data granularities but this scheme results in various internal storage requirement as illustrated in (8). Again, the results show that data storage requirement is inverse proportional to external data transfer rate. Hence, this exploration scheme can efficiently reduce external data transfer rate with a few overhead in internal storage.

Subsequently, according to the algorithmic characteristics which is utilized for different applications, design space can be systematically explored by changing different executing orders and various data granularities; furthermore, some parameters in algorithm are also took into consideration to determine design solution, e.g. N_H and N_V. For example, the line-based scan [5] stores the intermediate 1-D filter, which results in embedded line buffers to minimize the external data transfer rate. The block-based scan [15, 30] can further facilitate the trade-off between internal storage and external data transfer rate with appropriate data granularity, based on the size of the sliding windows of filters. In addition, the stripe-based scan [13] takes the data granularity and the executing order into consideration, so that it gives extra degree of freedom for exploring internal storage and external data transfer rate.

In contrast to average data transfer rate, external instantaneous data transfer rate is a critical complexity metric of algorithm since external peak data transfer which reveal the potential bandwidth which could affect the bus configuration and arbiter when design goes to lower level of abstraction. With a dataflow of an algorithm, external peak data transfer could be found by exploring various executing orders and data granularities although average data transfer could be identical.

For instant, by using a larger granularity to fetch data can smooth the discrete instantaneous data transfer rate; however, it also results in increased internal storage requirement. Consequently, the lowest external peak data transfer rate could be considered as an optimization problem whose objective function is trying to find the lowest external peak data transfer rate among all possible external peak data transfer rate with different executing orders and various data granularities. The theoretical lower bound is expressed by

$$min\{R_{peak}\} = min\{max\{R[n]\}\} \tag{11}$$

where R_{peak} is the external peak data transfer rate and $R[n]$ denotes data transfer rate of all possible executing orders and data granularities.

2.4 Intelligent Parallel and Reconfigurable Computing

As discussed in previous sections, AAC presents a technique, which based on spectral graph theory, systematically lays out the full spectrum of potential parallel processing components Eigen-decomposed into all possible data granularities. This makes possible the study of both quantitative and qualitative potentials for homogeneous or heterogeneous parallelization at different granularities as opposed to systolic array for homogeneous designs at only one single fixed granularity. In addition, we have also discussed on the capabilities of AAC in facilitating systematic analysis of dataflow models for flexible and efficient data transfer and storage.

Reconfigurable architectures including multicore and GPU platforms provide balance between flexibility, performance, and power consumption. Starting from

algorithm, the data granularity could be reduced so as to extract common functionalities among different algorithms. To reduce the granularity from the architectural side, the Eigen-decomposition of dataflow models described above could also be used to decompose connected graphs to disconnect components with different granularities. These commonalities would then require one design of either software and or hardware which could be share. These lower granularity commonalities also provide quantitative guidance in reconfiguring architectural resources such as in multicores or GPUs through graph component synthesis.

The Eigen-analysis of dataflow graphs and graph component synthesis in AAC for parallel and reconfigurable computing therefore provide a framework similar to the analysis and synthesis equations in Fourier analysis.

3 AAC Case Studies

In multicore platforms, the algorithmic complexity analysis, especially of the degree of parallelism helps map applications onto homogeneous or multigrain heterogeneous architectures. In addition, the complexity analysis also provides essential information to develop retargetable compilers for multicore platforms. Furthermore, it is capable of even facilitating porting operating systems onto the platforms, since designers are aware of the algorithmic intrinsic complexity, thereby understanding how to appropriately schedule the task.

As the data granularity of the dataflow studied is fine enough, the algorithmic complexity analysis can be used to extract features common to different algorithms and formats that are adaptive to versatile video contents. The commonality extracted can, of course, help in designing datapath and controllers from the hardware perspective, thereby resulting in highly efficient and flexible reconfigurable architectures in visual computing applications. For instance, the definition of functional units in MPEG RVC is done based on such a concept.

Consequently, building a dataflow model at a proper data granularity followed by thoroughly quantifying the complexity characterizing the algorithms reveals system architecture information and hence provides a systematic top-down design methodology for mapping visual applications onto the broad spectrum of platforms at different levels of granularity and performance. In addition, early understanding and if necessary feedback or back-annotation of architectural information or electronic ingredients enables optimization of algorithms. This section then shows case studies for illustrating mapping motion-compensated frame rate up-convertor onto multi-core platform via complexity metrics quantification and a reconfigurable interpolation.

3.1 *Mapping Motion-Compensated Frame Rate Up-Convertor onto Multi-Core Platform via Complexity Metrics Quantification*

Motion-Compensated Frame Rate Up-Convertor (MC-FRUC) [17] was an emerging technology that is used to enhance visual quality in temporal domain by interpolating virtual frames between the original frames. Visual signal processing algorithm which uses motion information is usually bandwidth-intensive and computation-intensive. MC-FRUC, whose block diagram is displayed in Fig. 13, hierarchically performs block-size ME, including Coarse ME (CME) and Refined ME (RME), to accurately extract Motion Vectors (MVs). The CME uses a spatial-temporal recursive ME to accurately track the motion trajectory of object; however, there is highly dependency between the processing of each coarse-grain block due to the fact that MVs are recursively updated by spatial neighboring blocks and temporal blocks. On the other hand, in algorithmic consideration, fixed block-size ME would suffer from the inaccurate MVs at objects boundaries; hence, the RME uses fine-grain block to refine coarse-grain MVs by re-examining neighboring coarse-grain MVs; the procedure of each fine-grain block is independent since one fine-grain block would use four coarse-grain blocks to refine or smooth MVs. Subsequently, upon having fine-grained MVs, Multiple Block Candidates (MBC) derivation would indicate several blocks located at two consecutive frames be the candidates of current to-be-interpolated block based on the motion trajectory of fine-grained MVs in both forward and backward directions. MBC resolves the problems in unilateral MVs, such as motion holes and motion block overlapped, by referencing neighboring block candidates. Subsequently, Motion Compensated Interpolation (MCI) performs pixel-wise filter among block candidates to fill out the to-be-interpolated frame.

We span design space from three perspectives, including degree of parallelism at thread-level, amount of data transfer, and storage size by varying data granularity. To explore design space, we establish the DFG to model MC-FURC. Take CME as an example, its DFG is illustrated in Fig. 14a. Every vertex is one task that computes ME of one coarse-grain block and edges denote the referenced spatial neighboring MVs.

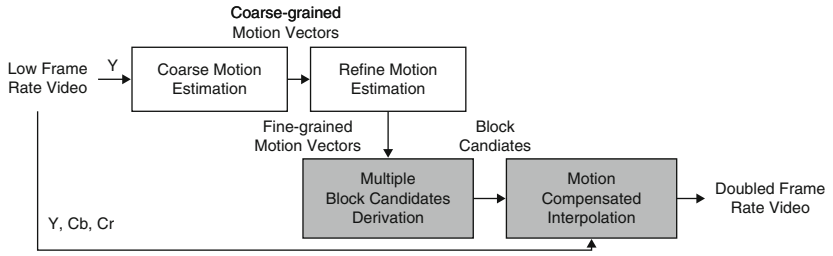


Fig. 13 Block diagram of MC-FRUC system

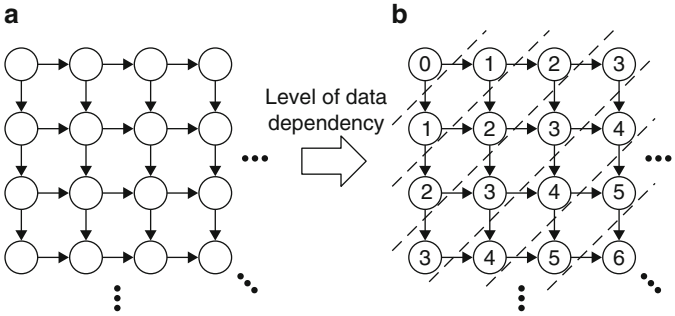


Fig. 14 DFG of CME and level of dependency of each vertex

We could apply the methodology developed by quantifying intrinsic parallelism using linear algebra for AAC [19] to quantify degree of parallelism at multi-grain granularity according to various level of data dependency. When data granularity is larger than one task, it is hard to exploit the degree of parallelism due to the fact all tasks are connected sequentially; in contrast, when we narrow down data granularity into one task, the parallelization possibility of CME is increased. In Fig. 14b, level of data dependencies are listed at each vertex and the dash lines split DFG according to level of data dependency; then, vertices in identical level of data dependency are independent. Hence, the degree of parallelism can be systematically quantified via dependency matrix of DFG. Consequently, the maximum degree of parallelism of CME is dynamic according to level of data dependency. In the beginning, degree of parallelism is 1 and then incremented to the bound of available processors, i.e., six in this case study. On the other hand, RME, MBC derivation, and MCI are also applied the same approach to exploit the degree of parallelism to maximize the performance on thread-level. We also use SIMD to enhance performance at data-level; however, we only apply SIMD for partial operations, such as similarity measurement in CME and RME or coarse-grain MVs refinement in RME, trajectory tracking for multiple blocks in MBC derivation, and multiple interpolations in MCI, due to the fact that we focus on thread-level parallelization in this subsection.

To reduce the data transaction between storages, we utilize the data flow model and linear algebra method of data transfer analysis in AAC on the transfer between local storage and external storage. We expand DFG over time to explore the data reusability; and then indicate that the data would be reused for previous Data Granularity (DG) and current DG. As a result, we could systematically determine the suitable DG with highest data reusability; then, we select this data granularity for our architecture. Although the number of data reuse is deterministic in this example due to the regular DFG of MC-FRUC; however, the data flow model and linear algebra method in AAC could also dynamically determine the ratio of data reuse when data flow of targeted algorithm is irregular or dynamic since the data flow model in AAC just depends on DFG. Take MBC derivation and MCI as an example, MBC derivation uses fine-grained MVs to derive MBC according to

motion trajectory for MCI. Hence, we investigate the DFG of MBC derivation for computing consecutive DGs, DG_{N-1} and DG_N, in Fig. 15 and the weights in DFG denotes the ratio of data size with respect to the maximum one. From the figure, a part of fine-grained MVs and reference pixels would be used for both DG_{N-1} and DG_N; that is, by using the proposed method, we could indicate how many data could be reused under the size of current DG and which data would be reused, then these data would be kept to avoid unnecessary data transaction from external storage. Then, dependency matrix of the DFG composing of DG_{N-1} and DG_N is built to systematically achieve smaller data transfer rate; in the implementation, we encapsulate 16 × 16 pixels as one vertex and 16 fine-grained MVs as one vertex in DFG to avoid huge dependency matrix. We utilize the proposed method for CME, RME, MBC derivation, and MCI, respectively, to significantly reduce data transfer rate with acceptable storage requirement.

3.2 *Reconfigurable Interpolation*

Figure 16 shows the block diagram of our reconfigurable interpolation architecture. The architecture is designed for the interpolation of one 4 × 4 block in MPEG-2, MPEG-4, and AVC/H.264. According to the macroblock partition information and motion vector for the current 4 × 4 block, the address generator determines the address(es) of the memory block(s) each reference row occupies in the cache memory. Let the memory reference row refer to the memory block(s) containing one reference row in the cache memory. The data transporter then loads each memory reference row from the cache memory to the internal memory. After loading all memory reference rows for the current 4 × 4 block, the data transporter transmits

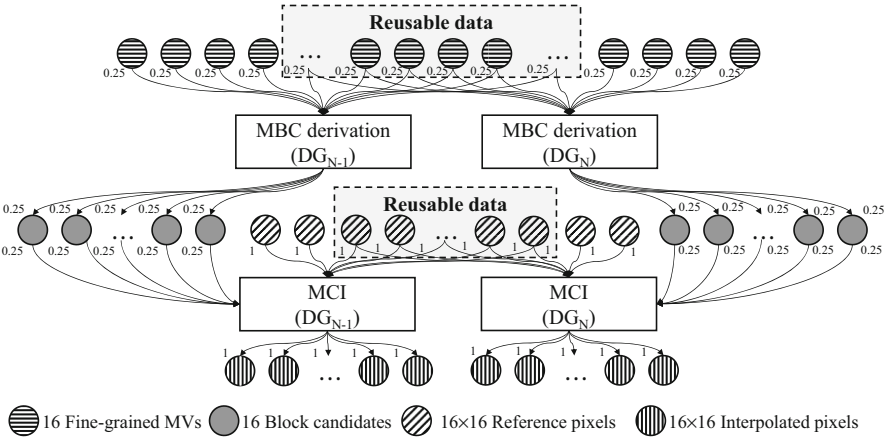


Fig. 15 DFG of MBC derivation for computing DG_{N-1} and DG_N

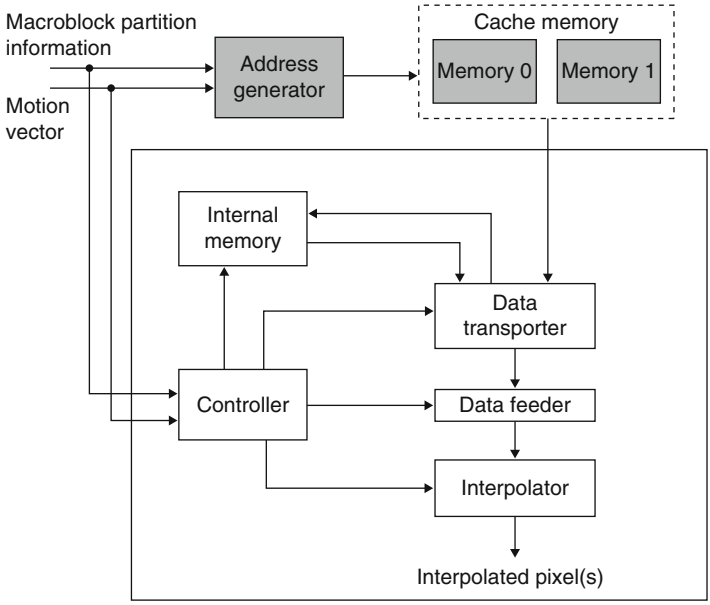


Fig. 16 Block diagram of reconfigurable interpolation architecture

each memory reference row from the internal memory to the data feeder, and in the same time loads each memory reference row for the next 4×4 block from the cache memory to the internal memory. The data feeder extracts each reference row from the input memory reference row. It then supplies the required integer-pixel samples to the interpolator properly so that the interpolator can perform subpixel sample interpolation for the target video standard. The controller controls the internal memory, the data transporter, the data feeder, and the interpolator for cooperation between them.

For the P-picture, the internal memory must store all memory reference rows for two 4×4 blocks, which are the current and the next 4×4 blocks. For the B picture, the required internal memory space is doubled. In our target video standards, the luminance and chrominance interpolations of one 4×4 block needs at most 11 and 3 memory reference row, respectively. Each memory reference row contains at most two 8-byte memory blocks. Therefore, the internal memory size is $2 \times 2 \times (11 + 3) \times 2 \times 8 = 896$ bytes. To support reading data for the current 4×4 block and writing data for the next 4×4 block simultaneously, the dual port memory is used for the internal memory.

The data feeder uses one register array to provide the required integer-pixel samples to the interpolator. The register array is divided into two parts. One is for luminance interpolation in MPEG-4 and AVC/H.264. The other is for luminance interpolation in MPEG-2 and chrominance interpolation. For luminance interpolation in MPEG-4 and AVC/H.264, the data feeder supplies integer-pixel

samples of one reference row in each cycle. One reference row in MPEG-4 and AVC/H.264 contains 11 and 9 samples, respectively. Thus, 11 bytes are used for the first part of the register array. For luminance interpolation in MPEG-2 and chrominance interpolation, any subpixel sample can be derived from 4 integer-pixel samples. The data feeder provides individual integer-pixel samples for each of 4 subpixel samples in each cycle. Thus, 16 bytes are used for the second part of the register array.

Figure 17 shows the interpolator design. The interpolator is composed of four interpolation units, one averaging and rounding (AR) unit, one dedicated buffer, and one averaging or bypassing (AB) unit. The four interpolation units derive four interpolated pixel samples simultaneously, similar to the design in [4]. The AR unit can perform the required averaging, rounding, or bypassing function for the interpolation. The dedicated buffer stores the interpolated pixel samples temporarily. The AB unit then can average or bypass the stored data to obtain the interpolated pixel samples for the B-picture or P-picture. Each interpolation unit has the same structure that mainly contains three 1-D reconfigurable FIR filter (RHFIR, RVFIR and RCFIR), one embedded averaging and rounding (EAR) unit, one 8-byte register array providing the input of RVFIR, and one 6-byte register array providing the input of RCFIR. Each reconfigurable FIR filter can be adapted to the target video standard. The pipeline register is used in the reconfigurable FIR filter. The EAR unit receives the integer-pixel samples from data feeder and the output of RHVIR. It then performs averaging and rounding operation for luminance interpolation in MPEG-4, and bypassing operation for other interpolations. Each interpolation unit receives the required integer-pixel samples from the data feeder. The interpolation process for each interpolation unit is similar.

The luminance or chrominance interpolation in MPEG-2, or chrominance interpolation in MPEG-4 use RHFIR for half-pixel sample interpolation, as shown in Fig. 18. The process is similar to that for the chrominance interpolation in AVC/H.264 with only RHFIR used. Both the Cr and Cb interpolated pixel samples are also derived at the same time.

According to commonality analysis in AAC, the reconfigurable FIR filter shown in Fig. 19 is designed for RHFIR, RVFIR, and FCFIR. The design only utilizes shifters and adders to realize the filter coefficients. The multiplexers are used to select the data path for each interpolation filter. In addition, the pipeline registers are added to reduce the critical path delay and achieve the performance for the design specification.

Figure 20 illustrates the configuration of the reconfigurable FIR filter for each interpolation filter. In Fig. 20a, b, the 8-tap and 6-tap filters support the derivation of one subpixel sample in MPEG-4 and AVC/H.264. In Fig. 20c, the filters with coefficients $(1, 1)$ and $(1, 1, 1, 1)$ support the derivation of 1 integer-pixel and 3 subpixel samples in MPEG-2 or MPEG-4, or the derivation of 4 neighboring samples for each recursive stage in AVC/H.264.

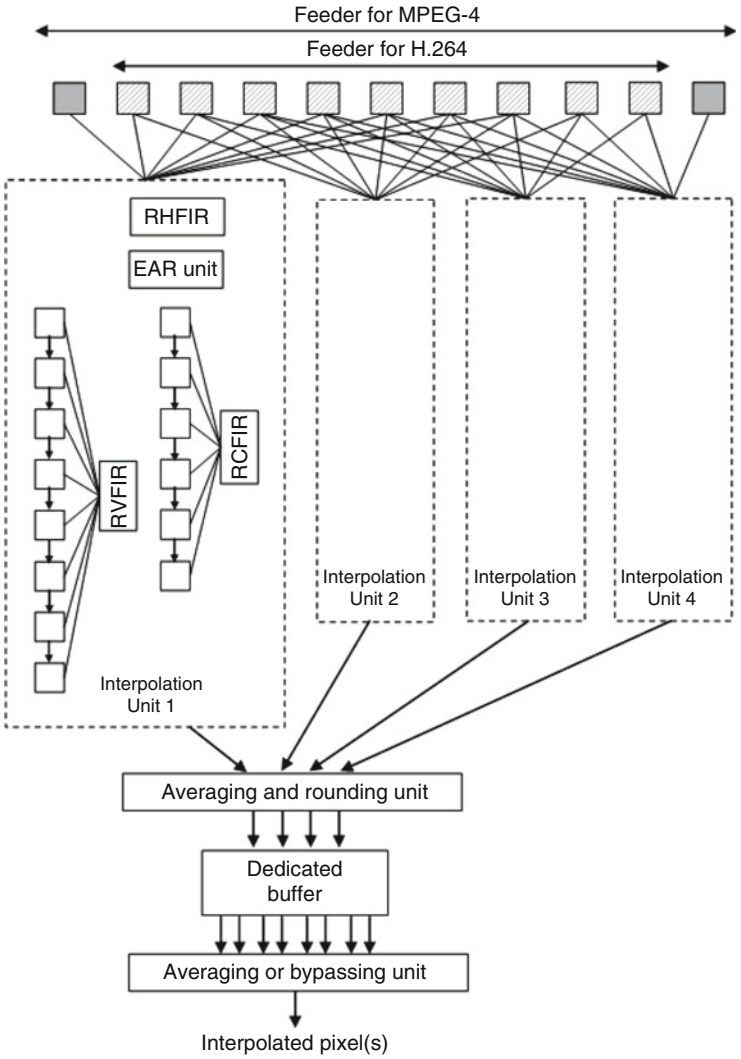


Fig. 17 The interpolator design

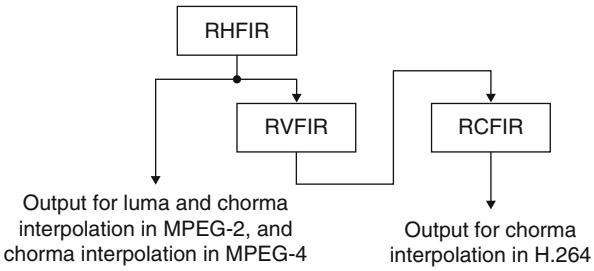


Fig. 18 Data path for luminance and chrominance interpolations in MPEG-2, chrominance interpolation in MPEG-4, and chrominance interpolation in AVC/H.264

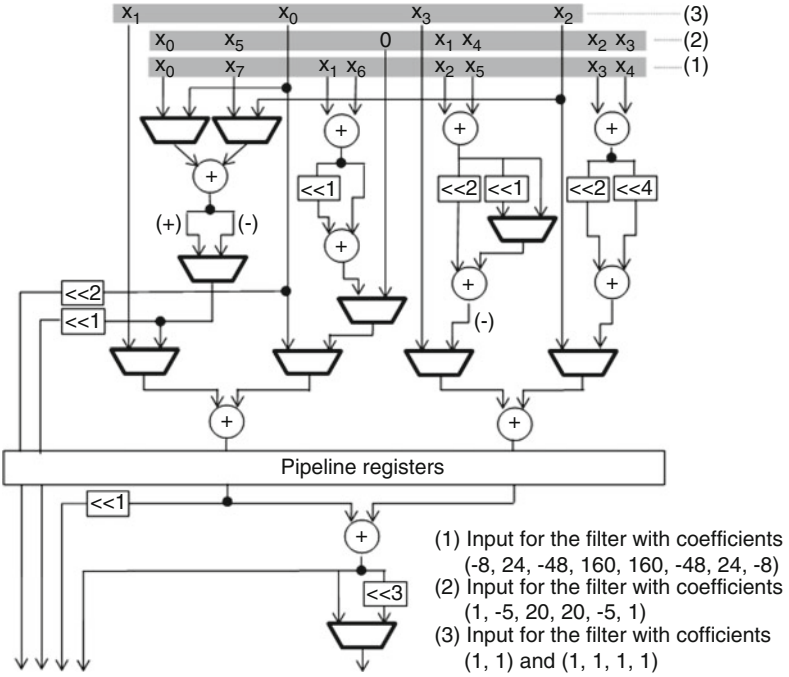


Fig. 19 Reconfigurable FIR filter structure

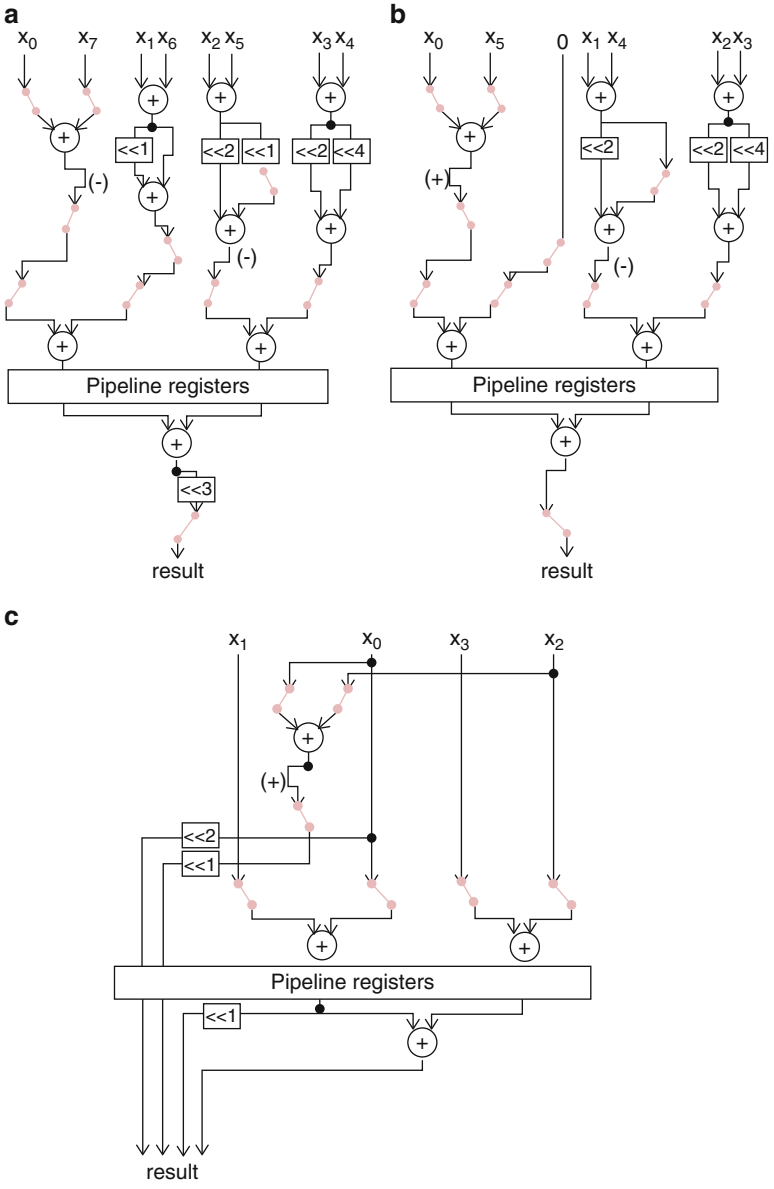


Fig. 20 Configuration of reconfigurable FIR filter for each interpolation filter. (**a**) The case for the filter with coefficients (8, 24, 48, 160, 160, 48, 24, 8). (**b**) The case for the filter with coefficients (1, 5, 20, 20, 5, 1). (**c**) The case for the filters with coefficients (1, 1) and (1, 1, 1, 1)

References

1. Amdahl, G.M.: Validity of single-processor approach to achieving large-scale computing capability. In: Proceedings of AFIPS Conference, pp. 483–485. Atlantic, New Jersey (1967)
2. Booth, A.D.: Signed binary multiplication technique. Quarterly Journal of Mechanics and Applied Mathematics **4**(2), 236–240 (1951)
3. Carron, L., Rutzig, M.B.: Multi-core system on chip. Handbook of Signal Processing Systems, 1st edition., Springer pp. 485–514 (2010)
4. Chen, J.W., Lin, C.C., Guo, J.I., Wang, J.S.: Low Complexity Architecture Design of AVC/H.264 Predictive Pixel Compensator for HDTV Application. In: Proc. ICASSP2006, vol. 3, pp. III–932–III–935 (2006)
5. Chrysafis, C., Ortega, A.: Line-based, reduced memory, wavelet image compression. IEEE Trans. on Image Processing **9**(3), 378–389 (2000)
6. Chung, F.R.K.: Spectral graph theory. Regional Conferences Series in Mathematics (92) (1997)
7. Edwards, S., Lavagno, L., Lee, E.A., Sangiovanni-Vincentelli, A.: Design of embedded systems: Formal models, validation and synthesis. In: Proceedings of the IEEE, vol. 85, pp. 366–390 (1997)
8. Escuder, V., Duran, R., Rico, R.: Quantifying ILP by means of graph theory. In: Proceedings of the 2nd International Conference on Performance Evaluation Methodologies and Tools, pp. 317–322. San Francisco, California (2007)
9. Fiedler, M.: Algebraic connectivity of graphs. Czechoslovakia Mathematical Journal **23**(2), 298–305 (1973)
10. Ha, S., Oh, H.: Decidable dataflow models for signal processing: synchronous dataflow and its extension. Handbook of Signal Processing Systems, 2nd edition., Springer pp. 1083–1110 (2013)
11. Horowitz, M., John, A., Kossentini, F., Hallapuro, A.: H.264/AVC baseline profile decoder complexity analysis. IEEE Transactions on Circuits and Systems for Video Technology **13**(7), 704–716 (2003)
12. Hu, A., Kung, S.Y.: Systolic Arrays. Handbook of Signal Processing Systems, 2nd edition., Springer pp. 1111–1144 (2013)
13. Huang, C.T., Tseng, P.C., Chen, L.G.: Analysis and VLSI architecture for 1-D and 2-D discrete wavelet transform. IEEE Trans. on Signal Processing. **53**(4), 1575–1586 (2005)
14. Janneck, J.W., Miller, D., Parlour, D.B.: Profiling dataflow programs. In: Proceedings of IEEE ICME 2008, pp. 1065–1068 (2008)
15. Jiang, W., Ortega, A.: Lifting factorization-based discrete wavelet transform architecture design. IEEE Trans. on Circuits and Systems for Video Technology **11**(5), 651–657 (2001)
16. Kung, S.Y.: VLSI Array Processor. Upper Saddle River, New Jersey: Prentice-Hall (1988)
17. Lee, G.G., Chen, C.F., Hsiao, C.J., Wu, J.C.: Bi-Directional Trajectory Tracking With Variable Block-Size Motion Estimation for Frame Rate Up-Convertor. IEEE Journal on Emerging and Selected Topics in Circuits and Systems **4**, 29–42 (2014)
18. Lee, G.G., Chen, Y.K., Mattavelli, M., Jang, E.S.: Algorithm/Architecture Co-Exploration of Visual Computing: Overview and Future Perspectives. IEEE Transactions on Circuits and Systems for Video Technology **19**(11), 1576–1587 (2009)
19. Lee, G.G., Lin, H.Y., Chen, C.F., Huang, T.Y.: Quantifying Intrinsic Parallelism Using Linear Algebra for Algorithm/Architecture Coexploration. IEEE Transactions on Parallel and Distributed Systems **23**, 944–957 (2012)
20. Lin, H.Y., Lee, G.G.: Quantifying Intrinsic parallelism via Eigen-decomposition of dataflow graphs for algorithm/architecture co-exploration. In: Proceedings of IEEE SiPS 2010 (2010)
21. Oppenheim, A.V., Schaefer, R.W.: Discrete-Time Signal Processing. Englewood Cliffs, NJ: Prentice-Hall (1989)
22. Parhi, K., Chen, Y.: Signal Flow Graphs and Data Flow Graphs. Handbook of Signal Processing Systems, 2nd edition., Springer pp. 1277–1302 (2013)

23. Parhi, K.K.: VLSI Digital Signal Processing Systems: Design and Implementation. New York: Wiley (1999)
24. Prihozhy, A., Mattavelli, M., Mlynek, D.: Evaluation of the parallelization potential for efficient multimedia implementations: dynamic evaluation of algorithm critical path. IEEE Transactions on Circuits and Systems for Video Technology **15**(5), 593–608 (2005)
25. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide, a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation. Seattle, Washington (2013)
26. Ravasi, M., Mattavelli, M.: High-level algorithmic complexity evaluation for system design. Journal of Systems Architecture **48/1315**, 403–427 (2003)
27. Ravasi, M., Mattavelli, M.: High-abstraction level complexity analysis and memory architecture simulations of multimedia algorithms. IEEE Transactions on Circuits and Systems for Video Technology **15**(5), 673–684 (2005)
28. Sutter, B.D., Praveen, P., Lambrechts, A.: Coarse-grain reconfigurable array architectures. Handbook of Signal Processing Systems, 2nd edition., Springer pp. 553–592 (2013)
29. Takala, J.: General purpose DSP processors. Handbook of Signal Processing Systems, 2nd edition., Springer pp. 779–802 (2013)
30. Yamauchi, H., et al.: Image processor capable of block-noise-free JPEG2000 compression with 30 frames/s for digital camera applications. In: Proc. IEEE Int. Solid-State Circuits Conf., pp. 46–47 (2003)

Architectures for Stereo Vision

Christian Banz, Nicolai Behmann, Holger Blume, and Peter Pirsch

Abstract Stereo vision is an elementary problem for many computer vision tasks. It has been widely studied under the following two aspects, increasing the quality of the results and accelerating the computational processes. This chapter provides theoretic background on stereo vision systems and discusses architectures and implementations for real-time applications. In particular, the computationally intensive part, the stereo matching, is discussed using one of the leading algorithms, the semi-global matching (SGM) as an example. For this algorithm two implementations are presented in detail on two of the most relevant platforms for real-time image processing today: Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs). Thus, the major differences in designing parallelization techniques for extremely different image processing platforms can be illustrated.

1 Introduction

The field of stereo vision is highly inspired by the capabilities of the human imaging system. It encompasses all aspects of computer vision processing data from stereo image pairs in one way or another. The goal is to estimate 3D information about the observed scene, which can be used for a number of applications such as e.g. distance measurement, 3D reconstruction, and arbitrary view interpolation. Crucial for stereo vision is the task of stereo matching which identifies the projection points of the same 3D real world point in both images of the stereo pair. The location difference (*the disparity)* in conjunction with a known stereo camera calibration allows to infer the depth information. Figure 1 gives an example.

The importance of stereo matching has been underlined by Szeliski and Scharstein stating that it is "one of the most widely studied and fundamental problems of computer vision" [93]. Active research in this field has resulted in a

C. Banz · N. Behmann · H. Blume (✉) · P. Pirsch
Institute of Microelectronic Systems, Leibniz University of Hannover, Hannover, Germany
e-mail: banz@ims.uni-hannover.de; behmann@ims.uni-hannover.de;
blume@ims.uni-hannover.de; pirsch@ims.uni-hannover.de

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_16

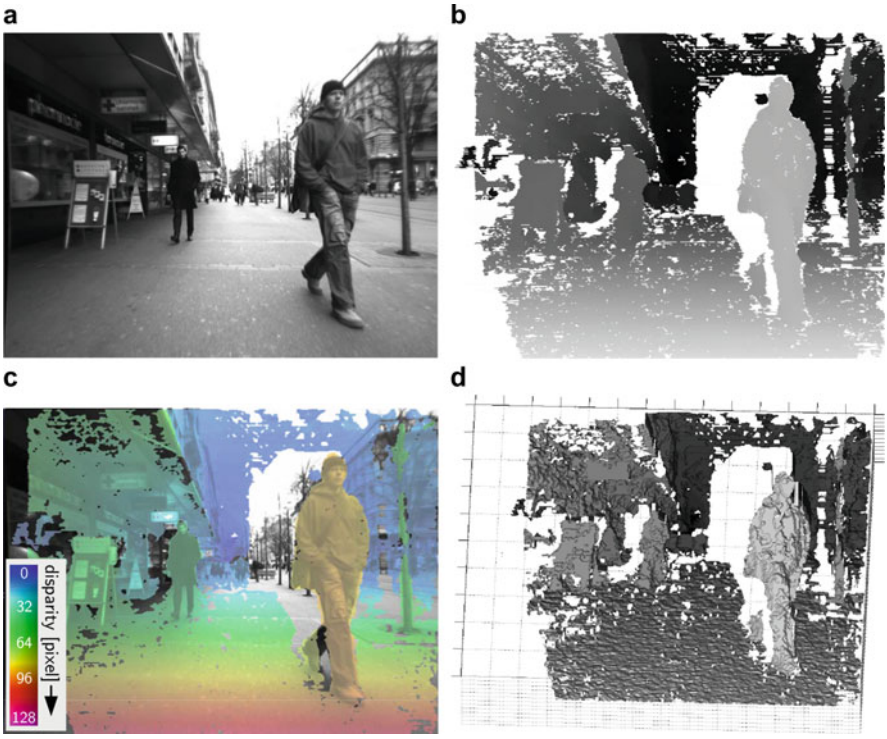


Fig. 1 Results for the stereo correspondence problem: (**a**) Left rectified input image (raw input images taken from [21]), (**b**) disparity map after left/right check where white denotes disparities marked as invalid, (**c**) false color representation of the disparity map, (**d**) untextured 3D view generated from the disparity map of (**b**)

wide range of disparity estimation algorithms using radically different approaches. A general taxonomy has been introduced [91] including a comprehensive survey, that resulted in the on-going online *Middlebury benchmark* [90], which includes ground truth disparities from a structured light system. Further surveys evaluated different algorithms and variations thereof [42, 96]. The *KITTI stereo benchmark* [27, 69, 70] focuses on automotive scenarios, generated from a front facing setup and semi automated ground truth disparities. Synthetically rendered stereo image sequences and reference disparity maps lower the effort of labeling, while abstracting environmental effects on real cameras [84, 103]. Major focus was the quality of the stereo matching in terms of accuracy, density of the disparity map, and robustness.

However, advances in robustness and accuracy were accompanied with significant increases in complexity and computational requirements making the use of specialized implementations for many of today's real-time applications an absolute necessity. Surveys on efficient implementations for selected types of algorithms have been conducted [31, 64, 74, 96] and many more specialized implementations

and architectures for individual algorithms and applications have been proposed. Considering all aspects (algorithmic performance, implementation performance, architectures) a huge design space is unfolded. For embedded systems the choice is invariably on low-power solutions, e.g. based on application-specific architectures implemented on FPGAs or ASICs. However, there has been a rise recently in the use of GPUs for high performance computing, offering a cost-efficient alternative for stationary systems where power consumption is not an issue.

This chapter addresses high performance disparity estimation considering both, algorithmic and implementation performance. The chapter is structured into an algorithmic and an architectural section; these being Sects. 2 and 3. An introduction to the fundamental principles of the stereo image matching (*epipolar geometry*) and a minimal practical stereo vision system is given in Sect. 2.1. The algorithmic and architecture sections both give a comprehensive overview of recent works. It is followed by a detailed discussion of the semi-global matching algorithm (SGM) [40] (Sect. 2.3) and two exemplary implementations on FPGA (Sect. 3.7) and GPU (Sect. 3.6), respectively.

2 Algorithms

A minimal system for disparity estimation from a real camera setup consists of two processing steps: The first step is usually camera lens undistortion and *rectification* of non-ideal stereo camera setup (Sect. 2.1) while the second step is the actual stereo matching (Sect. 2.2). All other image preprocessing steps (e.g. noise reduction, equalization) and disparity map post-processing steps (e.g. whole filling, interpolation of pixel with missing stereo information) are optional.

2.1 Epipolar Geometry and Rectification

The objective is to find corresponding pixels in the two images from a stereo camera setup. Due to the underlying epipolar geometry [37, 93] of a stereo camera setup, the search space for corresponding pixels is one-dimensional. As shown in Fig. 2a, for a given pixel in the base image all potential correspondences project onto the *epipolar line* (e_{bm}) in the match image and vice versa. Strictly speaking the possible projections are bound by the *epipole* and the viewing rays for a real-world point at infinity.

For efficient correspondence search implementations a preprocessing step, the *rectification*, is employed. Both images are warped such that epipolar lines in both images are parallel to the scanlines and are row-aligned, i.e. corresponding pixels are in the same horizontal line [37, 117]. Thus, efficient memory access patterns and parallelism over independent scanlines can be obtained. After rectification, the focal axis are parallel to each other and perpendicular to the line joining the two camera centers (*baseline*) and the disparity for points at infinity is 0.

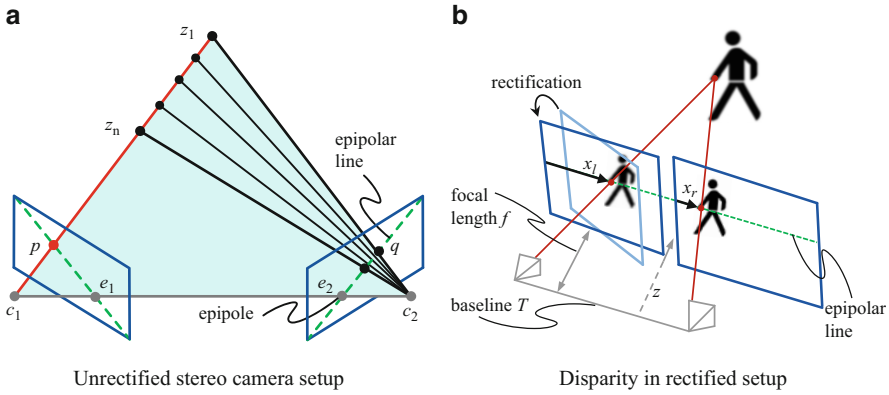


Fig. 2 Epipolar geometry: in (**a**) an unrectified setup and in (**b**) a rectified setup is shown. The rectification process in (**b**) to achieve row-aligned search space is illustrated only for the left projection plane

The rectified stereo setup is shown in Fig. 2b and the disparity is a purely horizontal offset $d = (x_l - x_r)$ with the unit [pixel]. With the rectified focal length f [pixel], the baseline T [m] of the camera pair, the distance z [m] between the baseline and the 3D point can be calculated as

$$z = \frac{fT}{(x_l - x_r)} = \frac{fT}{d}. \tag{1}$$

This is also referred to as *standard rectified geometry* [93]. Thus, extracting depth information from a stereo camera setup becomes the estimation of the *disparity map* $d(x, y)$.

In addition to a non-ideal camera setup, stereo vision systems have to handle camera-inflicted image distortions, of which the most common are radial lens distortion, sensor tilting and offset from the focal axis [12]. These must be compensated *before* rectification. However, when applying undistortion and rectification to a sequence of input images both steps can be combined. Reverse mapping assigns every pixel in the undistorted and rectified image a sub-pixel accurate origin in the input image. The rectified pixels are obtained using any desired pixel interpolation method. The bilinear interpolation for example, exhibits a reasonable trade-off between image quality and hardware implementation costs. Alternative interpolation methods are spline interpolation, which has higher silicon area requirements, and nearest-neighbor, which does not provide the required resolution for disparity estimation. Intermediate results from the processing steps are shown in Fig. 3.

The displacement vectors for undistortion and rectification are calculated using the intrinsic and extrinsic matrices, the tangential and the radial distortion parameters. These can be obtained through a separate camera calibration step (e.g. [118]) using a calibration pattern, such as a chessboard pattern employed in OpenCV [12].

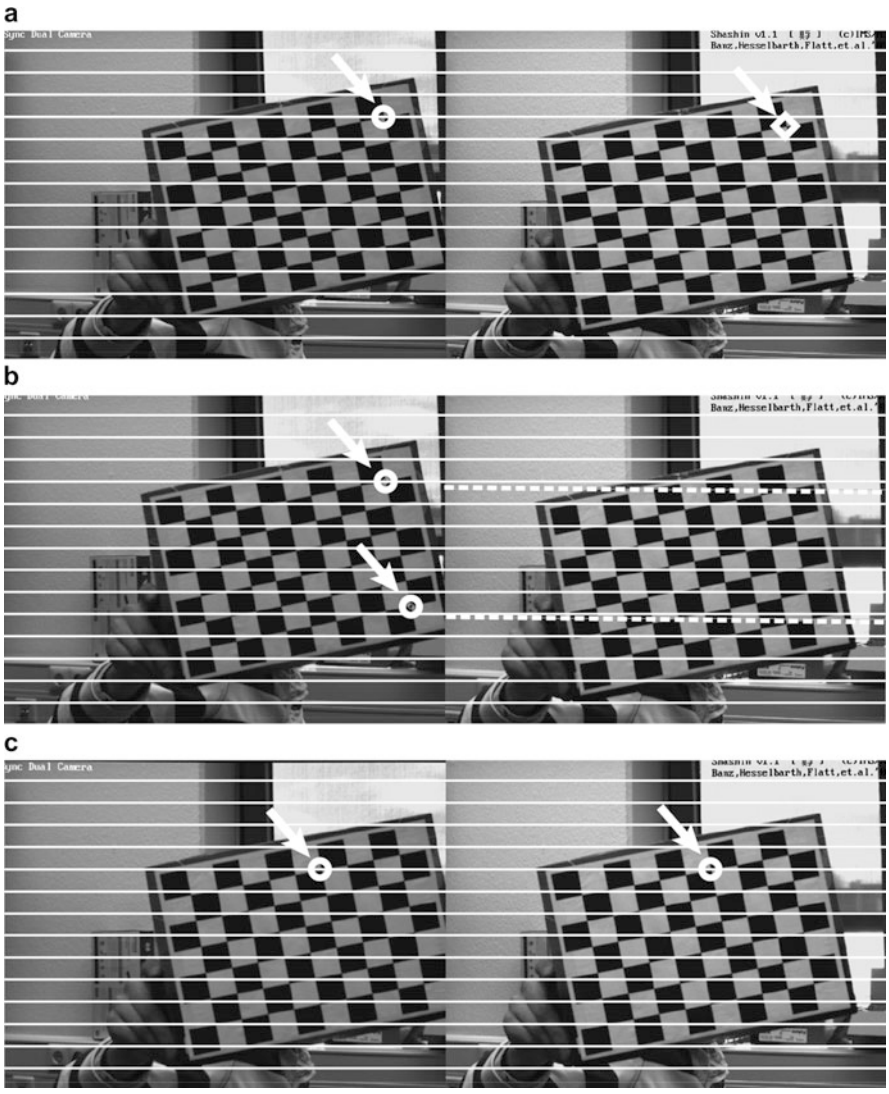


Fig. 3 Image results after undistortion and rectification: (**a**) input images showing that correspondences are not aligned (circle and square). (**b**) undistorted images showing the epipolar lines (dashed lines) for two exemplary points (circles). Here, the effect is minor but the epipolar lines are clearly not aligned to the scanlines (i.e. horizontal pixel rows, white). (**c**) final, undistorted, rectified images with row aligned epipolar lines

Alternatively, or additionally, camera self-calibration from scene structure can be employed for particular camera parameters. In e.g. cars, camera self-calibration or at least updating of the extrinsic parameters from scene structure is mandatory.

2.2 Stereo Correspondence

The origins of *classic* stereo correspondence were *sparse*, feature-based methods processing only a set of potentially highly discriminative image points. Today, most algorithms are *dense* methods, trying to infer a complete disparity map even for texture-less regions. Dense methods are typically classified into *local* and *global* approaches. However, for both classes of dense methods a common taxonomy and categorization has been introduced in [91]. Generally, a disparity estimation algorithm consists of the four processing steps:

1. matching cost computation,
2. cost (support) aggregation,
3. disparity computation and optimization, and
4. disparity refinement (or post processing).

The introduction of the census transform in 1994 [111] enables fast dense disparity estimation. Dynamic programming paradigms [9] and the usage of the graphcut algorithm [85] for stereo-based depth estimation further improved the accuracy, by optimizing individual of the before mentioned processing steps. Using path-aggregation for smoothing in the disparity error function, the semi-global matching algorithm by Hirschmüller marks a milestone in efficient and high-quality stereo disparity estimation [39]. In [116] a local shape-adaptive cost function is proposed to optimize the disparity map quality.

Using superpixels [108] for local texture-sensitive disparity smoothing, or prior knowledge [33], the quality of disparity estimation can be further improved. The rise of deep learning has, as in many other fields, opened a new approach to solving the stereo correspondence problem. In these approaches, some or all of the processing steps are computed using trained neural networks [54, 115].

The classic methods and the deep learning methods are discussed in Sects. 2.2.1 and 2.2.2, respectively. Figure 4 shows the timeline of research on stereo processing highlighting the introduction of major new concepts.

Fig. 4 Timeline of the stereo vision related algorithmic developments

2.2.1 Classical Disparity Estimation

It is of crucial importance to distinguish between the matching costs, which is the initial similarity measure between two pixels in the base and match image (or left and right image, respectively), and the aggregation method that uses these costs. The results of the matching cost computation are stored in the *disparity space image* $C(x, y, d)$. Cost aggregation of local (or area based) methods is performed on the information based in a local aggregation region (*support region*) from the matching costs $C(x, y, d)$. Global methods on the other hand perform one or more optimization steps on the matching costs often enforcing some kind of smoothness criterion. Depending on the algorithm, steps have varying importance and some might even be omitted. An example is given in Sect. 2.3.

For matching cost computation, a number of different window-based similarity measures can be employed. With rectified input images, the similarity of potentially corresponding pixels must be computed at location $\mathbf{p} = [x, y]^T$ in the left image and $\mathbf{q} = [x - d, y]^T$ in the right image. Initially often used and inspired by other areas of video processing are the sum of squared intensity differences (SSD), the sum of absolute differences (SAD), the normalized cross correlation (NCC) and their respective zero mean variations. More recently, measures specifically for stereo matching have been proposed. For example, rank and census transform [111] are non-parametric transforms, and are thus robust to a certain amount of intensity differences. A vast number of other measures based on gradients, phase correlations, ordinal measures, and dense feature descriptors exist. Entropy based measures (e.g. mutual information [40, 55]) have also been proposed. For those measures that compare absolute difference values, the approach of Birchfeld and Tomasi (BT) [10] can be used to include sampling insensitivity. For a more complete list of similarity measures refer to [93]. Detailed studies on the performance of the similarity measures in conjunction with different aggregation methods have been conducted [42, 91, 100].

Local methods focus on the cost aggregation step (step 2). A comprehensive comparison of aggregation methods can be found in [96] and of selected methods for GPU implementation in [31]. Support regions can be two- or three-dimensional windows from the disparity space with fixed or adaptive window sizes, shapes, anchor points or weights. Adaptation for example may be performed by a full search through multiple windows or from a number of cues, e.g. constant disparity constraints and color-based segmentation. A more complete list may also be found in [93]. After cost aggregation, the disparity computation (step 3) follows, of which the most basic form is selecting the disparity with minimal aggregated cost value for each pixel. Local methods are often well suited for hardware implementation due to the implicit parallelism and local data dependencies.

With global methods, the cost aggregation (step 2) is often omitted because the global smoothness constraints, which are enforced by the optimization process during the disparity computation (step 3), perform similar functions [93]. Global methods are often formulated within an energy-minimization framework:

$$E(D) = E_d(D) + \lambda E_s(D). \tag{2}$$

The objective is to find a solution d that minimizes the total energy E for a disparity map D, where E_d is the data term representing how well the solution fits to the input image and E_s represents the smoothness constraints made by the algorithm. These *regularization* or variational formulations are also employed in many other areas of image processing. In stereo processing, it is important to formulate $E_s(d)$ to allow for *discontinuity preservation* in the disparity map. Algorithms to find the solution to (2) include belief propagation, graph cuts, and total variation among others [93]. Unfortunately, the problem is NP-hard for many discontinuity preserving if E_s is formulated two-dimensionally [11]. Reducing E_s to one-dimension along the scanlines, allows for independent, parallel *scanline optimization* but suffers from streaking (inconsistency between scanlines). Other global methods are based on *dynamic programming*, which performs global optimization for independent scanlines. Dynamic programming also suffers from streaking, but several works have addressed this problem, e.g. [88].

For each approach several algorithms have been proposed and minute details influence the performance. As mentioned in Sect. 1, comparative studies have been performed (e.g. [31, 42, 91]) and popular benchmarks already exist [69, 70, 90, 103]. For a stereo vision system with high performance in terms of robustness, accuracy, and processing speed, several aspects have to be weighted against each other. While some local methods are more efficiently implementable, they can be challenged by areas with low or repetitive textures due to a high level of ambiguity [42]. Iterative, global minimization methods are often computationally intensive. However, Tombari et al. [96] express, that with sophisticated cost aggregation some local methods yield performance comparable with many global methods. The semi-global optimization strategy [40] is a solution resident in between by accumulating optimization results from multiple independent one-dimensional directions for each pixel. It produces very high quality results, although not the best in the Middlebury benchmark [90]. Furthermore, it is robust and it can be implemented efficiently for a global method. For robustness of the entire disparity estimation a suitable similarity measure must be chosen. Among the more robust measures in [41, 42] were census, rank, ordinal measures, and hierarchical mutual information.

Disparity refinement (step 4) often includes sub-pixel refinement, confidence or integrity checks, and interpolation measures. Since most stereo methods compute disparities at integer level, a *sub-pixel* refinement is necessary for many applications. An easy and computationally efficient way is to fit a curve through the discrete disparity space around the selected disparity. Interpolation functions are investigated in [35]. Several issues arising are discussed in [94]. An often computationally prohibitively expensive alternative is to start the computation with a disparity space already discretized to sub-pixel accuracy.

Several algorithms use the observation that depth discontinuities on object borders usually exhibit gray scale differences. This information is used by over-segmenting the grey scale image into superpixels. In [116], the superpixels are implicitly build in the cost aggregation step by adapting the size and shape of the aggregation window by analyzing the input image. Another segmentation approach is presented in [108], in which a sophisticated post-processing performs

Fig. 5 Oversegmentation of the camera image, representing similar textured regions within one superpixel (orange bounds)

a segmentation of the image jointly on input image and disparity information. Each segment is then fitted with planar surface in 3D space observing consistency constraints building a refined disparity map. This method is extended in [33] by fitting 3D shape information using deep learning in the disparity map optimizing textureless or reflective areas. Figure 5 illustrates this method and oversegmentation on the camera input image.

Foreground objects in the scene occlude different parts of the background when seen from the two camera perspectives. Consequently disparities cannot be computed for these occluded areas of the image due to missing stereo correspondences. This is visible in Fig. 1 by the halos around the foreground objects. It is often desirable to exclude these areas and areas with low confidence from the disparity map and optionally process them with sophisticated hole filling algorithms. Identification of these areas is performed with a *left/right check*, where the disparity maps for the left and right perspective are computed and only matching depth information from both perspectives to a 3D world point is allowed. With respect to the camera-to-camera projection in a rectified stereo pair the constraint for a valid disparity in the base image can be formulated as

$$D_{b,\text{check}}(x,y) = \begin{cases} D_b & \text{if} \quad |D_b(x,y) - D_m(e_{mb}(x, D_b(x,y)), y)| \leq \delta \\ invalid & \text{otherwise} \end{cases}, \tag{3}$$

where D_b and D_m are the disparity maps from the base and match perspective, respectively.

Further post-processing of the disparity map can be performed using basic median filtering to remove single outliers, peak removal and sophisticated whole filling algorithms, such as surface fitting. However, without a dense, highly accurate initial disparity map, post-processing will not provide reliable disparities.

2.2.2 Disparity Estimation Using Deep-Learning

Recently, deep learning techniques are used to train the networks for the disparity estimation. In [115], a siamese network architecture is used to compute the initial matching costs. The siamese network architecture uses shared weights across two sub-networks, one for the left and right image patch, followed by combination to calculate the similarity measurements, i.e. matching costs. The combination is either a fully connected layer for higher accuracy or a dot-product layer for computation speed. In [63], the concept of using a dot-product layer is further expanded upon. In both methods, cost aggregation and disparity selection are then performed with classic methods to compute the disparity map.

In contrast to these approaches, *end-to-end* learning approaches aim to learn the entire processing chain from input images to final disparity map generation. A end-to-end network consisting of a contractive part and an expanding part with long-range links between them is designed in [67]. The most recent end-to-end approach in [54] designs a network architecture guided by the classic steps of disparity estimation and their specific properties, currently giving best results in the KITTI stereo benchmark.

2.3 Algorithm Example: Semi-global Matching

As a specific example disparity estimation based on the highly relevant and top-performing combination of rank transform [111] and semi-global matching algorithm (SGM) [40] will be used to illustrate the matter of the previous sections. Simultaneously, SGM will be used as a case study for implementations on FPGA and GPU, resulting in a reduction of the execution time. This pushes the SGM algorithm in a quality-runtime design space exploration towards a pareto optimal point.

The matching costs $C(x, y, d)$ (step 1) are calculated from the rank transform of the base and match image R_b and R_m with absolute difference comparison:

$$C(\mathbf{p}, d) = \left| R_b\left(p_x, p_y\right) - R_m\left(p_x - d, p_y\right) \right|. \tag{4}$$

It is $\mathbf{p} = [p_x, p_y]^T$ the pixel location in the left image. The rank transform is defined as the number of pixels $\mathbf{p}'$ in a square $M \times M$ neighborhood $A(\mathbf{p})$ of the center pixel $\mathbf{p}$ with a luminous intensity I less than $I(\mathbf{p})$

$$R(\mathbf{p}) = \left\| \left\{ \mathbf{p}' \in A(\mathbf{p}) \mid I(\mathbf{p}') < I(\mathbf{p}) \right\} \right\|. \tag{5}$$

These initial pixel-wise calculated matching costs (i.e. locally calculated) yield non-unique or wrong correspondences due to low texture and ambiguity. Therefore, semi-global matching introduces global consistency constraints by aggregating matching costs along several independent, one-dimensional *paths* from different cardinal directions as shown in Fig. 6. A path $\mathbf{r}$ is formulated recursively by the

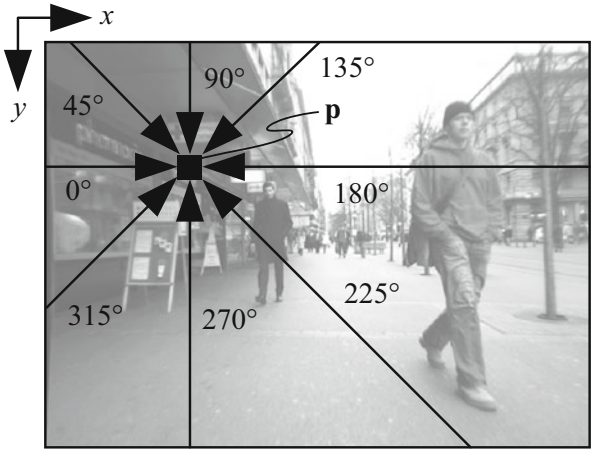


Fig. 6 The path cost aggregation is performed from eight cardinal directions to every pixel

definition of the path costs $L_r(\mathbf{p}, d)$, defined as:

$$\begin{aligned} L_r(\mathbf{p}, d) = C(\mathbf{p}, d) + \min [& L_r(\mathbf{p} - \mathbf{r}, d) \ , \\ & L_r(\mathbf{p} - \mathbf{r}, d - 1) + P_1, \\ & L_r(\mathbf{p} - \mathbf{r}, d + 1) + P_1, \\ & \min_i L_r(\mathbf{p} - \mathbf{r}, i) + P_2] - \\ & \min_l L_r(\mathbf{p} - \mathbf{r}, l) \end{aligned} \tag{6}$$

The first term, $C(\mathbf{p}, d)$, describes the initial matching costs. The second term adds the minimal path costs of the previous pixel $\mathbf{p}-\mathbf{r}$ including a penalty P_1 for disparity changes and P_2 for disparity discontinuities, respectively. Discrimination of small changes $|\Delta d| = 1$ pixel *[px]* and discontinuities $|\Delta d| > 1$ px allows for slanted and curved surfaces on the one hand and preserves disparity discontinuities on the other. The last term prevents constantly increasing path costs. For a detailed discussion refer to [40]. P_1 is an empirically determined constant. P_2 can also be an empirically determined constant or can be adapted to the image content. The selection of these penalty functions is investigated in [5] in detail.

Path costs are calculated from several cardinal directions to each pixel, as shown in Fig. 6, and are summed. The aggregated sum costs S are the sum of the path costs

$$S(\mathbf{p}, d) = \sum_{\mathbf{r}} L_r(\mathbf{p}, d). \tag{7}$$

By (6) and (7) SGM aims to approximate the following global energy minimization problem:

$$E(D) = \underbrace{\sum_{\mathbf{p}} C(\mathbf{p}, d)}_{E_d(D)} + \underbrace{\sum_{\mathbf{p}} \left(\sum_{\mathbf{p}' \in A} P_1 \mathrm{T}\left[\left| D_\mathrm{p} - D_{\mathrm{p}'} \right| = 1 \right] + \sum_{\mathbf{p}' \in A} P_2 \mathrm{T}\left[\left| D_\mathrm{p} - D_{\mathrm{p}'} \right| > 1 \right] \right)}_{E_s(D)}, \tag{8}$$

where E_s contains the 2D smoothness constraints on the disparity map. For a derivation of (8) see [40]. The resulting method of approximation resembles a scanline optimization approach but with excellent regard to interscanline consistencies.

Final disparity selection (step 3) is performed by a *winner-takes-all* (WTA) approach. The disparity map $D_b(p_x, p_y)$ from the perspective of the base camera is calculated by selecting the disparity with the minimal aggregated costs

$$\min_d S(p_x, p_y, d) \tag{9}$$

for each pixel. For calculating the disparity map from the perspective of the match camera $D_m(q_x, q_y)$, the minimal aggregated costs along the corresponding epipolar lines are selected:

$$\min_d S\left(q_x + d, q_y, d\right). \tag{10}$$

Alternatively, SGM can be applied again, but with the other image as the base image.

The effect of the path costs aggregation and the disparity selection is illustrated in Fig. 7. The initial matching costs $C(\mathbf{p}, d)$ (dashed line) exhibit a high level of ambiguity. Seven of the eight aggregated paths costs $L_r(\mathbf{p}, d)$ already show distinct minima. The summed path costs $S(\mathbf{p}, d)$ (thick black line) clearly identify the minimum at a disparity level of 32 resolving all ambiguities. However, the cost difference for the positions 32 and 33 is minimal indicating that the correct position is located at a subpixel precision.

Finally, left/right check according to (3) and post processing can be applied, e.g. a median filter in its most basic form. An overview of the processing steps is given in Fig. 8.

3 Architectures

The variety of architectures and implementations to compute the stereo correspondence easily rivals the variety of the underlying stereo matching algorithms. Today very efficient implementations for local and global stereo methods are available on FPGAs, ASICs, GPUs, and DSPs. For real-time image throughput, local methods

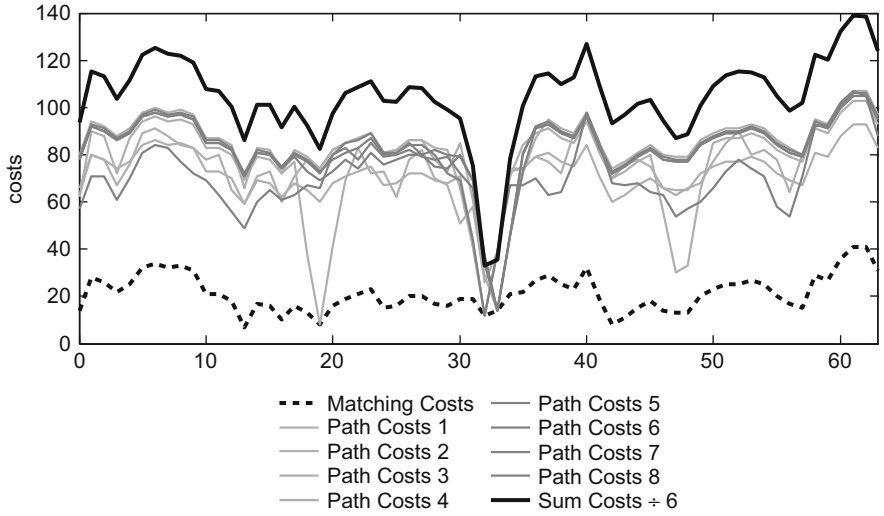


Fig. 7 Effect of path cost aggregation: matching costs, aggregated path costs, and sum costs (scaled by factor 1/6 for better presentation) for the pixel $\mathbf{p} = [183; 278]$ of the Teddy test image [92] calculated with SGM

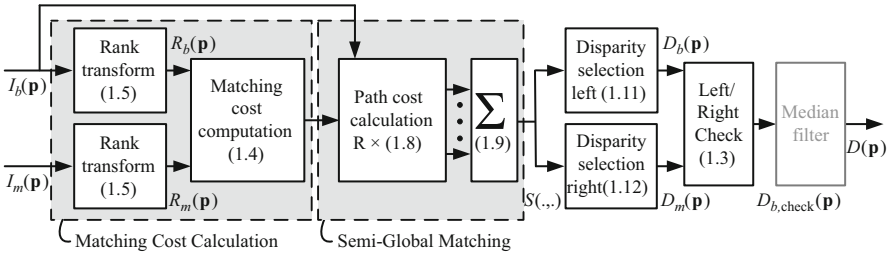


Fig. 8 Processing steps for disparity estimation using rank transform, semi-global matching, and optional median filter. Numbers in parenthesis refer to the respective equations and R denotes the number of paths

have been and continue to be favored by many researchers because of their efficient implementation possibilities. However, with advances in computational power, many global methods are also implementable in real-time.

Early work includes a complete stereo vision system from 1996 featuring rectification and stereo matching with an SSD variant on a custom hardware board consisting of off-the-shelf components and a DSP array [52]. At 30 fps 200×200 images with 5 −bit depth resolution could be computed. Other noteworthy early implementations have been presented in 1993 using a DSP array [22] and in 1997 using a single DSP [56]. In [56], an early overview of implementations is also provided. An FPGA array was used in 1997 to implement a census transform stereo matching method [106]. All of these implementations directly compute the

disparity information from the matching costs without cost aggregation. Early work includes a complete stereo vision system from 1996 featuring rectification and stereo matching with an SSD variant on a custom hardware board consisting of off-the-shelf components and an DSP array [52]. At 30 fps 200×200 images with 5−bit depth resolution could be computed. Other noteworthy early implementations have been presented in 1993 using a DSP array [22] and in 1997 using a single DSP [56]. In [56], also an early overview of implementations is provided. An FPGA array was used in 1997 to implement a census transform stereo matching method [106]. All of these implementations directly compute the disparity information from the matching costs without cost aggregation.

The following three subsections aim to give an overview of the research conducted on architectures and implementations for disparity estimation. Each subsection focuses on one specific hardware platform. Some key throughput values will be highlighted but without indication of algorithmic performance. A fair comparison must take into account architecture specific features, scalability, algorithmic performance under challenging imaging conditions (which are not present in the standard Middlebury data set), termination criterions on data dependent algorithms (e.g. belief propagation), and varying post processing steps. Thus, a comparison is an extremely complex task and beyond the scope of this chapter. The interested reader may consult the references themselves or one of the comparison studies.

3.1 GPU-Based Implementations

Commodity graphics processing hardware, nowadays superseded by *general-purpose graphics processing unit*s (GP-GPUs), have been used since the beginning to outsource, firstly, part of the computation and then the entire stereo matching. For the calculation of disparity maps with an image size of 200×200 pixels and 50 disparity levels (abbreviated 200×200×50), 106 ms were achieved using a variable window SSD method on a NVIDIA GeForce4 in 2003 [86]. Early implementations for scene reconstruction are [87] on a Nvidia GeForce4 and [112] on a ATI Radeon 9700Pro from 2002 and 2003, respectively.

For belief propagation (BP) an efficient technique has been proposed in [23] and implemented for CPUs. It has been extended with occlusion handling and adapted for GPU implementation [13]. The same technique is used in recent implementations [46] and [107] reaching 2.75 s for a 640×480×33 image on Nvidia GeForce GTX 280 and 93.98 s on an Intel Core 2 Duo (2.13 GHz). A fast converging hierarchical belief propagation is proposed and implemented in [109] reaching 16 fps for 320×240×16 images. New message passing schemes for BP have been applied in [59] for a GPU and VLSI implementation.

A dynamic programming solution with extensive use of MMX instructions on the CPU using color based cost aggregation has been presented in [24]. In 2005, the dynamic programming optimization step was still slower on the GPU than on the CPU [32] due to the limitations of the general-purpose computation capabilities

of the GPU, e.g. branching. Consequently, mixed CPU/GPU implementations performing cost aggregation on the GPU and dynamic programming optimization on the CPU have been presented [32, 60]. Scanline optimization in [113] also shows mixed performance results when comparing GPU versus CPU implementations. Recently, [51] presented a multi-resolution symmetric dynamic programming variant on a GTX 295 reaching 14 fps for $2048\times2048\times256$ images. A total variation algorithm with GPU implementation has been presented requiring between 15–60 s per image [81].

Variants of local methods examining the different techniques of adaptive weights or adaptive support regions have received much attention. Recent local approaches are census based with basic box filter cost aggregation [102] and a local truncated laplacian kernel approximation with adaptive cost aggregation [47]. Locally adaptive support regions have been used and speeded up with bitwise voting in [53]. Further work on local variants with adaptive cost aggregation methods includes [48, 71], and [43]. Instead of adaptive support regions on the input images [66] uses edge-preserving filtering on the matching costs. A comparison of 6 local methods in terms of algorithmic and computational performance on GPUs has been conducted [31]. A plane sweep algorithm with local depth connectivity in order to retain depth discontinuities has been examined in [17].

For SGM various implementations have been presented on a GeForce 8800 Ultra [20] (0.0057 fps at $640\times480\times128$), a Quadro FX5600 [29], a GTX 280 without [34] and with increased depth accuracy [75], and on a Tesla C2050 [4], which is the highest performing implementation with 63 fps for $640\times480\times128$ images. This allows a very interesting retrospective on the evolution of GPUs. Especially some of the new features of Nvidia's compute capability 2.0 graphics cards allow radically different parallelization schemes, which was exploited in [4]. We will have a detailed look at this implementation in Sect. 3.6. Furthermore, a combination of adaptive support regions with a reduced version of SGM is proposed in [68] reaching 10 fps for $450\times375\times64$ images.

3.2 Dedicated Architectures (FPGA and VLSI)

For dedicated architectures targeting FPGAs or ASICs, local methods are often favored because of their potential for very small designs. This goes as far as to omit the cost aggregation altogether despite the drawbacks in accuracy and robustness. Nevertheless, new cost aggregation concepts have also been investigated and incorporated in hardware. In the following implementations without cost aggregation are indicated with "w/o CA".

Some examples of early architectures using SAD based matching w/o CA are [2, 58, 72]. An SAD based stereo vision system with three cameras has been presented in [110]. Depending on the emphasis of the referenced work, the results vary in throughput and resolution up to $640\times480\times64$ and 31 fps. The so-called Tyzx ASIC for color-image census-based stereo-matching (w/o CA) achieves 200 fps for $512\times$

480 images and 52 disparity levels [104]. It forms the basis of an extended stereo vision system in [105].

Also for recent implementations local methods with and without cost aggregation are still popular. This includes [49] where a census transform (w/o CA) is employed as the basis of an entire stereo vision system on an FPGA. Another complete system based on SAD (w/o CA) is presented in [101]. Census with aggregation cues from the original and gradient images is investigated in [1]. Color SAD with a fuzzy logic disparity selection has been proposed and implemented on an FPGA [28]. Methods and architectures using adaptive support weights have been proposed in [15] employing a census variant and in [99] employing an absolute differences variant. In order to reduce the amount of data to be processed, [98] works on Sobel filtered images, which goes in the direction of sparse matching.

In [65], the architecture of [18], which is based on a local, phase-based method, is extended to large disparity ranges without significant additional hardware cost by adapting an offset of the smaller disparity search window across multiple frames. After large disparity changes, a latency of several frames occurs before correct disparity information can be regained. A bio-inspired method based on Gabor filters is introduced in [19].

Among the implementations of dynamic programming approaches a trellis-based implementation, using a single interline consistency constraint has been investigated [76]. A dynamic programming approach based on a maximum-likelihood method is implemented in [88] achieving 64 fps at 640×480 px with 128 disparity levels. And a symmetric dynamic programming variant, similar to the GPU implementation of [51], has been implemented on an FPGA [73].

An FPGA architecture for memory efficient belief propagation for stereo matching has been proposed in [79]. New concepts and architectures for the message passing in BP are proposed [97].

For semi-global matching two architectures have been proposed. The implementation of [26] utilizes a SGM variant with depth adaptive sub-sampling. It achieves 27 fps at 320×200 px and 64 px disparity range. A parameterizable parallelization scheme for SGM and a corresponding FPGA architecture have been proposed in [7] and [8]. It achieves, depending on the degree of parallelism, up to 176 fps for VGA images with 128 disparity levels and 4 SGM paths. This architecture will be studied in more detail in Sect. 3.7.

3.3 Other Architectures

The use of programmable architectures besides GPUs has also been investigated in some depth. Mühlmann et al. [74] investigated memory layout schemes for the disparity space and implementations schemes including MMX optimizations for SAD-based matching without cost aggregation (w/o CA).

A number of publications specifically target programmable embedded solutions: An SSD with multiple window selection has been implemented on the ClearSpeed

CSX700 architecture (250 MHz, 9 W) which provides massively parallel SIMD in multiple parallel processing elements [44]. The same algorithm has been implemented [89] on the Tilera TILEPro64, which is a MIMD architecture with 64 integer processing cores organized in a two dimensional mesh network running at MHz. A SAD w/o CA is also investigated on the Tilera TilePro 64 and on many-core CPUs [83]. SAD (w/o CA) for a VLIW processor (Texas Instruments TMS320C6414T, 1.0 GHz) has been shown in [14].

Application-specific processors (ASIP) have been investigated in two cases: For semi-global matching an instruction set extension for the Tensilica LX2 DSP template has been proposed [6] reporting 20 fps for $640\times480\times64$ images with reduced number of paths when run at 373 MHz, which is possible with the targeted TSMC 90 nm process. Similarly for SGM, architecture optimizations for a VLIW processor template, the MOAI, have been investigated in [77] reaching 30 fps when running at 400 MHz.

Apart from the original CPU implementation of SGM running at 1.3 s for $450\times375\times64$ images [39], a variant with depth adaptive sub-sampling has been proposed running at 14 fps for 320×160 images [25].

The cell broadband engine has been utilized for belief propagation and dynamic programming, both taking few seconds to process an image pair [62]. An SAD (w/o CA) implementation on the cell achieves 30 fps for VGA images with 48 disparity levels.

3.4 Comparison Studies

In addition to the algorithmic studies mentioned earlier, studies also taking into account the computational performance have been conducted. An evaluation of cost aggregation for local methods with focus on algorithmic performance and run-time on CPU can be found in [96]. Selected algorithms (various SAD variants, belief propagation, and dynamic programing) have been compared on a CPU in [64]. An evaluation of local algorithms on the GPU has been conducted in [31] and [61].

An implementation of belief propagation on GPU and for VLSI has been compared in [59]. Symmetric dynamic programing on GPU and FPGA has been compared in [50]. Comparison of a census based approach (w/o CA) on a DSP (TI C6416), a GPU (GeForce 9800 GT), and a CPU (Intel Core2Quad) has been conducted in [45]. In [83] SAD (w/o CA) has been studied on a GPU, two multi-core CPUs and the MIMD Tile architecture. Furthermore, in many of the references in the previous sections the GPU or FPGA implementation is compared to a regular CPU implementation.

3.5 Current Trends

When targeting real-world applications, the everlasting question is how to improve algorithmic performance while reducing computational requirements. This has already been addressed in many of the references above. A recent research direction is to integrate the computation of various information retrieval image processing tasks (e.g. disparity estimation with optical flow). In [30], an algorithm for joint computation of disparity estimation and optical flow is proposed and implemented on the GPU. A holistic architecture for phase based disparity estimation, optical flow, and more is presented in [95] and implemented on an FPGA. An holistic architecture for disparity estimation and motion estimation based on SAD is presented in [114].

3.6 Implementation Example: Semi-global Matching on the GPU

An example implementation of the semi-global matching algorithm for GPUs will be given based on the works in [4]. Since GPUs are becoming more and more common, an introduction of the architecture and the terminology will be skipped. Please refer to the Nvidia manuals and [38] for a detailed background on GPU architecture or directly to [4] for a short sketch. The evaluation platform in the following is a Nvidia Tesla C2050 with compute capability 2.0 providing 3 GB DDRRAM global memory with a maximum theoretical bandwidth of 144 GB/s.

3.6.1 Parallelization Principles

Banz et al. [4] formulate the following performance limiting factors for a kernel:

- **Effective memory bandwidth usage** for the payload data which is for example reduced by nonaligned, overhead-producing memory access
- **Instruction throughput** defined as the number of instructions performing arithmetics for the core computation and other non-ancillary instructions per unit of time
- **Latency of the memory interface** occurring e.g. when accessing scattered memory locations even if aligned and coalesced, warp-wise access is performed
- **Latency of the arithmetic pipeline** of the ALUs inside the GPU cores if arithmetic instructions depend on each other and can only be executed with the result from the previous instruction

Accordingly, kernels can be memory bound, compute bound, or latency bound. Kernels that are not limited by any of the three bounds are ill-adapted for GPU implementation and can be classified as bound by their parallelization scheme.

An efficient parallelization scheme guarantees inherently *aligned* and *coalesced* data access schemes without instruction overhead. Coalesced memory access is the simultaneous memory access to consecutive memory locations of all threads of a warp. It further includes a combination of parallel and sequential processing with independent arithmetic computation steps. An inner (sequential) loop in the otherwise parallel threads working on a set of data that is kept in shared memory or register facilitates data reuse, increases the instruction ratio, and keeps the pipeline filled. Further, coherent access schemes are ensured for the memory interface if results are written out with each loop iteration. Apart from an inner loop, executing several warps per streaming multiprocessor increases pipeline utilization.

3.6.2 Rank Transform and Median Filter Kernel

The rank transform and median filter are both non-linear, non-separable 2D image transforms. To generate the result of one output pixel, the data of a local $N \times N$-neighborhood from the input image is required.

The kernel for rank transform and median filter are based on the same principle which is based on the implementation of a separable convolution in [82]. It pre-fetches the data of a two-dimensional spatial locality from global memory into shared memory. Thus, data reuse is maximized because all filter kernels that fully reside in this spatial locality can be processed by a block of threads without additional global memory access. An aligned group of pixels is processed by a two-dimensional block of threads first loading the neighboring center pixels of all kernels. Left and right pixels outside the center area are always loaded with the warp width. Even though this causes minimal data to be loaded which is not used by the current block, it ensures inherent coalesced memory access without instruction overhead or warp divergence. An inner loop allows the processing of several pixels per thread (n_{ppt}) with a stride of the warp width. Adjusting n_{ppt} and the *launch configuration*, i.e. the number of threads per block in x-dimension (*tdx*) and y-dimension (*tdy*), allows navigation between the different optimization principles. Figure 9 shows the data layout and thread access scheme.

The median filter is always compute bound and performs best with $tdx \times tdy = 32 \times 4$ threads and $n_{\text{ppt}} = 4$. The results of the parameter study for $tdx = 32$ are shown in Fig. 10. Configurations with $n_{\text{ppt}} = 8$ perform slightly worse although redundant memory access is further reduced because of inefficient pipeline utilization. Processing times for a 3×3 median filter (i.e. kernel radius $K' = 1$) are given in Fig. 11 resulting in 0.64 ms for the new shared memory based kernel. For a texture-memory based kernel, which is the most often suggested way of implementing a 2D non-separable filter, processing time is 2.77 ms. In comparison, this yields a speed-up of 4.3 when processing a 1280×960 image.

For a 9×9 rank transform (i.e. $K' = 4$) experiments show that a block size of $tdx \times tdy = 32 \times 4$ with $n_{\text{ppt}} = 4$ yields the best performance. A speed up of 4.0 is obtained switching from the texture-based kernel (3.13 ms) to the shared memory kernel (0.78 ms) for 1280×960 images.

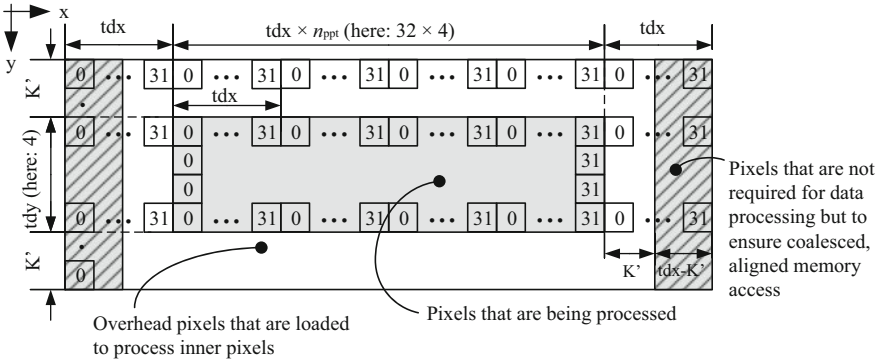


Fig. 9 Data fetching and accessing scheme for the 2D filter kernels processing $tdx \cdot n_{ppt} \times tdy$ kernel windows with a radius of K' where n_{ppt} is the number of pixels processed per thread and the launch configuration, which determines how threads are grouped and executed on the streaming multiprocessors, is $tdx \times tdy$. Each square represents a pixel and the number inside is the x-dimension thread ID which fetches the pixel from global memory

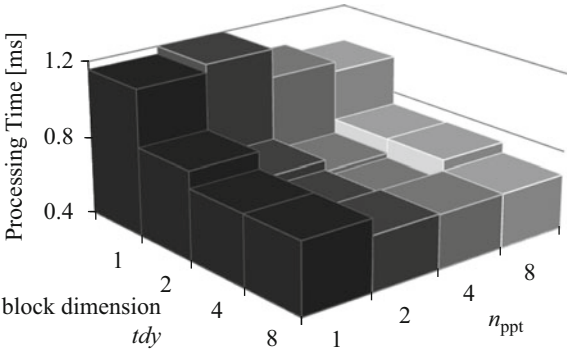


Fig. 10 Performance of the 3×3 median filter: on 1280×960 images as the parallelization configuration changes. Block width is fixed to $tdx = 32$. The best performance is achieved with $tdx \times tdy = 32 \times 4$ and $n_{ppt} = 4$

3.6.3 SGM Kernel

For every pixel location **p**, calculation of the matching cost $C(\mathbf{p}, d)$ according to Eq. (4) results in a vector with one entry for each disparity level d. Thus, the spatial directions (x and y) and the disparity range span the three-dimensional disparity space. The matching cost (MC) calculation for every point in this space can be performed independently allowing for parallelization in all three dimensions.

A straightforward parallelization is to assign each thread with the calculation of one entry in the 3D cost space of $C(\mathbf{p}, d)$. This kernel (`mc_unaligned`) reaches 16.3 ms and 48.6 GB/s which is far from the bandwidth limit due to inefficient, often misaligned memory access, lack of data reuse, and little latency hiding possibilities. This kernel is latency bound which can only be eliminated by a new parallelization scheme.

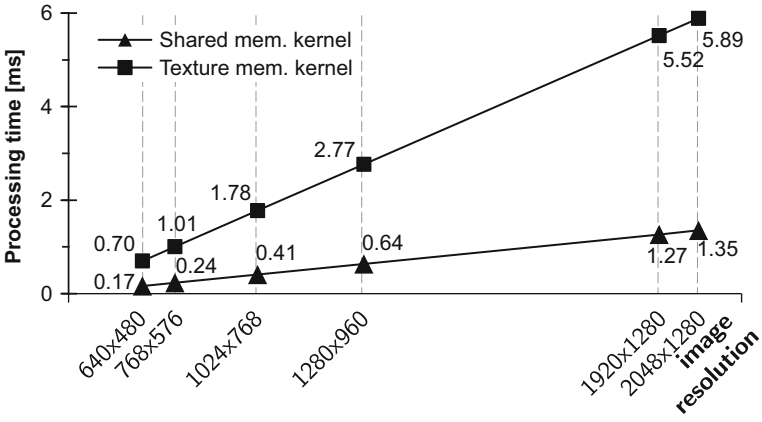


Fig. 11 Performance of the 3×3 median filter: comparison of the texture memory kernel and the proposed shared memory kernel on a Tesla C2050 GPU for the best-performing parallelization configuration

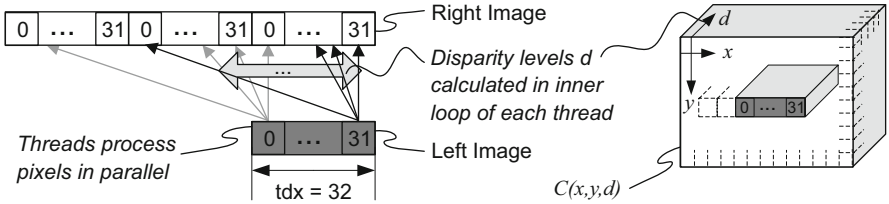


Fig. 12 Memory access scheme for calculating the matching costs for *tdx* pixels in parallel in a *tdx*-thread wide warp and $tdy = 1$. The location of the results in the 3D matching cost space is shown. Again the numbers in the squares represent the thread ID that fetches the according pixels from global memory

The new kernel (`mc_proposed`) processes all disparity levels of a group of *tdx* neighboring pixels synchronously in *tdx* threads. The disparity dimension itself is further separated into *tdy* groups each processing d_{range}/tdy disparity levels with an inner loop in the kernel. By adjusting *tdy* thread parallelism is substituted with inner loop complexity. Pixels from the base image are read aligned and coalesced over the *tdx* threads. The required pixels from the right image are loaded in groups of *tdx* aligned, coalesced pixels into the shared memory where they can be accessed and reused by all threads. The parallelization scheme is shown in Fig. 12. Furthermore, only 8−bit precision is required. Since performing arithmetic in non-native GPU data types (i.e. other than 32-bit integer and float) is slow, input images and computation are based on 32-bit integer and type conversion to `uchar` is performed just before writing out the result. Consequently, type conversion to `uchar` is performed just before writing out the result. Choosing *tdx* as a multiple of the warp size (i.e. 32) results in always aligned memory access. This kernel adheres the optimization approach of Sect. 3.6.1 by providing inherently aligned

Table 1 Performance results of the optimized kernels (MC: matching costs, PC: path costs, WTA: winner-takes-all) with optimal launch configuration for computing the semi-global matching algorithm for images with 1280×960 pixels and 128 disparity levels

Kernel	Time (ms)	Bandwidth (GB/s)	Bound by
MC unaligned	16.32	48.6	Parallelization scheme
MC proposed (`uchar4`)	1.80	107.3	Pipeline latency
MC+PC 8 path dir. (sequential)	75.68	20.9	Pipeline latency
MC+PC 8 path dir. (concurrent)	39.81	39.7	Pipeline latency
Sum, WTA left disparity map	15.09	117.4	Memory bandwidth

memory access, high data reuse, and efficient use of the arithmetic pipeline. With an obtained performance of 1.8 ms and 111.2 GB/s this is a speed-up of factor 9.2. A performance summary is given in Table 1.

The path costs (*PC*) calculation according to Eq. (6) is performed by individually traversing along each of the eight path directions updating the matching cost values and resulting in a new 3D cost space for each path direction. PC calculation must be done sequentially along the respective path direction (e.g. from left to right) because the previous pixel's path costs must be known. The parallel minimum search over the disparity levels has been implemented similarly to the parallel reduction scheme from [36]. Although the MCs are common to all PC directions and it seems obvious to separate MC and PC calculations, it is faster to integrate MC and PC calculations and recalculate the MCs on-the-fly for each PC direction. This drastically reduces pressure on the performance-limiting memory bandwidth since the MC data is never transferred via the external memory but can be kept locally in the shared memory. All eight path directions are executed concurrently using the CUDA concurrent kernel execution.

Due to the coalesced memory access necessity, only a group of horizontally neighboring pixels can be efficiently accessed in memory. The path costs kernels must be modified according to their path direction in order to maintain efficient memory access. For each diagonal path direction, processing is separated into rectangular tiles. Within each tile the processing direction is along the image columns, i.e. misaligned to the path direction, but ensuring aligned memory access. Tiles not sharing data dependencies can be processed in parallel as independent thread blocks. This is similar to the intrablock encoding scheme for video streams proposed in [57]. An example of the parallel processing order is shown in Fig. 13. Since block synchronization does not exist on GPUs, correct execution order is established by sequentially launching a kernel for each diagonal tile front (identical letters in Fig. 13) causing some minor overhead in time. The practical alternative of keeping the processing implementation unchanged but rearranging the data in the memory creates an inherently contradictory situation: if the GPU is used to rearrange the data, the re-sorting causes additional memory access with is not even coalesced.

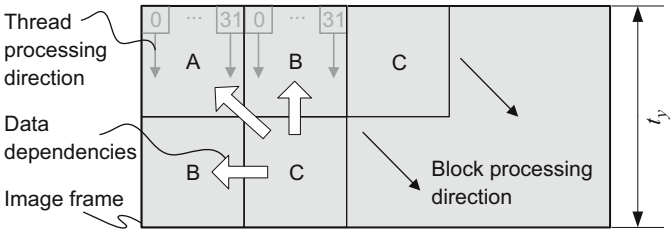


Fig. 13 Image tiling for the 45° path and $t_y = 2$ allowing divergent processing direction and path direction while tiles with the same letter can be processed in parallel. The processing direction ensures coalesced and aligned memory access

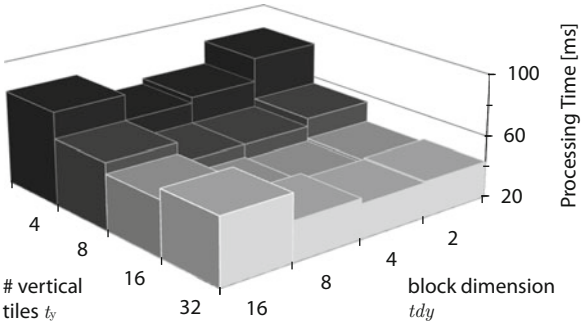


Fig. 14 Impact of the parallelization configuration on the performance of the concurrent path cost calculation for 8 paths of the SGM for 1280×960 images and 128 disparity levels. Block width and tile width are both fixed to $tdx = 32$. Best performance is achieved with $tdx \times tdy = 32 \times 4$ (i.e. each inner loop processes 32 disparity levels) and $t_y = 16$

Again, parameters adjustment allows navigation between the performance optimization principles. The first parameter (tdy) trades thread parallelism against sequential computation in the inner loop for all kernels. The second parameter (t_y) trades the number of parallelly processable blocks versus launch overhead and memory overhead for the four diagonal paths. Figure 14 shows the result of the parameter study. Choosing $tdy = 4$ and $t_y = 16$ results in best performance (39.8 ms and 39.7 GB/s) for a 1280×960 image. If the concurrent kernel execution is not used, performance is approximately halved (75.7 ms and 20.9 GB/s). Both kernel sets, concurrent and sequential, are latency bound.

Summation of the eight path cost spaces (7) and winner-takes-all disparity selection (9) can be performed independently for each pixel allowing for the same parallelization scheme as for the MC calculation. This kernel (`sum_wta`) requires 15.1 ms and is memory bound with 117.4 GB/s.

Table 2 Performance results for the entire disparity estimation algorithm using rank transform, semi-global matching and median filtering on a Nvidia Tesla C2050 GPU

	d_{range}		
Image size	64	128	256
640×480	9.7 ms	16.0 ms	29.0 ms
1024×768	21.5 ms	35.9 ms	67.1 ms
1280×960	32.9 ms	56.2 ms	105.7 ms

Results are k-mean values over multiple runs and images

3.6.4 Performance

The processing time for the complete disparity estimation including rank transform, semi-global matching for eight paths, disparity map generation (without left/right check (3)), and median filtering on a Tesla C2050 Fermi architecture GPU is summarized in Table 2. Overall, a 1280×960 image with 128 disparity levels requires 56.2 ms. The processing times do not include data transfer between host and GPU because it can be effectively hidden using concurrent data transfer when processing image streams. When processing 1280×960 image sets ca. 5 ms additional transfer time is required.

3.7 Implementation Example: VLSI Architecture for Semi-global Matching

In this section a parallelization scheme and corresponding VLSI architecture for semi-global matching will be discussed. It is based on the works of [7] and [8].

3.7.1 Parallelization

A crucial point for VLSI-implementation is the mapping of the algorithm into a *parallel-processable* and *stream-based* flow that only requires a single-pass across the input images. Further important aspects are the *regularity* and the *locality* of the architecture that implements this flow [80]. Challenges are imposed by the semi-global matching due to the recursively defined paths and their orientations within the images (see Sect. 2.3), which are not aligned to a stream-based flow.

First, the two-dimensional parallelization concept that enables stream-based processing will be introduced. Afterwards, an extension of the concept into the third dimension is presented, which significantly increases processing speed and throughput. The two-dimensional parallelization concept is shown in Fig. 15 and will be presented for the path directions of 0°, 45°, 90° and 135°. Pixels are processed from left to right along the image row (0° path). After processing pixel $\mathbf{p}_{-1} = [x - 1, y]$ of the upper row, all path costs over d of all directions are available

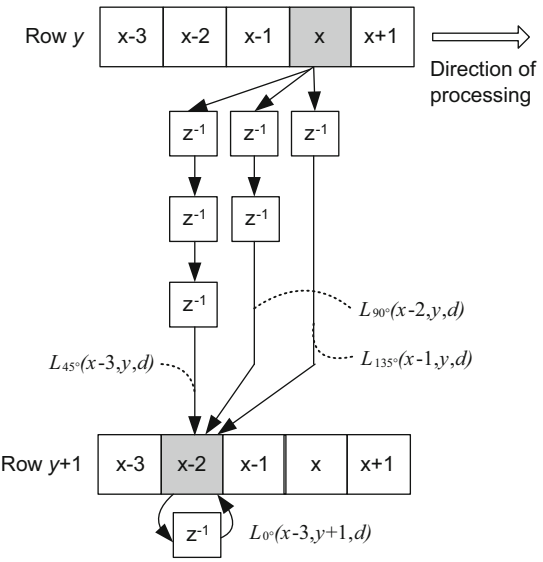


Fig. 15 Synchronized and parallel calculation of the path costs of the four paths $L_{0°}$, $L_{45°}$, $L_{90°}$, and $L_{135°}$ for the two pixels $\mathbf{p}_1 = [x, y]$ and $\mathbf{p}_2 = [x - 2, y + 1]$. Each delay element stores the respective path costs over all disparity levels for the duration of one processing step

in the path cost buffers $\left(\mathrm{z}^{-1}\right)$. Path costs are delayed, according to their path directions of 90° and 45° by one and two *additional* processing steps, respectively. Afterwards, path costs of $L_{45°}(x-3, y, d)$, $L_{90°}(x-2, y, d)$ and $L_{135°}(x-1, y, d)$ are available at the output of the path cost buffers. These are exactly those path costs needed for parallel and synchronous calculation of all path costs of all orientations for pixel $\mathbf{p}_2 = [x - 2, y + 1]$. Synchronous calculation allows direct summation of path costs in a pipeline that returns the aggregated costs S.

Therefore, all paths to the pixels $\mathbf{p}_1 = [x, y]$ *and* $\mathbf{p}_2 = [x - 2, y + 1]$ are calculated in parallel in a single processing step. This concept is extendable to an arbitrary number of rows. An additional delay by two pixels is introduced for each new row as illustrated in Fig. 15. Images are separated into image slices of N parallel rows in order to process whole images. Path costs of the last row of an image slice need to be stored and made available to the first row of the next slice.

Generalization of this concept is only limited by the fact that the maximum angle range must be within the half-closed interval $[0, 180°)$. This means that no paths in opposite directions can be directly supported without additional hardware. The two-dimensional parallelization allows regular data accesses of the input images and all intermediate values and will be further referred to as row parallelism. Moreover, this concept is independent of the processing method of the disparity levels, which can be either serial or parallel. Processing the disparity levels in parallel establishes a third dimension of parallelism, which will be referred to as disparity level parallelism. An approach of particular interest for dedicated hardware implementations is not to choose either extreme (none or all disparity levels in parallel) but to process the disparity levels in small groups (e.g. 2, 4, or 8). In this case, the size of the path cost buffers, as specified above, remains constant while

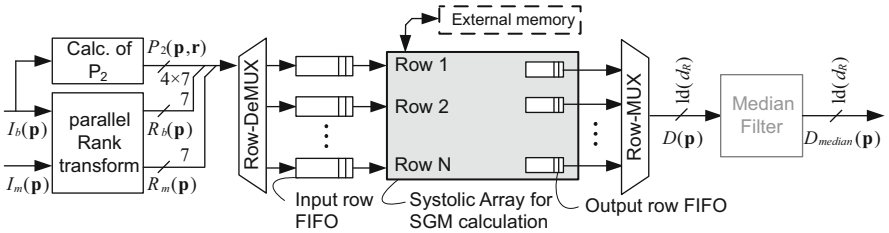


Fig. 16 Hardware architecture for calculation of disparity maps using rank-transform and semi-global matching. The median filter is optional

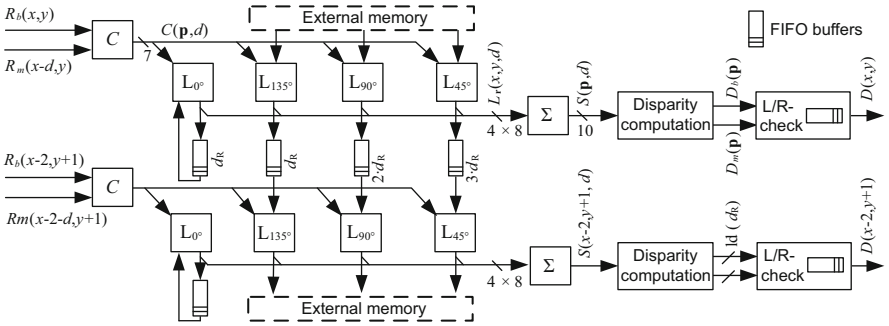


Fig. 17 Hardware architecture of the systolic array for parallel path cost calculation of the semi-global matching for two parallel rows

the throughput increases linearly with the number of parallelized disparity levels. However, some additional logic for the arithmetic computation of n paths in parallel will be required. The increase of logic requirements vs. performance of disparity level parallelism and row level parallelism will be investigated in Sect. 3.7.3.

3.7.2 Architecture

The hardware architecture for the entire stereo matching algorithm is given in Fig. 16. Computation of the rank transform of both images and calculation of the data dependent penalty term P_2 is done in parallel and synchronously utilizing the same data path.

A N-row buffer provides this data to the systolic array, which calculates the disparities of all N rows in parallel according to the parallelization concept introduced above. As a basic post processing step, a median filter is employed for outlier suppression.

A heterogeneous, completely synchronized systolic array realizes the parallelization concept for the semi-global matching utilizing path directions from 0°, 45°, 90° and 135°. Figure 17 shows the corresponding block diagram without utilization of disparity level parallelism. In this case, processing of a pixel **p** is carried out

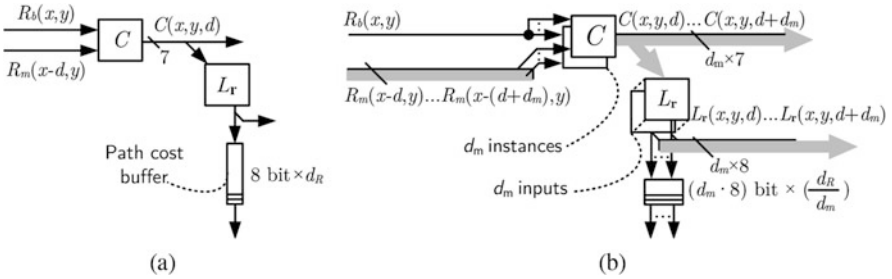


Fig. 18 Architectural extension (**b**) of the 2D-systolic array (**a**) for introducing disparity level parallelism where d_m specifies the number of disparity levels processed in parallel

sequentially over all disparities of this pixel. The first processing elements (*C-PEs*) calculate the matching costs $C(\mathbf{p}, d)$. Each of the following PEs (*L-PEs*) calculates the path costs L_r along a path $\mathbf{r}$ according to Eq. (6). The results are buffered in the appropriate path cost buffers. All L-PEs are completely identical and the path orientations are solely defined by the delays introduced by the path cost buffers. Path costs are summed to S and then processed by disparity computation PEs (*D-PEs*). D-PEs locate the minimum, i.e. the correct disparity, for the disparity maps D_b and D_m of the base and match camera, respectively. A final L/R-Check-PE projects the disparity map D_m to the perspective of the base camera, executes the left/right check including occlusion detection, and marks pixels accordingly. A local single row buffer is needed for the projection. It functions simultaneously as an output buffer.

In order to introduce disparity level parallelism in addition to the row level parallelism, the C-PEs and L-PEs are extended to process several consecutive disparity levels in parallel. These groups of parallel disparity levels are processed serially. This leads to an approximately linear increase in throughput. Furthermore, it is area efficient for two reasons. First, additional logic is only required for parts of the processing units. And second, the absolute size of local buffers does not change—only the depth-to-width ratio. This is the major advantage of disparity level parallelism. The architectural extension for disparity level parallelism is shown in Fig. 18.

Boundary treatment for pixels with missing stereo overlap (i.e. $x < d_{\max}$) significantly reduces the number of entries of the cost spaces $C(\mathbf{p}, d)$, $L_r(\mathbf{p}, d)$, $S(\mathbf{p}, d)$, and, consequently, leads to a computing time reduction. For VGA images and a disparity range of 128 px the reduction is 9.9% (without disparity level parallelism).

An external interim memory is required for storing the path costs of the three non-horizontal paths of the last row of an image slice and providing them to the first row of the consecutive image slice. Due to the extremely regular data transfer, obeying the FIFO-principle, and the low transfer rates, external SSRAM and SDRAM-memories can be used. Alternatively, on-chip memory can be considered due to the low absolute memory requirements.

Table 3 Minimum required clock frequencies of the SGM unit (including rank transform and median filter) for a fixed resolution of 640×480 px with 128 disparity levels at 30 fps and resource usage on a Xilinx Virtex-5 FPGA

	Min. clock frequency (MHz)				LUTs			
$p_r \setminus d_m$	1	2	4	8	1	2	4	8
5	219.6	112.3	58.5	31.6	5652	6621	10,110	17,214
10	111.9	57.4	30.0	16.8	11,595	13,398	20,565	34,589
20	58.3	30.2	17.2	13.4	23,379	26,986	41,292	69,578
30	40.7	21.4	14.9	12.4	35,119	40,700	61,930	103,504

The number of parallel rows and parallel disparity levels is denoted p_r and d_m, respectively

3.7.3 Performance

Performance of the complete system and scalability of the SGM unit are analyzed with the minimum clock frequency required to fulfill a fixed throughput constraint. This metric, i.e. the clock frequency normalized for a fixed throughput, allows direct and accurate comparison, and reflects the importance of performance while being independent from varying operating clock frequencies [78]. This also models a typical design constraint of real-world applications, where the throughput required is usually specified by external circumstances (e.g. by the cameras, required depth resolution, etc.). In this case, throughput-normalized metrics for clock frequency, resource usage, power, and latency enable straightforward identification of the Pareto-optimal point of operation. Table 3 provides the results for the SGM unit for a typical parameter set of 640×480 px at 30 fps. As metric for silicon area required, only Virtex5 LUTs are used. For more information (e.g. BRAMs) please refer to [8].

Interesting insights can be gained by studying the row parallelism vs. disparity parallelism trade-off. With increasing degree of parallelism, the SGM unit can be clocked with lower frequencies at the price of higher area requirements. However, there are significant differences between row level parallelism and disparity level parallelism. Each point in Fig. 19 is a specific configuration of the design representing the LUT requirements over the normalized clock frequency. This representation is considerably different from a typical AT-diagram, which would be inadequate for this comparison as it would not reflect the throughput constraint.

For a small number of parallel disparity levels, increasing disparity level parallelism is very efficient since it has a significantly smaller influence on the total resource usage than increasing row level parallelism. However, row parallelism is the key concept for stream-based processing and is crucial for a high base performance but increases linearly with the number of rows. The full potential of the parallelism approaches is exploited when using a combination of both, i.e. by using a small number of parallel rows and additionally introducing disparity level parallelism up to the configuration that does not yet require additional memory resources. For example, starting from the ($p_r = 10, d_m = 1$)-configuration a performance increase of approximately factor two can be achieved by doubling the number of either parallel rows or disparity levels. Increasing disparity level

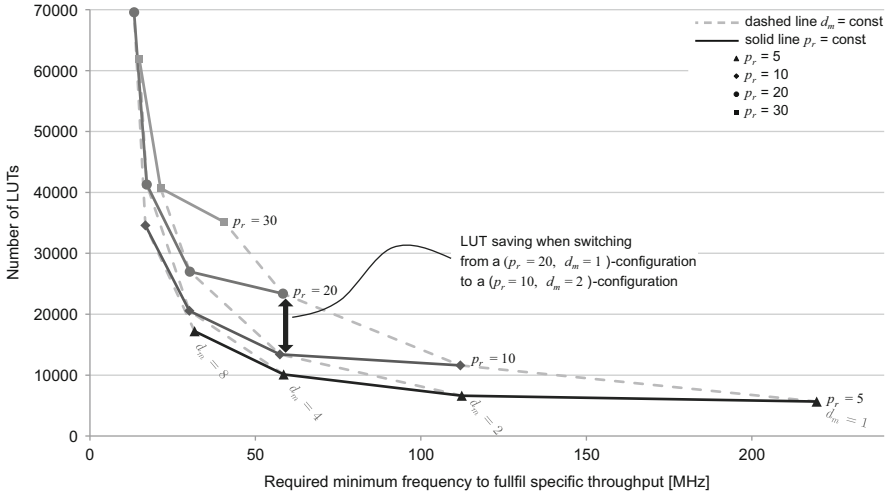


Fig. 19 Required number of LUTs of the systolic array of the SGM unit over the minimum required clock frequency to process 640×480 px with 128 disparity levels at 30 fps. The number of parallel rows and parallel disparity levels is denoted p_r and d_m, respectively. The diagrams effectively show the impact in area and performance when varying row parallelism and/or disparity level parallelism. The lower left border in the diagram reflects the Pareto-optimum configuration points

parallelism does not increase BRAM requirements (not shown, see [8]) and results in a LUT saving of factor 1.8. The major benefit of increasing disparity level parallelism is that local memory requirements remain constant for both, path costs buffers and input/output buffers.

A stereo vision system covering the entire stereo vision process including image acquisition, noise reduction, rectification, disparity estimation, post processing, and visualization has been integrated into a single FPGA. The system has been integrated on a custom build hardware platform show in Fig. 20. This work shows that it is possible to implement an algorithmically extremely high performing disparity matching algorithm in an FPGA with true real-time performance. More details on the implementation can be found in [8].

4 Summary

There has been and continues to be tremendous research in the field of computer vision, both on the algorithmic side and on the hardware side. Nowadays, many implementations for GPUs, FPGAs, ASICs, DSPs, and ASIPs are available. These cover a huge variety of algorithms and design aspects (e.g. algorithmic performance vs. silicon area). The two example implementations on the GPU and the FPGA for

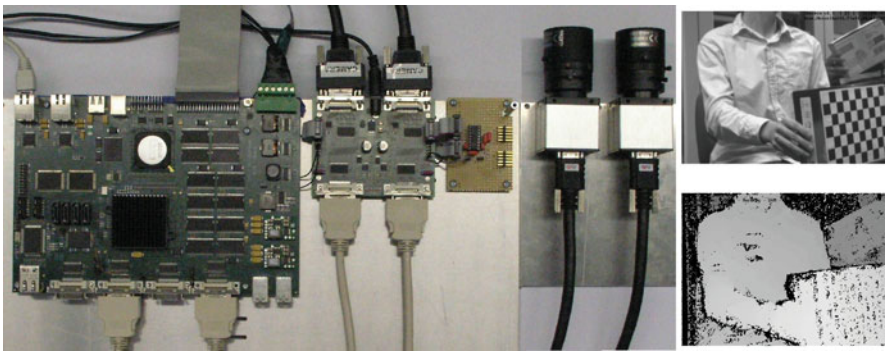


Fig. 20 Hardware setup of the stereo vision system with the system board and the stereo camera rig. On the right is the input image of lab scene and the computed raw disparity map before false color visualization and sending to display is conducted

semi-global matching based disparity estimation show, that it is possible to realize high quality stereo correspondence search in real-time. The GPU implementation enables SGM processing with 8 paths but without left/right check with more than 62 fps of images with a resolution of 640×480 and 128 disparity levels on a Nvidia Fermi architecture GPUs. The VLSI architecture is scalable and allows exact adaptation to the particular application. For the same image resolution frame rates of 1.7–319 fps are achieved at an operating frequency of 133 MHz. Which of the two architectures presented offers a more suitable solution depends on the external parameters.

5 Further Reading

A detailed algorithmic overview is provided in the textbook [93] and the surveys in [31, 91, 95]. Epipolar geometry and rectification is covered in [37, 117]. The OpenCV library provides many functions for stereo processing [12]. For multi-view stereo and 3D reconstruction [93] is a good starting point.

Dedicated image processing architectures including rectification and many more are covered in [3] and RTL hardware design in [16]. Various kinds of computer architectures including GPUs are found in the newest edition of [38].

References

1. Ambrosch, K., Kubinger, W.: Accurate hardware-based stereo vision. Computer Vision and Image Understanding, Elsevier **114**, 1303–1316 (2010)
2. Arias-Estrada, M., Xicotencatl, J., Brebner, G., Woods, R.: Multiple stereo matching using an extended architecture. Proc. Field-Programmable Logic and Applications **2147**, 203–212 (2001)

3. Bailey, D.G.: Design for embedded image processing on FPGAs. John Wiley & Sons, Singapore (2011)
4. Banz, C., Blume, H., Pirsch, P.: Real-time semi-global matching disparity estimation on the GPU. Proc. IEEE Intl. Conf. Computer Vision Workshops pp. 514–521 (2011)
5. Banz, C., Blume, H., Pirsch, P.: Evaluation of penalty functions for SGM cost aggregation. Intl. Archives of Photogrammetry and Remote Sensing (2012)
6. Banz, C., Dolar, C., Cholewa, F., Blume, H.: Instruction set extension for high throughput disparity estimation in stereo image processing. Proc. IEEE Intl. Conf. Architectures and Processors Application-Specific Systems pp. 169–175 (2011)
7. Banz, C., Hesselbarth, S., Flatt, H., Blume, H., Pirsch, P.: Real-time stereo vision system using semi-global matching disparity estimation: Architecture and FPGA-implementation. Proc. IEEE Intl. Conf. Embedded Computer Systems: Architectures, Modeling, and Simulation pp. 93–101 (2010)
8. Banz, C., Hesselbarth, S., Flatt, H., Blume, H., Pirsch, P.: Real-time stereo vision system using semi-global matching disparity estimation: Architecture and FPGA-implementation. Trans. High-Performance Embedded Architectures and Compilers, Springer (2012)
9. Belhumeur, P.N.: A bayesian approach to binocular steropsis. Intl. Journal of Computer Vision **19**(3), 237–260 (1996)
10. Birchfield, S., Tomasi, C.: A pixel dissimilarity measure that is insensitive to image sampling. IEEE Trans. Pattern Analysis and Machine Intelligence **20**(4), 401–406 (1998)
11. Boykov, Y., Veksler, O., Zabih, R.: Fast approximate energy minimization via graph cuts. Proc. IEEE Intl. Conf. Computer Vision **1**, 377–384 (1999)
12. Bradski, G., Kaehler, A.: Learning OpenCV, 1 edn. O'Reilly, Sebastopol (2008)
13. Brunton, A., Chang, S., Roth, G.: Belief propagation on the GPU for stereo vision. Proc. Canadian Conf. Computer and Robot Vision p. 76 (2006)
14. Chang, N., Lin, T.M., Tasi, T.H., Tseng, Y.C., Chang, T.S.: Real-time DSP implementation on local stereo matching. Proc. IEEE Intl. Conf. Multimedia and Expo pp. 2090–2093 (2007)
15. Chang, N., Tasi, T.H., Hsu, B., Chen, Y., Chang, T.S.: Algorithm and architecture of disparity estimation with mini-census adaptive support weight. IEEE Trans. Circuits and Systems for Video Technology **20**(6), 792–805 (2010)
16. Chu, P.P.: RTL hardware design using VHDL: Coding for efficiency, portability, and scalability. Wiley-Interscience, Hoboken and N.J (2006)
17. Cornells, N., van Gool, L.: Real-time connectivity constrained depth map computation using programmable graphics hardware. Proc. IEEE Conf. Computer Vision and Pattern Recognition **1**, 1099–1104 (2005)
18. Darabiha, A., MacLean, W., Rose, J.: Reconfigurable hardware implementation of a phase-correlation stereo algorithm. Machine Vision and Applications, Springer **17**, 116–132 (2006)
19. Diaz, J., Ros, E., Carrillo, R., Prieto, A.: Real-time system for high-image resolution disparity estimation. IEEE Trans. Image Processing **16**(1), 280–285 (2007)
20. Ernst, I., Hirschmüller, H.: Mutual information based semi-global stereo matching on the GPU. Proc. Intl. Symp. Visual Computing **5358**, 228–239 (2008)
21. Ess, A., Leibe, B., Schindler, K., van Gool, L.: A mobile vision system for robust multi-person tracking. Proc. IEEE Conf. Computer Vision and Pattern Recognition pp. 1–8 (2008)
22. Faugeras, O., Viéville, T., Theron, E., Vuillemin, J., Hotz, B., Zhang, Z., Moll, L., Bertin, P., Mathieu, H., Fua, P., Berry, G., Proy, C.: Real-time correlation-based stereo: Algorithm, implementations and applications (1993)
23. Felzenszwalb, P.F., Huttenlocher, D.P.: Efficient belief propagation for early vision. Intl. Journal of Computer Vision, Springer **70**, 41–54 (2006)
24. Forstmann, S., Kanou, Y., Jun, O., Thuering, S., Schmitt, A.: Real-time stereo by using dynamic programming. Proc. IEEE Conf. Computer Vision and Pattern Recognition Workshop p. 29 (2004)
25. Gehrig, S., Rabe, C.: Real-time semi-global matching on the CPU. Proc. IEEE Conf. Computer Vision and Pattern Recognition Workshop pp. 85–92 (2010)

26. Gehrig, S.K., Eberli, F., Meyer, T.: A real-time low-power stereo vision engine using semi-global matching. Proc. Intl. Conf. Computer Vision Systems **5815**, 134–143 (2009)
27. Geiger, A., Lenz, P., Urtasun, R.: Are we ready for autonomous driving? The KITTI vision benchmark suite. In: Proc. IEEE Conf. on Computer Vision and Pattern Recognition (2012)
28. Georgoulas, C., Andreadis, I.: A real-time fuzzy hardware structure for disparity map computation. Journal of Real-Time Image Processing, Springer **6**(4), 257–273 (2011)
29. Gibson, J., Marques, O.: Stereo depth with a unified architecture GPU. Proc. IEEE Conf. Computer Vision and Pattern Recognition Workshop pp. 1–6 (2008)
30. Gong, M.: Real-time joint disparity and disparity flow estimation on programmable graphics hardware. Computer Vision and Image Understanding, Elsevier **113**(1), 90–100 (2009)
31. Gong, M., Yang, R., Wang, L., Gong Mingwei: A performance study on different cost aggregation approaches used in real-time stereo matching. Intl. Journal of Computer Vision, Springer **75**, 283–296 (2007)
32. Gong, M., Yang, Y.H.: Near real-time reliable stereo matching using programmable graphics hardware. Proc. IEEE Conf. Computer Vision and Pattern Recognition **1**, 924–931 (2005)
33. Güney, F., Geiger, A.: Displets: Resolving stereo ambiguities using object knowledge. Proc. IEEE Conf. on Computer Vision and Pattern Recognition (2015)
34. Haller, I., Nedevschi, S.: GPU optimization of the SGM stereo algorithm. Proc. IEEE Intl. Conf. Intelligent Computer Communication and Processing pp. 197–202 (2010)
35. Haller, I., Nedevschi, S.: Design of interpolation functions for subpixel-accuracy stereo-vision systems. IEEE Trans. Image Processing **21**(2), 889–898 (2012)
36. Harris: Optimizing parallel reduction in CUDA (2007). Whitepaper included in Nvidia Cuda SDK 4.0
37. Hartley, R.I., Zisserman, A.: Multiple view geometry in computer vision, 2. ed., 7. print. edn. Cambridge Univ. Press, Cambridge (2010)
38. Hennessy, J.L., Patterson, D.A.: Computer architecture: A quantitative approach, 5 edn. Morgan Kaufmann, San Francisco and Calif and Oxford (2011)
39. Hirschmüller, H.: Accurate and efficient stereo processing by semi-global matching and mutual information. Proc. IEEE Conf. Computer Vision and Pattern Recognition **2**, 807–814 (2005)
40. Hirschmüller, H.: Stereo processing by semiglobal matching and mutual information. IEEE Trans. Pattern Analysis and Machine Intelligence **30**(2), 328–341 (2008)
41. Hirschmüller, H., Scharstein, D.: Evaluation of cost functions for stereo matching. Proc. IEEE Conf. Computer Vision and Pattern Recognition pp. 1–8 (2007)
42. Hirschmüller, H., Scharstein, D.: Evaluation of stereo matching costs on images with radiometric differences. IEEE Trans. Pattern Analysis and Machine Intelligence **31**(9), 1582–1599 (2009)
43. Hosni, A., Bleyer, M., Rhemann, C., Gelautz, M., Rother, C.: Real-time local stereo matching using guided image filtering. Proc. IEEE Intl. Conf. Multimedia and Expo pp. 1–6 (2011)
44. Hosseini, F., Fijany, A., Safari, S., Fontaine, J.: Fast implementation of dense stereo vision algorithms on a highly parallel SIMD architecture. Journal of Real-Time Image Processing, Springer pp. 1–15 (2011)
45. Humenberger, M., Zinner, C., Kubinger, W.: Performance evaluation of a census-based stereo matching algorithm on embedded and multi-core hardware. Proc. Intl. Symp. Image and Signal Processing and Analysis pp. 388–393 (2009)
46. Ivanchenko, V., Shen, H., Coughlan, J.: Elevation-based MRF stereo implemented in real-time on a GPU. Workshop Applications of Computer Vision pp. 1–8 (2009)
47. Jiangbo, L., Rogmans, S., Lafruit, G., Catthoor, F.: Real-time stereo correspondence using a truncated separable laplacian kernel approximation on graphics hardware. Proc. IEEE Intl. Conf. Multimedia and Expo pp. 1946–1949 (2007)
48. Jiangbo, L., Zhang, K., Lafruit, G., Catthoor, F.: Real-time stereo matching: a cross-based local approach. Proc. IEEE Intl. Conf. Acoustics, Speech and Signal Processing pp. 733–736 (2009)

49. Jin, S., Cho, J., Pham, X.D., Lee, K.M., Park, S.K., Kim, M., Jeon, J.W.: FPGA design and implementation of a real-time stereo vision system. IEEE Trans. Circuits and Systems for Video Technology **20**(1), 15–26 (2010)
50. Kalarot, R., Morris, J.: Comparison of FPGA and GPU implementations of real-time stereo vision. Proc. IEEE Conf. Computer Vision and Pattern Recognition Workshop pp. 9–15 (2010)
51. Kalarot, R., Morris, J., Gimel'farb, G.: Performance analysis of multi-resolution symmetric dynamic programming stereo on GPU. Proc. Intl. Conf. Image and Vision Computing New Zealand pp. 1–7 (2010)
52. Kanade, T., Yoshida, A., Oda, K., Kano, H., Tanaka, M.: A stereo machine for video-rate dense depth mapping and its new applications. Proc. IEEE Conf. Computer Vision and Pattern Recognition pp. 196–202 (1996)
53. Ke, Z., Jiangbo, L., Qiong, Y., Lafruit, G., Lauwereins, R., van Gool, L.: Real-time and accurate stereo: A scalable approach with bitwise fast voting on CUDA. IEEE Trans. Circuits and Systems for Video Technology **21**(7), 867–878 (2011)
54. Kendall, A., Martirosyan, H., Dasgupta, S., Henry, P., Kennedy, R., Bachrach, A., Bry, A.: End-to-end learning of geometry and context for deep stereo regression. arXiv preprint arXiv:1703.04309 (2017)
55. Kim, J., Kolmogorov, V., Zabih, R.: Visual correspondence using energy minimization and mutual information. Proc. IEEE Intl. Conf. Computer Vision pp. 1033–1040 (2003)
56. Konolige, K.: Small vision systems: Hardware and implementation. Proc. Intl. Symp. Robotic Research (1997)
57. Kung, M., Au, O., Wong, P., Chun, H.L.: Block based parallel motion estimation using programmable graphics hardware. Proc. Intl. Conf. Audio, Language and Image Processing pp. 599–603 (2008)
58. Lee, S.H., Yi, J., Kim, J.S.: Real-time stereo vision on a reconfigurable system. Proc. Intl. Conf. Embedded Computer Systems: Architectures, Modeling, and Simulation Workshops **3553**, 299–307 (2005)
59. Liang, C., Cheng, C., Lai, Y., Chen, L., Chen, H.: Hardware-efficient belief propagation. IEEE Trans. Circuits and Systems for Video Technology **21**(5), 525–537 (2011)
60. Liang, W., Miao, L., Minglun, G., Ruigang, Y., Nister, D.: High-quality real-time stereo using adaptive cost aggregation and dynamic programming. Proc. Intl. Symp. 3D Data Processing, Visualization, and Transmission pp. 798–805 (2006)
61. Liang, W., Mingwei, G., Minglun, G., Ruigang, Y.: How far can we go with local optimization in real-time stereo matching. Proc. Intl. Symp. 3D Data Processing, Visualization, and Transmission pp. 129–136 (2006)
62. Liu, J., Xu, Y., Klette, R., Chen, H., Vaudrey, T.: Disparity Map Computation on a Cell Processor (2009)
63. Luo, W., Schwing, A., Urtasun, R.: Efficient deep learning for stereo matching. Proc. IEEE Conf. on Computer Vision and Pattern Recognition (2016)
64. van der Mark, W., Gavrila, D.: Real-time dense stereo for intelligent vehicles. IEEE Trans. Intelligent Transportation Systems **7**(1), 38–50 (2006)
65. Masrani, D., MacLean, W.: A real-time large disparity range stereo-system using FPGAs. Proc. Intl. Conf. Computer Vision Systems p. 13 (2006)
66. Mattoccia, S., Viti, M., Ries, F.: Near real-time fast bilateral stereo on the GPU. Proc. IEEE Conf. Computer Vision and Pattern Recognition Workshop pp. 136–143 (2011)
67. Mayer, N., Ilg, E., Häusser, P., Fischer, P., Cremers, D., Dosovitskiy, A., Brox, T.: A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. Proc. IEEE Conf. on Computer Vision and Pattern Recognition (2016)
68. Mei, X., Sun, X., Zhou, M., Jiao, S., Wang, H., Zhang, X.: On building an accurate stereo matching system on graphics hardware. Proc. IEEE Intl. Conf. Computer Vision Workshops pp. 467–474 (2011)
69. Menze, M., Geiger, A.: Object scene flow for autonomous vehicles. Proc. IEEE Conf. on Computer Vision and Pattern Recognition (2015)

70. Menze, M., Heipke, C., Geiger, A.: Joint 3D estimation of vehicles and scene flow. ISPRS Workshop on Image Sequence Analysis (2015)
71. Minglun, G., Ruigang, Y.: Image-gradient-guided real-time stereo on graphics hardware. Proc. Intl Conf. 3D Digital Imaging and Modeling pp. 548–555 (2005)
72. Miyajima, Y., Maruyama, T.: A real-time stereo vision system with FPGA. Proc. Intl. Conf. Field Programmable Logic And Application **2778**, 448–457 (2003)
73. Morris, J., Jawed, K., Gimel'farb, G., Khan, T.: Breaking the 'Ton': Achieving 1% depth accuracy from stereo in real time. Proc. Intl. Conf. Image and Vision Computing New Zealand pp. 142–147 (2009)
74. Mühlmann, K., Maier, D., Hesser, J., Manner, R.: Calculating dense disparity maps from color stereo images, an efficient implementation. Intl. Journal of Computer Vision, Springer **47**(1–3), 79–88 (2002)
75. Pantilie, C., Nedevschi, S.: SORT-SGM: Subpixel optimized real-time semiglobal matching for intelligent vehicles. IEEE Trans. Vehicular Technology **61**(3), 1032–1042 (2012)
76. Park, S., Jeong, H.: Real-time stereo vision FPGA chip with low error rate. Proc. Intl. Conf. Multimedia and Ubiquitous Engineering pp. 751–756 (2007)
77. Paya Vaya, G., Martin Langerwerf, J., Banz, C., Giesemann, F., Pirsch, P., Blume, H.: VLIW architecture optimization for an efficient computation of stereoscopic video applications. Proc. Intl. Conf. Green Circuits and Systems pp. 457–462 (2010)
78. Paya-Vaya, G., Martin-Langerwerf, J., Pirsch, P.: A multi-shared register file structure for VLIW processors. Journal of Signal Processing Systems, Springer **58**(2), 215–231 (2010)
79. Perez, J., Sanchez, P., Martinez, M.: High memory throughput FPGA architecture for high-definition Belief-Propagation stereo matching. Proc. Intl. Conf. Signals, Circuits and Systems pp. 1–6 (2009)
80. Pirsch, P.: Architectures for digital signal processing. John Wiley & Sons, Inc., Chichester (2008)
81. Pock, T., Schoenemann, T., Graber, G., Bischof, H., Cremers, D.: A convex formulation of continuous multi-label problems. Proc. European Conf. on Computer Vision **5304**, 792–805 (2008)
82. Podlozhnyuk, V.: Image Convolution with CUDA (2007). Whitepaper included in Nvidia Cuda SDK 4.0
83. Ranft, B., Schoenwald, T., Kitt, B.: Parallel matching-based estimation - a case study on three different hardware architectures. Proc. IEEE Intelligent Vehicles Symposium pp. 1060–1067 (2011)
84. Ros, G., Sellart, L., Materzynska, J., Vazquez, D., Lopez, A.: The SYNTHIA dataset: A large collection of synthetic images for semantic segmentation of urban scenes. Proc. IEEE Conf. on Computer Vision and Pattern Recognition (2016)
85. Roy, S., Cox, I.: A maximum-flow formulation of the n-camera stereo correspondence problem. Proc. of Intl. Conf. on Computer Vision (1998)
86. Ruigang, Y., Pollefeys, M.: Multi-resolution real-time stereo on commodity graphics hardware. Proc. IEEE Conf. Computer Vision and Pattern Recognition **1**, I–211–I–217 (2003)
87. Ruigang, Y., Welch, G., Bishop, G.: Real-time consensus-based scene reconstruction using commodity graphics hardware. Proc. Pacific Conf. Computer Graphics and Applications pp. 225–234 (2002)
88. Sabihuddin, S., Islam, J., MacLean, W.: Dynamic programming approach to high frame-rate stereo correspondence: A pipelined architecture implemented on a field programmable gate array. Proc. Canadian Conf. Electrical and Computer Engineering pp. 001,461–001,466 (2008)
89. Safari, S., Fijany, A., Diotalevi, F., Hosseini, F.: Highly parallel and fast implementation of stereo vision algorithms on MIMD many-core Tilera architecture. Proc. IEEE Aerospace Conf. pp. 1–11 (2012)
90. Scharstein, D., Szeliski, R.: The Middlebury Stereo Pages. http://vision.middlebury.edu/stereo/

91. Scharstein, D., Szeliski, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. Intl. Journal of Computer Vision, Springer **47**(1), 7–42 (2002)
92. Scharstein, D., Szeliski, R.: High-accuracy stereo depth maps using structured light. Proc. IEEE Conf. Computer Vision and Pattern Recognition **1**, I–195–I–202 (2003)
93. Szeliski, R.: Computer vision: Algorithms and applications. Springer, London and New York (2011)
94. Szeliski, R., Scharstein, D.: Sampling the disparity space image. IEEE Trans. Pattern Analysis and Machine Intelligence **26**(3), 419–425 (2004)
95. Tomasi, M., Vanegas, M., Barranco, F., Daz, J., Ros, E.: Massive parallel-hardware architecture for multiscale stereo, optical flow and image-structure computation. IEEE Trans. Circuits and Systems for Video Technology **22**(2), 282–294 (2012)
96. Tombari, F., Mattoccia, S., Di Stefano, L., Addimanda, E.: Classification and evaluation of cost aggregation methods for stereo correspondence. Proc. IEEE Conf. Computer Vision and Pattern Recognition pp. 1–8 (2008)
97. Tseng, Y.C., Chang, T.S.: Architecture design of belief propagation for real-time disparity estimation. IEEE Trans. Circuits and Systems for Video Technology **20**(11), 1555–1564 (2010)
98. Ttofis, C., Hadjitheophanous, S., Georghiades, A., Theocharides, T.: Edge-directed hardware architecture for real-time disparity map computation. IEEE Trans. Computers **PP**(99), 1 (2012)
99. Ttofis, C., Theocharides, T.: Towards accurate hardware stereo correspondence: A real-time FPGA implementation of a segmentation-based adaptive support weight algorithm. Proc. Conf. Design, Automation & Test in Europe pp. 703–708 (2012)
100. Vaish, V., Levoy, M., Szeliski, R., Zitnick, C., Sing, B.K.: Reconstructing occluded surfaces using synthetic apertures: Stereo, focus and robust measures. Proc. IEEE Conf. Computer Vision and Pattern Recognition **2**, 2331–2338 (2006)
101. Villalpando, C., Morfopolous, A., Matthies, L., Goldberg, S.: FPGA implementation of stereo disparity with high throughput for mobility applications. Proc. IEEE Aerospace Conf. pp. 1–10 (2011)
102. Weber, M., Humenberger, M., Kubinger, W.: A very fast census-based stereo matching implementation on a graphics processing unit. Proc. IEEE Intl. Conf. Computer Vision Workshops pp. 786–793 (2009)
103. Wedel, A., Rabe, C., Vaudrey, T., Brox, T., Franke, U., Cremers, D.: Efficient dense scene flow from sparse or dense stereo data. In: 10th European Conf. on Computer Vision, pp. 739–751. Springer-Verlag, Berlin, Heidelberg (2008)
104. Woodfill, J., Gordon, G., Buck, R.: Tyzx DeepSea high speed stereo vision system. Proc. IEEE Conf. Computer Vision and Pattern Recognition Workshop p. 41 (2004)
105. Woodfill, J., Gordon, G., Jurasek, D., Brown, T., Buck, R.: The Tyzx DeepSea G2 vision system, a taskable, embedded stereo camera. Proc. IEEE Conf. Computer Vision and Pattern Recognition Workshop p. 126 (2006)
106. Woodfill, J., Herzen, B.v.: Real-time stereo vision on the PARTS reconfigurable computer. Proc. IEEE Symp. FPGAs for Custom Computing Machines pp. 201–210 (1997)
107. Xu, Y., Chen, H., Klette, R., Liu, J., Vaudrey, T.: Belief propagation implementation using CUDA on an Nvidia GTX 280. Proc. Advances in Artificial Intelligence **5866**, 180–189 (2009)
108. Yamaguchi, K., McAllester, D., Urtasun, R.: Efficient joint segmentation, occlusion labeling, stereo and flow estimation. Proc. European Conf. on Computer Vision (2014)
109. Yang, Q., Wang, L., Yang, R., Wang, S., Liao, M., Nister, D.: Real-time global stereo matching using hierarchical belief propagation. Proc. The British Machine Vision Conf. pp. 989–998 (2006)
110. Yunde, J., Xiaoxun, Z., Mingxiang, L., Luping: A miniature stereo vision machine (MSVM-III) for dense disparity mapping. Proc. IEEE Intl. Conf. Pattern Recognition **1**, 728–731 (2004)

111. Zabih, R., Woodfill, J.: Non-parametric local transforms for computing visual correspondence. Proc. European Conf. on Computer Vision pp. 151–158 (1994)
112. Zach, C., Klaus, A., Hadwiger, M., Karner, K.: Accurate dense stereo reconstruction using graphics hardware. Proc. EUROGRAPHICS pp. 227–234 (2003)
113. Zach, C., Sormann, M., Karner, K.: Scanline optimization for stereo on graphics hardware. Proc. Intl. Symp. 3D Data Processing, Visualization, and Transmission pp. 512–518 (2006)
114. Zatt, B., Shafique, M., Bampi, S., Henkel, J.: Multi-level pipelined parallel hardware architecture for high throughput motion and disparity estimation in Multiview Video Coding. Proc. Conf. Design, Automation & Test in Europe pp. 1–6 (2011)
115. Zbontar, J., LeCun, Y.: Stereo matching by training a convolutional neural network to compare image patches. Journal of Machine Learning Research (2016)
116. Zhang, K., Lu, J., Lafruit, G.: Cross-based local stereo matching using orthogonal integral images. Trans. Circuits Syst. Video Techn., IEEE **19**, 1073–1079 (2009)
117. Zhang, Z.: Determining the epipolar geometry and its uncertainty: A review. Intl. Journal of Computer Vision, Springer **27**(2), 161–195 (1998)
118. Zhang, Z.: A flexible new technique for camera calibration. IEEE Trans. Pattern Analysis and Machine Intelligence **22**(11), 1330–1334 (2000)

Hardware Architectures for the Fast Fourier Transform

Mario Garrido, Fahad Qureshi, Jarmo Takala, and Oscar Gustafsson

Abstract The fast Fourier transform (FFT) is a widely used algorithm in signal processing applications. FFT hardware architectures are designed to meet the requirements of the most demanding applications in terms of performance, circuit area, and/or power consumption. This chapter summarizes the research on FFT hardware architectures by presenting the FFT algorithms, the building blocks in FFT hardware architectures, the architectures themselves, and the bit reversal algorithm.

1 Introduction

The *Fourier transform* is one of the most important tools in digital signal processing. It is used to convert continuous signals in time domain into frequency domain. For discrete data, such as that in digital systems, the *discrete Fourier transform* (DFT) is used instead. The DFT transforms a finite sequence of equally spaced samples to a corresponding frequency domain representation as follows:

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, \quad k = 0, 1, \ldots, N-1, \tag{1}$$

where N denotes the DFT size, $x[n]$ is the input signal in the time domain, and $X[k]$ is the output signal in the frequency domain, which is defined for the frequencies

M. Garrido · O. Gustafsson
Linköping University, Linköping, Sweden
e-mail: mario.garrido.galvez@liu.se; oscar.gustafsson@liu.se

F. Qureshi (✉)
Tampere University of Technology, Tampere, Finland
e-mail: fahad@tut.fi

J. Takala
Department of Pervasive Computing, Tampere University of Technology, Tampere, Finland
e-mail: jarmo.takala@tut.fi

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_17

$k \in [0, N-1]$. Note that both $x[n]$ and $X[k]$ are discrete signals. The coefficients W_N^{nk} are called *twiddle factors* and correspond to rotations in the complex plane defined as

$$W_N^{nk} = e^{-j\frac{2\pi}{N}nk} = \cos\left(\frac{2\pi}{N}nk\right) - j\sin\left(\frac{2\pi}{N}nk\right), \tag{2}$$

where j denotes the imaginary unit.

The original signal $x[n]$ can be recovered from $X[k]$ by calculating the *inverse discrete Fourier transform* (IDFT):

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]W_N^{nk}, \qquad n = 0, 1, \ldots, N-1. \tag{3}$$

The arithmetic complexity of the DFT in (1) is $O(N^2)$. However, the DFT contains redundant operations.

The term *fast Fourier transform* (FFT) refers to various methods that reduce the computational complexity of the DFT. The most popular one is the Cooley-Tukey algorithm [16]. Section 2 discusses FFT algorithms and representations.

For the implementation of FFT hardware architectures, Sect. 3 discusses the building blocks that they consist of, i.e., butterflies, rotators and shuffling circuits. Later, Sect. 4 presents the FFT hardware architectures. They are divided into fully parallel, iterative and pipelined FFTs. The outputs of FFT hardware architectures are generally provided in bit-reversed order. Section 5 explain the bit reversal algorithm used to sort them out. Finally, Sect. 6 summarizes the main conclusions of this chapter.

2 FFT Algorithms

2.1 The Cooley-Tukey Algorithm

The Cooley-Tukey algorithm [16] decomposes the DFT into a set of smaller DFTs, when N is not a prime number. Let us assume that $N = N_2 \cdot N_1$ and consider that n and k are calculated as

$$\begin{aligned} n &= n_1 N_2 + n_2, \quad \text{with} \quad n_1 = 0, \ldots, N_1 - 1 \quad \text{and} \quad n_2 = 0, \ldots, N_2 - 1; \\ k &= k_2 N_1 + k_1, \quad \text{with} \quad k_1 = 0, \ldots, N_1 - 1, \quad \text{and} \quad k_2 = 0, \ldots, N_2 - 1. \end{aligned} \tag{4}$$

Then, (1) can be written as

$$X[k_2N_1 + k_1] = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x[n_1N_2 + n_2] W_N^{(n_1N_2+n_2)(k_2N_1+k_1)}. \tag{5}$$

By exploiting the periodicity of the twiddle factors, i.e., $W_N^{\phi N_2} = W_{N_1}^{\phi}$, $W_N^{\phi N_1} = W_{N_2}^{\phi}$ and $W_N^{\phi N_1 N_2} = 1$, the equation is transformed into

$$X[k_2N_1 + k_1] = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x[n_1N_2 + n_2] W_{N_2}^{n_2k_2} W_{N_1}^{n_1k_1} W_N^{n_2k_1}, \tag{6}$$

which finally results in

$$X[k_2N_1 + k_1] = \underbrace{\sum_{n_2=0}^{N_2-1} \left[\underbrace{\left(\sum_{n_1=0}^{N_1-1} x[n_1N_2 + k_1] W_{N_1}^{n_1k_1} \right)}_{N_1-\text{point DFT}} \underbrace{W_N^{n_2k_1}}_{\text{Twiddlefactor}} \right] W_{N_2}^{n_2k_2}}_{N_2-\text{point DFT}}, \tag{7}$$

where the N_1-point and N_2-point DFTs are referred to as inner and outer DFTs, respectively. As a result, an N-point DFT is broken down into N_2 DFTs of size N_1 and N_1 FFTs of size N_2, with twiddle factor multiplications in the middle. This is illustrated in Fig. 1 for $N = 16$, $N_2 = 8$ and $N_1 = 2$.

In general, N can be the product of several numbers, i.e., $N = N_{m-1} \cdot N_{m-2} \cdot \ldots \cdot N_0$. *Radix-r* refers to the case in which $N_i = r, \forall i = 0 \ldots m-1$. The radix-$r$ FFT is derived by expressing n and k as

$$\begin{aligned} &n = n_{m-1} \cdot r^{m-1} + n_{m-2} \cdot r^{m-2} + n_1 \cdot r + n_0, \\ &n_i \in [0, \ldots, r-1], \forall i = 0, \ldots, m-1; \\ &k = k_{m-1} \cdot r^{m-1} + k_{m-2} \cdot r^{m-2} + k_1 \cdot r + k_0, \\ &k_i \in [0, \ldots, r-1], \forall k = 0, \ldots, m-1. \end{aligned} \tag{8}$$

This results in $m = \log_r N$ nested r-point DFTs with twiddle factors in between. When $r = 2$, each nested 2-point DFT is calculated on $N/2$ pairs of data and requires a total of N additions. This leads to an arithmetic complexity of $O(N \log N)$ for the entire FFT, compared to $O(N^2)$ in the DFT in (1).

Finally, the recursive application of Cooley-Tukey principle can be done by starting from the time domain sequence, which results in a decimation-in-time (DIT) decomposition. In a similar fashion, the decimation-in-frequency (DIF) decomposition is obtained by starting from the frequency sequence.

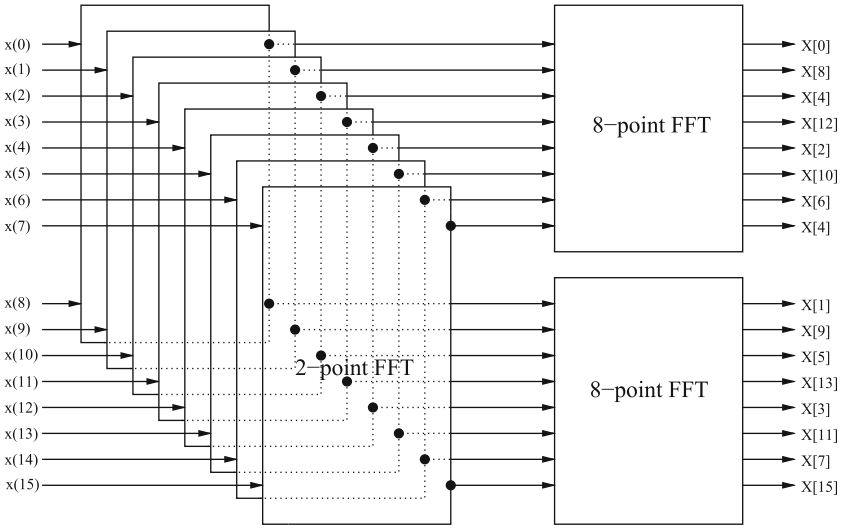


Fig. 1 Graphical representation of the Cooley-Tukey algorithm

2.2 *Representation Using Flow Graphs*

Figures 2 and 3 show the flow graphs of the radix-2 DIF and DIT FFT algorithms, respectively. The numbers at the input of the graph represent the indexes of the input sequence, $x[n]$, whereas those at the output are the frequencies, k, of the output signal $X[k]$. The flow graphs consist of a series of n stages, $s \in \{1 \ldots n\}$. At each stage, additions, subtractions and rotations are calculated. Additions and subtractions come in pairs, forming the so called *butterflies*, which have the shape $\begin{matrix}\nabla \\ \Delta\end{matrix}$. The upper output of the butterfly provides the sum of the inputs and the lower part subtracts the lower input from the upper one.

Each number ϕ in between butterflies represents a rotation, which corresponds to a complex multiplication by

$$W_N^{\phi} = e^{-j\frac{2\pi}{N}\phi}. \tag{9}$$

Rotations by $\phi \in \{0, N/4, N/2, 3N/4\}$ are called *trivial rotations*, due to the fact that they correspond to multiplications by $1, -j, -1$ or j. Trivial rotations can easily be calculated by interchanging the real and imaginary parts of the inputs and/or changing their signs.

Finally, an index I is added to the left of the flow graph, together with its binary representation $b_{n-1}b_{n-2}\ldots b_1 b_0$. This index together with the stage is used to refer to the rotations in the flow graph. For instance, the rotation with index $I = 14$ at stage $s = 1$ in Fig. 2 is $\phi_s(I) = \phi_1(14) = 6$. Through the chapter, the symbol

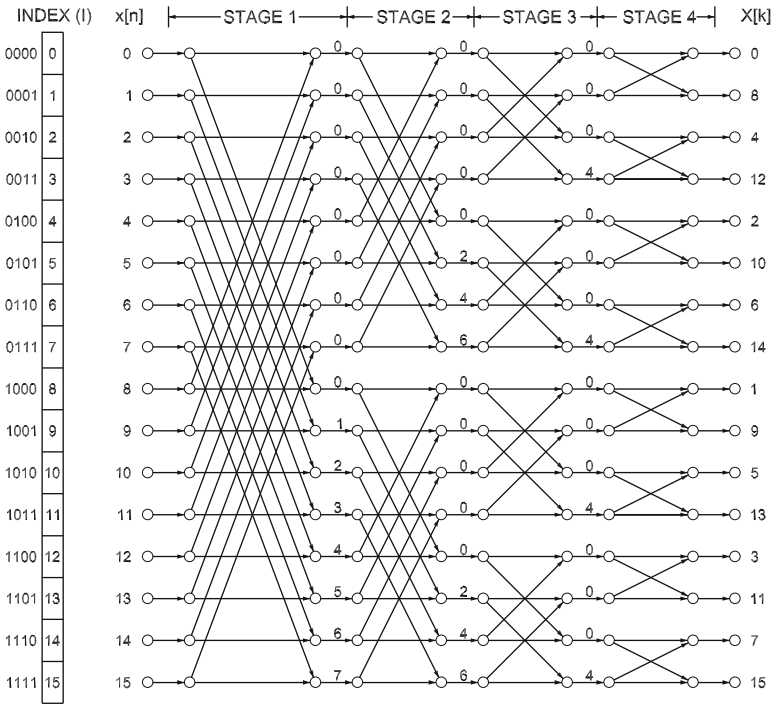


Fig. 2 Flow graph of a 16-point radix-2 DIF FFT

($\equiv$) is used to relate decimal numbers and their binary representation, e.g., $I \equiv b_{n-1}b_{n-2}\ldots b_1 b_0$.

When the FFT size N is large, it is not feasible to represent the FFT by a flow graph. In this case, the binary tree and the triangular matrix representations are useful tools to represent the algorithms in a simple manner.

Note that any FFT flow graph of the same radix will look the same. The only difference is where the twiddle factor multiplications are positioned. This can be seen by comparing Figs. 2 and 3. Hence, all the following algorithms only differ in the twiddle factor multiplications, although this may provide significant differences when implemented.

2.3 Binary Tree Representation

The binary tree representation [61, 83] is a generalization of the Cooley-Tukey algorithm. In the Cooley-Tukey algorithm, N is decomposed into a product of factors, i.e., $N = N_{m-1} \cdot N_{m-2} \cdot \ldots \cdot N_0$. This results in nested DFTs where each DFT contains the previous one. Conversely, in the binary tree representation, N is only split in two factors, i.e., $N = P \cdot Q$, which is analogous to (7). Then, both P

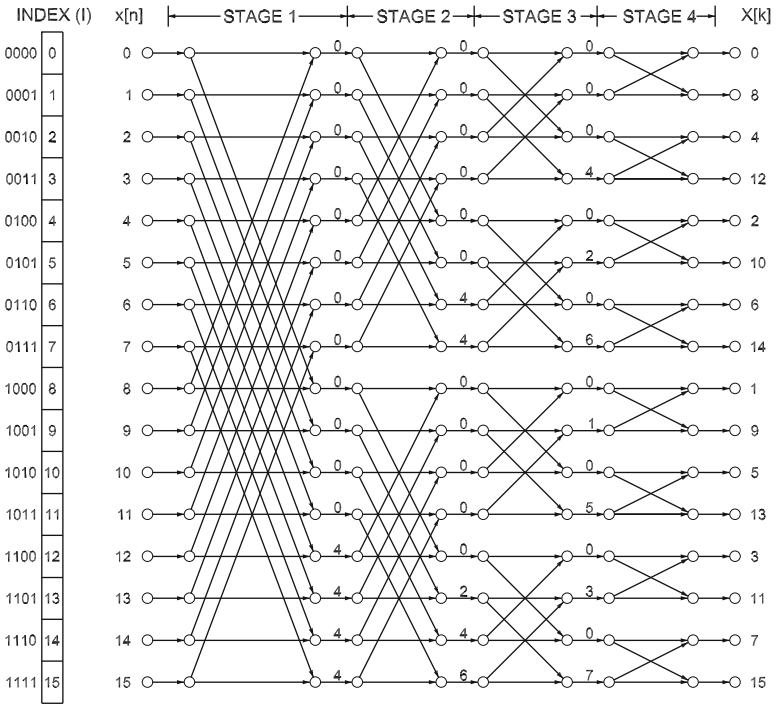


Fig. 3 Flow graph of a 16-point radix-2 DIT FFT

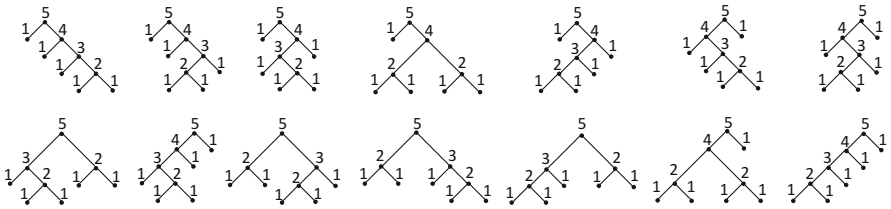


Fig. 4 Binary tree diagram of all possible algorithm for $N = 32$

and Q are again decomposed in two factors each. This process repeats iteratively forming a tree in which each node is split in two unless it is a leaf node.

A binary tree diagram [61] is an effective way of understanding the difference between FFT algorithms. For instance, Fig. 4 shows all the binary tree representations for $N = 32$. Note that each node has at most two branches.

In the binary tree representation, the upper node is assigned a value n to represent the 2^n-point DFT. Then, n is split into p and q, where $n = p + q$, $P = 2^p$ and $Q = 2^q$. After the first iteration, the remaining DFTs are again divided into smaller DFTs using the same procedure, so that the value of a node is the same as the sum of the sub-nodes.

The values of the sub-nodes can be chosen arbitrarily at each iteration. This leads to a large number of alternatives. In general, for $N = 2^n$, the number of N-point

FFT algorithms generated by using binary trees is [83]

$$\frac{(2(n-1))!}{n!(n-1)!}, \tag{10}$$

which comes from all the possible selections at each iteration. For a 32-point FFT, the number of FFT algorithms according to (10) is 14, which corresponds to the cases in Fig. 4.

The twiddle factors at each stages are directly obtained from the binary tree. This is done by going across the binary tree from left to right. In this process, the final leafs of the tree represented by 1 are skipped, as they correspond to radix-2 DFT operations. For example, the sequence of numbers in Fig. 4a is 5, 4, 3, 2. Given this sequence, the number of angles of the corresponding twiddle factors is the power of these numbers. Thus, for this example the twiddle factors are W_{32}, W_{16}, W_8, and W_4. This corresponds to the DIF decomposition, as it is obtained by decimating a 2-point DFT at each iteration from the input samples towards the output frequencies. How to generate the rotation coefficients ϕ for any FFT stage is described in [83].

2.4 Triangular Matrix Representation

The triangular matrix representation [24] is based on the ideas that rotations can be moved among FFT stages. The triangular matrix representation of some typical 16-point FFT algorithms is shown in Fig. 5. Rows are numbered as $x = 1, \ldots, n-1$ from top to bottom and columns as $y = 1, \ldots, n-1$ from left to right. Each element in row x and column y, M_{xy}, corresponds to a set of rotations that can be moved through several stages. Its value is the stage where these rotations are placed, which must be in the range $x \leq M_{xy} \leq y$.

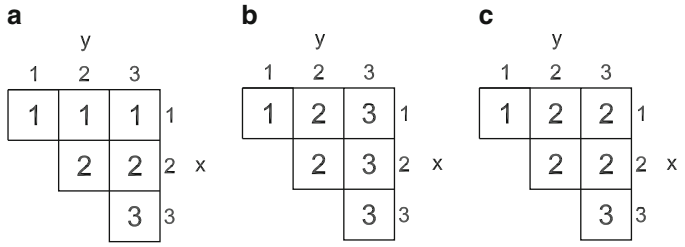


Fig. 5 Triangular matrix representation of typical 16-point FFT algorithms: (**a**) radix-2 DIF, (**b**) radix-2 DIT, (**c**) radix-2^2 DIF

Accordingly, the rotation coefficients $\phi_s(I)$ at any FFT stage s are calculated as

$$\phi_s(I) = \sum_{M_{xy}=s} b_{n-x} \cdot b_{n-y-1} \cdot 2^{n+(x-y)-2}. \tag{11}$$

Note that the numbering of rows and columns differs from the original paper [24], where the variables $i = n - x$ and $j = n - y - 1$ are used instead of x and y.

By applying Eq. (11) to Fig. 5a, the rotation coefficients are obtained as

$$\begin{aligned} \phi_1(I) &= b_3 \cdot b_2 \cdot 2^2 + b_3 \cdot b_1 \cdot 2^1 + b_3 \cdot b_0 \cdot 2^0 = b_3 \cdot [b_2 b_1 b_0]; \\ \phi_2(I) &= b_2 \cdot b_1 \cdot 2^2 + b_2 \cdot b_0 \cdot 2^1 = b_2 \cdot [b_1 b_0 0]; \\ \phi_3(I) &= b_1 \cdot b_0 \cdot 2^2 = b_1 \cdot [b_0 00], \end{aligned} \tag{12}$$

which correspond to the radix-2 DIF algorithm in Fig. 2. For instance, if $s = 1$ and $I = 14 \equiv 1110 = b_3 b_2 b_1 b_0$, then according to (12), $\phi_1(14) = b_3 \cdot [b_2 b_1 b_0] = 1 \cdot [110] = 6$. This corresponds to the rotation by 6 for $s = 1$ and $I = 14$ in Fig. 2. The rotations for the other algorithms in Fig. 5 can be derived in the same way.

In the triangular matrix representation, the radix-2 DIF FFT is the case where all the rotations are in the lowest possible stage, i.e., $\forall x, y, M_{xy} = x$. Analogously, the radix-2 DIT FFT is the case where all the rotations are in the highest possible stage, i.e., $\forall x, y, M_{xy} = y$. Finally, the radix-2^2 DIF FFT is the case where the rotations of the radix-2 DIF algorithm in odd stages are moved to the next even stage, except for those in the main diagonal, which cannot be moved.

Given that $x \leq M_{xy} \leq y$, each element M_{xy} can take $y - x + 1$ different values. By multiplying all the alternatives, the total number of algorithms that are represented by the triangular matrix as a function of n is

$$\prod_{k=1}^{n-1} (n-k)^k. \tag{13}$$

This is a large number of algorithms that includes, among others, all the algorithms representable by a binary tree.

2.5 The Radix in FFTs

The concept of radix has been used since the beginning of the FFT. It serves to distinguish different FFT algorithms. The radix is represented with a base and an exponent, i.e., $r = \rho^\alpha$. The base ρ indicates the size of the butterflies, i.e., the smallest DFT size that is used for the decomposition. Both base and exponent together provide the rotations at the FFT stages.

The first radices to be considered were radix-2, radix-4 and higher powers of two. These algorithms can be derived by the Cooley-Tukey algorithms as in (8).

Split radix [20] was also proposed in the early days. It combines radix-2 and radix-4 into an L-shaped butterfly. Split radix is known for having the least number of non-trivial rotations among FFT algorithms. However, it is seldom used in FFT hardware architectures due to the irregularity in the distribution of rotation.

Some years later, radix-2^2 was introduced for FFT hardware architectures [45]. From the algorithmic point of view, radix-2^2 is the same algorithm as radix-4 as both carry out exactly the same arithmetic operations [24]. However, both radices lead to different FFT architectures, due to the fact that radix-2^2 groups the calculations into 2-point butterflies and radix-4 groups them into 4-point butterflies.

The binary tree decomposition provides radix-2^k, for $k \in \mathbb{N}$. Radix-2^k is obtained when any node v of the binary tree is split into k and $v - k$, being k constant. Notice that the concept of radix is not unique any more, as radix-2^k may refer to different trees. Furthermore, many FFT algorithms based on the binary tree cannot be described by a single radix, but by a *mixed radix*, such as radix-2^4,2^3. As a result, many algorithms are better referred to by their tree than by their radix. The same happens to the triangular matrix representation, where there are even algorithms that cannot be described by a radix.

2.6 Non-power-of-two and Mixed-Radix FFTs

Most of the FFT designs consider FFT sizes that are powers of two and the rest of the chapter focuses on them. However, there are cases when N is a non-power-of-two and/or is a combination of powers of different radices [11, 88, 105, 106, 111, 112].

When N is a power of the radix, i.e., $N = r^k$, the Cooley-Tukey algorithm is the most suitable approach, even when it is a non-power-of-two. When N is a product of powers of coprime numbers, i.e.,

$$N = \prod_i r_i^{k_i}, \tag{14}$$

where r_i and r_j are coprime $\forall i \neq j$, then the Cooley-Tukey algorithm leads to twiddle factors between blocks of different radices. Conversely, the twiddle factors in between those blocks do not appear when using the prime factor algorithm (PFA) [6, 38]. Therefore, the PFA algorithm is recommended under these circumstances.

3 Building Blocks for FFT Hardware Architectures

FFT hardware architectures consist of butterflies, rotators and circuits for data shuffling. In the architectures, butterflies/rotators may be used to calculate one or several of the butterflies/rotations of the flow graph. Circuits for data shuffling are used to generate the data order to the butterflies and rotators in the architecture.

3.1 Butterflies

Butterflies in FFT architectures are characterized by their radix. A radix-ρ butterfly is a circuit with ρ inputs and ρ outputs that calculates an ρ-point DFT. Therefore, it corresponds to a direct implementation of the ρ-point DFT flow graph, where each addition/rotation in the flow graph is translated into an adder/rotator.

Radix-ρ butterflies are used in radix-ρ^α FFTs, $\alpha \geq 0$. The most common radices for butterflies are radix-2 and radix-4, which cover all radix-2^k and radix-4^k FFTs. Radix-2 and radix-4 butterflies have the advantage that they only consist of adders, and no rotator is needed. In case of radix-4, its trivial rotation by $-j$ can be embedded by changing the routing of the signals and the signs in the butterfly operations. Conversely, higher-radix butterflies include non-trivial rotators, which increases their cost.

3.2 Rotators

In a digital system with complex signals, a rotation by an angle α can be described as

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} C & -S \\ S & C \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}, \tag{15}$$

where $X + jY$ is the result of the rotation and $C, S \in \mathbb{Z}$ are the real and imaginary part of the rotation coefficient $C + jS$.

Due the finite word length effects, the rotation by $C + jS$ provides an approximation of the angle α with a certain error, being

$$\begin{aligned} C &= R(\cos\alpha + \epsilon_c); \\ S &= R(\sin\alpha + \epsilon_s), \end{aligned} \tag{16}$$

where ϵ_c and ϵ_s are the relative approximation errors of the cosine and sine components, respectively, and R is the magnitude. The approximation error for a given rotation can then be calculated as [30]

$$\epsilon = \sqrt{\epsilon_c^2 + \epsilon_s^2}. \tag{17}$$

Here, it should be noted that although it is common that R is a power-of-two, any magnitude can be used for the rotators, as long as the magnitude is the same for all the rotators at the same FFT stage [34].

It is often common to map the angle range into one or two quadrants. In this way, the rotators can often be simplified at the cost of a W_2 or W_4 rotator at the end. Which in turn sometimes may be integrated with the preceding butterfly as discussed above. Hence, in the following, the quadrant discussion is sometimes neglected.

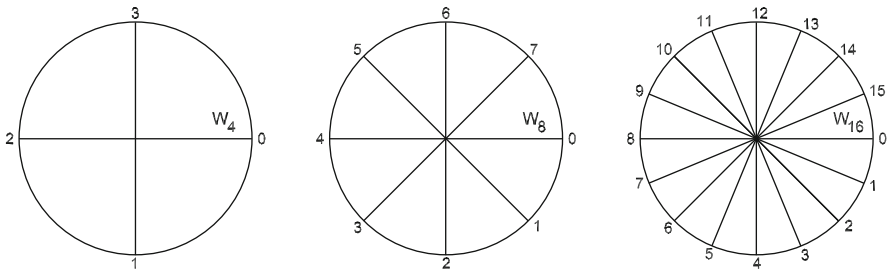


Fig. 6 Twiddle factors W_4, W_8 and W_{16}

The sine and cosine values are typically either stored in a memory or computed using an approximation, e.g., one of the methods discussed in the chapter "Arithmetic" [44]. The size of the sine and cosine memory can be reduced by utilizing octave symmetry, i.e.,

$$\begin{aligned} \sin(\alpha) &= -\cos\left(\alpha + \tfrac{\pi}{2}\right) \\ \cos(\alpha) &= \sin\left(\alpha + \tfrac{\pi}{2}\right) \end{aligned}. \tag{18}$$

In this way, only sin and cos values for inputs between 0 and $\frac{\pi}{4}$ must be stored or approximated.

Rotators in FFT hardware architecture usually calculate rotations by different angles at different time instants. These rotation angles are part of a the set of rotations

$$W_L^{\phi} = e^{-j\frac{2\pi}{L}\phi},\ \phi \in \{0, 1, \ldots, L-1\}, \tag{19}$$

where L is the resolution of the twiddle factor. Note that W_L refers to the entire set of L angles, whereas W_L^{ϕ} refers to a single angle which is the ϕ-th angle of the set. Any twiddle factor W_L divides the circumference in L equal parts and it contains the angles that create these divisions. The twiddle factors W_4, W_8 and W_{16} are shown in Fig. 6.

The number of different rotation angles that a rotator has to compute depends on the selected algorithm and architecture. When this number is large, general rotators are used. When the number of angles is small, the rotators may be simplified. Next sections describe different techniques to implement general and simplified rotators. They are mainly based on the use of multipliers or the CORDIC algorithm. An overview of techniques to implement rotators can be found in [73].

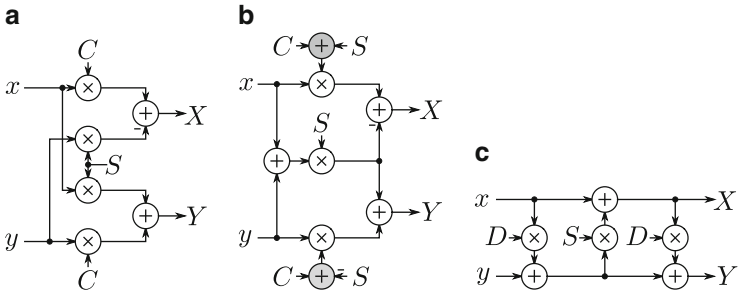


Fig. 7 Multiplier-based rotators. (**a**) Standard complex multiplier using four multiplications and two additions based on (20). (**b**) Complex multiplier using three multiplications and five additions based on (21) (three additions if the gray ones are replaced by memories). (**c**) Lifting-based rotator

3.2.1 Multiplier-Based General Rotators

The most straightforward approach is the direct implementation of Eq. (15), i.e.,

$$\begin{aligned} X &= xC - yS; \\ Y &= xS + yC. \end{aligned} \tag{20}$$

This requires four real-valued multipliers and two real-valued additions, as shown in Fig. 7a.

C and S are generally obtained as $C = \lfloor R\cos\alpha \rceil$ and $S = \lfloor R\sin\alpha \rceil$, where $\lfloor\cdot\rceil$ represents a rounding operation and being R a power of 2. Allowing R to be a non-power-of-two, widens the search for efficient rotation coefficients and the approximation errors can be reduced [30].

Other alternatives are based on rewriting (20) [101] to reduced the number of multipliers from four to three. Among them, the more interesting ones are

$$\begin{aligned} X &= x(C + S) - (x + y)S; \\ Y &= y(C - S) + (x + y)S, \end{aligned} \tag{21}$$

and

$$\begin{aligned} X &= (x + y)C - y(C + S); \\ Y &= (x + y)C - x(C - S). \end{aligned} \tag{22}$$

Both cases include a common term in the equations for X and Y that only need to be calculated once. Thus, when $C + S$ and $C - S$ are precomputed, these cases require three real-valued multiplications and three real-valued additions. Otherwise, they require three real-valued multiplications and five real-valued additions. The structure for (21) is shown in Fig. 7b.

Lifting-based rotators [8, 41, 79] are another way to obtain low-complexity rotations. There exist several different ways to rewrite (15) using lifting, one being

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} 1 & D \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ S & 1 \end{bmatrix} \begin{bmatrix} 1 & D \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}, \tag{23}$$

where $S = \sin(\alpha)$ and $D = \frac{1-\cos(\alpha)}{\sin(\alpha)} = \tan(\alpha/2)$. This requires three real-valued multiplications and three real-valued additions as shown in Fig. 7c. The other three standard alternatives have a similar form, but differ in how the coefficient corresponding to D is derived. Based on the angle, a structure can be selected to make sure that $|D| \leq \frac{1-\cos(\frac{\pi}{4})}{\sin(\frac{\pi}{4})} \approx 0.414$. This avoids the large magnitudes obtained for certain angles, as in the example structure $D \to \infty$ when $\alpha \to \pi$.

3.2.2 Multi-Stage General Rotators

As opposed to the multiplier-based rotators in the previous section, multi-stage rotators perform only a part of the total rotation in each stage. Step k of such a rotator can rotate with a set of angles, typically $\delta_k \alpha_k$, where $\delta_k \in \{-1, 1\}$ or $\delta_k \in \{-1, 0, 1\}$, i.e., a fixed angle, α_k, can be rotated in either direction or rotated in either direction or no rotation at all. Based on the remaining angle to be rotated, z_k, δ_k is determined and the remaining angle after the rotation, z_{k+1}, is updated.

The general structure of a rotation stage includes the calculation of x_{k+1}, y_{k+1} and z_{k+1} and is shown in Fig. 8. Multi-stage rotators mainly involve the CORDIC algorithm and its variations.

The total angle of rotation is then

$$\alpha = \sum_{k=0}^{M} \delta_k \alpha_k + \epsilon_\phi, \tag{24}$$

where ϵ_ϕ is the phase error of the approximation.

The available techniques optimize for different goals. For example, the CORDIC algorithm initially discussed, selects the angle of rotation such that the rotator becomes simple. A later discussed technique instead selects the angles to be suited for FFT computations with length power-of-two, i.e., the angle resolution is on a grid with power-of-two resolution.

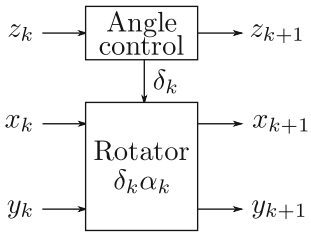


Fig. 8 General rotation stage in multi-stage rotator

In the CORDIC algorithm [97], discussed from a general perspective in the chapter "Arithmetic" [44], each stage multiply with the coefficients $C_k + jS_k = 2^k + j\delta_k$, where $\delta_k \in \{-1, 1\}$. The corresponding angles are then

$$\alpha_k = \tan^{-1}\left(2^{-k}\right). \tag{25}$$

These angles have the property that any angle α can be expressed as a sum of them. This enables the CORDIC algorithm to rotate by any rotation angle. Furthermore, due to the simplicity of the coefficients $C_k + jS_k$, each rotation stage is calculated with only 2 adders/subtracters as

$$\begin{aligned} x_{k+1} &= x_k C_k - y_k S_k = 2^k x_k - \delta_k y_k; \\ y_{k+1} &= x_k S_k + y_k C_k = 2^k y_k + \delta_k x_k. \end{aligned} \tag{26}$$

According to this, the scaling of each rotation stage is

$$R_k = \sqrt{C_k^2 + S_k^2} = C_k\sqrt{1 + (S_k/C_k)^2} = C_k\sqrt{1 + \tan(\alpha_k)^2} = \frac{2^k}{\cos(\alpha_k)}. \tag{27}$$

The term 2^k is generally compensated by removing the k LSBs after each rotation stage. The product of the scalings by $1/\cos(\alpha_k)$ at all the stages produces a total scaling of approximately 1.64, which can be compensated by multiplying the output of the CORDIC rotator by

$$K = \prod_{k=0}^{M} \cos(\alpha_k) = \prod_{k=0}^{M} \cos(\tan^{-1}(2^{-k})) = 0.6073. \tag{28}$$

The CORDIC assumes that the initial angle $z_0 = \alpha$ is in the interval $[-90°, 90°]$. Otherwise, this is easily achieved by a trivial rotation by 180°. Then, direction of the rotations δ_k are calculated for $k = 0, \ldots, M$ according to

$$\begin{aligned} \delta_k &= -\text{sign}(z_k); \\ z_{k+1} &= z_k + \delta_k \alpha_k, \end{aligned} \tag{29}$$

where z_k is the remainder rotation angle at the input of stage k, $z_{M+1} = \epsilon_\phi$, and $\text{sign}(\eta) = 1$ if $\eta \geq 0$ and $\text{sign}(\eta) = -1$ if $\eta < 0$.

There are multiple variations of the CORDIC algorithm. Some of the main modifications to the CORDIC algorithms are introduced next, and surveys on CORDIC techniques can be found in [2, 76]. For some of the mentioned approaches it is not straightforward to determine the rotation parameters at run-time. Hence, for these methods it is required to perform the design offline and store the control signals in memory rather than the angles. This approach is naturally possible for all techniques, and, as the sequence of angles is often known beforehand, most likely advantageous compared to storing the angle values.

The redundant CORDIC [63, 91] considers that $\delta_k \in \{-1, 0, 1\}$ [91] or even $\delta_k \in \{-2, -1, 0, 1, 2\}$ [63]. This enables several rotation angles at each CORDIC stage. However, the scaling for different angles is different, which demands a specific circuit for scaling compensation. The extended elementary angle set (EEAS) CORDIC [104] and the mixed-scaling-rotation (MSR) CORDIC [66, 81] also follow the idea of increasing the number of rotation angles per rotation stage.

The memoryless CORDIC [27] removes the need for rotation memory to store the FFT rotation angles. Instead, the control signals δ_k are generated from a counter. This is advantageous for large FFTs, which have stages with a large number of rotations.

The modified vector rotational (MRV) CORDIC [103] allows for skipping and repeating CORDIC stages, whereas the hybrid CORDIC [47, 89] divides the rotations into a coarse and a fine rotations. These techniques reduce the number of stages and, therefore, the latency of the CORDIC.

The CORDIC II [33] proposes new types of rotation stages: Friend angles, uniformly scaled redundant (USR) CORDIC and nanorotations. They allow for both a low latency and a small number of adders.

Finally, the base-3 rotators [57] consider an elementary angle set that is different to that of the CORDIC. All the rotations are generated by combining a small set of FFT angles. This set fits better the rotation angles of the FFT than that of the CORDIC, which results in a reduction in the rotation error, number of adders and latency of the circuit.

3.2.3 Simplified Multiplier-Based Rotators

Naturally, the real-valued multipliers in Sect. 3.2.1 can be implemented using shift-and-add multiplication, as discussed in the chapter "Arithmetic" [44]. This is specially useful when the rotator only needs to rotate by a single angle.

Initially, consider a rotation by $\frac{\pi}{4}$. As $\sin\left(\frac{\pi}{4}\right) = \cos\left(\frac{\pi}{4}\right)$ only one multiplication coefficient is required for each input (or output). Hence, either each input is multiplied by $\sin\left(\frac{\pi}{4}\right)$ and then the results are added and subtracted, as shown in Fig. 9a, or the inputs are first added and subtracted and then multiplied by $\sin\left(\frac{\pi}{4}\right)$, as shown in shown in Fig. 9b. The constant multiplication can be implemented using a optimal single constant multiplier from [42], while the best shift-and-add approximation with a given number of additions for the exact coefficient $\sin\left(\frac{\pi}{4}\right)$ can be found using the approach in [43].

For a general angle, the rotator shown in Fig. 7a can be used as a starting point. Now, both the multipliers sharing the same input can be simultaneously realized using multiple constant multiplication (MCM) techniques. This is illustrated in Fig. 10a, where the dashed box indicates the two multipliers realized using MCM. An identical MCM block is used for the other input. General MCM algorithms include [40, 98], while the algorithm in [18] is specifically tailored for two constants. An alternative view of the problem is to implement a sum-of-product block for the two multipliers going to the same output, as illustrated in Fig. 10b. However,

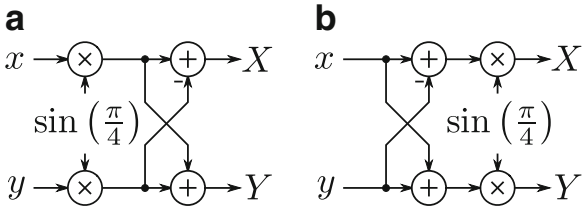


Fig. 9 Single constant rotations by $\pi/4$. (**a**) Multiplication followed by addition. (**b**) Addition followed by multiplication

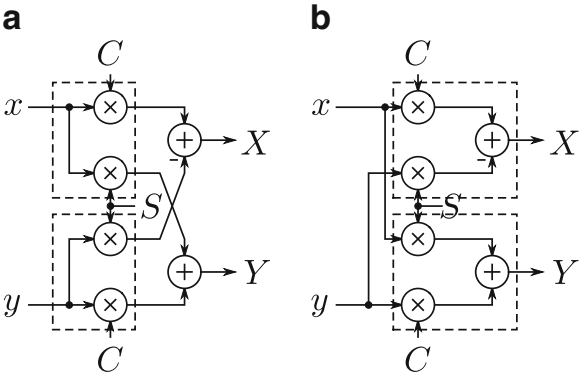


Fig. 10 Single constant rotations by a general angle. (**a**) Rotation implemented by using two MCM blocks as indicated by the dashed box. (**b**) Rotation implemented using two sum-of-product blocks as indicated by the dashed box

as the MCM and the sum-of-product problems are dual to each other, exactly the same number of adders are expected. A third option is to consider the problem as a constant matrix multiplication problem [5, 60].

For the rotators in Fig. 7b, c, no sharing can be done between the multipliers. However, due to the initially reduced complexity of the rotator, it may still happen that the total complexity is reduced. It should also be noted that the computations of the lifting-based structure in (23) can be merged to one matrix.

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} E & F \\ S & E \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}, \tag{30}$$

where $E = 1 + DS$ and $F = D(2 + SD)$. This forms another option to realize in any of the ways mentioned above.

When more than one angle is considered it is still possible to use shift-and-add techniques. Consider a W_8 rotator with indices $\phi \in \{0, 1\}$. For $\phi = 0$, the inputs are simply bypassed to the outputs. For $\phi = 1$ one of the approaches in Fig. 9 can be used. Then, the correct rotation result is selected by using a multiplexer, as shown in Fig. 11, where the $\frac{\pi}{4}$-rotator in Fig. 9a is used.

For several non-trivial angles, the naïve approach is to implement all different coefficients by using shift-and-add as an MCM problem and then select the correct angle by a multiplexer. However, the multiplexers can be merged with the shift-and-add network to significantly reduce the complexity [77, 96].

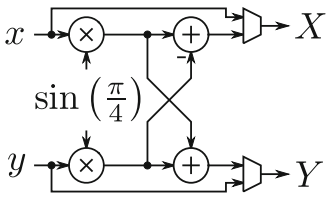


Fig. 11 W_8 rotator based on Fig. 9a

Of special interest is to note that it may be possible to find coefficients with longer word length, but with the same or smaller number of adders [34, 43]. Also, as earlier discussed, the magnitude may be selected as a non-power-of-two to further simplify the computations [34].

3.2.4 Simplified Multi-Stage Rotators

All the techniques mentioned in Sect. 3.2.2 are based on simple rotator stages. Clearly, for a constant coefficient the best selection of stages can be found. This has been explicitly proposed for CORDIC-based rotators [48, 75], although it can be easily generalized to arbitrary types of stages. If several angles are to be realized at different time instances it is of benefit having similar structure for the stages such that multiplexers can be easily introduced.

3.2.5 Rotators Based on Trigonometric Identities

Approaches based on trigonometrical identities [78, 84] search for expressions that are shared among different rotation angles. As a result, a simplified version of the rotator is obtained, which includes a reduced number of adders, multiplexers and multiplications by real constants. For instance, the twiddle factor W_{16} can be reduced to constant multiplications by $\cos(\pi/8)$ and/or $\sin(\pi/8)$ [84].

3.3 Shuffling Circuits

The purpose of the shuffling circuits in FFT architectures is to provide the data in the correct order needed for FFT stages. At each FFT stage, butterflies operate on pairs of data whose index I differ in b_{n-s} [29]. This can be observed in Fig. 2. For instance, the index of pairs of inputs to butterflies in stage 1 differs in $b_{n-s} = b_{4-1} = b_3$. As $I \equiv b_3b_2b_1b_0$, $I = 0 \equiv 0000$ and $I = 8 \equiv 1000$ differ in b_3 and are processed together in a butterfly at the first stage.

As different FFT stages demand different data orders, circuits for data permutation need to be included in between stages. These circuits have been studied by using life-time analysis and register allocation [74, 80], Kronecker products [39, 52–54, 82, 87, 92, 93] and bit-dimension permutations [21–23]. The following explanation is based on the latter, and follows the theory in [23].

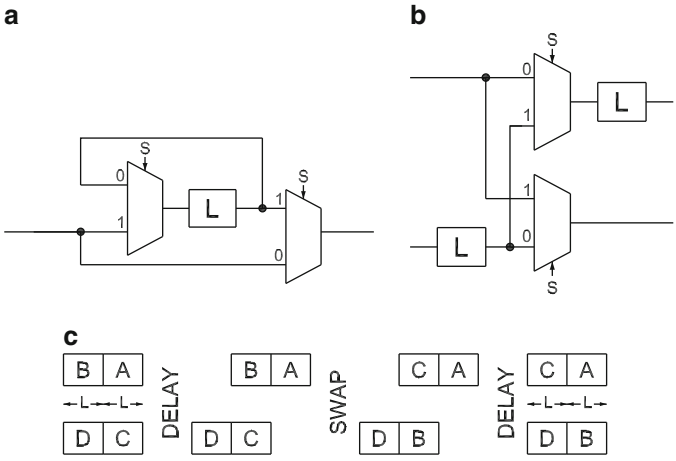


Fig. 12 Shuffling circuits. (**a**) Serial-serial permutation. (**b**) Serial-parallel permutation. (**c**) Dataflow of the serial-parallel permutation

Bit-dimension permutations are permutations on a group of 2^n data defined by a permutation of the n bits that represent the index of those data in binary. In FFTs, each stage processes $N = 2^n$ data that we index with $I = b_{n-1} \dots b_1 b_0$ as in Fig. 2. For these data, we can define their position [23]. For instance, $\mathrm{P} \equiv b_0 b_3 b_2 b_1$ defines a specific data order, where each index $I \equiv b_3 b_2 b_1 b_0$ is in position P. Thus, $I = 8 \equiv 1000$ is in $\mathrm{P} \equiv 0100$. For parallel data, a vertical bar separates serial and parallel data, e.g., $\mathrm{P} \equiv b_0 b_3 b_2 | b_1$. The first part of the position until the bar indicates the time of arrival, which is defined as the relative time to the arrival of the first sample to a given point of the circuit. The second part after the bar indicates the terminal of arrival. The number of parallel dimensions p corresponds to the number of bits after the bar. Thus, for $\mathrm{P} \equiv b_0 b_3 b_2 | b_1$, the number of parallel dimensions is $p = 1$ and $I = 8 \equiv 1000$ is in $\mathrm{P} = 010|0$, i.e., it arrives at terminal $T(\mathrm{P}) = 0$ at time $t(\mathrm{P}) = 2 \equiv 010$.

Different positions define different orders, and the data order is changed when permuting the bits of the position. This is achieved by the shuffling circuits in FFT architectures. Figure 12 shows the main shuffling circuits used in FFT hardware architectures.

The circuit in Fig. 12a is used to calculate a serial-serial bit-dimension permutation. It consists of a buffer of length L and two multiplexers. This circuit is used to change the position of pairs of data separated by L clock cycles. This is done when one of the data is at the input of the circuit and the other one is at the output of the buffer. Then $S = 0$ is selected so that the position change. Otherwise, when $S = 1$ data passes through the buffer maintaining the order.

The length of the buffer defines the bit-dimension permutation that is carried out. If x_{n-1} is the first bit from the left and x_0 is the last one, then a serial-serial permutation that interchanges x_j and x_k, $j > k \geq p$ has a buffer length [23]

$$L = (2^j - 2^k)/2^p. \tag{31}$$

The circuit in Fig. 12b carries out a serial-parallel permutation. Figure 12c shows how it works: The groups of data A, B, C and D have L data in series each. First, the L data A and C arrive to the circuit at the upper and lower inputs, respectively. Then the L data B and D arrive. The circuit first delays the lower data, then it swaps the groups B and C, and finally it delays the upper part. The result is that data in groups B and C are interchanged.

The length of the buffers for interchanging x_j and x_k, $j \geq p > k$ is [23]

$$L = 2^{j-p}. \tag{32}$$

Finally, parallel-parallel permutations interchange parallel data flows. This does not require any hardware, as it can be hard wired.

4 FFT Hardware Architectures

There are three main types of FFT hardware architectures: Pipelined, iterative and fully parallel. Next section discusses when to choose each type and the following sections describe the different types.

4.1 Architecture Selection

Table 1 classifies the types of FFT hardware architectures in terms of the input data flow and the number of parallel samples, P. The higher P, the higher the throughput and also the larger the area of the circuit.

Iterative FFT hardware architectures loads data into a memory, then processes them and finally outputs them. During the processing, new data cannot be loaded. Therefore, iterative FFT architectures are suitable for processing data bursts.

Table 1 FFT architecture selection

FFT architecture type	Data flow	P
Iterative	Burst	≥ 1
Serial pipelined	Continuous flow	1
Parallel pipelined	Continuous flow	>1
Fully parallel	Continuous flow	N

The other architectures process data in a continuous flow. The difference among them is the parallelization P. The selection of the architecture depends on the expected performance. The throughput is calculated as $\text{Th} = P \cdot f_{\text{clk}}$ where f_{clk} is the clock frequency of the system and P is generally a power of 2. Thus, if the minimum throughput that the system must achieve is Th_{min}, then P is obtained as

$$P = 2^{\lceil \log_2 (\text{Th}_{\text{min}}/f_{\text{clk}}) \rceil}. \tag{33}$$

4.2 Fully Parallel FFT

Fully parallel FFT architectures correspond to the direct implementation of the FFT flow graph, i.e., each multiplication/addition in the flow graph is implemented by a separated multiplier/adder. Therefore, the number of hardware components is of order $O(N \log N)$. Fully parallel FFT architectures offer the maximum parallelization of the FFT algorithm. As a consequence, they provide the maximum throughput and the minimum latency among FFT architectures.

The implementation of rotators as shift-and-add and the selection of the FFT algorithm play an important role in the design of fully parallel FFTs. As each rotator calculates a rotation by a single angle, it is beneficial to use simplified rotators. Complementary to this, the selection of the FFT algorithms determines the number of non-trivial rotations in the fully parallel FFT. Therefore, algorithms with a small number of non-trivial rotations lead to more hardware-efficient implementations.

4.3 Iterative FFT Architectures

Iterative FFT architectures [15, 35, 46, 49, 55, 71, 72, 85, 93, 95] are also called memory-based or in-place FFT architectures. We suggest to call them iterative, as this is the term that reflects their nature better. Although the term memory-based is widely used, we prefer to avoid it due to the facts that non-iterative FFTs may also use memories, and iterative FFTs may use delays instead of memories.

Iterative FFT architectures generally consist of a memory or bank of data memories. Data are read from memory, processed by butterflies and rotators, and stored again in memory. This process repeats iteratively until all the stages of the FFT algorithm are calculated. The advantage of iterative FFTs is the reduction in the number of butterflies and rotators, as they are reused for different stages of the FFT.

A simple iterative FFT architecture is shown in Fig. 13. It consists of a memory and a processing element (PE), which computes the butterfly and rotation. As seen in Fig. 13, after every iteration data are stored in memory so it is necessary to compute whole FFT before it receives new samples. Thus, the memory-based architecture is unable to compute the FFT when data arrives continuously at the input. For a

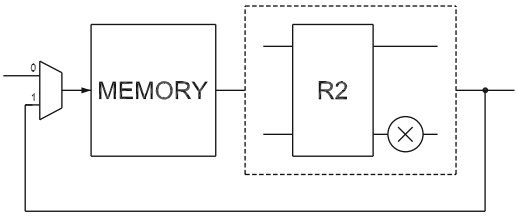


Fig. 13 Iterative (memory-based) FFT architecture

continuous flow, the iterative FFT requires additional memory to store the incoming data while the FFT is being calculated. If the processing time is longer than the time between FFTs, a larger processing element is also needed in order to handle the data flow.

Considering that the processing element is a radix-r butterfly and a set of rotators, the number of iterations of an iterative FFT architecture is calculated as

$$\mathrm{It} = \frac{\log_2 N}{\log_2 r} = \frac{n}{\log_2 r}, \tag{34}$$

and the number of clock cycles to process each iteration is N/r, which leads to a total processing time of

$$T_{\mathrm{proc}} = \frac{N \log_2 N}{r \log_2 r}. \tag{35}$$

The loading time depends on the FFT size and on the number of input samples that are loaded in parallel to the memory bank:

$$T_{\mathrm{load}} = \frac{N}{P}. \tag{36}$$

When reading output data from the memories and writing the new input data are done simultaneously, the latency of the iterative FFT in clock cycles is

$$\mathrm{Lat} = T_{\mathrm{load}} + T_{\mathrm{proc}} = \frac{N}{P} + \frac{N \log_2 N}{r \log_2 r}, \tag{37}$$

and, as N samples are processed every Lat clock cycles, the throughput in samples per clock cycle is

$$\mathrm{Th} = \frac{N}{\mathrm{Lat}} = \frac{rP}{r + P\frac{\log_2 N}{\log_2 r}}. \tag{38}$$

To increase the throughput and decrease the latency in iterative FFT architectures, high-radix processing elements are used. For instance, radix-16 is used in [49]. However, this also increases the amount of hardware of the FFT. Therefore, there is a trade-off between performance and hardware complexity.

The highest degree of parallelization for iterative FFT architectures consists in calculating simultaneously all the operations of the same stage of the FFT. This is done by the so called *column FFT* [3, 90], which computes the FFT by means of a column of processing elements composed of butterflies and rotators.. This architecture allows fast computation of the FFT, but requires a large number of hardware components, being the number of butterflies and rotators of order $O(N)$.

Finally, the radix-r butterfly in the PE processes r samples in parallel. Thus, r samples must be read from and written to memory each clock cycle. This requires to divide the memory into r memory banks that are accessed simultaneously. Furthermore, data must be written in memory in the same addresses that are read. This demands a conflict-free access strategy. Such memory organization can be studied from [15, 35, 46, 49, 55, 71, 72, 85, 93, 95].

4.4 Pipelined FFT Architectures

Pipelined FFT architectures [1, 13, 14, 17, 23, 25, 26, 29, 31, 32, 36, 45, 51, 58, 62, 64, 65, 67–69, 86, 94, 99, 107–109] consist of a set of $n = \log_\rho N$ stages connected in series, where ρ is the base of the radix $r = \rho^\alpha$. In a pipelined FFT, each stage of the architecture computes one stage of the FFT algorithm. The main advantage of pipelined architectures is that they process a continuous flow of data, with a good trade-off between performance and resources.

There are three main types of pipelined FFT architectures: feedback (FB), feedforward (FF) and serial commutator (SC). First, feedback architectures [13, 14, 17, 26, 45, 62, 64, 68, 69, 86, 94, 99, 107, 109] are characterized by their feedback loops, i.e., some outputs of the butterflies are fed back to the memories at the same stage. Feedback architectures are divided into single-path delay feedback (SDF) [17, 26, 45, 86, 109], which process a continuous flow of one sample per clock cycle, and multi-path delay feedback (MDF) [13, 14, 62, 64, 68, 69, 94, 99, 107], which process several samples in parallel. Second, feedforward architectures [1, 9, 10, 17, 25, 29, 31, 36, 45, 51, 58, 70, 86, 108] do not have feedback loops and each stage passes the processed data to the next stage. Single-delay commutator (SDC) FFTs are used for serial data [9, 10, 17, 70] and, multi-path delay commutator (MDC) FFTs [1, 25, 29, 31, 36, 45, 51, 58, 86, 108] are used to process several data in parallel. Finally, SC FFT architectures [32] are characterized by the use of circuits for bit-dimension permutation of serial data.

Pipelined FFT architectures can also be classified into serial pipelined and parallel pipelined FFT architectures. Next sections use this classification, which allows for comparing the hardware resources of FFTs with the same performance.

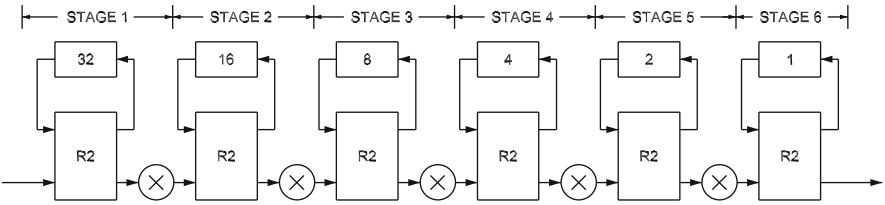
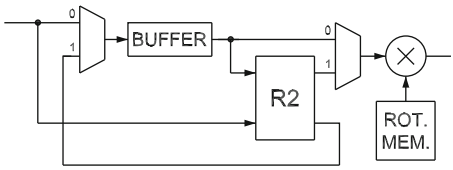


Fig. 14 64-Point SDF FFT architecture

Table 2 Twiddle factors of a 64-point DIF FFT for different radices

	FFT stage				
FFT algorithm	1	2	3	4	5
Radix-2 DIF	W_{64}	W_{32}	W_{16}	W_8	W_4
Radix-2^2 DIF	W_4	W_{64}	W_4	W_{16}	W_4
Radix-2^3 DIF	W_4	W_8	W_{64}	W_4	W_8
Radix-2^4 DIF	W_4	W_{64}	W_4	W_{16}	W_4

Fig. 15 Internal structure of a SDF stage



4.4.1 Serial Pipelined FFT Architectures

Serial pipelined FFT architectures consists of SDF, SDC and SC FFT architectures. These architectures are characterized by their relatively low number of components (adders, rotators and memory) and a throughput of 1 sample per clock cycle, which allows for high data rates of MSamples/s.

An example of SDF FFT architecture is shown in Fig. 14 for $N = 64$. It consists of $n = \log_2 N$ stages with radix-2 butterflies (R2), rotators and buffers. This architecture can implement radix-2^k FFT algorithms, including radix-2. The difference among FFT algorithms is reflected in the rotations calculated at each stage. Table 2 shows the twiddle factors for typical DIF algorithms. Note that W_4 corresponds to trivial rotations, which leads to simple rotators. Thus, the complexity of radices 2^2, 2^3 and 2^4 is smaller than that of radix-2.

The twiddle factors for DIT algorithms are the same as DIF ones, where the twiddle factor at stage s in the DIT case corresponds to that one in stage $n - s$ in the DIF case.

The internal structure of a SDF stage is shown in Fig. 15. It consists of a buffer, a radix-2 butterfly, two multiplexers, a rotator and eventually its rotation memory.

The buffer at stage s has length

$$L_s = 2^{n-s}. \tag{39}$$

The reason is that L_s input data are loaded to the buffer, causing a delay of those data. After L_s clock cycles, the output of the buffer is processed in the butterfly together with the input data for L_s clock cycles. During these clock cycles, one output of the butterfly is sent to the rotator and the other output of the butterfly is fed back to the buffer. Later, the values that go through the buffer are sent to the rotator, while a new sequence is loaded to the buffer. Therefore, the buffer is reused for inputting and outputting data.

This data management can be related to the data flow of the FFT algorithm: If we consider data arriving in series from top to bottom in the data flow of Fig. 2 then, at each stage, groups of L_s data must be delayed L_s clock cycles. This makes them arrive to the input of the butterfly at the same time as the data that they have to be operated with. For instance, if sample $x[0]$ at stage $s = 1$ is delayed $L_s = 2^{n-s} = 2^{4-1} = 8$ clock cycles, then it may be input to the butterfly together with $x[8]$. After the butterfly calculation, data are ordered in series again by delaying the lowest output of the butterfly L_s clock cycles.

In terms of hardware components, the radix-2^k SDF FFT requires one butterfly per stage, which results in $2\log_2 N$ complex adders, a total memory of $N - 1$, which is the sum of all the buffer lengths, and a number of rotators that depends on the algorithm itself.

Figure 16 shows a 64-point radix-4 SDF FFT architecture. In this case, the number of stages is $\log_\rho N = \log_4 64 = 3$. The data management is analogous to the radix-2^k SDF FFT: Data are delayed so that all 4 data into a butterfly arrive at the same clock cycle.

At each stage, the radix-4 butterfly requires 8 complex adders, leading to a total of $8(\log_4 N)$ complex adders for the entire FFT. For the data, radix-4 uses 3 memories of size 4^{n-s} at stage s, which leads to a total FFT memory of $N - 1$.

The second type of serial FFT architectures is SDC [9, 10, 17, 70]. It is based on separating the data stream in two parallel data streams with the real and imaginary parts of the samples, respectively. An explanation of radix-2^k SDC and SDF architectures can be found in [17]. Generally, SDF FFTs are preferred to SDC ones, due to the larger data memory in SDC FFT architectures.

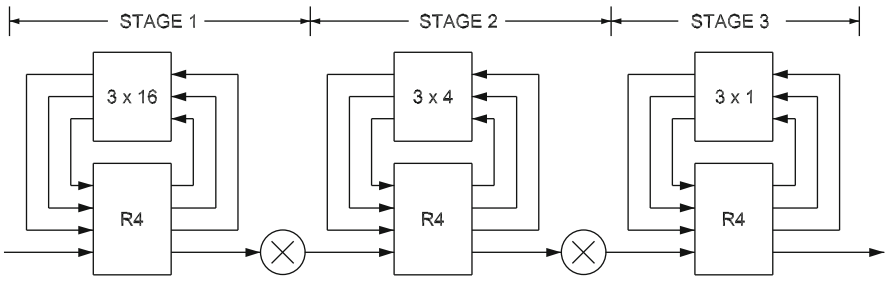


Fig. 16 64-Point radix-4 SDF FFT architecture

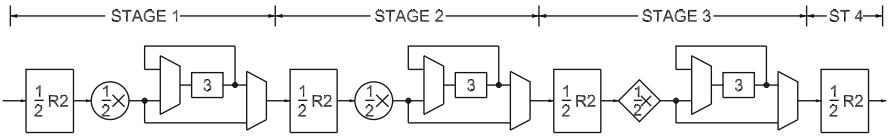


Fig. 17 16-Point radix-2 DIF SC FFT architecture

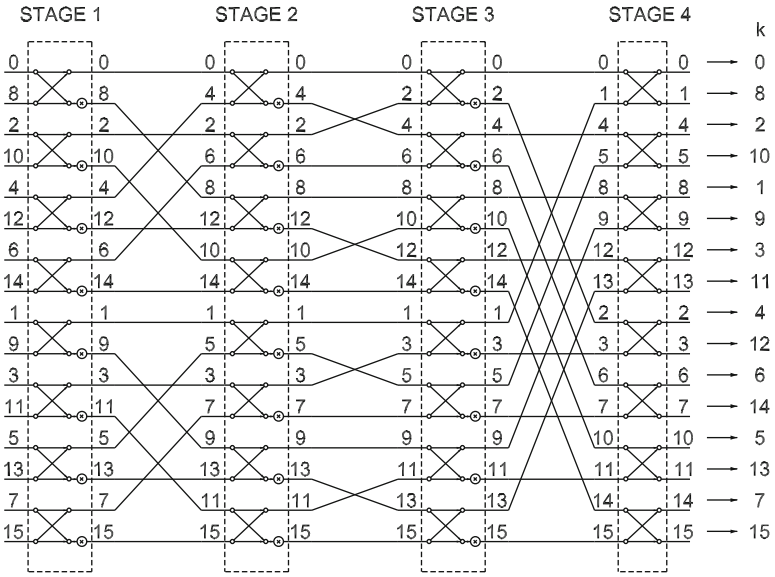


Fig. 18 Data management of the 16-point radix-2 DIF SC FFT

The third type of serial FFT architectures is SC [32]. Figure 17 shows a 16-point DIF serial commutator FFT. It uses circuits for bit-dimension permutation of serial data, which were described in Sect. 3.3.

The SC FFT is based on the idea of placing in consecutive clock cycles pairs of data that must be processed in the butterflies. Figure 18 shows the data management of the SC FFT in Fig. 17. The last column in Fig. 18 shows the frequencies k of $X[k]$. The rest of the numbers represent the data index I according to the definition in Fig. 2. The order of arrival to each FFT stage is from top to bottom. Therefore, $x[0]$ and $x[8]$ are the first and second inputs to the first stage, respectively. Butterflies operate on consecutive data. This allows for calculating the butterflies in two clock cycles, which halves the hardware complexity with respect to SDF FFTs. Note that the architecture in Fig. 17 uses half-butterflies (1/2 R2) instead of (R2). Rotators are also calculated in two clock cycles and its hardware is halved. The shuffling circuits in Fig. 17 delay three, one, and seven clock cycles, respectively. This can be observed in Fig. 18, where data that are exchanged are separated these numbers of clock cycles at the corresponding stages. Further details are shown in [32].

Table 3 Comparison of pipelined serial FFT architectures

	Area			Performance	
Pipelined architecture	Complex rotators	Complex adders	Complex sample memory	Latency (cycles)	Throughput (samples/cycle)
Radix-2 SDF [45]	$2(\log_4 N - 1)$	$4(\log_4 N)$	N	N	1
SDF Radix-2 [109]	$\log_4 N - 1$	$2(\log_4 N)$	$4N/3$	$4N/3$	1
SDF Radix-4 [19, 86]	$\log_4 N - 1$	$8(\log_4 N)$	N	N	1
SDF Radix-2^2 [45]	$\log_4 N - 1$	$4(\log_4 N)$	N	N	1
SDF Split-radix [110]	$\log_4 N - 1$	$4(\log_4 N)$	N	N	1
SDC Radix-2 [9, 10]	$2(\log_4 N - 1)$	$2(\log_4 N)$	$3N/2$	$3N/2$	1
SDC Radix-2 [70]	$2(\log_4 N - 1)$	$2(\log_4 N)$	$3N/2$	$3N/2$	1
SDC Radix-4 [4]	$\log_4 N - 1$	$3(\log_4 N)$	$2N$	N	1
SDC-SDF Radix-2 [100]	$\log_4 N - 1$	$2(\log_4 N) + 1$	$3N/2$	$3N/2$	1
SC Radix-2 [32]	$\log_4 N - 1$	$2(\log_4 N)$	N	N	1

Table 3 compares serial pipelined N-point FFT architectures. The table shows the trade-off between area and performance. Area is measured in terms of the number of complex rotators, adders and memory addresses, whereas performance is represented by throughput and latency. The throughput is 1 sample per clock cycle for all the architectures, which makes them comparable in terms of hardware resources.

As can be observed in the table, the order of magnitude of all parameters is the same for all architectures. The number of rotators and adders has order $O(\log N)$ and the memory has order $O(N)$. For large FFTs, the data memory takes up most of the area of the circuit, so it is preferable to use an architecture with a small memory. For small N, most of the FFT area is due to rotators, whose area is always larger than the area of the adders.

4.4.2 Parallel Pipelined FFT Architectures

This section discusses parallel FFT architectures, i.e., MDF and MDC. These architectures are characterized by their high throughputs. They can process P parallel samples in continuous flow and achieve a throughput of $\text{Th} = P \cdot f_{\text{clk}}$, reaching rates of GSamples/s.

MDF FFT architectures [13, 14, 62, 64, 68, 69, 94, 99, 102, 107] consists of multiple SDF paths in parallel. Thus, they work in a similar way as SDF FFT

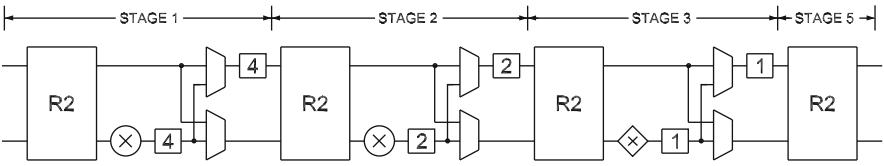


Fig. 19 16-Point 2-parallel radix-2 DIF MDC FFT architecture

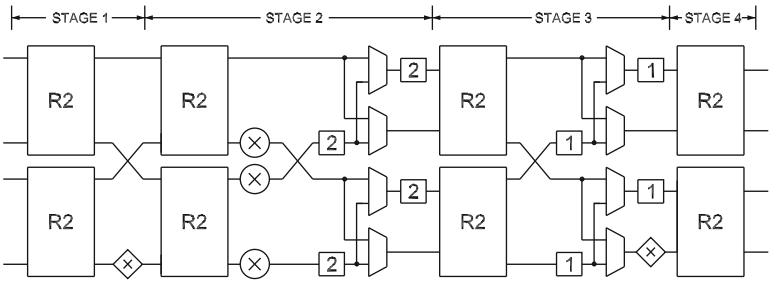


Fig. 20 16-Point 4-parallel radix-2^2 MDC FFT

architectures. The only difference is that the parallel SDF paths are interconnected either at some intermediate stages [102] or in the last stages.

MDC FFT architectures [1, 25, 29, 31, 36, 45, 51, 58, 86, 108] forward data to the next stage instead of feeding it back to the buffer of the same stage. Figure 19 shows a 16-point 2-parallel radix-2 DIF MDC FFT architecture. It consists of radix-2 butterflies, rotators and shuffling circuits for serial-parallel permutations. This architecture processes 2 samples per clock cycle in a continuous flow.

Higher throughput is achieved by increasing the parallelization. Figure 20 shows a 16-point 4-parallel radix-2^2 feedforward architecture. This architecture processes 4 samples per clock cycle in a continuous flow.

Table 4 compares parallel pipelined FFT hardware architectures. The architectures are classified into 4-parallel and 8-parallel ones. The table includes the number of adders and rotators. W_{16} and W_8 rotators are separated from general rotators due to its lower complexity. The number of complex adders is related to the architecture type: MDC FFTs have 100% utilization of butterflies and usually require less adders than MDF FFTs. The amount and complexity of the rotators depend on the radix and on the FFT size. Radices-2^4 and 2^3 are usually the best options. They achieve the least number of general rotators with some overhead of W_{16} and W_8 rotators. New approaches focus on reducing the amount rotators in parallel pipelined FFT architectures even further [31]. Finally, for most parallel pipelined FFT architectures the total data memory size is $N - P$.

Table 4 Comparison of parallel pipelined FFT architectures

Pipelined architecture	Area			
	Complex adders	Rotators		
		General	W_{16}	W_8
4-Parallel architectures				
R2 MDC, [36]	$4(\log_2 N)+4$	$2(\log_2 N)-4$	0	0
R4 MDC, [86]	$4(\log_2 N)$	$3\lceil(\log_2 N)/2\rceil-3$	0	0
R4 MDC, [108]	$4(\log_2 N)$	$3\lceil(\log_2 N)/2\rceil-3$	0	0
$R2^2$ MDC, [29]	$4(\log_2 N)$	$3\lceil(\log_2 N)/2\rceil-3$	0	0
$R2^3$ MDC, [29]	$4(\log_2 N)$	$4\lceil(\log_2 N)/3\rceil-4$	0	$2\lfloor(\log_2 N)/3\rfloor$
$R2^4$ MDF, [99]	$4(\log_2 N)$	$4\lceil((\log_2 N)-2)/4\rceil-1$	$4\lfloor((\log_2 N)-2)/4\rfloor$	(2)[a]
$R2^4$ MDF, [68]	$8(\log_2 N)$	$4\lceil(\log_2 N)/4\rceil-4$	$4\lfloor(\log_2 N)/4\rfloor$	(1)[b]
$R2^4$ MDF, [14]	$8(\log_2 N)$	$4\lceil(\log_2 N)/4\rceil-4$	$4\lfloor(\log_2 N)/4\rfloor$	(1)[b]
$R2^4$ MDC, [29]	$4(\log_2 N)$	$4\lceil(\log_2 N)/4\rceil-4$	$3\lfloor(\log_2 N)/4\rfloor$	(2)[b]
8-Parallel architectures				
R2 MDC, [56]	$8(\log_2 N)$	$4(\log_2 N)-8$	0	0
R2 MDF, [102]	$16(\log_2 N)$	$4(\log_2 N)-8$	0	0
$R2^2$ MDC, [29]	$8(\log_2 N)$	$6\lceil(\log_2 N)/2\rceil-6$	0	0
R8 MDC, [86]	$8(\log_2 N)$	$7\lceil(\log_2 N)/3\rceil-7$	0	$2\lfloor(\log_2 N)/3\rfloor$
$R2^3$ MDC, [29]	$8(\log_2 N)$	$7\lceil(\log_2 N)/3\rceil-7$	0	$2\lfloor(\log_2 N)/3\rfloor$
$R2^4$ MDF, [99]	$8(\log_2 N)$	$8\lceil((\log_2 N)-3)/4\rceil-1$	$8\lfloor((\log_2 N)-3)/4\rfloor$	2+(4)[c]
$R2^4$ MDF, [94]	$16(\log_2 N)$	$8\lceil(\log_2 N)/4\rceil-8$	$8\lfloor(\log_2 N)/4\rfloor$	(2)[b]
$R2^4$ MDC, [29]	$8(\log_2 N)$	$8\lceil(\log_2 N)/4\rceil-8$	$6\lfloor(\log_2 N)/4\rfloor$	(2)[b]

[a] Additional W_8 rotators required only when $\mathrm{mod}((\log_2 N)-2,4)=3$
[b] Additional W_8 rotators required only when $\mathrm{mod}((\log_2 N),4)=3$
[c] Additional W_8 rotators required only when $\mathrm{mod}((\log_2 N)-3,4)=3$

5 Bit Reversal for FFT Architectures

The outputs of FFT hardware architectures are generally provided in bit-reversed order. The bit reversal algorithm [37] is used to sort them out.

5.1 The Bit Reversal Algorithm

The bit reversal of $N = 2^n$ indexed data is an algorithm that reorders the data according to a reversing of the bits of the index [37]. This means that any sample with index $I \equiv b_{n-1} \dots b_1 b_0$ moves to the place $\mathrm{BR}(I) \equiv b_0 b_1 \dots b_{n-1}$. Note that the bit reversal is an inversion operation, i.e., $\mathrm{BR}(x) = \mathrm{BR}^{-1}(x)$. Therefore, if data are in natural order, the bit reversal algorithm obtains them in bit-reversed order and vice versa. For instance, the bit reversal of $(0, 1, 2, 3, 4, 5, 6, 7)$ is $(0, 4, 2, 6, 1, 5, 3, 7)$ and the bit reversal of the latter set is the former.

5.2 Bit Reversal for Serial Data

For a hardware circuit that receives a series of N data in bit-reversed order, the bit reversal of the data is calculated by the permutation:

$$\sigma(u_{n-1} \dots u_1 u_0) = u_0 u_1 \dots u_{n-1}. \tag{40}$$

A first option to calculate the bit reversal of a series of data is to use a double buffering strategy [9, 59]. This consists of 2 memories of size N where even and odd FFT output sequences are written alternatively in the memories. The bit reversal can also be calculated using a single memory of size N. This is achieved by generating the memory address in natural and bit-reversed order, alternatively for even an odd sequences [7]. For SDC FFT architectures, the output reordering can be calculated by using two memories of $N/2$ addresses [9, 70]. Alternatively, the output reordering circuit can be integrated with the last stage of the FFT architecture [9, 10].

The optimum solution in terms of memory/delays for the bit reversal of serial data [28] consists of using a series of $j = \lfloor n/2 \rfloor - 1$ circuits for serial-serial bit-dimension permutations. Each of them carries out a permutation of the bits x_j and x_{n-1-j}, which requires a buffer of length

$$L = 2^{n-1-i} - 2^i. \tag{41}$$

By adding the buffer lengths, the total number of delays for even n is

$$(N-1)^2, \tag{42}$$

and for odd n it is

$$\left(\sqrt{2N}-1\right)\left(\sqrt{\frac{N}{2}}-1\right). \tag{43}$$

5.3 Bit Reversal for Parallel Data

For parallel data, the bit reversal permutation is the same as for serial data, with the difference that the p less significant bits are parallel dimensions. As for serial data, some solutions are based on using memories [50, 108] and other ones are based on using buffers [12].

The optimum solution in terms of memory/delays for the bit reversal of parallel data [12] uses circuits for serial-serial permutations and parallel-parallel permutation. As in the bit reversal of serial data, these circuits are used to interchange bits x_j and x_{n-1-j} for $j = \lfloor n/2 \rfloor - 1$. Assuming that $p < n/2$, a serial-serial permutation is carried out when $0 \leq j < p$ and a serial-parallel one when $p \leq j \leq \lfloor n/2 \rfloor - 1$. As a result, the total numbers of delays for even n is

$$D(\sigma) = N - 2\sqrt{N} + P, \tag{44}$$

and for odd n it is

$$D(\sigma) = N - \sqrt{2N} - \sqrt{\frac{N}{2}} + P. \tag{45}$$

6 Conclusions

More than 50 years after the first FFT algorithms were proposed, the design of FFT hardware architectures is still an active research field that involves multiple research topics. They include the study and selection of FFT algorithms, the design of rotators in hardware, the design of new FFT architectures and the data shuffling, including the bit reversal algorithm. Nowadays, new FFT algorithms as well as new representations for these algorithms are explored. The most common FFT algorithms are radix-2^k, and there is an increasing interest in non-power-of-two FFTs. The area in FFT architectures is reduced by implementing rotators as shift-and-add operations. Most approaches are based on simplifying a complex multiplier or on the CORDIC algorithm. New FFT hardware architectures that achieve fully utilization of butterflies and reduction of the number of rotators and their complexity have been proposed during the last years. Likewise, the optimum circuits for bit reversal have been proposed recently, and the research on shuffling circuits for the FFT is still an open research field.

References

1. Ahmed, T., Garrido, M., Gustafsson, O.: A 512-point 8-parallel pipelined feedforward FFT for WPAN. In: Proc. Asilomar Conf. Signals Syst. Comput., pp. 981–984 (2011)
2. Andraka, R.: A survey of CORDIC algorithms for FPGA based computers. In: Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays, pp. 191–200. ACM (1998)
3. Argüello, F., Bruguera, J., Doallo, R., Zapata, E.: Parallel architecture for fast transforms with trigonometric kernel. IEEE Trans. Parallel Distrib. Syst. **5**(10), 1091–1099 (1994)
4. Bi, G., Jones, E.: A pipelined FFT processor for word-sequential data. IEEE Trans. Acoust., Speech, Signal Process. **37**(12), 1982–1985 (1989)
5. Boullis, N., Tisserand, A.: Some optimizations of hardware multiplication by constant matrices. IEEE Trans. Comput. **54**(10), 1271–1282 (2005)
6. Burrus, C., Eschenbacher, P.: An in-place, in-order prime factor FFT algorithm. Proc. IEEE Int. Symp. Circuits Syst. **29**(4), 806–817 (1981)
7. Chakraborty, T.S., Chakrabarti, S.: On output reorder buffer design of bit reversed pipelined continuous data FFT architecture. In: Proc. IEEE Asia-Pacific Conf. Circuits Syst., pp. 1132–1135. IEEE (2008)
8. Chan, S.C., Yiu, P.M.: An efficient multiplierless approximation of the fast Fourier transform using sum-of-powers-of-two (SOPOT) coefficients. IEEE Signal Process. Lett. **9**(10), 322–325 (2002)
9. Chang, Y.N.: An efficient VLSI architecture for normal I/O order pipeline FFT design. IEEE Trans. Circuits Syst. II **55**(12), 1234–1238 (2008)
10. Chang, Y.N.: Design of an 8192-point sequential I/O FFT chip. In: Proc. World Congress Eng. Comp. Science, vol. II (2012)
11. Chen, J., Hu, J., Lee, S., Sobelman, G.E.: Hardware efficient mixed radix-25/16/9 FFT for LTE systems. IEEE Trans. VLSI Syst. **23**(2), 221–229 (2015)
12. Cheng, C., Yu, F.: An optimum architecture for continuous-flow parallel bit reversal. IEEE Signal Process. Lett. **22**(12), 2334–2338 (2015)
13. Cho, S.I., Kang, K.M.: A low-complexity 128-point mixed-radix FFT processor for MB-OFDM UWB systems. ETRI J. **32**(1), 1–10 (2010)
14. Cho, S.I., Kang, K.M., Choi, S.S.: Implementation of 128-point fast Fourier transform processor for UWB systems. In: Proc. Int. Wireless Comm. Mobile Comp. Conf., pp. 210–213 (2008)
15. Cohen, D.: Simplified control of FFT hardware. IEEE Trans. Acoust., Speech, Signal Process. **24**(6), 577–579 (1976)
16. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19**, 297–301 (1965)
17. Cortés, A., Vélez, I., Sevillano, J.F.: Radix r^k FFTs: Matricial representation and SDC/SDF pipeline implementation. IEEE Trans. Signal Process. **57**(7), 2824–2839 (2009)
18. Dempster, A.G., Macleod, M.D.: Multiplication by two integers using the minimum number of adders. In: Proc. IEEE Int. Symp. Circuits Syst., vol. 2, pp. 1814–1817 (2005)
19. Despain, A.M.: Fourier transform computers using CORDIC iterations. IEEE Trans. Comput. **C-23**, 993–1001 (1974)
20. Duhamel, P., Hollmann, H.: 'Split radix' FFT algorithm. Electron. Lett. **20**(1), 14–16 (1984)
21. Edelman, A., Heller, S., Johnsson, L.: Index transformation algorithms in a linear algebra framework. IEEE Trans. Parallel Distrib. Syst. **5**(12), 1302–1309 (1994)
22. Fraser, D.: Array permutation by index-digit permutation. J. Assoc. Comp. Machinery (ACM) **23**(2), 298–309 (1976)
23. Garrido, M.: Efficient hardware architectures for the computation of the FFT and other related signal processing algorithms in real time. Ph.D. thesis, Universidad Politécnica de Madrid (2009)
24. Garrido, M.: A new representation of FFT algorithms using triangular matrices. IEEE Trans. Circuits Syst. I **63**(10), 1737–1745 (2016)

25. Garrido, M., Acevedo, M., Ehliar, A., Gustafsson, O.: Challenging the limits of FFT performance on FPGAs. In: Int. Symp. Integrated Circuits, pp. 172–175 (2014)
26. Garrido, M., Andersson, R., Qureshi, F., Gustafsson, O.: Multiplierless unity-gain SDF FFTs. IEEE Trans. VLSI Syst. **24**(9), 3003–3007 (2016)
27. Garrido, M., Grajal, J.: Efficient memoryless CORDIC for FFT computation. In: Proc. IEEE Int. Conf. Acoust. Speech Signal Process., vol. 2, pp. 113–116 (2007)
28. Garrido, M., Grajal, J., Gustafsson, O.: Optimum circuits for bit reversal. IEEE Trans. Circuits Syst. II **58**(10), 657–661 (2011)
29. Garrido, M., Grajal, J., Sánchez, M.A., Gustafsson, O.: Pipelined radix-2^k feedforward FFT architectures. IEEE Trans. VLSI Syst. **21**(1), 23–32 (2013)
30. Garrido, M., Gustafsson, O., Grajal, J.: Accurate rotations based on coefficient scaling. IEEE Trans. Circuits Syst. II **58**(10), 662–666 (2011)
31. Garrido, M., Huang, S.J., Chen, S.G.: Feedforward FFT hardware architectures based on rotator allocation. IEEE Trans. Circuits Syst. I **65**(2), 581–592 (2018)
32. Garrido, M., Huang, S.J., Chen, S.G., Gustafsson, O.: The serial commutator (SC) FFT. IEEE Trans. Circuits Syst. II **63**(10), 974–978 (2016)
33. Garrido, M., Källström, P., Kumm, M., Gustafsson, O.: CORDIC II: A new improved CORDIC algorithm. IEEE Trans. Circuits Syst. II **63**(2), 186–190 (2016)
34. Garrido, M., Qureshi, F., Gustafsson, O.: Low-complexity multiplierless constant rotators based on combined coefficient selection and shift-and-add implementation (CCSSI). IEEE Trans. Circuits Syst. I **61**(7), 2002–2012 (2014)
35. Garrido, M., Sánchez, M., López-Vallejo, M., Grajal, J.: A 4096-point radix-4 memory-based FFT using DSP slices. IEEE Trans. VLSI Syst. **25**(1), 375–379 (2017)
36. Glittas, A.X., Sellathurai, M., Lakshminarayanan, G.: A normal I/O order radix-2 FFT architecture to process twin data streams for MIMO. IEEE Trans. VLSI Syst. **24**(6), 2402–2406 (2016)
37. Gold, B., Rader, C.M.: Digital Processing of Signals. New York: McGraw Hill (1969)
38. Good, I.J.: The interaction algorithm and practical Fourier analysis. J. Royal Statistical Society B **20**(2), 361–372 (1958)
39. Granata, J., Conner, M., Tolimieri, R.: Recursive fast algorithm and the role of the tensor product. IEEE Trans. Signal Process. **40**(12), 2921–2930 (1992)
40. Gustafsson, O.: A difference based adder graph heuristic for multiple constant multiplication problems. In: Proc. IEEE Int. Symp. Circuits Syst., pp. 1097–1100. IEEE (2007)
41. Gustafsson, O.: On lifting-based fixed-point complex multiplications and rotations. In: Proc. IEEE Symp. Comput. Arithmetic (2017)
42. Gustafsson, O., Dempster, A.G., Johansson, K., Macleod, M.D., Wanhammar, L.: Simplified design of constant coefficient multipliers. Circuits Syst. Signal Process. **25**(2), 225–251 (2006)
43. Gustafsson, O., Qureshi, F.: Addition aware quantization for low complexity and high precision constant multiplication. IEEE Signal Processing Letters **17**(2), 173–176 (2010)
44. Gustafsson, O., Wanhammar, L.: Arithmetic. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
45. He, S., Torkelson, M.: Design and implementation of a 1024-point pipeline FFT processor. pp. 131–134 (1998)
46. Hsiao, C.F., Chen, Y., Lee, C.Y.: A generalized mixed-radix algorithm for memory-based FFT processors. IEEE Trans. Circuits Syst. II **57**(1), 26–30 (2010)
47. Hsiao, S.F., Lee, C.H., Cheng, Y.C., Lee, A.: Designs of angle-rotation in digital frequency synthesizer/mixer using multi-stage architectures. In: Proc. Asilomar Conf. Signals Syst. Comput., pp. 2181–2185 (2011)
48. Hu, Y., Naganathan, S.: An angle recoding method for CORDIC algorithm implementation. IEEE Trans. Comput. **42**(1), 99–102 (1993)
49. Huang, S.J., Chen, S.G.: A high-throughput radix-16 FFT processor with parallel and normal input/output ordering for IEEE 802.15.3c systems. IEEE Trans. Circuits Syst. I **59**(8), 1752–1765 (2012)

50. Huang, S.J., Chen, S.G., Garrido, M., Jou, S.J.: Continuous-flow parallel bit-reversal circuit for MDF and MDC FFT architectures. IEEE Trans. Circuits Syst. I **61**(10), 2869–2877 (2014)
51. Jang, J.K., Kim, M.G., Sunwoo, M.H.: Efficient scheduling scheme for eight-parallel MDC FFT processor. In: Proc. Int. SoC Design Conf., pp. 277–278 (2015)
52. Järvinen, T.: Systematic methods for designing stride permutation interconnections. Ph.D. thesis, Tampere Univ. of Technology (2004)
53. Järvinen, T., Salmela, P., Sorokin, H., Takala, J.: Stride permutation networks for array processors. In: Proc. IEEE Int. Applicat.-Specific Syst. Arch. Processors Conf., pp. 376–386 (2004)
54. Järvinen, T., Salmela, P., Sorokin, H., Takala, J.: Stride permutation networks for array processors. J. VLSI Signal Process. Syst. **49**(1), 51–71 (2007)
55. Jo, B.G., Sunwoo, M.H.: New continuous-flow mixed-radix (CFMR) FFT processor using novel in-place strategy. IEEE Trans. Circuits Syst. I **52**(5), 911–919 (2005)
56. Johnston, J.A.: Parallel pipeline fast Fourier transformer. In: IEE Proc. F Comm. Radar Signal Process., vol. 130, pp. 564–572 (1983)
57. Källström, P., Garrido, M., Gustafsson, O.: Low-complexity rotators for the FFT using base-3 signed stages. In: Proc. IEEE Asia-Pacific Conf. Circuits Syst., pp. 519–522 (2012)
58. Kim, M.G., Shin, S.K., Sunwoo, M.H.: New parallel MDC FFT processor with efficient scheduling scheme. In: Proc. IEEE Asia-Pacific Conf. Circuits Syst., pp. 667–670 (2014)
59. Kristensen, F., Nilsson, P., Olsson, A.: Flexible baseband transmitter for OFDM. In: Proc. IASTED Conf. Circuits Signals Syst., pp. 356–361 (2003)
60. Kumm, M., Hardieck, M., Zipf, P.: Optimization of constant matrix multiplication with low power and high throughput. IEEE Transactions on Computers **PP**(99), 1–1 (2017)
61. Lee, H.Y., Park, I.C.: Balanced binary-tree decomposition for area-efficient pipelined FFT processing. IEEE Trans. Circuits Syst. I **54**(4), 889–900 (2007)
62. Lee, J., Lee, H., in Cho, S., Choi, S.S.: A high-speed, low-complexity radix-2^4 FFT processor for MB-OFDM UWB systems. In: Proc. IEEE Int. Symp. Circuits Syst., pp. 210–213 (2006)
63. Li, C.C., Chen, S.G.: A radix-4 redundant CORDIC algorithm with fast on-line variable scale factor compensation. In: Proc. IEEE Int. Conf. Acoust. Speech Signal Process., vol. 1, pp. 639–642 (1997)
64. Li, N., van der Meijs, N.: A radix 2^2 based parallel pipeline FFT processor for MB-OFDM UWB system. In: Proc. IEEE Int. SOC Conf., pp. 383–386 (2009)
65. Li, S., Xu, H., Fan, W., Chen, Y., Zeng, X.: A 128/256-point pipeline FFT/IFFT processor for MIMO OFDM system IEEE 802.16e. In: Proc. IEEE Int. Symp. Circuits Syst., pp. 1488–1491 (2010)
66. Lin, C.H., Wu, A.Y.: Mixed-scaling-rotation CORDIC (MSR-CORDIC) algorithm and architecture for high-performance vector rotational DSP applications. IEEE Trans. Circuits Syst. I **52**(11), 2385–2396 (2005)
67. Lin, Y.W., Lee, C.Y.: Design of an FFT/IFFT processor for MIMO OFDM systems. IEEE Trans. Circuits Syst. I **54**(4), 807–815 (2007)
68. Liu, H., Lee, H.: A high performance four-parallel 128/64-point radix-2^4 FFT/IFFT processor for MIMO-OFDM systems. In: Proc. IEEE Asia Pacific Conf. Circuits Syst., pp. 834–837 (2008)
69. Liu, L., Ren, J., Wang, X., Ye, F.: Design of low-power, 1GS/s throughput FFT processor for MIMO-OFDM UWB communication system. In: Proc. IEEE Int. Symp. Circuits Syst., pp. 2594–2597 (2007)
70. Liu, X., Yu, F., Wang, Z.: A pipelined architecture for normal I/O order FFT. Journal of Zhejiang University - Science C **12**(1), 76–82 (2011)
71. Ma, Y., Wanhammar, L.: A hardware efficient control of memory addressing for high-performance FFT processors. IEEE Trans. Signal Process. **48**(3), 917–921 (2000)
72. Ma, Z.G., Yin, X.B., Yu, F.: A novel memory-based FFT architecture for real-valued signals based on a radix-2 decimation-in-frequency algorithm. IEEE Trans. Circuits Syst. II **62**(9), 876–880 (2015)

73. Macleod, M.D.: Multiplierless implementation of rotators and FFTs. EURASIP J. Appl. Signal Process. **2005**(17), 2903–2910 (2005)
74. Majumdar, M., Parhi, K.K.: Design of data format converters using two-dimensional register allocation. IEEE Trans. Circuits Syst. II **45**(4), 504–508 (1998)
75. Meher, P.K., Park, S.Y.: CORDIC designs for fixed angle of rotation. IEEE Trans. VLSI Syst. **21**(2), 217–228 (2013)
76. Meher, P.K., Valls, J., Juang, T.B., Sridharan, K., Maharatna, K.: 50 years of CORDIC: Algorithms, architectures, and applications. IEEE Trans. Circuits Syst. I **56**(9), 1893–1907 (2009)
77. Möller, K., Kumm, M., Garrido, M., Zipf, P.: Optimal shift reassignment in reconfigurable constant multiplication circuits. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. (2017). Accepted for publication
78. Oh, J.Y., Lim, M.S.: New radix-2 to the 4th power pipeline FFT processor. IEICE Trans. Electron. **E88-C**(8), 1740–1746 (2005)
79. Paeth, A.W.: A fast algorithm for general raster rotation. In: Proc. Graphics Interface, pp. 77–81 (1986)
80. Parhi, K.K.: Systematic synthesis of DSP data format converters using life-time analysis and forward-backward register allocation. IEEE Trans. Circuits Syst. II **39**(7), 423–440 (1992)
81. Park, S.Y., Yu, Y.J.: Fixed-point analysis and parameter selections of MSR-CORDIC with applications to FFT designs. IEEE Trans. Signal Process. **60**(12), 6245–6256 (2012)
82. Püschel, M., Milder, P.A., Hoe, J.C.: Permuting streaming data using RAMs. J. ACM **56**(2), 10:1–10:34 (2009)
83. Qureshi, F., Gustafsson, O.: Generation of all radix-2 fast Fourier transform algorithms using binary trees. In: Proc. Europ. Conf. Circuit Theory Design, pp. 677–680 (2011)
84. Qureshi, F., Gustafsson, O.: Low-complexity constant multiplication based on trigonometric identities with applications to FFTs. IEICE Trans. Fundamentals **E94-A**(11), 324–326 (2011)
85. Reisis, D., Vlassopoulos, N.: Conflict-free parallel memory accessing techniques for FFT architectures. IEEE Trans. Circuits Syst. I **55**(11), 3438–3447 (2008)
86. Sánchez, M., Garrido, M., López, M., Grajal, J.: Implementing FFT-based digital channelized receivers on FPGA platforms. IEEE Trans. Aerosp. Electron. Syst. **44**(4), 1567–1585 (2008)
87. Serre, F., Holenstein, T., Püschel, M.: Optimal circuits for streamed linear permutations using RAM. In: Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays, pp. 215–223. ACM (2016)
88. Shih, X.Y., Liu, Y.Q., Chou, H.R.: 48-mode reconfigurable design of SDF FFT hardware architecture using radix-3^2 and radix-2^3 design approaches. IEEE Trans. Circuits Syst. I **64**(6), 1456–1467 (2017)
89. Shukla, R., Ray, K.: Low latency hybrid CORDIC algorithm. IEEE Trans. Comput. **63**(12), 3066–3078 (2014)
90. Stone, H.: Parallel processing with the perfect shuffle. IEEE Trans. Comput. **C-20**(2), 153–161 (1971)
91. Takagi, N., Asada, T., Yajima, S.: Redundant CORDIC methods with a constant scale factor for sine and cosine computation. IEEE Trans. Comput. **40**(9), 989–995 (1991)
92. Takala, J., Järvinen, T.: Stride Permutation Access In Interleaved Memory Systems. Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation, S. Bhattacharyya, E. Deprettere and J. Teich. CRC Press (2003)
93. Takala, J., Jarvinen, T., Sorokin, H.: Conflict-free parallel memory access scheme for FFT processors. In: Proc. IEEE Int. Symp. Circuits Syst., vol. 4, pp. 524–527 (2003)
94. Tang, S.N., Tsai, J.W., Chang, T.Y.: A 2.4-GS/s FFT processor for OFDM-based WPAN applications. IEEE Trans. Circuits Syst. II **57**(6), 451–455 (2010)
95. Tsai, P.Y., Lin, C.Y.: A generalized conflict-free memory addressing scheme for continuous-flow parallel-processing FFT processors with rescheduling. IEEE Trans. VLSI Syst. **19**(12), 2290–2302 (2011)
96. Tummeltshammer, P., Hoe, J.C., Püschel, M.: Time-multiplexed multiple-constant multiplication. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **26**(9), 1551–1563 (2007)

97. Volder, J.E.: The CORDIC trigonometric computing technique. IRE Trans. Electronic Computing **EC-8**, 330–334 (1959)
98. Voronenko, Y., Püschel, M.: Multiplierless multiple constant multiplication. ACM Trans. Algorithms **3**, 1–39 (2007)
99. Wang, J., Xiong, C., Zhang, K., Wei, J.: A mixed-decimation MDF architecture for radix- 2^k parallel FFT. IEEE Trans. VLSI Syst. **24**(1), 67–78 (2016)
100. Wang, Z., Liu, X., He, B., Yu, F.: A combined SDC-SDF architecture for normal I/O pipelined radix-2 FFT. IEEE Trans. VLSI Syst. **23**(5), 973–977 (2015)
101. Wenzler, A., Luder, E.: New structures for complex multipliers and their noise analysis. In: Proc. IEEE Int. Symp. Circuits Syst., vol. 2, pp. 1432–1435 (1995)
102. Wold, E., Despain, A.: Pipeline and parallel-pipeline FFT processors for VLSI implementations. IEEE Trans. Comput. **C-33**(5), 414–426 (1984)
103. Wu, C.S., Wu, A.Y.: Modified vector rotational CORDIC (MVR-CORDIC) algorithm and architecture. IEEE Trans. Circuits Syst. II **48**(6), 548–561 (2001)
104. Wu, C.S., Wu, A.Y., Lin, C.H.: A high-performance/low-latency vector rotational CORDIC architecture based on extended elementary angle set and trellis-based searching schemes. IEEE Trans. Circuits Syst. II **50**(9), 589–601 (2003)
105. Xia, K.F., Wu, B., Xiong, T., Ye, T.C.: A memory-based FFT processor design with generalized efficient conflict-free address schemes. IEEE Trans. VLSI Syst. **25**(6), 1919–1929 (2017)
106. Xing, Q., Ma, Z., Xu, Y.: A novel conflict-free parallel memory access scheme for FFT processors. IEEE Trans. Circuits Syst. II (2017)
107. Xudong, W., Yu, L.: Special-purpose computer for 64-point FFT based on FPGA. In: Proc. Int. Conf. Wireless Comm. Signal Process., pp. 1–3 (2009)
108. Yang, K.J., Tsai, S.H., Chuang, G.: MDC FFT/IFFT processor with variable length for MIMO-OFDM systems. IEEE Trans. VLSI Syst. **21**(4), 720–731 (2013)
109. Yang, L., Zhang, K., Liu, H., Huang, J., Huang, S.: An efficient locally pipelined FFT processor. IEEE Trans. Circuits Syst. II **53**(7), 585–589 (2006)
110. Yeh, W.C., Jen, C.W.: High-speed and low-power split-radix FFT. IEEE Trans. Signal Process. **51**(3), 864–874 (2003)
111. Yu, C., Yen, M.H.: Area-efficient 128- to 2048/1536-point pipeline FFT processor for LTE and mobile WiMAX systems. IEEE Trans. VLSI Syst. **23**(9), 1793–1800 (2015)
112. Zheng, W., Li, K.: Split radix algorithm for length 6^m DFT. IEEE Signal Process. Lett. **20**(7), 713–716 (2013)

Programmable Architectures for Histogram of Oriented Gradients Processing

Colm Kelly, Roger Woods, Moslem Amiri, Fahad Siddiqui, and Karen Rafferty

Abstract There is an increasing demand for high performance image processing platforms based on field programmable gate array (FPGA). The Histogram of Orientated Gradients (HOG) algorithm is a feature descriptor algorithm used in object detection for many security applications. The chapter examines the implementation of this key algorithm using an FPGA-based soft-core architecture approach. Firstly, the HOG algorithm is described and its performance profiled from a computation and bandwidth perspective. Then the IPPro soft-core processor architecture is introduced and a number of mapping strategies are covered. A HOG implementation is demonstrated on a Zynq platform, resulting in a design operating at 15.36 fps; this compares favorably with the performance and resources of hand-crafted VHDL code.

1 Introduction

Image processing techniques have long existed in industrial, commercial and military domains to name but a few. As sensor resolution and frame rate have dramatically increased, manufacturing costs have fallen to such a level that complete high performance, image processing systems have become commonplace. Modern image processing algorithms are often very computationally demanding and consequently have very high performance requirements. Such systems can be found in anywhere from relatively inexpensive children's toys to 3D magnetic resonance

C. Kelly
Thales Air Defence, Belfast, UK
e-mail: Colm.Kelly@uk.thalesgroup.com

R. Woods (✉) · F. Siddiqui · K. Rafferty
Queen's University of Belfast, Belfast, UK
e-mail: r.woods@qub.ac.uk; f.siddiqui@qub.ac.uk; k.rafferty@qub.ac.uk

M. Amiri
University of Bristol, Bristol, UK
e-mail: ma17215@bristol.ac.uk

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_18

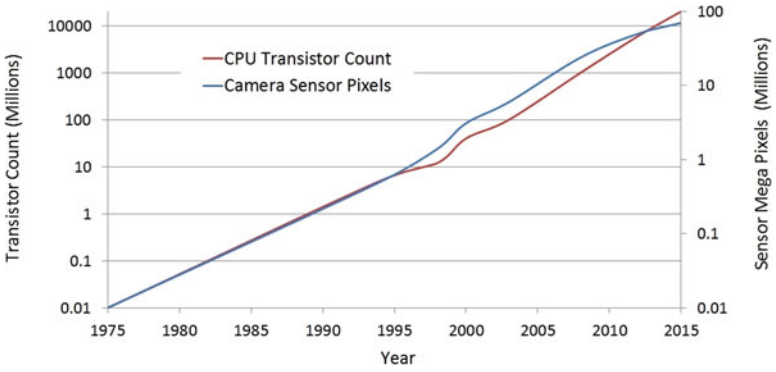


Fig. 1 CMOS image sensor pixel and CPU transistor count [6]

imaging systems in hospitals to smart cameras (chapter "Distributed Smart Cameras and Distributed Computer Vision") and have been one of the enablers for Industry 4.0. This chapter should be read in conjunction with other chapters in this book which cover the design of architectures for such systems such as light field displays (chapter "Signal Processing Methods for Light Field Displays").

Transistor miniaturisation has enabled CMOS sensor manufacturers to increase the density of the crystallized silicon elements that make up the individual pixel sensors on a single silicon die. Thankfully, sensor and processor densities have increased in tandem as shown in Fig. 1. With HD resolutions now commonplace, there is a demand for real-time, low cost, low power, versatile, embedded image processing platforms with increased computation rates. Such systems are particularly demanding at the front end of the system where large frame sizes of 8.3 M pixels for ultra-high density are delivered at 60 fps. Stereo vision (chapter "Architectures for Stereo Vision") and video coding (chapters "High Dynamic Range Video Coding" and "MPEG Reconfigurable Video Coding") represent key examples of such challenging systems.

The Graphics Processing Unit (GPU), the multimedia extended central processing unit (MMX CPU) and the multimedia specific digital signal processor (DSP) [10] have now emerged as platforms for implementing image processing systems. The field programmable gate array (FPGA) [1] represents a compromise between the low power consumption of the application specific integrated circuit (ASIC) and the programmability of such processors [12, 16]. Moreover, recent FPGA devices offer processing resources in the form of the ARM processors [8] as well as programmable logic.

A major limitation of FPGAs though, is the need to learn specialized hardware design (HDLs) languages, although this is being addressed by high level synthesis tools such as Vivado HLS and Intel FPGA SDK for OpenCL. Intellectual property (IP) blocks approaches such as System Generator for DSP from Xilinx [4] and HDL Coder from MathWorks [5] have emerged to try to satisfy this requirement. However, one of the remaining challenges is the time to perform the place and

route process which occurs at the end of the design space exploration phase where the estimated performance is translated into real performance. An alternative approach is to employ predesigned soft-core processors which simply need to be programmed. The aim is to provide a software design route, whilst still offering FPGA performance associated with conventional synthesis routes. It is believed that this work is the first to implement the HOG descriptor using a soft-core processor orientated FPGA framework, providing performance in a truly programmable manner [18].

1.1 Chapter Breakdown

The chapter is organized as follows. The HOG algorithm is introduced in Sect. 2 and is profiled in order to reveal the processing and data communications requirements. Section 3 introduces the IPPro architecture and shows how it effectively utilizes the dedicated FPGA resources. The text also categorizes the algorithm in terms of processing requirements and shows how it suits a soft-core implementation approach. The mapping of the HOG algorithm onto a multicore implementation is then given in Sect. 4 and followed by a detailed analysis of the performance (Sect. 5), highlighting the limitation of the IPPro instruction set. A number of optimisations are then presented in Sect. 6 with the aim of overcoming these limitations and followed by conclusions in Sect. 7.

2 HOG Algorithm

Dalal and Triggs [17] illustrate that the human form can be characterized rather well by the distribution of local intensity gradients or edge directions. Their algorithm converts the pixel intensity information to gradient information, as illustrated by the image of gradients for the pedestrian's heel (Fig. 2). Gradients have both magnitude and direction.

For this example and in most instances in the literature, the detection window is fixed at 64×128 pixels and is divided up into small 8×8 pixel spatial regions or cells. Each cell generates a histogram of gradient directions (edge orientations) over the 64 pixels of the cell as shown in Fig. 3.

The algorithm shown in Fig. 3 comprises six stages. In the *Normalize Gamma and color* stage, the camera image is normalized for human vision and luminosity values are extracted from the RGB values. The gradient of each pixel relative to its surrounding pixels is calculated in the *Compute Gradients* stage. A 1D kernel $[1 \;\; 0 \;\; -1]$ is convolved in the x- and then y-axis to produce the respective gradients, G_x and G_y.

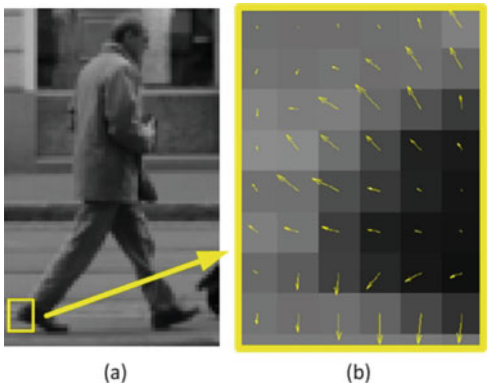


Fig. 2 Image example of the HOG algorithm. (**a**) Image of intensities. (**b**) Image of gradients [19]

The *Weighted vote into spatial and orientated cells* stage requires the calculation of the magnitude of each x and y gradient pair where magnitude, M_{xy}, is given as $\sqrt{G_x^2 + G_y^2}$. The angle of the vector $M \times y$ is then used to allocate the value of $M \times y$ for each pixel into one of nine bins, each spanning 20° between 0° and 180°. Over an 8 × 8 pixel cell, a single 9-element vector is produced which is referred to as the histogram of orientated gradients or HOG.

In the *Normalisation over overlapping spatial blocks* stage, blocks are generated by locally normalising groups of four cells i.e. 2 × 2 cells, in order to improve the invariance to illumination and shadowing; a L2 normalisation technique is employed. The resultant block vector has 36 elements.

Collation of the blocks over the full detection window (7 × 15 blocks) is carried out in the 5th stage, *Collect HOGs over detection window* to produce HOG descriptors. These HOG descriptors are the concatenation of the normalized 36 element vectors produced in the 4th stage. The new 3780 (7 × 15 × 36) element vector is called a feature descriptor, fv.

In the final stage, an off-line pre-trained support vector machine (SVM) classifier receives the 3780 element vectors and multiplies them with set weights generating a window score s:

$$s = \sum_{i=1}^{n=3780} (fv_i \cdot w_i) + b. \tag{1}$$

Presence or absence of a human in the detection window is represented by the sign of s whereas the level of confidence of the detection is measured by the magnitude.

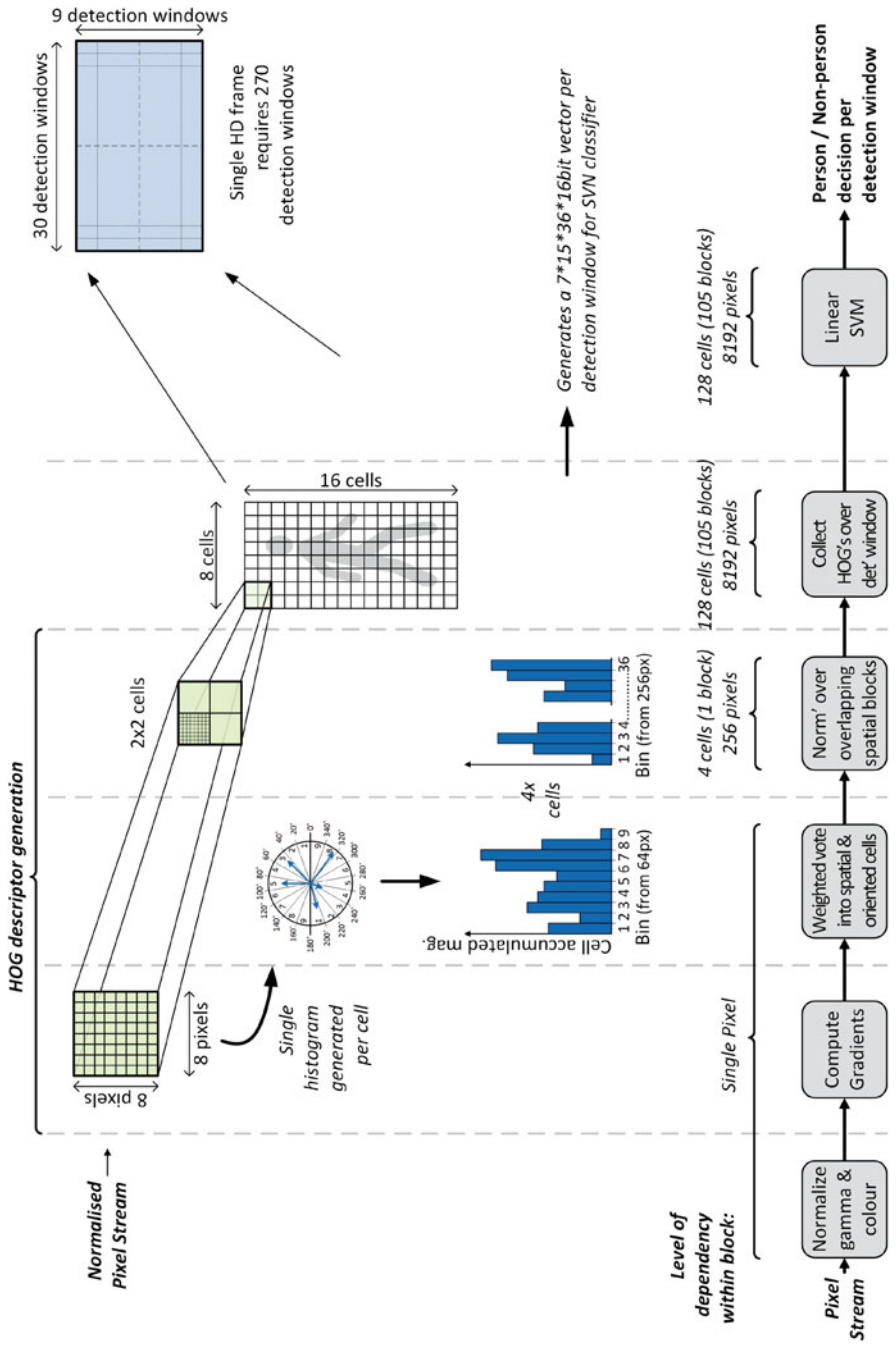


Fig. 3 HOG data dependencies as per Dalal and Triggs implementation [17]

2.1 *Profiling HOG*

The data bandwidth between each of the algorithm functions was calculated by generating a profile of the data flowing through the entire algorithm. The results of the analysis on the HOG algorithm are illustrated in Fig. 4. It shows that the most intensive calculations are performed in the *Compute Gradients* and *Weighted Vote into Spatial and Orientation Cells* blocks, so their acceleration would provide the largest processing performance gain. The algorithm reduces the data transmission needs from 1.5 Gbps (at the camera output) to 30 bps (at the output of the SVM module); this is important in remote surveillance applications where long range RF communications have very low bandwidth and minimal energy levels.

Figure 4 also indicates that computation hotspots are to be found between the 2nd and 3rd blocks where a peak bandwidth of 1990 Mbps is required. Conventional software approaches struggle to achieve real-time implementations with these computational hotspots; thus, the HOG algorithm remained the preserve of high performance computing systems but now it has been realized in real-time FPGA implementations [20, 21, 24]. These fixed implementations are, however, created using hand written HDL which is a time consuming process requiring very specialist knowledge and experience.

The lower part of Fig. 4 outlines the IPPro accelerator concept. The 'Compute Gradients' and 'Weighted vote into Spatial and orientation cells' functions is off-loaded to the IPPro processing arrays before the results are passed back to the host ARM processor to complete the remainder of the algorithm processing.

3 IPPro Introduction

The IPPro is a 16-bit, signed fixed-point, 5-stage, balanced pipelined RISC architecture that exploits the FPGA dedicated DSP resource, in this case a Xilinx DSP48E1, and provides balance among performance, latency and efficient resource utilization [22]. The architecture here is modified to support the mapping of dataflow graphs by replacing the previously memory-mapped, data memory by stream-based, blocking input/output first in, first out FIFOs that support data transfer (Fig. 5). The in-order pipeline simplifies the compiler development compared to out-of-order architectures, supports the identified execution and memory access patterns and can be used as a coarse-grained processing core. The IPPro has the following memory areas: a register file of size 32×16-bits to store pixels and intermediate results and; a kernel memory of size 32×16-bits to store the kernel coefficients, constant values and input/output FIFOs to stream pixel data in and out of IPPro.

The programming methodology is based on the CAL dataflow programming language [11] which is detailed in chapter "MPEG Reconfigurable Video Coding". It employs a reprogrammable model comprising multicore processors supporting

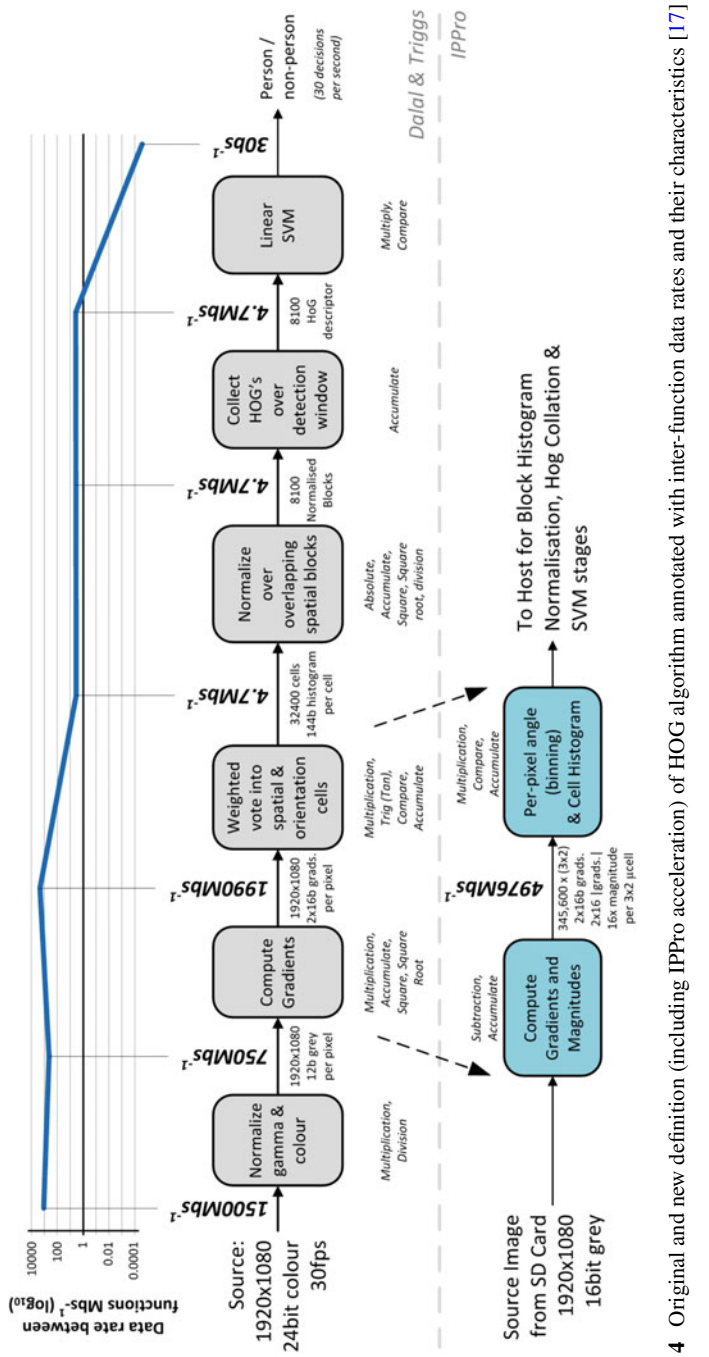


Fig. 4 Original and new definition (including IPPro acceleration) of HOG algorithm annotated with inter-function data rates and their characteristics [17]

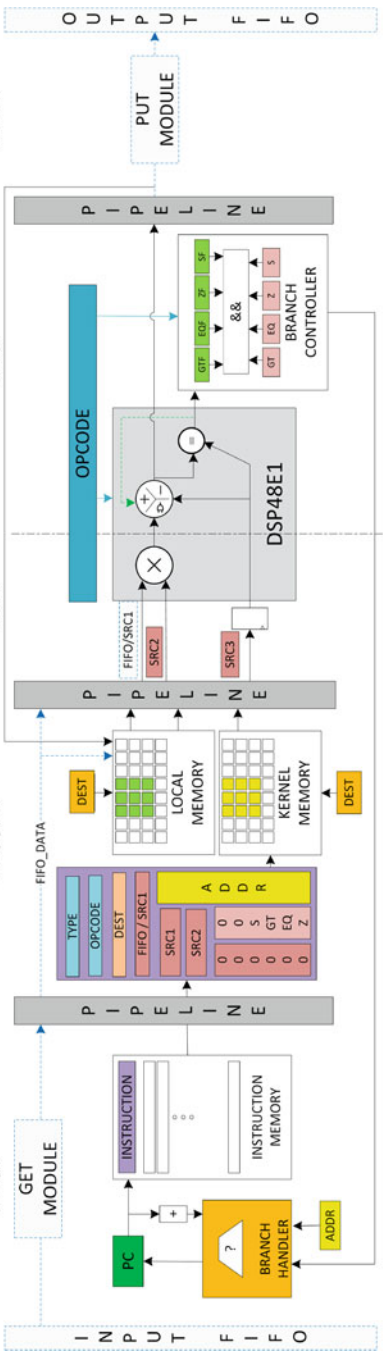


Fig. 5 Block diagram showing detailed datapath of IPPro

single-input, multiple-output (SIMD) operation and an associated inter-processor communication methodology. A dataflow model allows algorithms to be realized as actors with specific firing rules that are mapped into directed graphs where the nodes represent computations and arcs represent the movement of data. Combining actors with a set of connections between them allows the construction of a network where communication is made using infinite size, FIFO components.

Table 1 outlines the relationship between programmable abstraction and the addressing modes, along with some supported instructions for the IPPro architecture, facilitating programmable implementation of point and area image processing algorithms. The "stream access" reads a stream of pixels from the input FIFO using a `GET` instruction and allows processing either with constant values (Kernel Memory-FIFO) or neighbouring pixel dependent values (Register File-FIFO or Register File-Register File). The processed stream is then written to the output FIFO using the `PUSH` instruction. IPPro supports arithmetic, logical, branch and data handling instructions. The presented instruction set was optimized after profiling use-cases presented in [22, 23].

Table 2 shows how the input/output operands of addressing modes of Table 1 are encoded and used by the "Instruction Decoder" (ID) to control the IPPro datapath

Table 1 Relationship of addressing modes, programmable abstraction and instruction set of IPPro

Addressing mode	Programmable abstraction	Supported instructions
FIFO handling	Stream access	get, push
Register file-FIFO	Stream and randomly accessed data	addrf, subrf, mulrf, andrf, orrf, minrf, maxrf etc.
Register file-register file	Randomly accessed data	str, add, sub, mul, mulacc, muladd, and, min, max etc.
Kernel memory-FIFO	Stream and fixed values	addkm, subkm, mulkm, muladdkm, minkm, maxkm etc.

Table 2 Instruction frame structure

Addressing modes	34-Bit IPPro instruction encoding						
	33 to 31	30 to 26	25 to 21	20 to 16	15 to 11	10 to 6	5 to 0
Register File-FIFO	INSTR_TYPE	OPCODE	RD	RB	00000	00000	000000
			RD	RB	00000	RC	000000
Register File-Register File	INSTR_TYPE	OPCODE	RD	RB	RA	RC	000000
			RD	RB	RA	00000	000000
Kernel Memory-FIFO	INSTR_TYPE	OPCODE	RD	Kn	00000	RC	000000
			RD	Kn	00000	00000	000000
			00000	Kn	16-bits value		
FIFO	INSTR_TYPE	OPCODE	RD	00000	00000	00000	000000
			00000	Kn	00000	00000	000000
Branch/Jump	INSTR_TYPE	OPCODE	00000	00000	16-bit address		

to execute the instruction. INSTR_TYPE is a 3-bit field that allows differentiation between addressing modes. OPCODE is a 5-bit field that defines what operation shall be executed on data operands. The terms. RA, RB, RC, Kn and RD are 5-bit fields which define source and destination data operands located in Register File and Kernel Memory; RA, RB, RC are always source registers, RD is always a destination register and Kn represents that data operand should be fetched from Kernel Memory instead of Register file. In some cases, there are operations required for better accuracy, e.g. incorporating a coprocessor block such as a divider (see later).

IPPro supports branch instructions to support control flow execution patterns as they are commonly used to implement conditional statements and loops. IPPro has four flags (zero, equal, greater than and sign) that are generated from the DSP48E1 pattern detector. It compares the input operands or output results and sets/resets the PATTERNDETECT (PD) bit. Branch instructions are handled by branch controller and branch handler as shown in Fig. 5. The IPPro was coded in Verilog HDL and synthesized using Xilinx Vivado v2015.4 design suite giving a clock rate of 337 MHz which delivers 1.6–3.3 times higher operating frequency than comparative processors.

4 HOG Deployment on IPPro

The IPPro System functions as the computational stage between the video source, e.g. the surveillance thermal or day-time camera and the distribution of the processed video data to a host over a network. For this example, a Xilinx Zedboard™ was used with images from the onboard SD memory card in lieu of an actual camera source and fed directly into the DDR and stored as frames. The ARM Cortex A9 host processor manages the flow of data from the frame buffer to the programmable fabric that accommodates the IPPro cores.

4.1 Algorithm Partitioning

The mapping to the IPPro began with a functional breakdown of the algorithm as a Simulink model. The initial mapping used the partitioning of Dalal and Triggs [17] but was modified to facilitate the reuse of already computed gradient, absolute gradient and magnitude data (Fig. 6). The additional data is passed between the two IPPro functional blocks, giving an increase in required bandwidth from 1990 to 4976 Mbps. As this is to be implemented within the FPGA fabric where several Tbps of bandwidth are available, it is more beneficial to trade bandwidth for reduction in computation time. The algorithm segments are then further broken down into IPPro instructions. The IPPro core Program Memory (PM) code is generated to produce an

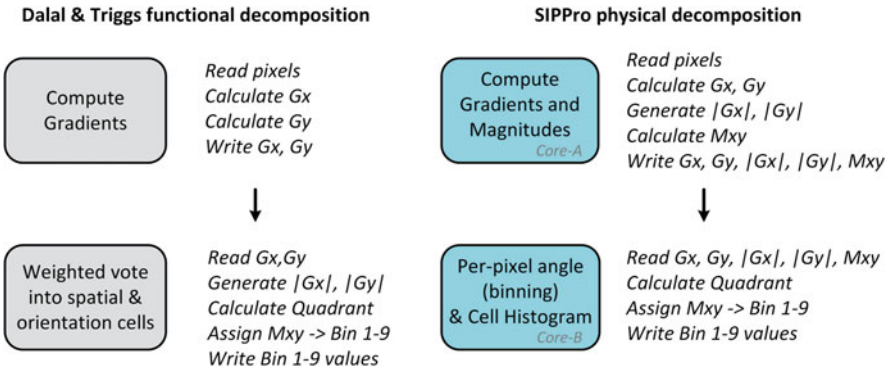


Fig. 6 Mapping of accelerated Dalal and Triggs functions to IPPro cores

efficient ratio of read/write to ALU instructions whilst maximizing the quantity of input data to be processed. Partitioning of the input image data is such that smaller blocks and their computation fit within the IPPro core registers.

The FPGA's limited memory encourages reuse of intermediate results. In order to establish the highest throughput, PMs were coded using different decompositions. A limited number of configurations were explored and the configuration with the highest throughput of 175 frames per second (fps) was chosen.

Temporal parallelism is exploited by the reuse of local data in the IPPro core registers. For example, during exploration, it was found that the *Pixel Gradients* function could be appended with the *Magnitude* ($M \times y$) function, thus saving costly additional load and store of the Gradient values, as the IPPro core already had these pixel intensities loaded with gradients stored in its register file. Within the IPPro local memory, there were enough unused register locations to store the results. This increased the throughput by 6% which in real terms for this 90-core IPPro implementation, is a increase to 10.6 fps for a 1920×1080 pixel frame.

The following mathematical optimization was also incorporated when calculating the gradient values $G_x = [-1, 0, 1]$ and $G_y = [-1, 0, 1]^T$; as the gradient calculation kernel values use only -1, 0 and 1 as factors, we only need one subtraction instruction. Research in [24] and [20] has already shown that these optimizations have negligible effect on the accuracy of the algorithm. Using the example data of Fig. 3, G_x and G_y are calculated as moderately positive values of 167 and 129 respectively, indicating that the light to dark gradients are in both the left to right and top to bottom directions in almost equal amounts (Fig. 7).

The next stage is that of binning which allocates each of the derived pixel gradient magnitudes to one of the nine available bins as shown in the lower left corner of Fig. 3. The tangent for each input pixels G_x, G_y is normally evaluated to determine to which bin in Fig. 8 that the respective M_{xy} belongs. As $\tan\theta$ is symmetrical about the G_y axis, Bin 1 magnitudes shall capture all θ values between 0–20° and 180–200°, Bin 2 magnitudes shall capture all θ values between 20–40° and 200–220° and so on.

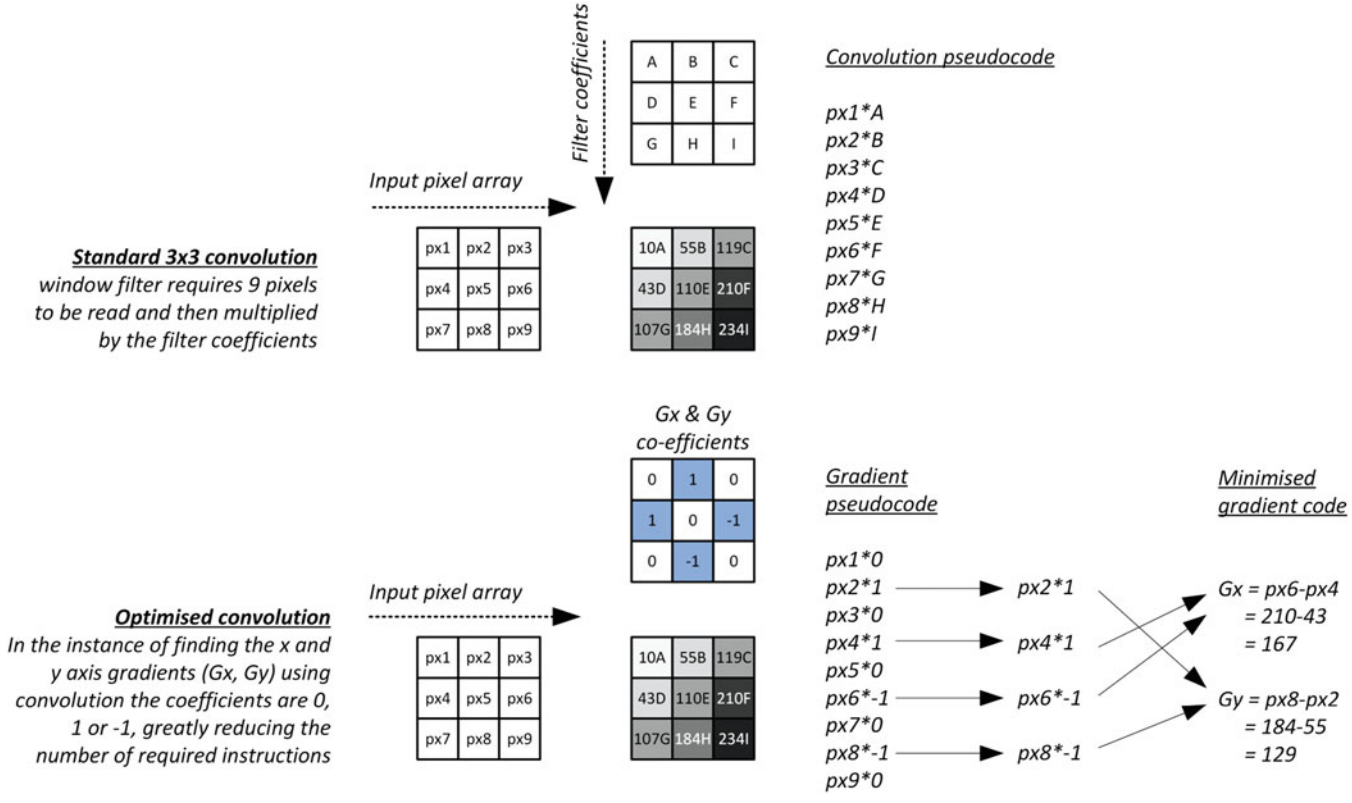


Fig. 7 Gradient filter code optimisation

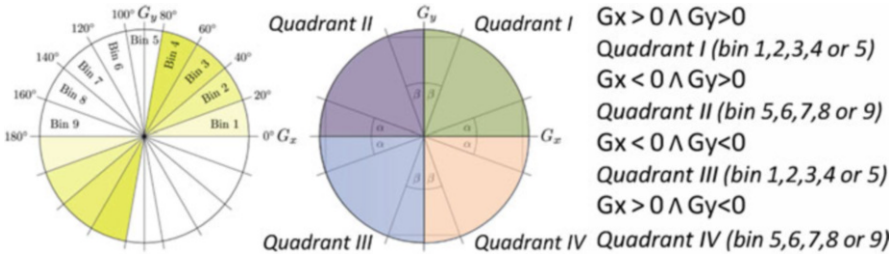


Fig. 8 Gradient magnitude quadrant evaluation

Rather than perform a trigonometric calculation to determine to which 20° bin the magnitude belongs, the $M \times y$ values are first assigned to one of four quadrants as shown in Fig. 8. A series of comparisons are then run to check if the angle is greater than a predefined threshold [19]. Cell histograms are generated by accumulating M_{xy} value of each pixel at the appropriate bin for that pixel over a cell.

4.2 Instruction Mapping and Scheduling on a Single IPPro

The approach for mapping into the IPPro is summarized in Fig. 9. The design flow starts with an initial functional partition and is then explored iteratively. The IPPro approach favours algorithms with deterministic behaviour due to its considerably minimized control mechanisms. As the register size is tightly constrained, it is necessary to establish whether or not it is more efficient to heavily process small amounts of data (complex PM) or lightly process large amounts of data (simple PM). Complex PMs result in longer instruction lengths and reduce the granularity of the high-level functional blocks which reduces the ability to explore the mapping across multiple IPPro cores.

The next step is the translation of the image processing functions to the IPPro instruction set. A key objective is to reduce the no operation (NOP) instructions by interleaving the sub-tasks within the functional blocks. Sub-tasks begin when sufficient registers are loaded; the remaining input data for subsequent sub-tasks is loaded by paying attention to the pipeline delays during sub-tasks. Interleaving of IPPro processors is also employed in a multi-IPPro core implementation such that scheduling of each of the processors is initialized serially; this reduces the bandwidth requirements on the higher level memory accesses.

The final stage of the process involves mapping the generated PMs onto the available hardware. With a single IPPro core, there are two approaches. The first relies upon each of the PMs carrying out tasks that are not data-dependent, and thus events are scheduled to execute sequentially on a small volume of data. If, however, each functional stage is dependent on a group or entire frame of results from the current PM, then the second approach is adopted to perform the function across

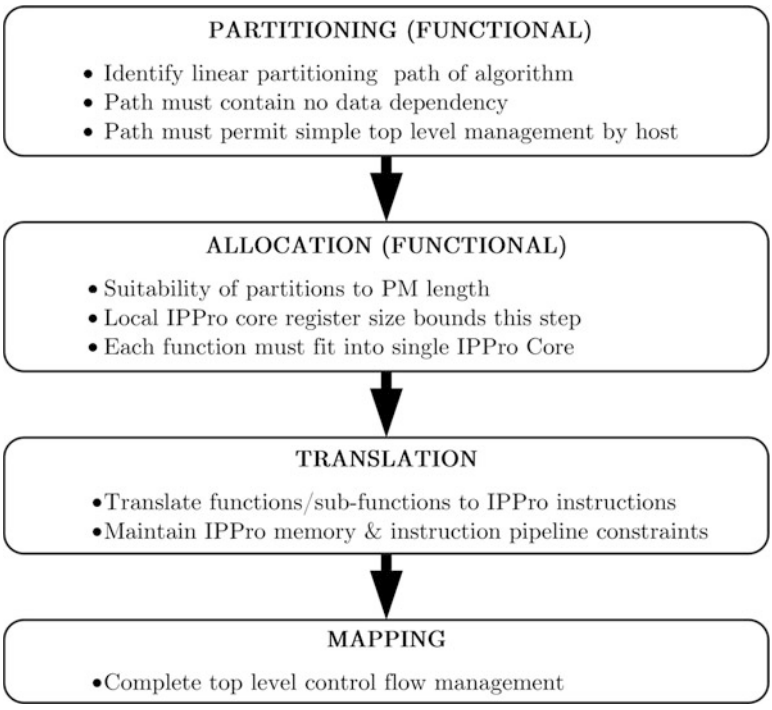


Fig. 9 Key features in algorithm to IPPro mapping and scheduling process

the entire frame, producing intermediate results which are stored in the higher level memory. The subsequent PM then retrieves the data from the high level memory and executes the next stage of dependent processing from the entire frame or block of intermediate results. Whilst this process is very similar to the traditional mapping of functions, an emphasis is put on maximizing the utilization of DSP48E1s whilst minimizing the memory accesses overhead.

4.3 Instruction Mapping and Scheduling on Multiple IPPro

Initially, the mapping of the HOG algorithm onto a single core was explored. The performance of an array of cores which utilizes the parallelism offered through FPGA implementation, is now considered. The same IPPro PM code and functional decomposition as highlighted earlier in this section is used, but spatial and temporal parallelism are now also considered (Fig. 10). Mapping involves allocating the number of processing steps to processor elements and generating a schedule. In the instance of one processor element, hardware reuse is employed whereas as in the multicore case, parallelism is exploited by duplicating the functionality across single

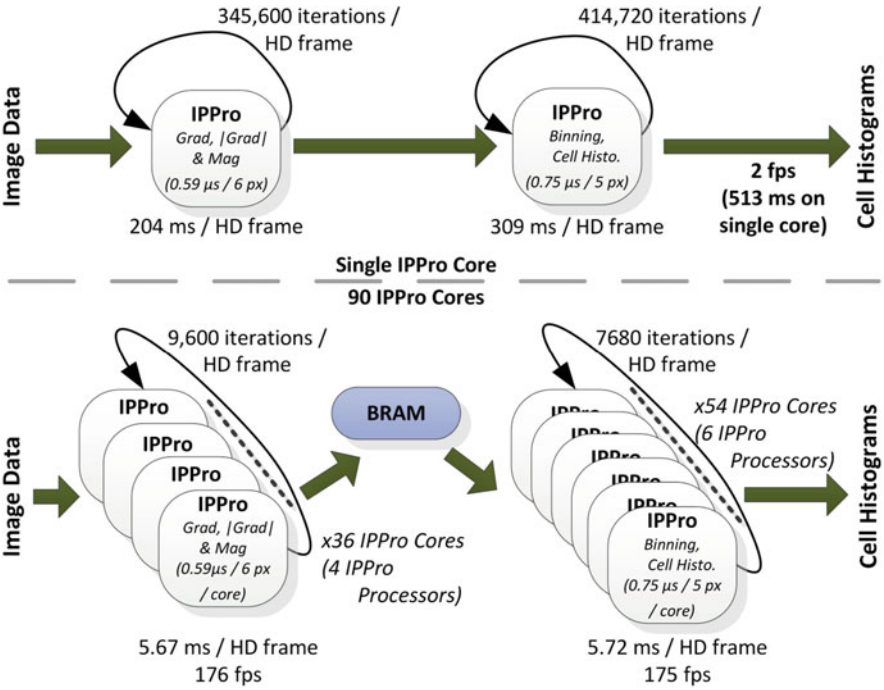


Fig. 10 HOG architecture for single-IPPro (top) and multi-IPPro (bottom)

IPPro cores as illustrated in Fig. 10. During the manycore IPPro implementation, data is streamed so careful scheduling and packing of the PM is essential to balance the computation phases and avoid blocking. For now, this is achieved in a deterministic manner by inserting NOP instructions into the PM code.

4.4 Results Generation: Initial Architecture

Two versions of the functional blocks were explored, a hand-coded VHDL description and an IPPro implementation. Both designs were coded in and taken through, Xilinx ISE 14.6 Place and Route (PAR) tools with the results recorded in Table 3. In each case, the target platform was the programmable fabric within the Zynq 7020 used in the Diligent Zedboard™ Zynq-7000 development board. After place and route, the single IPPro core operated at 337 MHz for both functions (PMs) and the hand coded implementation operated at 288 MHz for the *Gradients and Magnitude* function and at 164 MHz for the *Binning and Cell* Histogram. The metric "fps per 1k Lookup Tables (LUT)" is created to allow fair comparison per resource between approaches and is quoted in Table 3. The frame rate of 4.9 fps for the single IPPro corresponds to 204 ms to compute the frame.

Table 3 Algorithm resource usage on a single IPPro versus a hand coded approach for HOG functions as standalone units

Function	Resource type	Single IPPro			Hand coded		
		Usage	Frame rate	HD fps/ 1K LUT	Usage	Frame rate	HD fps/ 1K LUT
Gradient and magnitude	LUTS	140	4.9	35	422	139	329
	DSPs	1			0		
	BRAMs	0			0		
Binning and cell histogram	LUTS	140	3.2	23	1463	74.4	54.3
	DSPs	1			0		
	BRAMs	0			0		

The PMs for the *Gradients and Magnitude* function required 199 instructions to generate the values for 6 output pixels and for the *Binning and Cell Histogram*, 251 instructions are needed to generate the values for 5 output pixels; the data was verified on the Diligent Zedboard™ Zynq-7000 development board. For the architecture in Fig. 10, it is possible to achieve a maximum throughput of 175 fps at a 1920 × 1080 pixel (HD) resolution. This architecture uses 90 IPPro cores and can be implemented on the smallest Xilinx 7 Series FPGA (XC7A35TCPG236) which has 90 DSP48E1 blocks available. A resource comparable, 16 core IPPro solution (16 × 140 LUTs) is compared against a hand coded VHDL solution (requiring 1885 LUTs). The total processing time per frame is 65.1 ms for IPPro versus 19.78 ms for the hand coded design equivalent to frame rates of 15.36 and 50.56 fps, respectively.

Although the multicore IPPro design achieves just under half of the performance of the hand coded solution, it was achieved in a matter of days as opposed to weeks. The speed and high degree of re-programmability possible with an IPPro solution allows the designer to more rapidly explore the design space with much greater flexibility.

5 Profiling of Initial HOG Implementation

Detailed profiling of the HOG algorithm is performed by observing the number of instructions for all phases of the HOG algorithm and highlights the impact of a limited IPPro instruction set on the overall system performance, and of the potential of using a coprocessor. For accurate comparison against our existing analysis, the throughput of the entire IPPro system design is required to match that of the throughput achieved by the 90 core design (see Fig. 10). This involves apportioning the cores as follows:

- *Compute Gradient* function—36 IPPro cores
- *Weighted Vote into Spatial and Orientation cells* function—54 IPPro Cores

The IPPro code was manually compiled from the standard IPPro Instruction Set Architecture (ISA) [22]. The initial approach was to read all of the input variables

once, at the start of a program, and then produce the results in one group, at the end of the program. The newly profiled code is structured to read in only enough input variables to initialize the first calculations. This allows logical and arithmetic instructions to begin to execute, some of which require wait states due to the datapath which feature as NOPs. Where possible, these NOPs are replaced by read and write operations hence improving efficiency; this is a function that an automated compiler would employ. Similarly, the expectation was to remove the NOPs which are used to balance branching sections of the code. Unfortunately on simulation, it was found that this was not possible as in this instance, the branch was not taken and the READ or WRITE instructions would not be executed. This restructured code is now analysed and profiled for each function separately. The results are then summarized to show their contribution to the overall signal processing effort.

5.1 Normalize Gamma and Color

This step of the datapath in the image processing chain corrects linear characteristics of the sensor within the camera such that they align better with the human eye's perception of light intensities. In the instance of a digital camera when twice the number of photons strike the sensor, the generated signal is doubled; however, for our eyes, doubling the amount of light is only perceived as being slightly brighter. The outcome is that our vision can operate over a larger range of luminance which is required in the outdoors.

This non-linear relationship is approximated by taking the square root of the incoming pixels. If a large enough LUT is available, then the square root function can be avoided. In this instance, the 8-bit input pixel values are mapped to 16-bit locations yielding 256 possible results; thus a relatively small, 4096-bit (256×16-bit) LUT is required. The most time efficient solution is to use the pixel value to address the available LUT which contains the precomputed square root values from 0 to 255; this then requires one clock per input pixel.

Should the square root function be executed in native IPPro instructions and not by a bespoke LUT for each pixel, the IPPro would require of the order of 160 instructions to calculate each result. Later in this section, we detail the number of cycles required for the division and square root functions using native IPPro instructions; these account for the 160 instructions through shifts, compares, additions and loop control.

With this approximation, the square root function in native IPPro instructions is no longer a major overhead; it decreases the time taken for the *Normalize Gamma and color* function by a factor of 160. For this study, the *Normalize Gamma and color* function is performed by a LUT stored in a readily available BRAM within the ZYNQ device.

5.2 *Compute Gradients*

The x and y gradients of the pixel of interest are generated by means of convolution where the x-axis gradient is given by, $G_x = [1, 0 - 1]$ and the y-axis gradient by $G_y = [1, 0, -1]^T$. On examination of the matrices, it can be seen that they can be implemented by a single subtraction rather than three multiplication and an addition instructions. This simplification of the arithmetic reduces the *Compute Gradients* function down to a simple subtraction of the pixel values north and south, or east and west of the central pixel of interest.

The Gradient Magnitude M_{xy} of two vectors x and y is calculated by a variation of the Pythagoras' theorem and in our case, $M_{xy} = \sqrt{G_x^2 + G_y^2}$, is approximated as $M_{xy} \approx |G_x| + |G_y|$ [2]. As M_{xy} is only used as a scaling factor and the error scales linearly, the approximation error is negligible. Should M_{xy} be calculated using the ideal method, two multiplication, one addition and a square root instruction are required. Once again, if we approximate that the square root instruction takes 160 instructions, a total of 163 instructions are required versus the 19 instructions for the approximation (9 per absolute and 1 addition). This approximation is 8.6 times more efficient considering the number of instructions. After computing both the x-axis and y-axis gradients, the absolute values of the gradients are calculated in order to approximate the magnitude values.

For a 16 input pixel window as shown in Fig. 11, each IPPro core executes a single iteration of the *Compute Gradient* PM code, generating 30 outputs:

- 6 * G_x: Gradient in x-axis
- 6 * G_y: Gradient in y-axis
- 6 * g_x: Absolute value of gradient in x-axis
- 6 * g_y: Absolute value of gradient in y-axis
- 6 * M_{xy}: Magnitude of the gradient for pixel of interest

The number of pixels of interest calculated per loop of the related PM is only limited by the amount of registers available to store the calculated results at the end of the program loop. In this instance, 32 registers are available. As each pixel of interest generates 5 items of data, only 6 pixels of interest are analysed and reported per program loop.

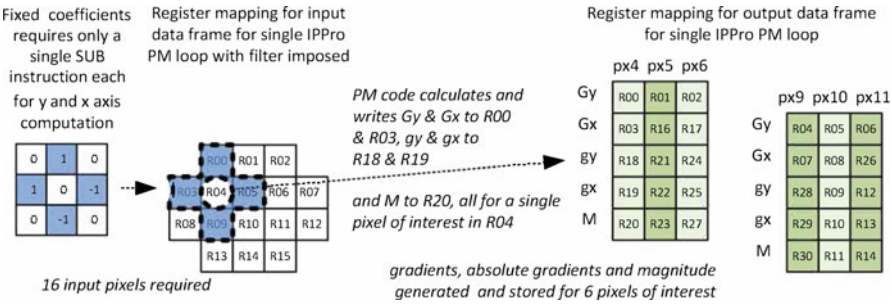


Fig. 11 Gradient calculation using simple coefficients

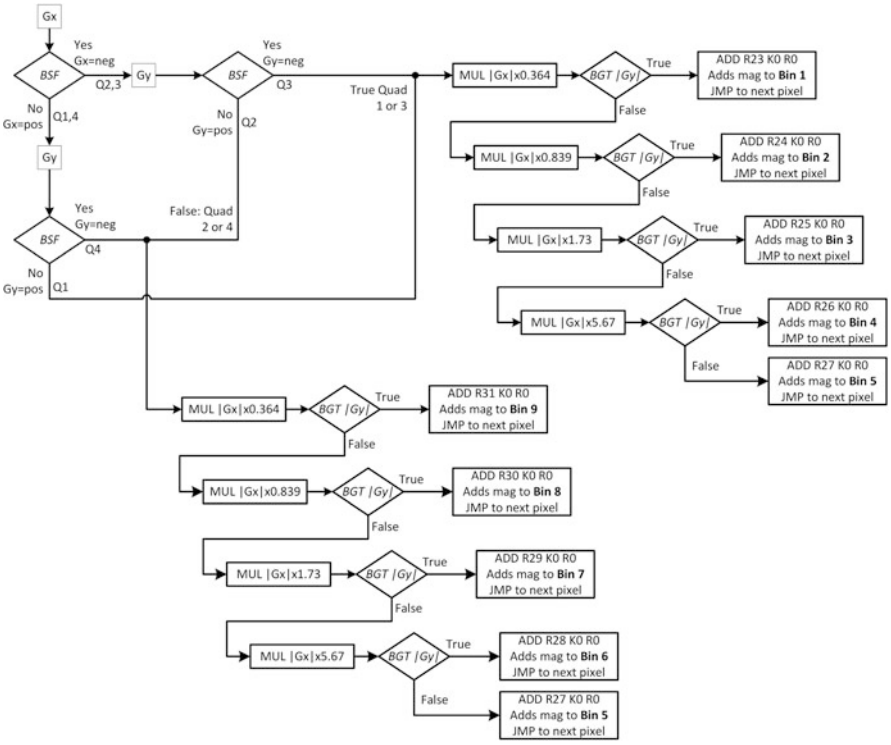


Fig. 12 Gradient magnitude quadrant evaluation algorithm

5.3 Weighted Vote into Spatial and Orientation Cells

The absolute gradient values G_x and G_y calculated in the previous stage are used as inputs to the M_{xy} quadrant evaluation logic of this stage and involved in determining to which quadrant the M_{xy} belongs (as illustrated in Fig. 8). When profiling, it is important to consider that each of the four equations of Fig. 8 is fully evaluated in order to maintain an overall balanced program. Whilst this makes the program deterministic in terms of execution time, it comes at the cost of increased instructions per loop. The IPPro system currently cannot handle variable length loops which are often presented by non-deterministic code. Such an adaptation would require complex control mechanisms which would act to both increase the resource usage and reduce the clock frequency. This overhead is not desired as the majority of image processing functions are deterministic.

To aid the translation of the logic equations described in Fig. 8 into IPPro instructions, a flow graph is constructed as shown in Fig. 12. The evaluation of values for G_x and G_y point towards two main branches which place the magnitude into bin group 1, 2, 3, 4 or 5 (Quadrant 1 or 3), or into bin group 5, 6, 7, 8 or 9

(Quadrant 2 or 4). the values, $|G_x|$ and $|G_y|$; thus, these can be used to establish which bin within the appropriate quadrant that the magnitude should be assigned to by evaluating five equations. Each of the two branches performs the same five evaluations.

Rather than evaluating a tangent function for each M_{xy}, the algorithm makes use of the post synthesis evaluated values of tan 20°, 40°, 60°and 80° to determine to which bin the magnitude of gradient belongs. Knowing M_{xy}'s precise angular position in the bin is not important, only its bin location. Note that two major branching possibilities for binning exist, but only one will be ever taken; however, the PM includes all the branch eventualities and has 229 instructions. The output from this functional section is the data for 128 histograms for one detection window, where each histogram describes the gradient magnitude profile for a single cell.

5.4 Normalize over Overlapping Spatial Blocks

Using a sliding window, the 128 histograms generated by the previous function are taken to normalize each cell with its surrounding three cells in that spatial block, as shown in Fig. 13. When the initial 9 × 16-bit histogram B_n is generated for an 8 × 8-pixel cell, B_n is then combined with the three surrounding cells of that block to make a single 36 × 16-bit histogram b_v. This is then normalized to produce b_n.

Normalisation is done using the L2 norm which helps to reduce noisy data to a linear approximation and is achieved in the following two steps;

$$b_n{}' = \frac{b_v}{\sqrt{|b_v|_2^2 + e}} \quad \text{then} \quad b_n = \frac{b_n{}'}{\sqrt{|b_n{}'|_2^2 + e}}.$$

Here the 16-bit datapath is configured for 10-bit integer data and 6-bit fractional data (1024 maximum and 0.015625 minimum); hence, all input data must be scaled up.

In the instance of the initial IPPro configuration there are 32 register locations. As a result, the normalisation can only be realized by performing it in three separate PMs, hence each block histogram requires three iterations of this PM to achieve the cell of interest normalisation against its three surrounding neighbours. One cell histogram consists of 9 × 16-bit register values, therefore it is not possible to store the entire block histogram (36 × 16-bit; 4 cells) in one IPPro core. Once again, in this profile, the IPPro is afforded 160 native IPPro instructions to process each DIV instruction. This segmentation of the function is necessary but results in an inefficient implementation.

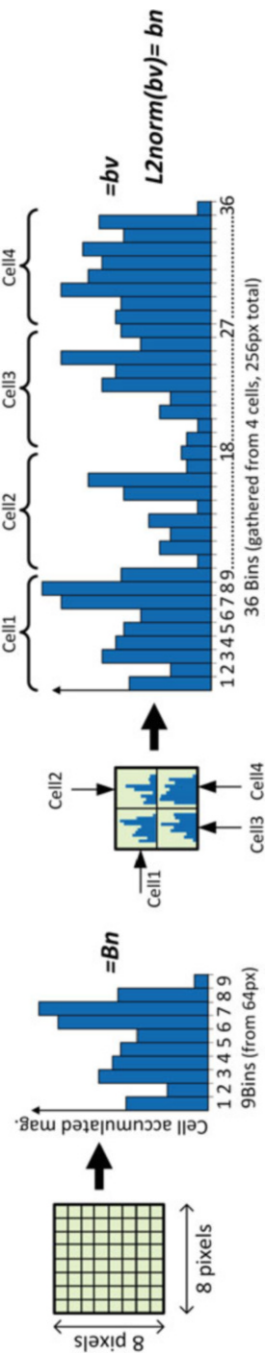


Fig. 13 Normalisation of a cell histogram with its surrounding three cell histograms

5.5 Collect HOGs over Detection Window

The prior *Normalisation over overlapping spatial blocks* group of calculations is executed 105 times per detection window producing a single vector of 3780(105 $*$ 36) $\times$ 16-bit elements per detection window (64 $\times$ 128 pixels).

A single 1920 $\times$ 1080 pixel HD frame contains (round up 8.43 to 9 hence 30*9) 270 detection windows, thus the *Normalisation over overlapping spatial blocks* calculation is executed 28,350 (105*270) times per HD frame. Note, however, that the dependency of this implementation of the HOG algorithm is limited to a detection window, so it is not necessary to calculate the entire HD frame at one time; the process can be passed across the HD frame one detection window at a time.

There is no specific calculation required to achieve this algorithm step other than looping the previous three steps. For this reason, the profile is set as zero effort at the host of the IPPro System (in this case the ARM processor in the ZYNQ fabric) controls the higher level memory management of the data generated from the previous stage.

5.6 Linear SVM

The HOG algorithm then applies the 3780 $\times$ 16-bit vector which describes a single detection window to the SVM. The SVM is trained offline with a known dataset to establish the required coefficients. The SVM is then configured with these preset coefficients in order to determine the likelihood of whether or not a pedestrian is present in the detection window. This training normally takes place in an external environment such as MATLAB but was not carried out in this study as it was not deemed relevant to the HOG algorithm execution profiling.

In the SVM, the 3780 coefficients are multiplied with the detection window's 3780 $\times$ 16-bit vectors and compared against fixed thresholds. The outcome of these comparisons generates a confidence rating on whether or not the detection window contains a human form or not.

5.7 Summary of HOG Profiling

The cumulative number of profiled instructions illustrated in Fig. 14 is 2,597,976. These instructions generate the HOG descriptors required to represent a single detection window. To generate HOG descriptors for a complete high definition (HD) frame, 701M IPPro instructions are required using the initial architecture.

The input data is 8-bit. In our example, the most efficient way to achieve the first functional block, *Normalize Gamma and color* function, is to use a look up table (256 $\times$ 8-bits in capacity) as it requires a square root function. As this is at the front end of the algorithm, there is no communications penalty for breaking in or out from

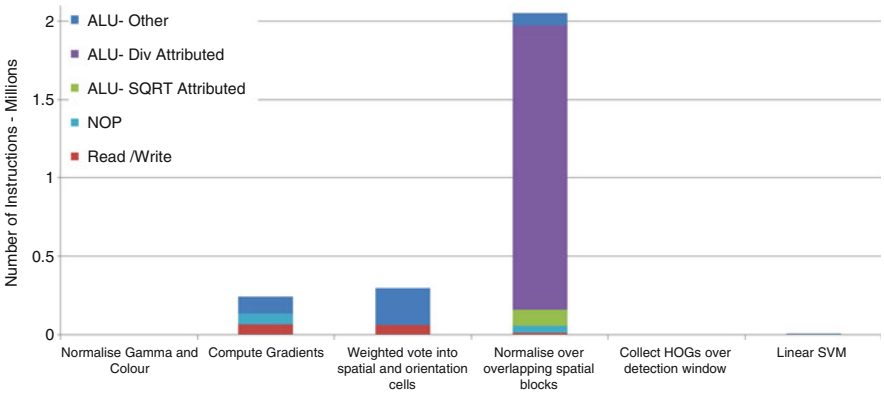


Fig. 14 Instruction profile of HOG implementation on IPPro

the IPPro datapath. For this reason, no instructions are reported in Fig. 14 for this function. Collecting HOGs over the detection window simply requires structuring of the data located in the BRAM. This task involves rearrangement of large datasets and is more suited to execution by the host ARM as it has access to larger memories. Similarly, whilst the histogram data is in higher level memory, we choose not to return the data to the soft-cores to execute the SVN as the communications delay incurred will outweigh any potential speed-up. Neither of these functions are computationally demanding and do not warrant being transferred to the IPPro for acceleration.

It must be noted that 77.3% of the total HOG implementation IPPro instructions belong to the *Normalize overlapping spatial blocks* function. More specifically, 72.2% of the total IPPro instructions belong to either the division or square root operations. The *Compute gradients* and *Weighted vote into spatial orientation* cells require 9.1% and 11.2% respectively of the total IPPro instruction count. This identifies the division and square root operations as being the computational bottleneck in the implementation and suggests that targeting these operations shall achieve the greatest positive impact on the performance.

6 IPPro Optimisations

In this section, the three optimisations are detailed and with reference to a single detection window, as this is considered to be the fundamental element in HOG and can be used to tile any frame size.

6.1 Register Size

The mapping strategy of Fig. 9 aims to reduce the transfer of data in and out of the core registers, whilst reserving adequate register locations to execute intermediate calculations before forwarding processed results to the next functional stage. This increases the utilization for processing over data transfer. Whilst the soft-core would benefit from larger local registers, this would have a negative impact on the FPGA clock rate. Current mid-range FPGAs provide 5–15 Mb of on-chip memory which does not permit storage of a full 33 Mb HD frame.

DDR transfers should ideally only occur at the start and end of the algorithm datapath. Unfortunately, as we can see from Fig. 3, some HOG functions require a large number of pixels. The initial IPPro 32 × 16-bit register configuration can only read blocks of 16 input pixels as 5 data values are generated for each processed input pixel during the *Compute Gradients* function (gradient convolution needs surrounding pixels to the pixel of interest). The limit is bounded by the IPPro core register size. A maximum of 6 pixels of interest (6 pixels × 5 = 30 data points) generating 30 values can be stored during a single PM loop as the register file size is limited to 32, hence the limitation of storage prior to output.

In the optimized IPPro core, functionality and instructions have been added to allow run-time read and write capability to the under-utilized 32 × 16-bit kernel registers by the ALU. This enables the generation of a maximum of 12 pixels of interest (12 pixels × 5 = 60 data points) as opposed to only 6 pixels of interest using a 32 location register file per PM loop. The increase in register size increases data reuse in the *Compute Gradients* function by reducing the total instructions per detection window for this function by 4% from 242,176 to 232,544. The revised architecture whilst producing the same output data required for a detection window, now achieves this in less instructions as a consequence of increasing the read/write to ALU ratio through better data reuse.

In the *Normalize* function, the cell of interest is normalized with its three surrounding cells by splitting into three consecutive, identical loops, one per surrounding cell due to the limitation of the IPPro register size. To store a block (4 cells), the histogram needs 4 × 9 × 16-bit register locations. The existing IPPro core cannot support this function within one core as the algorithm requires a minimum of 36 register locations. With the optimized core having access to 64 register locations, the *Normalize* function can be executed in a single core. More significantly, in the *Normalize* as opposed to the *Compute Gradients* function, an increase from 32 to 64 register locations is required to provide random access to the existing 32 × 16-bit kernel locations; this reduces the instructions for this function by 35% from 2,051,280 to 1,333,920 instructions.

In both the *Normalize* and *Compute Gradients* functions, the addition of read and write access to the Kernel resisters increases IPPro core efficiency through both data reuse and the reduction of read and writes instructions without the need for any additional register resource.

6.2 Mapping Strategy, Input Data Pattern

When mapping the HOG algorithm to IPPro, all valid permutations of input data patterns were explored; this allowed us to arrive at the most efficient core in terms of data processing i.e. the most efficient use of the limited number of registers, REG. This is specific to windowing operations only as these generally occur with an overlapping behaviour where a chosen pixel of interest uses a limited number of pixels in the area of interest. According to this concept, the Number of Outputs per Filter Window ($NOFW$) is given by REG divided by the number of required registers, FR, and represents the main optimization target.

The next step is to estimate register usage by considering the window function as well as input image size. The image size is important as the aspect ratio will define the amount of surplus computation executed in generating "wasted" pixels that are required to be read, computed and stored due to the overlapping behaviour and pixel packet processing of the algorithm mapping into the IPPro. Given an $X \times Y$ input window of pixels where most windowing operations use overlapping elements, an overlapping constant O_v is an important feature. The input window height, HI, and width, WI, define the input pixel pattern, whereas the processed pixels and the output pattern height, HO, and width, WO, define the output.

$$HI = HO + (O_v \times 2) \tag{2}$$

$$WI = WO + (O_v \times 2) \tag{3}$$

where WO's and HO's maximum number is defined by the term, $NOFW$, as

$$NOFW = HO \times WO. \tag{4}$$

Then the main aim becomes to find the optimum HO and WO considering the input window size and the register use:

$$Optimum \;\; Read \;\; Window = \left\{ \left\lfloor \frac{X}{WO} \right\rfloor \right\} * \left\{ \frac{Y}{HO} \right\}. \tag{5}$$

The pixel wastage can be defined as the rounding error which is calculated as shown in (6) and (7). The most optimum result for efficient register usage and execution time can be achieved by choosing the least wastage where the smallest rounding errors should be chosen by interchanging WO and HO values, if the result minimizes Δ_X and Δ_Y as below:

$$\Delta_X = \left\{ \frac{X}{WO} - \left\lfloor \frac{X}{WO} \right\rfloor \right\} \tag{6}$$

Table 4 Instruction profile for Compute Gradient function for all patterns

Input pattern		Reads	Instructions per PM	PM's per detection window		Instructions per detection window
x-Axis	y-Axis			x-Axis	y-Axis	
14	3	38	350	5.33	128	268,800
3	14	38	350	64	10.67	246,400
8	4	28	340	10.67	64	239,360
4	8	28	340	32	21.33	239,360
6	5	26	338	16	42.67	232,544
5	6	26	338	21.33	32	237,952

$$\Delta_Y = \left\{ \frac{Y}{WO} - \left\lfloor \frac{Y}{WO} \right\rfloor \right\}. \tag{7}$$

With the new 64 × 16-bit IPPro register configuration, it is possible to generate a maximum of 60 output data values from the 12 input pixels of interest for the *Compute Gradients* task. Only six possible combinations of input pattern achieve this maximum output and are detailed in Table 4.

In this instance, the *Compute Gradients* function can have a worst case pattern requiring 268,800 instructions or a best case pattern requiring 232,544 instructions per detection window. Conscious pattern choice has the potential to reduce the instruction count by 13.5% in this function without any architecture changes or additional resource usage.

6.3 Coprocessor Development

Earlier, it was established that only two of the six functional blocks of the HOG algorithm could be implemented when using the existing native IPPro instructions to explicitly translate the HOG algorithm from mathematical expressions into IPPro instructions. This is primarily due to the absence of division and square root instructions in the IPPro ISA. With slight alterations to the IPPro datapath detailed later in this section, long hand methods of division and square root are possible using the existing IPPro instructions. Our profiling, however, shows that they would be very time consuming and therefore not practical.

From a bespoke logic perspective, the non-restoring division algorithm [13] is the fastest and less complex of the radix-2 division algorithms [15]. As this supports our requirement of retaining a high throughput yet simple architecture, the non-restoring algorithm was chosen for our implementation. Variants such as SRT [14], Newton-Raphson, Gold Schmitt and CORDIC were examined and found to result in FPGA designs with slower clock rates and requiring resources in excess of the simple shift architecture based on the non-restoring algorithm [3].

The logic describing the non-restoring divider is easily achieved in the FPGA [3]. In the IPPro methodology, our datapath and register widths are constrained

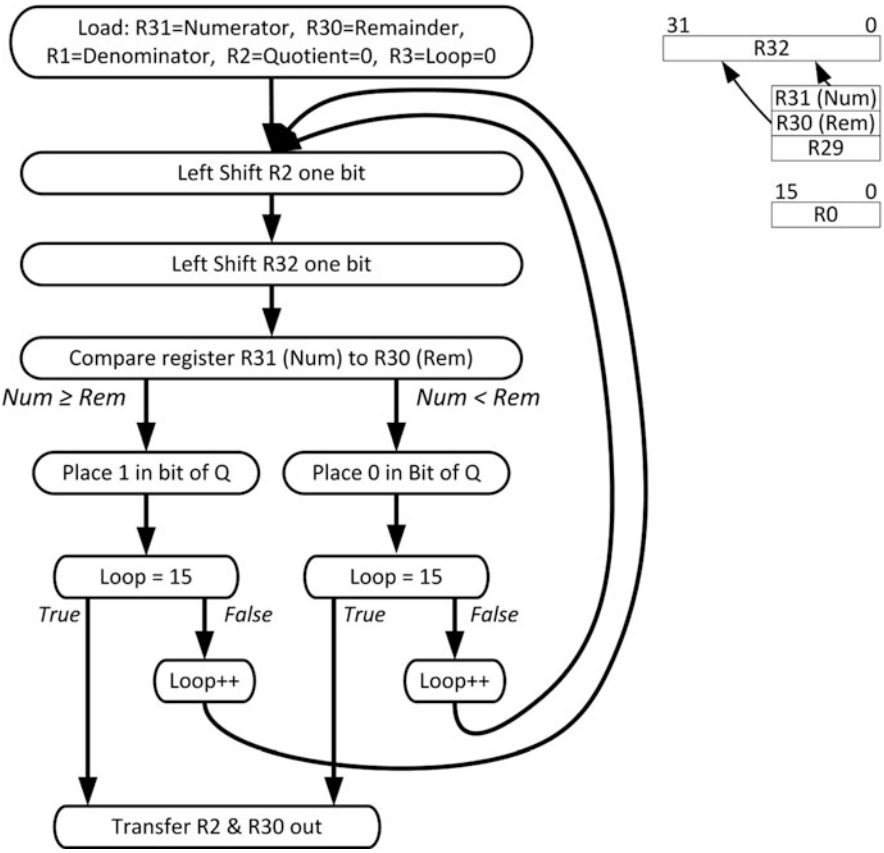


Fig. 15 Division algorithm using bespoke 32-bit register

to allow predictability and higher throughput. Prior to our profiling, two methods were considered for incorporating division in IPPro native instructions using this datapath.

The first practical method requires the addition of a 32-bit register to facilitate the left shift otherwise a single 16-bit division would require 151 instructions. This could be formed by a real-time copy of the top two registers R30 and R31 (Fig. 15), but would require the entire datapath to become 32-bit wide. Such an increase in the interconnect for the datapath would both significantly reduce the operational frequency and increase the resource usage.

The second practical method uses the existing 16-bit registers and datapath and requires 167 instructions, 16 more than the version requiring the addition of a 32-bit register as a consequence of the additional write into the second 16-bit register. We conclude that by using the current DSP48E1 on its own, it is not possible to significantly reduce the number of clock cycles required to execute a 16-bit division or square root function.

In the profiling exercise, a compromise between the two solutions offering 151 and 167 instructions of 160 instructions per division and square root function is used for all profiling calculations. As identified in the profiling instance of the HOG algorithm in Fig. 14, 79% of the total processing time is attributed to the *Normalize over Overlapping Spatial Blocks* operation and 70+% of the total algorithm time is spent on division. As the most significant contributor to instructions per detection window, the division function is the prime candidate for acceleration by means of bespoke logic. It is acknowledged that the square root operation is similar to division and can be implemented by similar logic, thus there is the opportunity for hardware sharing in a multifunctional coprocessor. In this study, the focus is solely on the division operation as it is the most influential.

6.4 Implementation of Coprocessor

The approach chosen has been to implement the division as a coprocessor in order to provide a speedup. Two architectural approaches have been considered, one that incorporates a single coprocessor shared temporally by an array of IPPro cores and the other where a single coprocessor is assigned to each IPPro core creating spatial parallelism (see Fig. 16). In both cases, a non-restoring integer, radix-2 divider which operates on 16-bit signed data was chosen as it provided a lightweight, fast implementation for small bit widths such as our 16-bit datapath.

For wider datapaths e.g. 32-bit and 64-bit, the SRT algorithm which requires a lookup table or an approximation algorithm such as Newton Raphson or Gold Schmidt, provides better performance at further FPGA resources expense but in the instance of 16-bit data they do not [3]. CORDIC dividers were also implemented but they too were found to require more LUTs than non-restoring methods when simple integer results were required. On a Xilinx Virtex 7 [7], the 16-bit Cordic divider required 117 LUTs with a maximum operating frequency, F_{max}, of 275 MHz.

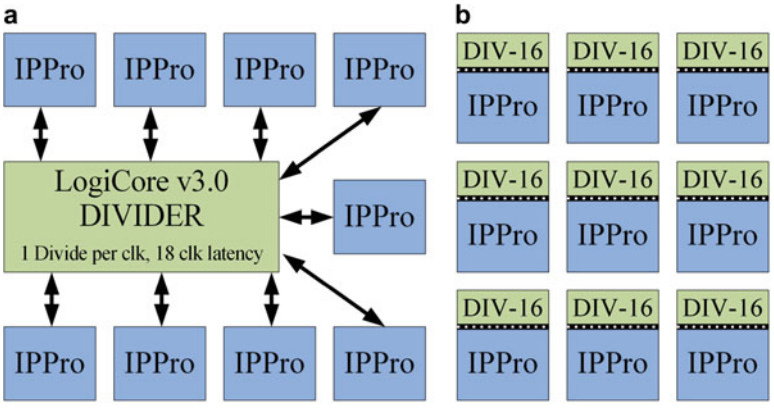


Fig. 16 Serial and parallel coprocessor topology. (**a**) Serial coprocessor. (**b**) Parallel coprocessor

Table 5 LogiCore Divider implementation performance

Clock cycles per division	Latency clocks	LUTs	F_{max} (MHz)	Divisions/s
1	18	463	328	328M
4	9	228	264	66M
4	4	385	125	31.25M
8	19	146	267	33.4M

6.4.1 Serial Coprocessor (Temporal Parallelism)

The main advantage of using a serial coprocessor is that any core can implement division at any time, whereas the parallel coprocessor needs to be shared and needs careful scheduling. Sharing the coprocessor does, however, potentially provide better utilization for longer periods of time, making more efficient use of LUTs. The LogiCore Divider (v3.0) [9] was implemented as it does not require an AXI bus interface and provides a simple parallel interface primarily consisting of DIVIDEND, DIVISOR, QUOTIENT, FRACTIONAL buses along with the required control lines. This IP achieves Radix-2 integer division using only LUTs. Four configurations detailed in Table 5 were considered and implemented.

High throughput of the serial architecture comes with disadvantages; in particular where single spurious divisions are required, a high penalty of 18 instructions cycle latency is experienced. As this type of coprocessor is highly pipelined, a larger footprint of 463 LUTs is required. In order to service many IPPro cores, a high coprocessor fan-out is required which impacts the overall clock speed. The 328 MHz result in Table 5 refers to the speed of the LogiCore v3.0 Divider when placed and routed in isolation and not connected to an array of 9 cores. Connecting the divider to 9 cores further reduces the F_{max}.

Results recorded in Table 5 show that higher division throughput for the Logicore divider is proportional to resource utilization (LUTs). This is due to the high level of pipelining required to achieve more divisions per clock. The 1 clock cycle per division configuration is ideal for algorithms with sustained division requirements but the high latency will be very inefficient when division is spurious and only occasional.

6.4.2 Parallel Coprocessor

The main advantage of a parallel divider topology is local computation and therefore minimal penalty for data transportation. Here the scheduler effort is light and there is no possibility of inter-core contention as there is with a serial divider. The parallel coprocessor also has a small footprint, namely 55 LUTs per IPPro core (89 LUTs with control and interfacing logic) which is just slightly greater than one ninth of the footprint of the serial coprocessor without interface and control logic. The parallel divider also has a higher frequency, F_{max}, than the 9 IPPro core divider.

Unfortunately, these advantages are challenged by the fact that the divider is not pipelined and requires 20 instructions cycles per division. Unlike the serial divider logic which will potentially be employed regularly during a typical algorithm execution by an array of cycling IPPro cores, the logic which makes up the parallel divider will be dormant for longer durations. This is an inefficient use of the resources should the design be challenged by logic resource, thus driving the decision to exclude the coprocessor from cores with no division requirement.

6.4.3 Architecture Choice

Whilst this latter option was not implemented, consideration was given to how streamlined version of the IPPro core could be used to execute only the instructions required for division. This solution frees up the main IPPro during division, but costs 160 cycles of latency. It is estimated that by removing the unused decode and control logic, a footprint of 180 LUTs is achievable. Combining the latency and excessive footprint issues makes this unviable. If the overall impact on resources is considered for an array consisting of 9 IPPro cores and coprocessor(s) arranged similar to that shown in Fig. 16, then the overall LUT footprint is calculated as per Fig. 17.

An 8% difference in the total LUT resource is not deemed significant enough to influence choosing serial over parallel implementation. The coprocessor architecture choice influences the overall algorithm implementation performance due to the overhead of interconnection and scheduling. For the normalization function in HOG, the serial processor both requires division at the same instance and also for the majority of the processing time within that functional block without stalling the IPPro Cores for considerable periods. Whilst a serial coprocessor has very high standalone potential in the instance of HOG, it will be impossible due to the required concurrent access. This can be considered to be a general case for most parallelized image processing algorithms hence ruling out a serial coprocessor.

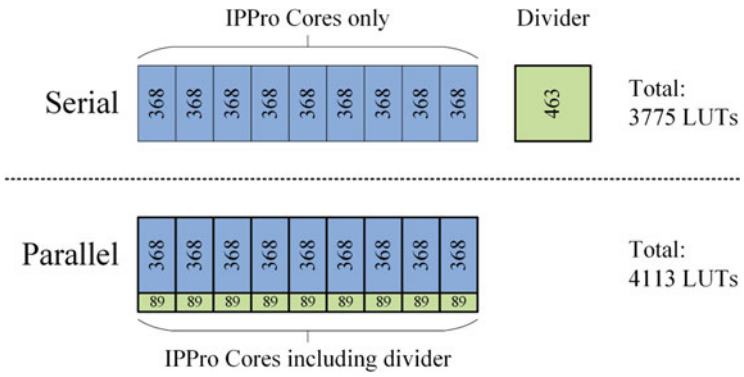


Fig. 17 Total cost of Divider architectures

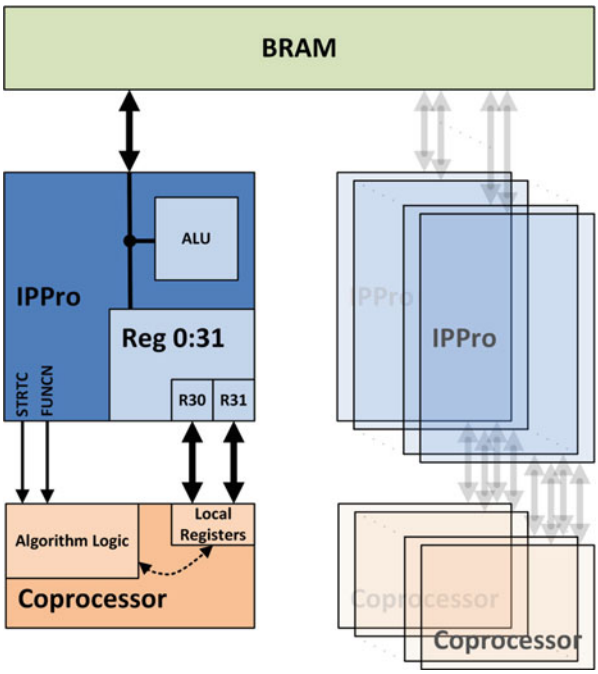


Fig. 18 Parallel coprocessor interface ports definition

Whilst the Parallel coprocessor has not the ability to reduce the instruction count to 1 where in the extreme cases the Serial coprocessor can, it still reduces the instruction count per division function to 16 plus the overhead of the interface. This is a significant reduction from the initial 160 instructions required when using native IPPro instructions. In conclusion, for ease of integration into a potential compiler and scheduling system and also due to the compactness of the Parallel implementation, IPPro shall utilize the Parallel Architecture.

6.4.4 IPPro Coprocessor Interface Design

A small compact multifunctional (division and square root) coprocessor was connected to each IPPro core. Two control signals are required; STRTC (Start coprocessor) and FUNCN (coprocessor function) as detailed in Fig. 18.

The interaction between the coprocessor and IPPro core is as follows and illustrated in Fig. 19;

1. Program memory decodes an instruction specific to the coprocessor (DIV).
2. IPPro core control logic asserts the STRTC signal and sets the FUNCN signal to the appropriate state depending on the function. For a dual function coprocessor, this only needs to be a binary signal.

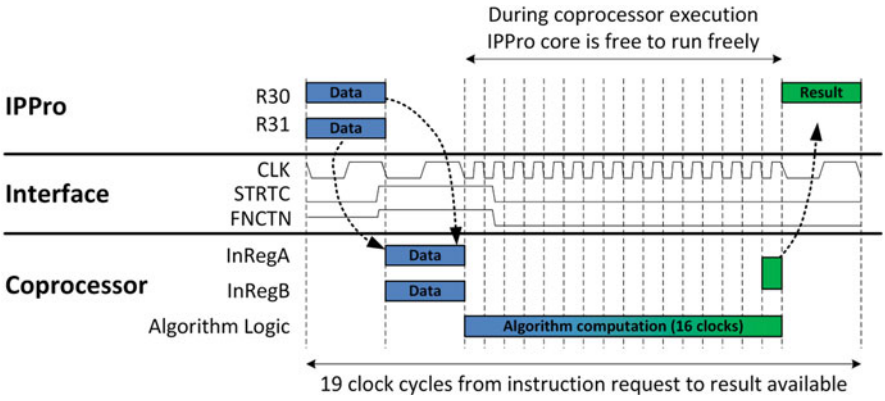


Fig. 19 Parallel coprocessor interface ports definition

3. Coprocessor copies the data in R30 and R31 into its local memory.
4. Coprocessor acts on the data.
5. Result is returned to IPPro core back into R30 after a known number of states.

The following must be observed during operation;

- Program code must be structured so that the operands targeted at the coprocessor are written (or copied) to R30 and R31 and the code is such that it expects the returned result at R30 after a predefined number of instructions.
- Deterministic coprocessor functionality, branching or interrupts do not happen within the coprocessor execution.
- Concurrent parallel code executing on the IPPro main core is not dependent on the coprocessor result (which can only be valid after a defined wait period).

6.4.5 Summary of Coprocessor Impact

In addition to reducing the instruction count for division from 160 to 19 instructions, a coprocessor solution also allows the main IPPro core to continue operation in the background. Furthermore, the division is available in the coprocessor FIFO for up to 16 cycles after the first division result is delivered, hence giving the scheduler greater freedom to efficiently organize the fine grained tasks of the algorithm. Unfortunately, this cannot be exploited in the HOG algorithm as all division instructions have to be calculated sequentially, but it should be useful in other algorithms with different arithmetic patterns.

Using a coprocessor to off load the division effort of the *Normalize over Overlapping Spatial* function shows that our preferred parallel coprocessor implementation reduces the associated IPPro Core instructions by 82% from 1,344,420 to 246,120 for a single detection window. This reduction assumes the previous optimisation of

Table 6 Divider coprocessor resource and power

	Core parameters			Energy power	
IPPro core implementation	LUTS	DSPs	Clock (MHz)	Latency	Energy (nJ)
Non-coprocessor	368	1	337	183	4.3
With parallel coprocessor	457	1	337	18	0.6

increasing the registers to 64 × 16-bit has already been incorporated. This saving is attributed to the introduction of the coprocessor at the cost of 89 LUTs per core as shown in Table 6.

In order to maintain the 175 fps that was achieved by the initial 54 + 36 core acceleration of the Compute Gradients and Weighted vote into spatial and orientation cells functions (detailed in Sect. 3) either 26 accelerated IPPro cores or 101 standard IPPro cores are required. The ability of the coprocessor to significantly improve the throughput for minimal resource increase widens the system architect's design space in a positive manner.

7 Conclusions

The design and implementation of a HOG using a lean, FPGA-based soft-core called IPPro is presented. It is shown how a performance of 15.3 fps can be achieved using a multicore processor implementation which compares high favorably with handcrafted VHDL code. A detailed profiling of the implementation is then carried out, showing that a number of optimisations in terms of increased register file size, profiling of the input data to remove data redundancy and introduction of an arithmetic coprocessor can act to considerably improve performance.

Acknowledgements This work has been undertaken in collaboration with Heriot-Watt University in a project funded by the Engineering and Physical Science Research Council (EPSRC) through the EP/K009583/1 grant. Colm Kelly has received support from Thales Air Defence.

References

1. Woods R, McAllister J, Lightbody G and Yi Y (2017) FPGA-based Implementation of Signal Processing Systems. 2nd edn. Wiley, UK.
2. Jain R, Kasturi R and Schunck B G (1995) Machine Vision. McGraw-Hill, Inc.
3. Deschamps J P, Sutter G D and Cantó E. (2012) Guide to FPGA Implementation of Arithmetic Functions. Springer.
4. Xilinx Inc. (2016) System Generator for DSP. Available via http://www.xilinx.com. Cited 29 April 2017.
5. MathWorks (2016) HDL Coder. Available via http://uk.mathworks.com/products/hdl-coder/index.html. Cited 29 April 2017.

6. McKinsey and Company (2012) McKinsey on Semiconductors. Available via http://www.mckinsey.com. Cited 29 April 2017.
7. Xilinx Inc. (2015) DS183: Viretx-7 and XT FPGAs Data Sheet: DC and AC Switching Characteristics. Available via http://www.xilinx.com. Cited 29 April 2017.
8. ARM Ltd. ARM7TDMI Technical Reference Manual (ARM DDI 0029G). Available via http://www.atmel.com. Cited 29 April 2017.
9. Xilinx Inc. (2011) LogiCORE IP Divider Generator v3.0. Available via http://www.xilinx.com. Cited 29 April 2017.
10. Texas Instruments (2010) TMS3206678 Rev.E. Available via http://www.ti.com. Cited 29 April 2017.
11. Eker J and Janneck J (2003) CAL language report. University of California at Berkeley Technical Report UCB/ERL M, (3).
12. Blair C, Robertson N M and Hume D (2013) Characterizing a Heterogeneous System for Person Detection in Video Using Histograms of Oriented Gradients: Power Versus Speed Versus Accuracy. IEEE Journal on Emerging and Selected Topics in Circuits and Systems. 3(2), 1236–247.
13. Oberman S F and Flynn M (1997) Division algorithms and implementations. IEEE Transactions on Computers, 46(8), 833–854.
14. Robertson J E (1958) A New Class of Digital Division Methods. IRE Transactions on Electronic Computers, EC-7(3), 218–222.
15. Macii E, Paliouras V and Koufopavlou O (2004) Power Aware Dividers in FPGA. Proc. of Power and Timing Modeling, Optimization and Simulation, 574–584.
16. Thomas D B, Howes L and Luk W (2009) A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation. Proc. of ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 63–72.
17. Dalal N and Triggs B (2005) Histograms of oriented gradients for human detection. Proc. of IEEE Conference on Computer Vision and Pattern Recognition, 886–893.
18. Hahnle M, Saxen F, Hisung M, Brunsmann U and Doll K (2013) FPGA-Based Real-Time Pedestrian Detection on High-Resolution Images. Proc. of IEEE Conference on Computer Vision and Pattern Recognition, 629–635.
19. Bauer S, Brunsmann U and Schlotterbeck-Macht S (2009) FPGA Implementation of a HOG-based Pedestrian Recognition System. Proc. of IMPC-Workshop, Karlsruhe.
20. Xie S, Li Y, Jia Z and Ju L (2013) Binarization based implementation for real-time human detection. Proc. of International Conference on Field-Programmable Technology, 1–4.
21. Kadota R, Sugano H, Hiromoto M, Ochi H, Miyamoto R and Nakamura Y (2009) Hardware Architecture for HOG Feature Extraction. Proc. of International Conference on Intelligent Information Hiding and Multimedia Signal Processing, 1330–1333.
22. Siddiqui F M, Russell M, Bardak B, Woods R and Rafferty K (2014) IPPro: FPGA based image processing processor. Proc. of IEEE Workshop on Signal Processing Systems, 1–6.
23. Kelly C, Siddiqui F M, Bardak B and Woods R (2014) Histogram of oriented gradients front end processing: an FPGA based processor approach. Proc. of IEEE Workshop on Signal Processing Systems, 1–6.
24. Negi K, Dohi K, Shibata Y and Oguri K (2011) Deep pipelined one-chip FPGA implementation of a real-time image-based human detection algorithm. Proc. of International Conference on Field-Programmable Technology, 1–8.

Part III
Design Methods and Tools

Methods and Tools for Mapping Process Networks onto Multi-Processor Systems-On-Chip

Iuliana Bacivarov, Wolfgang Haid, Kai Huang, and Lothar Thiele

Abstract Applications based on the Kahn process network (KPN) model of computation are determinate, modular, and based on FIFO communication for inter-process communication. While these properties allow KPN applications to efficiently execute on multi-processor systems-on-chip (MPSoC), they also enable the automation of the design process. This chapter focuses on the second aspect and gives an overview of methods for automating the design process of KPN applications implemented on MPSoCs. Whereas previous chapters mainly introduced techniques that apply to restricted classes of process networks, this overview will be dealing with general Kahn process networks.

1 Introduction

Multi-processor system-on-chip (MPSoC) is one of the most promising and solid paradigm for implementing embedded systems for signal processing in communication, medical, and multi-media applications. MPSoC platforms are heterogeneous by nature as they use multiple computation, communication, memory, and peripheral resources. They allow the parallel execution of (multiple) applications and, at the same time, they offer the flexibility to optimize performance, energy consumption, or cost of the system. Nevertheless, to optimize an MPSoC in the presence of tight time-to-market and budget constraints, a systematic design flow is required.

To deal with this challenge, Kienhuis et al. [1] suggested to structure the design flow in a certain manner, now commonly referred to as the Y-chart approach. It is a systematic methodology for selecting an embedded system implementation from a set of alternatives, a process often denoted as design space exploration. One key idea underlying this approach is to explicitly separate application and architecture

I. Bacivarov · W. Haid · K. Huang · L. Thiele (✉)
Computer Engineering and Networks Laboratory, ETH Zurich, Zurich, Switzerland
e-mail: wolfgang.haid@tik.ee.ethz.ch; huangk36@mail.sysu.edu.cn; thiele@ethz.ch

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_19

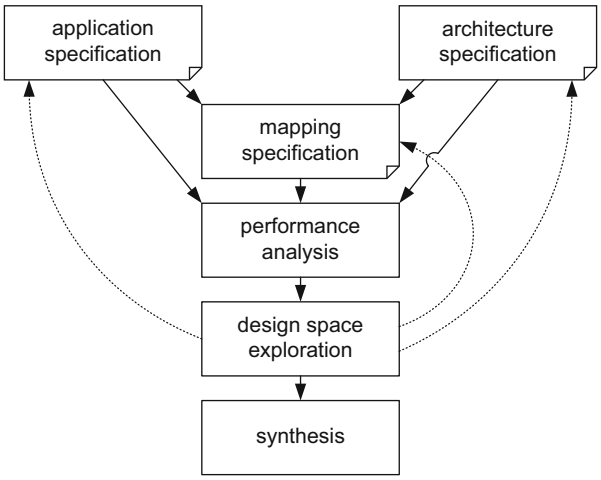


Fig. 1 Y-chart approach for designing MPSoC

specifications. A separate mapping specification describes how the application is spatially (binding) and temporally (scheduling) executed on the architecture. Design space exploration is then performed by iteratively analyzing and optimizing the application, the structure of the underlying (hardware) architecture as well as candidate mappings, as shown in Fig. 1.

Many design flows implementing the Y-chart approach have been proposed. For a review, see [2]. These flows have in common that they impose a set of system-level concepts to facilitate design space exploration, such as the use of a formal model of computation, providing restrictions on the set of scheduling policies, and relying on modular specifications. For instance, the application may be formally specified as a data flow model, a synchronous model, or a discrete event model, in order to enable automated performance analysis. In a similar way, resource sharing policies may be limited to an event-triggered or a time-triggered policy, to prune the design space. Finally, using modular system-level specifications will enable quick system modifications concerning the application, architecture, and mapping.

In the context of (array) signal processing applications executing on MPSoC, the Kahn process network (KPN) model of computation [3] is frequently used. Assuming a network of autonomous, concurrently executing processes that communicate point-to-point via unbounded FIFO channels, the KPN model has additional favorable properties. The KPN model is determinate, i.e. the functional behavior is independent on the scheduling of processes. The inter-process communication via FIFO channels using blocking read semantics can be efficiently implemented either in software, hardware, or in heterogeneous HW/SW systems. Computation and control are completely distributed, requiring no global synchronization, communication, or memory. The resulting modularity allows applications to be scaled easily and opens up many degrees of freedom for implementing a system.

Due to these properties, the KPN model of computation is "compatible" with the Y-chart approach and has led to numerous design flows. Although they share the same model of computation, these design flows consider different design objectives, they focus on different aspects, and leverage different properties of the KPN model or one of its subclasses. In this chapter, an overview of KPN-based design flows is given, emphasizing both, similarities and differences in these flows. The following section reviews existing design flows and the way they relate to the Y-chart. Afterwards, a closer look at individual steps in the design flow is taken and several methods to tackle them are presented. Finally, for exemplification, a specific design flow is considered in detail.

2 KPN Design Flows for Multiprocessor Systems

Several design flows based on the Y-chart approach and the KPN model have been developed. Table 1 shows a (non-exhaustive) list of design flows targeted at the implementation of KPN applications on MPSoC platforms. In addition to the listed design flows, there are other approaches related to the KPN model with different aims. Ptolemy [4] and Metropolis [5] allow the analysis and simulation of applications specified as KPNs, among other models of computation. However, they are targeted more towards hardware/software codesign and in particular towards the system synthesis and verification. The design space exploration is not the main focus of these frameworks. The Mathworks Real-Time Workshop [6] and the National Instruments LabVIEW Microprocessor SDK [7] target the implementation of signal processing applications on single-processor systems. SystemCoDesigner [8] and PeaCE [9] are HW/SW codesign flows based on a model of computation that combines the KPN model with finite state machines, see Ref. [10]. Note that even though the focus of this chapter is on the design flows for MPSoC listed in Table 1, many of the presented ideas also apply to the other mentioned design flows.

Generally, KPN design flows for MPSoCs respect the four design phases of the Y-chart: system specification, performance analysis, design space exploration, and system synthesis, as shown in Fig. 1.

Based on these four phases, the design process can be described as follows: The starting point of the design flow is a parallelized KPN specification of the application. In this specification, the coarse-grain data and functional parallelism of the application is made explicit. Fine-grained word or instruction-level parallelism can effectively be handled by today's compilers. Usually, the KPN is manually specified by the programmer. There are, however, also tools available that allow deriving a KPN from sequential programs, such as the Compaan [19] and pn [20] tools. KPN design flows usually provide a functional simulation capability that enables the execution of KPN specification on a standard single-processor machine in a multi-tasking environment. Due to the determinacy of KPNs, the timing-independent functionality of the application can be validated this way.

Table 1 KPN design flows for MPSoCs

Design flow	Web page
Artemis [11]	http://daedalus.liacs.nl
Distributed operation layer (DOL) [12]	http://www.tik.ee.ethz.ch/~shapes/dol.html
Embedded system-level platform synthesis and application mapping (ESPAM) [13]	http://daedalus.liacs.nl
Koski [14]	Not available online
Multiapplication and multiprocessor synthesis (MAMPS) [15]	http://www.es.ele.tue.nl/mamps
Open dataflow (OpenDF) [16]	http://opendf.sourceforge.net
Software/hardware integration medium (SHIM) [17]	Not available online
StreamIt [18]	http://www.cag.lcs.mit.edu/streamit

Second, the architecture needs to be specified. This is frequently done in form of a system-level specification describing the architectural resources, such as processors, memories, interconnects, and I/O devices. This specification can either describe a fixed MPSoC or the template of a configurable MPSoC platform. In both cases, the architecture specification needs to contain all the information required for design space exploration and performance analysis. In the case of a configurable platform, the architecture specification is also the basis for the synthesis of the final target platform later in the design flow. Hence, it needs to contain information required by the RTL synthesis tool, such as references to VHDL or Verilog code of hardware components, complete IP blocks, and configuration files.

The application and architecture specification phase is followed by defining a mapping of the application onto the architecture. In this step, processes are bound to processors and channels are bound to communication paths containing memories and interconnects. In addition, the scheduling and arbitration policies for shared resources are defined.

Usually, the final mapping is the result of a design space exploration, which is done based on the system performance analysis. The methods applied for performance analysis range from simple back-of-the-envelope calculations to formal analysis methods, simulations, and measurements. In KPN design flows, performance analysis during design space exploration is possible and is usually done at a rather high level of abstraction. As shown in the next section, different methods targeted towards KPN applications have been proposed in this context that achieve high accuracy within short analysis times. Being able to defer the use of simulation or measurements until late in the design cycle is one of the key advantages of KPN design flows.

After manual or automated design space exploration, the system is finally implemented by making use of appropriate synthesis techniques. For this purpose, KPN design flows feature powerful synthesis tools that implement a system based on the application, architecture, and mapping specification in software, hardware,

Table 2 Case studies that use some of the design flows from Table 1

Design flow	Case study application	Target platform	Performance analysis	Exploration method
Artemis [21]	Motion-JPEG encoder	Molen architecture on Xilinx Virtex-II Pro FPGA	Trace-driven simulation	Evolutionary algorithm
DOL [12]	MPEG-2 decoder	Atmel DIOPSIS 940	Real-time analytic model	Evolutionary algorithm
ESPAM [13]	Motion-JPEG encoder	Multi-MicroBlaze on Xilinx Virtex-II Pro FPGA	Measurement	Exhaustive search
Koski [14]	WLAN terminal	Multi-NIOS on Altera Stratix-II FPGA	High-level simulation	Simulated annealing
MAMPS [15]	H263 and JPEG decoders	Multi-MicroBlaze on Xilinx Virtex-II Pro FPGA	High-level simulation	Dedicated heuristic
OpenDF [22]	MPEG-4 SP decoder	FPGA (no particular type specified)	Not applicable	Not applicable
SHIM [23]	JPEG decoder	Sony/Toshiba/IBM Cell BE	Not applicable	Not applicable
StreamIt [18]	12 streaming applications	RAW architecture	SDF analytic model	Simulated annealing

or both hardware and software. Clearly, this is a key advantage of KPN design flows because the pitfalls of implementing a parallel system, such as hardware-software interface generation, deadlocks, starvation, and data races are handled in an automated way.

The design flows listed in Table 1 implement this basic Y-chart approach in different ways: On the one hand, the methods that are applied in each of the four phases differ between the design flows, as discussed in the next section. On the other hand, the scope (set of optimization variables) of design space exploration is different. Basically, one can distinguish between software design flows, where the target platform is fixed, and hardware/software co-design flows, where a template of a target platform is given and the instantiation of a specific platform is part of the design space exploration. This is shown in Table 2 where a few case studies are summarized that have been performed using the design flows listed in Table 1. DOL, SHIM, and StreamIt assume fixed hardware platforms, whereas the scope of the other design flows encompasses the implementation of the target platform on FPGAs.

3 Methods

KPN design flows attempt to assist a system designer in implementing an application as a hardware/software system by offering support for several activities such as:

- system specification,
- system synthesis,
- performance analysis, and
- design space exploration.

For each of these activities, methods have been proposed that differ in goal, scope, degree of automation, and complexity. In the previous chapters, mainly methods for subclasses of KPNs have been discussed. In this chapter, we give an overview of methods that are applicable to general KPNs in the context of MPSoCs. For each of the activities mentioned above, we discuss the challenges and proposed solutions.

3.1 System Specification

Developing applications that run correctly and efficiently on MPSoCs is challenging. The difficulty consists in finding an appropriate level of abstraction that balances the conflicting goals of (a) developing applications in a productive manner and of (b) enabling efficient automated implementation. While productivity is usually achieved by programming at a high abstraction level, efficiency is usually achieved by optimizing code at a low abstraction level. Many case studies provide evidence that for streaming applications, the KPN model of computation achieves a good trade-off between these two goals. On the one hand, streaming applications can often naturally be modeled as a KPN which promotes productivity. On the other hand, runtime environments have been developed that efficiently implement processes and channels.

Specifically, the KPN model can be seen as a *coordination* model [24] which considers the programming of a distributed system as the combination of two distinct activities: the actual *computing* part comprising a number of processes involved in manipulating data and a *coordination* part reflecting the communication and cooperation between processes. The coordination model allows reuse of components because the application programmer can easily build new algorithms by a new composition of existing processes. Furthermore the coordination model allows applications to be ported to different target architectures because usually only the glue-code that implements the coordination part is architecture dependent.

Due to these reasons, KPN applications are usually specified in a way that reflects the coordination model. Two different approaches can be distinguished, namely specification using a host as well as a coordination language, and specification using a domain-specific language. When using distinguished host and coordination languages, the KPN processes are specified in a host language (often in C or C++) whereas the coordination part is specified separately using a coordination language (often in XML or UML). This is, for instance, the approach taken in the Artemis and DOL design flows, where C and XML are used. When using a domain-specific language, computation and coordination are expressed in a single language that provides constructs for both parts. OpenDF and StreamIt, for instance, are based on domain-specific languages.

In both cases, applications are usually expressed based on the principles of encapsulation and explicit concurrency: Each process completely encapsulates its own state together with the code that operates on it and operates independently from other processes in the system, except for the data dependencies that are made explicit by channels. This allows for modular, scalable, and platform-independent application specifications.

System specification for KPNs is thus different from two other frequently used approaches for MPSoC software development, namely specification based on a board support package and specification based on a high-level application programming interface (API). When developing an application based on the board support package that is usually shipped with an MPSoC, the abstraction level is rather low. The focus is thus often on correctly implementing an application using low-level primitives for initialization, communication, or synchronization, rather than on optimizing an application. When using a high-level API the designer is relieved from dealing with low-level details (provided that the API has been ported to the target MPSoC). Compared to the KPN based approach, however, automatically optimizing programs written using a high-level API, such as MPI or OpenMP, is more difficult: Due to the lack of an underlying model of computation, the basis for automatically analyzing and optimizing a program is essentially missing.

3.2 System Synthesis

The Y-chart approach opens a gap between the system-level specification and the actual implementation of the design, sometimes referred to as the implementation gap. The challenge in bridging this gap is to preserve the KPN semantics on the one hand and achieve the desired performance on the other hand. Also, the pitfalls of parallel programming, such as deadlocks, starvation, and data races need to be handled. This is the task of system synthesis.

Different approaches for the synthesis of KPNs for software, hardware, and in combined hardware/software platforms have been proposed. The target architectures have been comprised of single-processors, multi-processors, and FPGAs. In all cases, system synthesis deals with the implementation of processes and channels as well as the arbitration of resources in case that processes and channels are mapped to shared resources. If not all parameters of an implementation are fixed before synthesis, the remaining degrees of freedom need to be exploited during system synthesis. In that case, system synthesis is often considered as an optimization problem where frequently considered optimization goals are the minimization of code size, the minimization of buffer requirements, or finding the schedules that minimize delays and maximize system throughput.

While many of these problems can be solved only for restricted subclasses, a few observations apply to general KPNs:

- First, KPN applications can be efficiently implemented on architectures with different processor, interconnect, and memory configurations, as shown in Table 2. As an example, KPN applications can be implemented on distributed memory (message-passing) architectures as well as shared memory architectures. The FIFO communication can be implemented using dedicated hardware FIFOs or buses, but also more complex communication topologies, such as hierarchical buses or networks-on-chip.
- Second, KPN applications can be executed in a purely data-driven manner based on their determinacy. This means that resources can operate independently from each other without any global synchronization. Pair-wise synchronization is only needed between processes that are directly connected by channels. From another perspective, this means that KPN applications can be scheduled with any scheduling policy that prevents deadlocks, i.e., preemptive, non-preemptive, or cooperative scheduling could be used. Due to these very relaxed requirements, KPN applications can usually be implemented easily on top of existing (real-time) operating systems. On the other hand, implementing a runtime-environment for a new platform from scratch is also possible because not many services need to be provided by the runtime-environment.
- Third, KPN applications can be easily partitioned into processes running in hardware and processes running in software. This is due to the parallel specification of the KPN application on the one hand, and due to the simple interaction of processes over FIFO channels on the other hand which facilitates the synthesis of the HW/SW interface.

The observations above indicate that synthesizing a KPN is conceptually not a very difficult task. Implementing a KPN based on a multi-processor operating system, for instance, is rather simple: Processes can be implemented as operating system processes or threads, and channels can be implemented using existing inter-process communication schemes. The difficulties in KPN synthesis origin from optimizing an implementation by minimizing the overhead for FIFO communication and the runtime environment. This can be achieved by considering low-level details of an implementation, for instance by efficiently using the hardware communication infrastructure (e.g. DMA engines) or by efficiently using the memory hierarchy (e.g. caches or scratchpad memories). On the other hand, optimizations can also be done at a high level, for instance, by (automatically) adjusting the granularity and topology of a KPN to the target architecture. This includes the replication of processes to increase the parallelism in a KPN or the merging of processes to reduce inter-process communication. We refrain from giving further details here and refer to the previous chapters for details on applicable techniques.

Finally, a further problem needs to be considered in the synthesis of KPNs: The denotational semantics of the Kahn model is based on FIFO channels with unbounded capacity. Since unbounded channels cannot be realized in physical implementations, however, KPNs need to be transformed in a way that allows for an implementation on channels with finite capacity. It can be shown that an operational semantics of KPNs based on channels with finite capacity matches the denotational

semantics when artificial deadlocks can be avoided. An artificial deadlock is a deadlock caused by one or more channels having insufficient capacity. Due to the Turing-completeness of KPNs, it is in general not possible to determine sufficient channel sizes at design time, however. One possibility to deal with this situation are runtime approaches that detect and resolve artificial deadlocks during execution [25]. Another possibility is to restrict the communication behavior of the processes such that the channels become amenable to analysis at design time.

3.3 Performance Analysis

During the design process, a designer is typically faced with questions such as whether the timing properties of a certain design will meet the design requirements, what architectural element will act as a bottleneck, or what the memory requirements will be. Consequently, one of the major challenges in the design process is to quantitatively analyze specific characteristics of a system, such as end-to-end delays, buffer requirements, throughput, energy consumption, or temperature rises due to application activities. We refer to this analysis as performance analysis.

The performance analysis of KPNs executing on MPSoCs poses a major challenge due to multiple and heterogeneous hardware resources, the distributed execution of the application, and the interaction of computation and communication on shared resources. To deal with these challenges, multiple methods have been successfully used in the context of KPN design flows. These methods differ in accuracy, evaluation time, set-up effort, and scope.

In Fig. 2, the scope of different performance analysis methods is compared. Leftmost, the interval of values for a performance metric as occurring in the

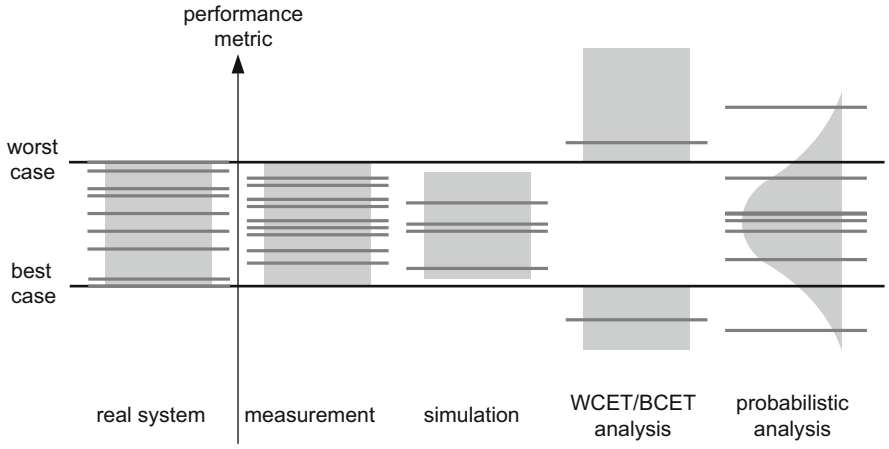


Fig. 2 Scope of different performance analysis methods for MPSoC

real system is shown. This performance metric could be the end-to-end delay of a system, the utilization of a computation or communication resource, or the occupation of a channel buffer, for instance. Different performance analysis methods now differ regarding the values that can be obtained.

When taking measurements of the real system, the measured values only represent a subset of all possible values. Most likely, due to insufficient coverage of corner cases and the limited number of measurement samples, the interval bounds can only be estimated based on the measurements. This observation applies to simulation as well. Best-case and worst-case analysis methods take a different approach by providing safe results about the interval bounds, i.e. upper and lower bounds on the worst-case and best-case behavior, respectively. On the other hand, usually not all parts of a system can be accurately modeled. In that case, (safe) optimistic and pessimistic assumptions need to be made, leading to bounds on system performance measures that are not tight. Finally, also probabilistic methods are used to provide quantitative statements about system behavior. In the following, we take a closer look at simulation and best-case/worst-case analysis due to their frequent application in KPN design flows.

Simulation is presumably the most frequently used method for performance analysis. This is reflected by the availability of a wide range of simulation tools that are applicable to different levels of abstraction. The most accurate but also slowest class are cycle-accurate simulators. Instruction-accurate simulators (also referred to as instruction-set simulators or virtual platforms) provide a good trade-off between speed and accuracy which allows entire MPSoCs to be modeled and simulated. An example is the so-called full system simulator of the Cell Broadband Engine which also allows switching between different simulation modes with different accuracies [26]. Besides performance analysis, virtual platforms can also be used for software development and debugging. For this purpose, the full system simulator of the Cell Broadband Engine provides a fast, purely functional simulation mode in which timing is not considered.

At higher levels of abstraction, also other kinds of simulation are used for performance analysis. One example is trace-based simulation in the Artemis design flow [11] or in DOL [27], for instance. In trace-based simulation, first an untimed execution trace of the application is recorded that contains computation and communication events of processes and channels. Based on an architecture description, the mapping of the application onto the architecture, and estimates about the time to process events, this trace is refined towards timing behavior. This technique allows designers to estimate the system performance. Depending on the level of detail in the trace and the modeling of the execution platform, estimation errors of less than 5% have been reported with a significantly reduced simulation time compared to instruction-accurate simulation.

For the design of hard real-time systems, worst-case guarantees on the system timing need to be given. As stated above, worst-case bounds are difficult to obtain from simulation due to insufficient corner case coverage and often prohibitively long execution times of a simulation run. Therefore, analytic methods appear to

be a promising method for providing worst-case guarantees even in the case of complex and large-scale MPSoC implementations. Prominent methods for analytic performance analysis are listed in the following.

- *Holistic Methods:* Holistic analysis is a collection of techniques for the analysis of distributed systems. The principle is to extend concepts of classical single-processor scheduling theory to distributed systems, integrating the analysis of computation and communication resource scheduling. Several holistic analysis techniques have been aggregated in the modeling and analysis suite for real-time applications (MAST) [28].
- *Compositional Performance Analysis Methods:* The basic idea of compositional performance analysis methods is to construct an analysis model of small components and propagate timing information between these components. Typical components model the execution of processes on a processor, the transmission of data packets on interconnects, or traffic shapers. Timing information is described by event models, such as periodic, periodic with jitter and bursts, or more general models in terms of arrival curves. Prominent methods of compositional performance analysis are modular performance analysis (MPA) [29] and symbolic timing analysis for systems (SymTA/S) [30]. Both methods support a rich set of scheduling policies, such as preemptive and non-preemptive fixed priority scheduling, earliest deadline first scheduling, or time division multiple access. MPA is used in the DOL design flow, for instance.
- *Automata Based Analysis:* Performance analysis of MPSoCs has also been tackled using state-based formalisms. One example are timed automata [31]: The approach is to model a system as a network of interacting timed automata and formally verify its behavior by means of reachability analysis using the Uppaal model checker [32].

A comparison of these performance analysis methods is provided in [33]. Note that beside being suited for the analysis of real-time systems, analytic models are often used as the basis for performing system optimization, such as scheduling parameter optimization [34] or robustness optimization [35].

Finally, one can observe that none of the methods shown in Fig. 2 can fulfill all the requirements concerning accuracy, scope, and set-up effort. Therefore, combinations of the different methods have been proposed: Simulation has been coupled with native execution on the target platform to reduce simulation time [36, 37]. Different analytic methods have been coupled to broaden the analysis scope [38, 39]. Subsystems in simulation have been replaced by analytic models to reduce simulation time and eliminate the need to generate a detailed simulation model of a component [40]. In these efforts, the modularity of KPNs is often leveraged by using the FIFO channels as the interface between the different performance analysis methods.

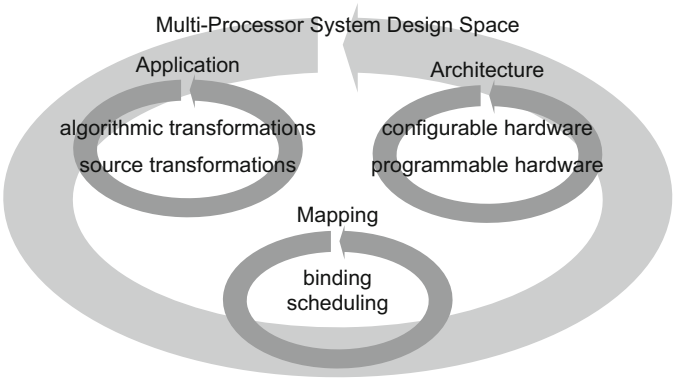


Fig. 3 Application, architecture, and mapping design space

3.4 Design Space Exploration

Designers of MPSoCs face a large design space due to the combinatorial explosion related to the available degrees of freedom. At several points in the design flow and at various levels of abstraction, they need to decide between design alternatives. Specifically, the design space of MPSoCs can be roughly divided into three domains: the application design space, the architecture design space, and the mapping design space. These three domains can be further split up, as shown in Fig. 3.

Exploration of the *application design space* can be split up into two main kinds of transformations, namely algorithmic and source transformations. Algorithmic transformations make explicit the coarse-grained parallelism in a sequential application by transforming it into a KPN. Given a KPN application, source transformations split and merge processes to trade-off parallelism and communication overhead.

Exploration of the *architecture design space* attempts to find an optimized architecture for a given application. The goal is to instantiate programmable and (re-)configurable hardware components that allow an efficient implementation of an application.

Exploration of the *mapping design space* is the last step in the design space exploration. Given a KPN application and an architecture, processes and channels of the KPN are bound to processors and interconnects in the architecture, and scheduling policies are defined on shared communication or computation resources.

Usually, design space exploration is a multi-objective optimization problem. The goal is thus to find a set of Pareto-optimal designs which represent solutions with different trade-offs between the optimization goals such as performance, cost, energy consumption, or peak temperature. The final choice is left to the designer who needs to decide which of the Pareto-optimal designs to implement or to refine to the next level of abstraction.

Available approaches to the exploration of design spaces can be characterized as follows. Gries [41] presents a more detailed survey of automated design space exploration and performance analysis in different design flows.

- *Manual Exploration:* The selection of design points is done by the designer. When taking this approach, the advantage of using a KPN design flow lies in efficient performance analysis and automated synthesis of selected designs.
- *Exhaustive Search:* All design points in a specified region of the design parameters are evaluated. Very often, this approach is combined with local optimization in one or several design parameters in order to reduce the size of the design space. Due to the availability of fast performance analysis techniques for KPNs, exhaustive search is a realistic option if the design space is limited (or can be pruned) to roughly a few thousand designs.
- *Reduction to Single Objective:* For design space exploration with multiple conflicting criteria, there are several approaches available that reduce the problem to a single criterion optimization. For example, manual or exhaustive sampling is done in one (or several) directions of the search space and a constraint optimization, e.g. iterative improvement or analytic methods is done in the other. One may also combine the various objectives to a single criterion by means of a weighted sum where the weights express the preferences of the designer.
- *Black-box Randomized Search:* The design space is sampled and searched via a black-box optimization approach, i.e. new design points are generated based on the information gathered so far and by defining an appropriate neighborhood function (variation operator). The properties of these new design points are estimated which increases the available information about the design space. Examples of sampling and search strategies are Pareto simulated annealing, Pareto tabu search, or evolutionary multi-objective optimization. These black box optimization methods are often combined with local search methods that optimize certain design parameters or structures. This approach is most frequently used in KPN design flows, as illustrated in Table 2.
- *Problem-Dependent Approaches:* In addition to the above methods, one can find also a close integration of the exploration with a problem-dependent performance analysis of implementations. This approach is often used in design flows that are based on subclasses of KPNs. The StreamIt and MAMPS design flow, for instance, are based on SDF (Synchronous Data Flow) graphs and use adopted techniques for design space exploration, see Ref. [42].

4 Specification, Synthesis, Analysis, and Optimization in DOL

Until now, this chapter introduced KPN design flows and the corresponding main design activities. This section will provide additional technical details by means of a concrete example of a typical design flow: the Distributed Operation Layer

(DOL) [12, 43]. The underlying concepts for system specification, synthesis, performance analysis, and design space exploration will be considered, as well as a few typical experimental results for the size of the implementation, the runtime, and accuracy of the applied methods. DOL is currently being extended towards scenario-based design flow [44], and supports the design, optimization, and simultaneous execution of multiple dynamic applications on a MPSoC starting with a similar programming model as [45]. However, this section does not discuss these extensions, focusing on the typical design flow for single Kahn process networks.

4.1 Distributed Operation Layer

The distributed operation layer (DOL) [12, 43] is a platform independent MPSoC design flow based on the Kahn process network (KPN) model of computation [3] and targeted at real-time multimedia and (array) signal processing applications.

The DOL design cycle, as shown in Fig. 4, follows the Y-chart approach in which the application specification is platform-independent and needs to be related to a concrete architecture by means of an explicit mapping. As usual, the design starts with the specification of the application and architecture (and sometimes even a mapping). Then, code for the functional simulation of the application is automatically generated for testing and debugging the parallel application code with standard debugging tools on a standard PC/workstation.

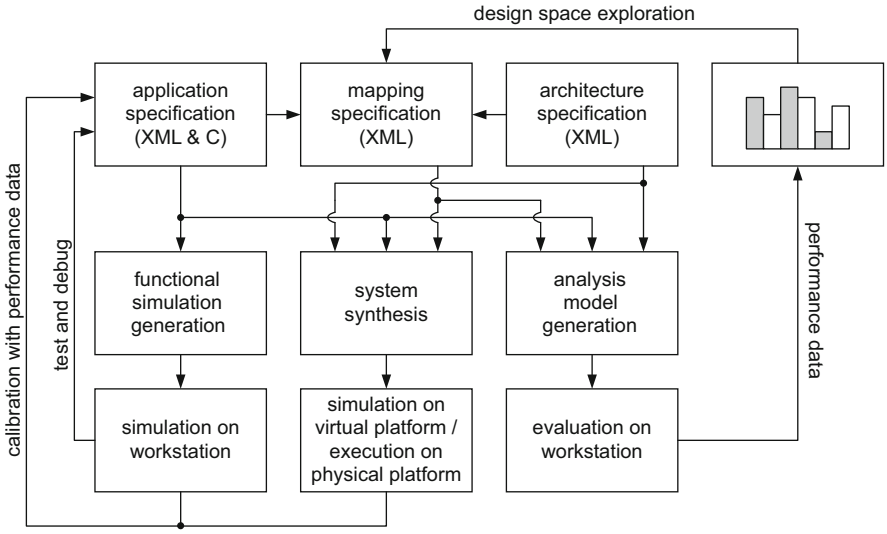


Fig. 4 Overview of the DOL design flow

Once the application is functionally correct, it can be mapped onto the target architecture. Based on the architecture and the mapping specification, the system is synthesized by generating the corresponding binaries. Note that, here, system synthesis refers to software synthesis only as the architecture specification is considered to be unaltered during the exploration phase. Then, the synthesis involves the generation of the mapping-dependent source code for processors, the compilation, and the linking to platform specific libraries as well as to the run-time environment. Generated binaries can either be executed on a simulator of the target platform or on the real MPSoC. Both, the functional and the low-level simulation provide performance figures that will enrich the application specification. This information will be used in later phases for the calibration of the analysis model.

The design flow described so far is typical for MPSoC design and very similar to the other design flows listed in Table 1 and explained in the previous chapters. What is different in DOL is its focus on the design and analysis of real-time signal processing applications. To this end, an analytic worst-/best-case performance analysis method has been embedded into the design flow. Besides enabling the analysis of real-time systems, using an analytic method for performance analysis facilitates rapid design space exploration due to short analysis times. The resulting performance data are embedded in a design space exploration loop in search of the optimal mapping.

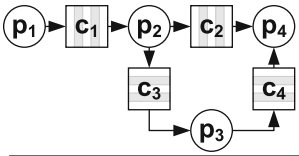
4.2 System Specification

For designing the specification format of an MPSoC, one has to consider three criteria. First, the specification format should be expressive enough to represent the class of envisioned applications, i.e. (real-time) signal processing applications. Second, the specification should facilitate automation of system synthesis and analysis. The third criterion is the possibility of mapping an application in different ways onto an architecture. In the DOL framework, these criteria are met by specifying the application as a Kahn Process Network [3] and by specifying the application independently of architecture.

When designing parallel applications irrespective of architectures, an important feature is the ability to specify different topologies of the process network with different degrees of parallelism. For this reason, the KPN coordination part is kept separately (described in XML) from the source code of the individual processes (described in C/C++), see Fig. 5. Similar hybrid XML/C formats are employed by other frameworks as well (e.g. in Artemis [11], ESPAM [13], and MAMPS [15]).

While the syntax of the XML file is specified using an XML schema, the C code is based on a simple API. As shown in Fig. 5, this API basically consists of four functions, two of which concern computation, namely `INIT` and `FIRE`, and two of which concern communication inside the `FIRE` procedure, namely `READ` and `WRITE`:

```
01: <process name="p1">
02:   <port type="output" name="out"/>
03:   <source type="c" location="p1.c"/>
04: </process>
05:
06: <process name="p2">
07:   <port type="input" name="in"/>
08:   <source type="c" location="p2.c"/>
09: </process>
10:
11: <connection name="c1_fifo">
12:   <origin name="p1"><port name="out"/></origin>
13:   <target name="p2"><port name="in"/></target>
14: </connection>
```




```
01: procedure INIT(Process p)
02:   //initialize process state
03: end procedure
04:
05: procedure FIRE(Process p)
06:   READ(input, size, buf)
07:   //processing
08:   WRITE(output, size, buf)
09: end procedure
```

Fig. 5 Kahn process network model. Left: XML description of the process network structure. Right top: example of a process network. Right bottom: C code of individual processes

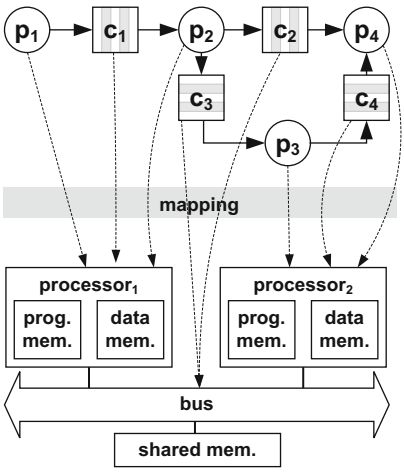
- `INIT` contains the code that is executed once at start-up to initialize a process.
- `FIRE` contains the code that is repeatedly called by the scheduler.
- `READ` implements the blocking read from a FIFO channel.
- `WRITE` implements the blocking write to a FIFO channel.

A similar API is defined by Y-API [46], for instance, a library for specifying and executing Kahn process networks.

The architecture model in DOL is an abstract representation of the underlying execution platform. Its purpose is to determine at a system-level the consequences of the application mapping. This abstract architecture models the topology (i.e. the set of processors and communication paths between processors) and includes performance figures of the underlying platform useful for performance analysis, e.g. the clock frequency and throughput of architectural resources. The architecture model is a structural description that does not express the functional behavior, and which is specified in XML, similar to the application model. This XML architecture representation is not specific to DOL, but also encountered in other frameworks, such as Artemis and MAMPS.

The application model is brought in correspondence to the architecture model by a mapping (see Fig. 6) which can be either established manually by an experienced designer or generated automatically by design space exploration. This mapping fixes the allocation of hardware resources, the binding of the application elements onto these resources, and the scheduling on shared resources. For the mapping specification, once more, the XML format is used. The mapping XML serves as intermediate format and interface between tools, i.e. the design space exploration tool generates a mapping XML as an output, which is the input for the software synthesis tool.

The application XML, the architecture XML, and the mapping XML are the basis for the following DOL synthesis steps, i.e. for the functional simulation and the implementation of the final MPSoC, but also for the generation of the analytic performance analysis model (see Fig. 4).




```
01: <binding name="b_p1" type="computation">
02:   <process name="p1"/>
03:   <processor name="processor1"/>
04: </binding>
 ...
16: <binding name="b_c2" type="communication">
17:   <sw_channel name="c2"/>
18:   <writepath name="dmem1_bus_shm"/>
19:   <readpath name="shm_bus_dmem2"/>
20: </binding>
...
30: <schedule name="sched1" type="fixedpriority">
31:   <resource name="processor1"/>
32:   <origin name="p1">
33:    < configuration name="priority" value="0"/>
34:   </origin>
35:   <origin name="p2">
36:     <configuration name="priority" value="1"/>
37:   </origin>
38: </schedule>
```

Fig. 6 Mapping of a Kahn process network onto a two-processors architecture and an example of a corresponding mapping XML file

4.3 System Synthesis

Similar to other frameworks, an application specified in DOL cannot be directly executed by just compiling the provided source code of the processes. A synthesis step is required that generates the "glue code" implementing the processes and channels, the bootstrapping and the scheduling of the application. Specifically, synthesis is done first for a standard PC/workstation to support the functional verification and debugging of the application (in which case it should be rather termed functional simulation generation) and second for the target MPSoC (which is properly known as system synthesis). However, due to similarities, the two steps are treated together in the following subsections as facets of system synthesis in the DOL design flow. Note that when the behavior of a part of an application can be restricted to a subclass of KPN, the general approach described below could be combined with one of the corresponding synthesis techniques described in the previous chapters.

4.3.1 Functional Simulation Generation

The purpose of providing a functional simulation that can be executed on a standard PC/workstation is to provide the application developer with a convenient approach to test and debug the application. Specifically, functional bugs within the application can be exposed and debugged by running a functional simulation on a standard PC and using standard debugging tools, e.g. the GNU debugger gdb.

A second role of the functional simulation is to obtain architecture-independent application parameters for performance analysis, such as the amount of data transferred between processes or the number of activations of processes. These parameters can easily be obtained from functional simulation by monitoring the calls of the `READ`, `WRITE`, and `FIRE` methods. By back-annotating these parameters to the application specification as shown in Fig. 4, they can be referred to during performance analysis, as explained later in this section.

Using the DOL design flow, a functional simulation can be automatically generated according to the application specification: For each process, an execution thread is instantiated. To implement software channels, inter-thread communication channels are used. The execution of the application is then controlled by a simple data-driven scheduler. Since Kahn process networks are determinate, this is a viable possibility because the scheduling does not influence the input/output behavior of the application.

In DOL, the functional simulation is based on the SystemC library. Therefore, processes can be implemented as user-space threads which incurs less runtime overhead compared to using an operating system thread library, such as the pthread library. Figure 7 shows the software architecture of the functional simulation based on SystemC: Each Kahn process is embedded into a SystemC thread, whereas each Kahn software channel is implemented as a SystemC channel. Moreover, the main file that bootstraps the process network and implements the scheduler to coordinate the quasi-parallel execution of processes is generated automatically as well.

Another frequently chosen library is the pthreads library. On multicore multiprocessors where single operating system threads can be executed on different cores, a functional simulation based on pthreads can even achieve a speed-up compared to the sequential version of the application. In [47], speeds-ups of more than 3 have been reported for executing applications specified in SHIM on a quad-core Intel Xeon processor.

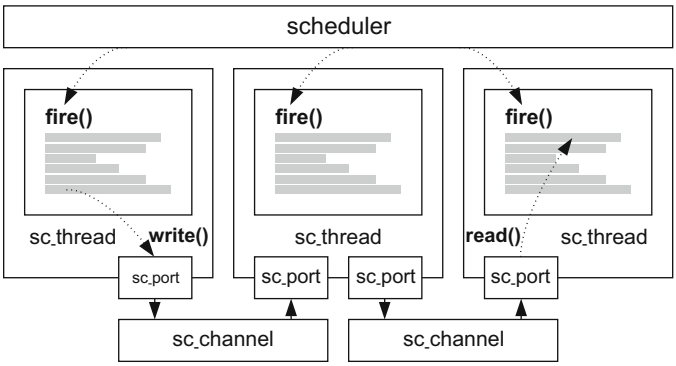


Fig. 7 Software architecture of the functional simulation of a KPN application based on SystemC

4.3.2 Software Synthesis

After the application has been functionally verified by functional simulation, it is ported to the target platform. This requires an architecture dependent runtime environment in which the application is executed. The role of the runtime environment is to hide architectural details of the MPSoC platform by providing a set of high-level services enabling the execution of an application on the platform, such as task scheduling, inter-process communication, or inter-processor communication. Depending on the target platform, developing (parts of) the runtime environment might be necessary to create the basis for software synthesis.

In case of the DOL design flow, different hardware MPSoC platforms are supported:

- *Cell Broadband Engine [48]:* MPSoC consisting of a PowerPC-based Power Processor Element and eight DSP-like Synergistic Processing Elements interconnected via a ring bus.
- A*tmel Diopsis 940 [49]:* tile-based MPSoC, where a single tile is composed of an ARM9 processor and a DSP interconnected by an AMBA bus; up to eight tiles are interconnected via a network-on-chip.
- *MPARM [50]:* Homogeneous MPSoC consisting of identical ARM7 processors connected by an AMBA bus.
- *Intel SCC [51]:* Many-core homogeneous architecture with 48 cores organized in 24 tiles, each tile embedding two cores. Tiles are connected via a mesh on-chip network, and each tile has also a message passing buffer.

Figure 8 depicts a block diagram of the MPARM architecture. Software synthesis for MPARM is based on the RTEMS (Real-Time Executive for Multi-processor Systems) [52] operating system. Basic services provided by RTEMS are the scheduling of processes, device drivers for inter-process communication, and device drivers for system input/output. Based on these services, it is rather simple to bootstrap and execute a process network. As an example, Listing 1 illustrates parts of the code for bootstrapping a process network based on the RTEMS API. Software synthesis for the Cell Broadband Engine is described in [53] in the context of the DOL design flow, and in [23] in the context of the SHIM design flow, for instance.

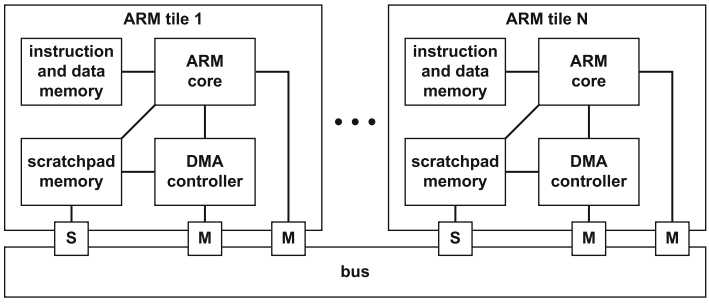


Fig. 8 Block diagram of the MPARM architecture

Listing 1 shows parts of a main file for a producer-consumer type application running on MPARM. In lines 1–2, memory is allocated for the local data of the producer and consumer processes. In lines 5–8, two tasks are created for the processes by allocating a task control block, by assigning a task name and a task ID, by allocating a stack, and by setting initial attributes like the task priority and the task mode. Lines 10–11 show the creation of a message queue. In lines 13–17, the `rtems_task_start` directive puts the tasks into the ready state, enabling the scheduler to execute them. Finally, the initialization tasks deletes itself (line 19).

Listing 1 RTEMS initialization task in which two tasks are bootstrapped to run a producer and consumer process of a process network

```
1   producer_wrapper ← malloc(sizeof(RtemsProcessWrapper));
2   consumer_wrapper ← malloc(sizeof(RtemsProcessWrapper));
3
4   for (j ← 0; j < 2; j++) {
5     status ← rtems_task_create(j + 1, 128,
6                RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
7                RTEMS_DEFAULT_ATTRIBUTES, &(task_id[j]));
8   }
9
10  status ← rtems_message_queue_create(1, 10, 1,
11             RTEMS_DEFAULT_ATTRIBUTES, &queue_id[0]);
12
13  status ← rtems_task_start(task_id[1], producer_task,
14             (rtems_task_argument)producer_wrapper);
15
16  status ← rtems_task_start(task_id[2], consumer_task,
17             (rtems_task_argument)consumer_wrapper);
18
19  rtems_task_delete(RTEMS_SELF);
20 }
```

For all the mentioned platforms, the main challenge is to provide an efficient FIFO channel implementation that allows overlapping computation and communication in order to reduce the runtime overhead as much as possible. Aspects that play an important role in this context are the size and location of channel buffers, the efficient use of DMA controllers for data transfers between processors, and the minimization of synchronization messages.

4.4 Performance Analysis

The DOL design flow is targeted towards the design of real-time multi-media and signal processing applications. These systems must meet real-time constraints, which means that not only the correctness and performance of a system are of major

concern but also the timeliness of the computed results. Typical questions in this context are:

- What is the response time to certain events? Is this response time within the required real-time limits?
- Can the system accept additional load and still meet the quality-of-service and real-time constraints?
- Is a system schedulable, that is, are all real-time constraints met?

To be able to answer these questions, a suitable combination of system design and performance analysis is required. To this end, it is essential that the architecture, application, and runtime-environment of a system are amenable to formal analysis, because simulation or measurements are not able to provide guarantees about timing properties. On the other hand, performance analysis methods with a reasonable scope and accuracy need to be employed such that effects occurring in the system implementation can be faithfully modeled. For MPSoC applications, this includes the modeling of heterogeneous resources and their sharing, the modeling of complex timing behavior arising from variable execution demands and interference on shared resources, or the modeling of different processing semantics.

Many approaches have been proposed to solve this problem, see [54] for an overview. Frequently used approaches are time-triggered and synchronous approaches, for instance. In purely time-triggered approaches, such as the time-triggered architecture [55] or Giotto [56], processing time of resources is allocated to tasks in fixed time slots. This fixed allocation facilitates analysis, but dimensioning of the slots turns out to be difficult for varying workloads. For instance, using the worst-case workload for setting the slot sizes might lead to over-dimensioned systems. Purely synchronous approaches implemented in synchronous languages, such as Esterel, Lustre, and Signal, rely on a global clock that divides the execution of a system into a sequence of atomic processing steps [57]. While synchronous approaches are successfully used for single-processor systems, applying them to MPSoCs is difficult because MPSoCs are usually split up into different (asynchronous) clock domains such that the synchronous assumption does not hold.

The approach taken in DOL relies on using compositional performance analysis where a system is modeled as a set of processing components that interact via event streams. This is a good match for MPSoCs as well as for KPNs. Contrary to other approaches, the approach is rather flexible in that it is not limited to a certain system architecture, scheduling policy, or execution semantics. Specifically, Modular Performance Analysis (MPA) is integrated into the DOL design flow. In the following, the basic concepts of MPA are reviewed. Afterwards, it is summarized how MPA is integrated into the DOL design flow.

4.4.1 Modular Performance Analysis (MPA)

MPA [29, 58] is an analytical approach for the analysis of real-time systems. It is based on real-time calculus [59] which has its foundations in network calculus, a

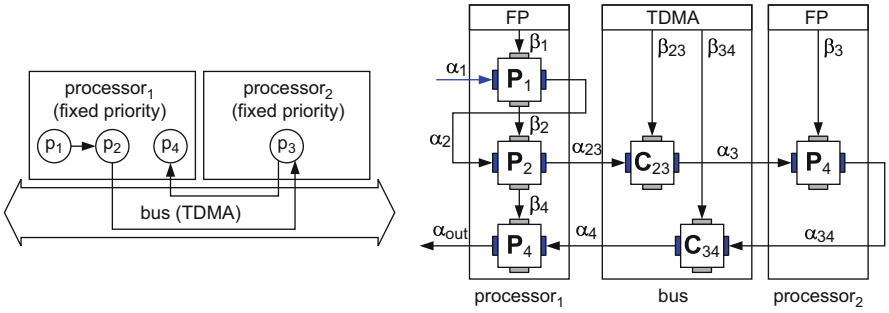


Fig. 9 MPA model of a system with two processors connected by a bus on which a KPN with four processes is executing. In the MPA model, horizontal edges represent event streams whereas vertical edges represent resource streams

method for worst-case analysis of communication networks [60, 61]. With MPA, hard upper and lower bound for performance metrics of a distributed real-time system can be computed. As shown in Fig. 9, the performance model of a system is decomposed into a network of abstract processing components that model the computation and communication in a system. These processing components are connected by abstract event streams that model the timing properties of the data streams flowing through the system. Finally, resources are modeled by resource streams that model the availability of processing resources to computation and communication tasks. Different scheduling policies can be modeled by differently connecting processing components and resource streams. As an example, Fig. 9 illustrates fixed priority (FP) scheduling on processors and time-division multiple-access (TDMA) scheduling on the bus.

The processing components modeling computation and communication are characterized by the worst-case/best-case execution demand and the minimal and maximal token size, respectively. Event streams are characterized by so-called arrival curves and resource streams by service curves. Summarizing, these abstractions allow the modeling of computation and communication on heterogeneous resources in a unified manner.

Based on these abstractions, the system is analyzed by consecutive propagation of event streams between components. Depending on the mapping of processing components to a resource and its scheduling/arbitration, the timing properties of the streams change. System properties such as resource utilization, system throughput, end-to-end delays, or buffer sizes can be derived this way. Tool support for actually performing the analysis is provided by a freely available Matlab toolbox [62] that implements the underlying algebraic operations.

4.4.2 Integration of MPA into the DOL Design Flow

It has been mentioned that the goal of system synthesis is to bridge the implementation gap, that is, to refine a high-level system specification into an actual system implementation. Similarly, there is an "abstraction gap" between an MPA model and the implementation: The execution of sequential processes on a processor is modeled by an abstract processing component, the availability of resources is abstracted by service curves, and the dataflow through the systems by arrival curves. Bridging this abstraction gap, that is, creating an analysis model that correctly models the implementation is a non-trivial task. On the one hand, the high-level system specification is conceptually similar to the analysis model but does not contain all the parameters required to generate an MPA model, such as best-case/worst-case task execution times or token sizes. The implementation, on the other hand, implicitly contains this information, but extracting the information is not straightforward.

In the DOL design flow, the abstraction gap is bridged by analysis model generation and calibration. In model generation, the high-level system specification is translated into a corresponding MPA model. In model calibration, the required model parameters are obtained. In both steps, the modular structure of the application and architecture specification is leveraged. The basic approach is depicted in Fig. 10. The analysis model generation represents a branch in the design flow that is parallel to the system synthesis. The analysis model calibration makes use of this feature later on in order to build a database with necessary performance data for the formal model. In the following, the basic approach is described. Further details are provided in [63].

The goal of model generation is to translate the high-level application, architecture, and mapping specification into a corresponding MPA model implemented as a Matlab script. Due to the modular specification of the application that is made explicit in the process network XML description, this is straightforward: Each

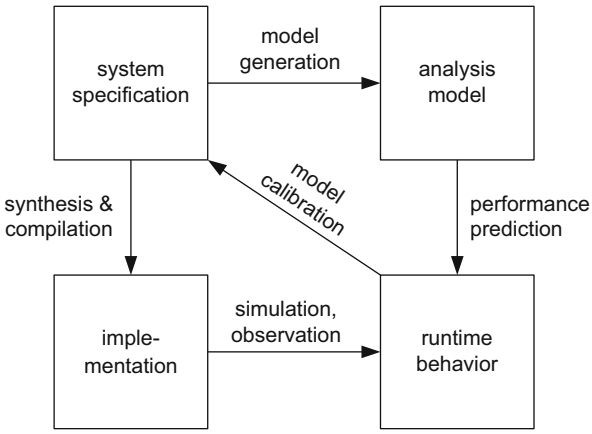


Fig. 10 Analysis model generation and calibration in the DOL flow

process is simply modeled as an abstract processing component. Similarly, the communication channels between processes are modeled as abstract communication components and connected to the processing components according to the topology of the KPN.

The aim of model calibration is to obtain the quantities to parameterize the generated model such that it correctly models the implementation. Basically, three different types of parameters can be distinguished:

- First, there are the application parameters that are architecture and timing independent. An example is the minimal and maximal size of tokens transmitted over each channel.
- Second, there are the parameters that depend only on the architecture and the runtime environment. Examples are the throughput of the different communication resources or the context switch time of the runtime environment.
- Third, there are the application parameters that depend on the architecture and the mapping. Basically, whenever the architecture or mapping changes, new parameters need to be determined. An example is the worst-case/best-case execution time of a process on a processor.

Depending on the parameter type, there are different ways to obtain them. Timing independent parameters can be obtained from the functional simulation, due to the determinism of KPN applications. Architecture dependent parameters need to be obtained once a new hardware architecture or runtime environment is employed for realizing a system. The parameters of the third category, i.e. application parameters that depend on the architecture and the mapping, are more difficult to determine. Similar to system-level performance analysis, different methods for worst-case/best-case execution time analysis have been proposed, for instance [64]. In the DOL design flow, timed simulation on a virtual platform is employed. Note that compared to formal methods, this approach is not suitable for the calibration of hard real-time system models unless complete coverage of corner cases is exhibited in the calibration simulation runs. One can observe that similar approaches are taken in other design flows. In the Artemis design flow, for instance, model generation and calibration is used to create a model for trace-based simulation [65].

Finally, the DOL framework have been extended with capabilities for worst-case thermal analysis, as nowadays providing guarantees on maximum temperature is as important as functional correctness and timeliness. Aware of the performance-temperature correlation, DOL is optimizing the system design with respect to both worst-case performance and worst-case temperature, analyzed in the same MPA framework. The basic worst-case peak temperature analysis method in MPA for a single processor under a broad range of uncertainties in terms of task execution times, task invocation periods, and jitter in task arrivals is described in [66]. Extensions are then proposed in [67] for analysis of MPSoC platforms by considering both the self-heating of the processor and the heat transfer between neighboring processors. In the same manner as it is done for timing, thermal analysis models are automatically generated from the same set of specifications as used for software synthesis. To increase the model accuracy, both analysis models

are calibrated with data corresponding to real system parameters obtained in an automatic manner, prior to design space exploration. The calibration tool-chain for the thermal model is described in [68].

4.5 Design Space Exploration

The final piece of the DOL flow is design space exploration, built on top of analysis and synthesis tools to find an optimal mapping. In general, the problem of optimally mapping an application to a heterogeneous distributed architecture is known to be NP-complete. Even for systems of modest complexity, one thus needs to resort to heuristics to solve the problem. In addition, the mapping problem is usually multi-objective such that there is no single optimal solution but a set of Pareto-optimal solutions constituting a so-called Pareto-front.

In DOL, the aim of the design space exploration is to compute the set of Pareto-optimal solutions representing different trade-offs in the design space. Based on the (approximated) Pareto-front, the designer chooses the final solution to implement. Therefore, the mapping problem is specified as a multi-objective optimization problem.

Formally, a multi-objective optimization problem is defined on the decision space X which contains all possible design decisions, i.e. architectures, applications and mappings. To each implementation $x \in X$ there is associated an objective vector f in the objective space Z that consists of n objectives $f = (f_1, \ldots, f_n)$ which should be minimized (or maximized). An order relation $\leq$ is defined on the objective space Z, which induces a preference relation $\preceq$ on the decision space X: $x_1 \preceq x_2 \Leftrightarrow f(x_1) \leq f(x_2)$, for $x_1, x_2 \in X$. In other words, for the mapping problem for instance, if mapping x_1 is better (minimal) in all objectives than mapping x_2, the optimization algorithm will "preferentially" select mapping x_1.

The design search space, symbolized with Ψ, is the set of all subsets of X, i.e. it includes all possible solution sets $A \subseteq X$. The final goal is to determine an optimal element of Ψ, i.e. an optimal subset of all possible implementations X. This subset should reflect all trade-offs induced by the multiple objectives. The preference relation $\preceq$ on X that has been defined above can now be used to define a corresponding set preference relation, symbolized with $\preccurlyeq$, on the search space Ψ.

This set preference relation provides the information on the basis of which two candidate Pareto sets can be compared: $A \preccurlyeq B \Leftrightarrow \forall b \in B, \exists a \in A : a \preceq b$. This property reflects the concept of Pareto-dominance: A design point dominates another one if it is equal or better in all criteria and strictly better in at least one. Moreover, the search in the design space will be pursued until a good Pareto-optimal set approximation $A \in \Psi$ is found.

In DOL, evolutionary algorithms are used to solve the mapping optimization problem. Evolutionary algorithms find solutions to a given problem by iterating two main steps [69]: (1) selection of promising candidates, based on an a-priori evaluation of candidate solutions and (2) generation of new candidates by variation

of previously selected candidates. The principle of the selection in evolutionary multi-objective optimization is sketched in Algorithm 1. For a complete description, we refer to [70]. The starting point is a randomly generated population $P \in \Psi$ of size m. During optimization, a heuristic mutation operator based on selection and variation generates another set $P' \in \Psi$, which is wanted to be better than P in the context of the predefined set preference relation $\preccurlyeq$, i.e. $P' \preccurlyeq P$. Finally, P is replaced by P', if the later is preferable to the former (i.e., $P' \preccurlyeq P$), or P it remains unchanged in the opposite case.

Algorithm 1 Main optimization function

```
1:  randomly choose A ∈ Ψ                          ▷ generate initial set P of size m
2:  set P ← A
3:
4:  while termination criterion not fulfilled do   ▷ main optimization loop
5:      P' ← heuristicSetMutation(P)
6:      if P' ≼ P then
7:          P ← P'
8:      end if
9:  end while
10: return P
```

Algorithm 2 Heuristic set mutation function

```
1: function HEURISTICSETMUTATION(()P)
2:     generate {r1, ..., rk} ∈ X based on P
3:     P' ← P ∪ {r1, ..., rk}
4:     while |P'| > m do
5:         choose p ∈ P' with fitness(p) = min_{a∈P'}{fitness(a)}
6:         P' ← P' \ {p}
7:     end while
8:     return P'
9: end function
```

The heuristic set mutation operator is detailed in Algorithm 2. First, k new solutions are created based on P, after an appropriate selection and variation operation. While the variation is problem-specific, the selection is independent of the problem, using either an uniform random selection or a fitness-based selection. Then, the k new solutions are added to P, resulting in a set P' of size $m + k$. P' is iteratively truncated to size m by removing the solutions with worst fitness values. Note that the fitness values are associated in a performance evaluation process, i.e., in DOL we use the MPA framework as described in Sect. 4.4.

While the selection algorithm described is domain independent, specific methods are used to include domain specific knowledge into the search process and select "best" solutions among a population. These are the domain representation (i.e., the system mapping), the evaluation of designs (i.e., the MPA analysis), and the variation of a population of solutions by mutation and cross-over operations.

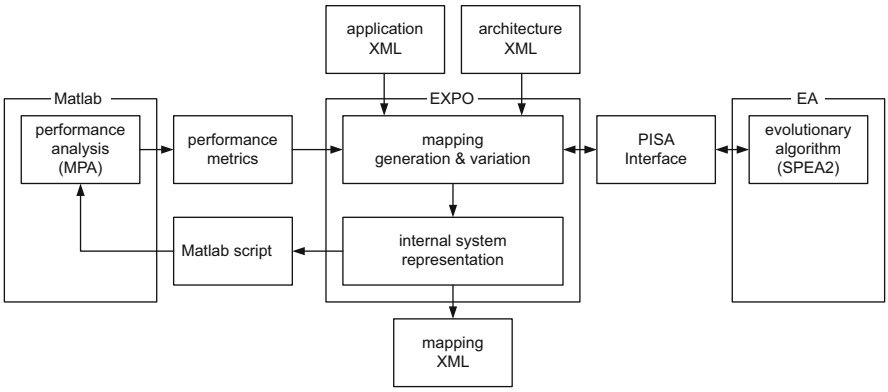


Fig. 11 Design space exploration in the DOL framework

Although standard variation schemes exist for mutation and crossover, their implementation is strongly dependent on system properties. The mutation generates a local neighborhood of selected design points. In the DOL context, the mutation affects the mapping solutions; for instance, different mappings can be generated with different bindings of processes onto processors. The crossover recombines two selected solutions to generate a new one. Note that during mutation or crossover, infeasible (mapping) solutions can be generated. In this situation, a repair strategy is invoked, which, in conformity with the evolutionary algorithm principle, attempts to maintain a high diversity in the population. An example could be the rerouting of inter-process communication, when during re-mapping a process was bound to a new location.

In DOL, the design space exploration framework includes several tools, as shown in Fig. 11. In particular, the EXPO [71] tool is the central module of the framework. As underlying multi-objective search algorithm, Strength Pareto Evolutionary Algorithm (SPEA2) [72] is used that communicates with EXPO via the PISA [73] interface. Similarly, the design space exploration framework in Artemis [11] is also based on PISA and SPEA2.

Using the frameworks of EXPO and PISA relieves the designer from implementing those parts of the design space exploration that are independent of the actual optimization problem. For example, the selection may be handled inside the multi-objective optimizer SPEA2. The designer just needs to focus on the problem specific parts, that is, the generation, variation, and evaluation of solutions. The implementation of problem specific parts starts with the specification, where the application and architecture (and later on, the mapping) are automatically extracted from the corresponding XML files and represented in the design space exploration framework. Then, candidate mappings are inspected (as described above) based on the provided variation methods. Finally, during design space exploration, the objective values of all candidate mappings are computed by generating the corresponding Matlab MPA scripts and interfacing Matlab for their evaluation. In DOL, all these

operations are automatically parameterized using the application and architecture specification. Note that the approach described above is a heuristic search procedure. Therefore, it does not guarantee the optimality of the final solution, i.e., the final set of solutions. However, in our experiments we have identified that after several design space exploration cycles, the found solutions are close to optimal even for large problem complexities.

4.6 Results of the DOL Framework

In this section, a few results are highlighted that have been obtained by applying the DOL design flow described above. Specifically, the design and analysis of a Motion-JPEG (MJPEG) decoder [74] running on MPARM [50] is considered. For the execution of the system, we used a 31-frame input bitstream encoded using the QVGA (320×240) YUV 444 format.

The MJPEG decoder decompresses a sequence of frames by applying JPEG decompression to each frame. Because of the inherent parallelism in the MJPEG algorithm, the decoding is done in a pipeline with five stages, each stage being implemented as a Kahn process. The first and last stages are the splitting of streams into frames (`ss`) and the merging of frames back to streams (`ms`). The variable length decoding and the splitting of frames into macroblocks form the second stage (`sf`). The zigzag scan, inverse quantization and the inverse discrete cosine transform form the third (`zii`), while combining macroblocks back to frames forms the fourth stage (`mf`).

Using the design space exploration framework of DOL based on the PISA interface and SPEA2, one can compute the Pareto optimal mappings of the MJPEG application onto an MPARM system with a variable number of processors. The mapping has been optimized in conformity with two design objectives: (1) end-to-end delay in the system computed as the result of MPA analysis, which is an upper bound of the actual end-to-end delay and (2) the cost of the system evaluated as a sum of costs associated with the used processors, memories, and the bus. In the experiments, a population size of 60 individuals has been chosen and the algorithm has been executed for 50 generations. These parameters generally depend on the complexity of the problem to solve. The obtained Pareto front is shown in Fig. 12, consisting of six mapping solutions onto a different number of processors. The search in this design space took about 2 h.

For illustration purposes only, we employ a simple configuration with a small number of processes that can be mapped in different ways onto the architecture and can communicate via different hardware communication paths. However, a more efficient implementation can be obtained if the same application is specified with a scalable number of processes processing data in parallel. This would enlarge the parallelism of the design but also the dimensionality of the design space. A more complex design space exploration with the DOL framework is shown in [12], where a scaled version of an MPEG-2 decoder has been mapped onto a tile-based heterogeneous architecture.

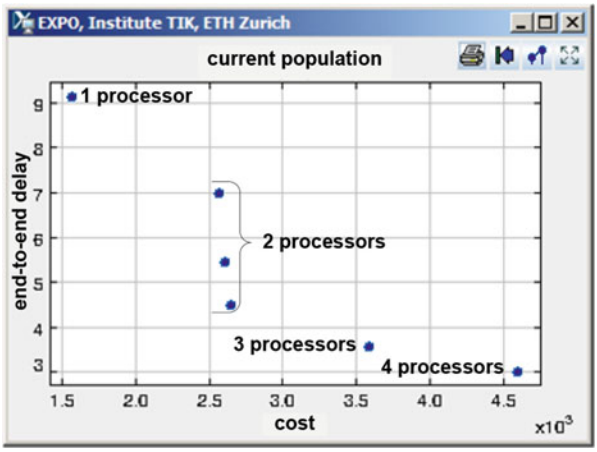


Fig. 12 Pareto optimal solutions resulted after the design space exploration (screenshot of the EXPO tool)

Other KPN flows, like Sesame in the Artemis project [11] that use exactly the same optimization frameworks of PISA and SPEA2, report comparable parameters and results for the design space exploration. However, their design space exploration is considerably shorter, i.e. 5 s for a design with eight processes, because they use a much simpler additive performance model. Of course, for larger problem sizes all the parameters will scale and the design space exploration can take much longer if more accurate analysis methods, like MPA or simulation, are used. However, this is an acceptable cost since designers are exploring the entire design space only once.

In the remainder of this section, a mapping of the MJPEG application onto a 3-tile MPARM system, that is, three ARM processors interconnected via a shared AMBA bus, is considered. The resulting MPA model is shown in Fig. 13.

To evaluate the efficiency of the design flow (i.e., the time spent in obtaining results), Table 3 lists the durations of the different design steps for performance analysis of the different design solutions. Several conclusions can be drawn:

- Automated software synthesis can be done fast. Actually, most of the time required to generate the functional simulation or the binaries for the target platform is required to compile the generated source code rather than to generate this code.
- The table shows that timed simulation on the virtual platform is the most time-consuming step in the design flow. Minimization of simulation time is thus paramount and actually possible in a systematic design flow, as has been shown above. Conversely, simulation time can become a major bottleneck in MPSoC design when following a less systematic design flow requiring many design iterations involving timed simulation.
- The generation and analysis of a system's MPA model is a matter of seconds. Note that similar times have been reported for alternative performance analysis

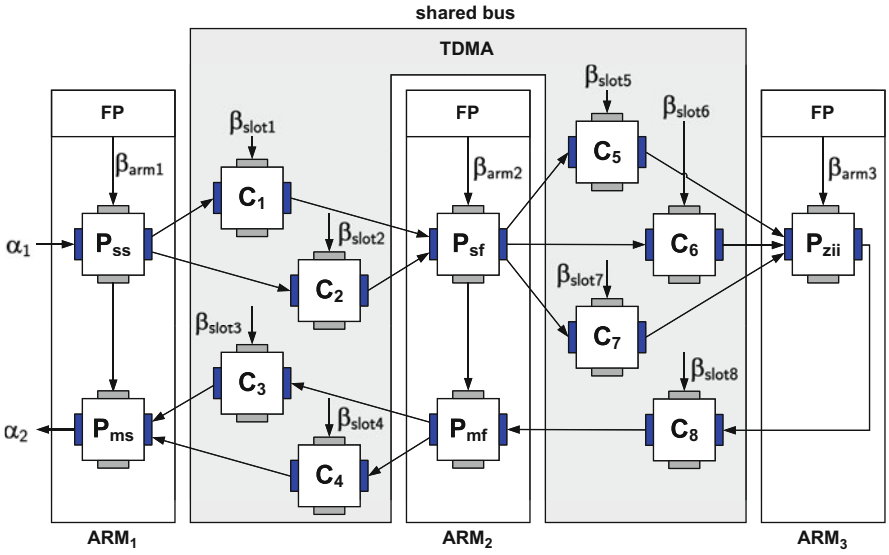


Fig. 13 MPA model of MJPEG application mapped onto a 3-tile MPARM platform. P_x are the five processes of the MJPEG decoding pipeline that communicate via the C_y software channels

Table 3 Duration of different design steps in the MJPEG design, measured on a 1.86 GHz Intel Pentium Mobile machine with 1 GB of RAM

Step		Duration (s)
Model calibration (one-time effort)	Functional simulation generation	42
	Functional simulation	3.6
	Synthesis (generation of binary)	4
	Simulation on MPARM	13,550
	Log-file analysis and back-annotation	12
Model generation		1
Performance analysis based on generated model		2.5

The simulations were executed to decode a 31-frame input bitstream encoded using the QVGA (320×240) YUV 444 format

methods like trace-based simulation in the Artemis design flow, for instance. While further reducing this time is desirable, it is a reasonable time frame for performance analysis within a design space exploration loop.

- The one-time calibration to obtain the parameters for the MPA model takes several seconds albeit being completely automated. Extracting these parameters manually would be a major effort.

In order to evaluate the accuracy of MPA estimations, the performance bounds computed with MPA are compared to actual (average-case behavior) quantities observed during system simulation. The differences are in a range of 10–20%, which is typical for a compositional performance analysis. Differences in the same range have been observed for several systems in [33], for instance. There are two main reasons for these differences. First, several operators in the formal performance analysis do not yield tight bounds. Second, the simulation of a complex system in general cannot determine the actual worst-case and best-case behavior. The simulations on the system level do not use exhaustive test patterns and do not cover all possible corner cases in the interference through joint resources.

Moreover, to illustrate the connection between the worst-case chip temperature and worst-case latency, we represent eight selected mapping configurations of the MJPEG decoder application together with their worst-case chip temperature and worst-case latency calculated in MPA (Fig. 14). Interesting here is the effect of the physical placement that cannot be ignored anymore. So even if the mapping is already defined, the system designer might still optimize the system (i.e., reduce the temperature) by selecting an appropriate physical placement. This is highlighted by solution pairs where only the placement of the processing components has changed but temperature differences of 8 K can still appear [75].

Finally, the DOL framework itself is evaluated in terms of code size of the prototype implementation. The DOL design flow and the associated tools are implemented in Java. To give an indication about the size of the implementation, Table 4 shows the code size of different parts of the design flow (excluding the plug-ins for design space exploration and thermal analysis). One can see that apart from the tool-internal representations of the system specification, the largest part is the MPA code generator for performance analysis. The software synthesizers and the monitoring for the MPA model calibration are comparatively small. Similar observations can be made for other design flows, as well.

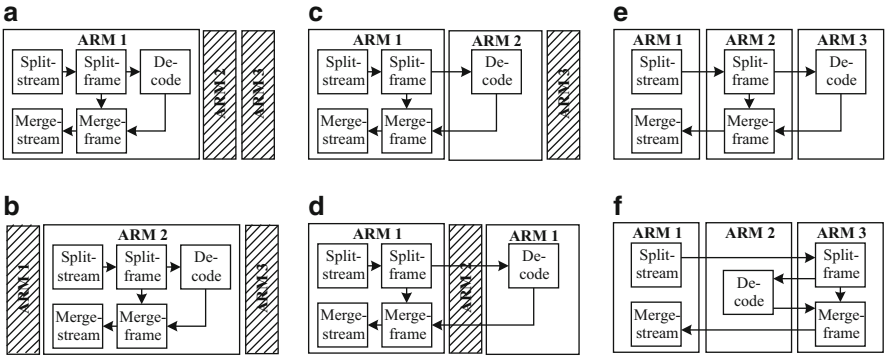


Fig. 14 Worst-case latency versus worst-case peak temperature for similar bindings but different placements, of an MJPEG decoder evaluated on MPARM platform [50]. (**a**) T*=342.2 K, l*=11.7 s. (**b**) T*=350.6 K, l*=11.7 s. (**c**) T*=359.9 K, l*=3.1 s. (**d**) T*=358.2 K, l*=3.1 s. (**e**) T*=364.5 K, l*=2.4 s. (**f**) T*=365.0 K, l*=2.4 s

Table 4 Java code size of different parts of the DOL design flow

Part of design flow	Lines of code
DOL representation of system specifications	6200
Functional simulation generator	4100
MPARM code generator	2100
MPA Matlab code generator	5000
Log-file analysis of functional and timed simulation	1200

5 Concluding Remarks

The mapping of process networks onto multi-processor systems requires a systematic and automated design methodology. This chapter provides an overview over different existing methods and tools, which are all starting from a general Kahn process network (KPN) model of computation and are implementing the established Y-chart approach. Due to fundamental properties of the Kahn model, many problems in the design process can be solved in an automated manner. Thus, the system specification, synthesis, performance analysis, and design space exploration can be implemented in a fully automated way.

After an overview over all these activities, this chapter provides a practical illustration of their implementation in the distributed operation layer (DOL) framework. The design steps followed by DOL are somewhat common to all the KPN flows. What is typical to the DOL framework is the embedding of an accurate formal performance analysis model into the design flow. This presents a clear advantage over the standard simulation-based approaches employed for performance analysis, which typically take more time to execute than a formal model and cannot offer guarantees for (hard) real-time signal-processing applications, due to the incomplete coverage of the design space.

Another key point is the need for a scalable design flow which allows to design large and complex MPSOC systems, which can clearly be noticed from Table 2. As soon as we are faced with more complex MPSoCs, this forthcoming difficulty needs to be considered in all steps of the design trajectory. In particular, it will have an impact at the system-level, where basic design decisions are taken. In this sense, the Kahn model and design methods based on it are promising candidates due to the modular system specification. It offers a great potential for compositional (and fast) performance analysis and design space exploration. By taking a closer look at the DOL framework, it can be observed that it features a specification format that can easily be scaled (i.e. provided by the XML and C basis), it includes a compositional formal performance analysis in the design, and the optimization is done with the support of modular tools such as EXPO and PISA. These features provide the basis for scalable mappings and mapping optimizations.

References

1. B. Kienhuis, E. Deprettere, K. Vissers, P. van der Wolf, in *Proc. Int'l Conf. on Application-Specific Systems, Architectures and Processors (ASAP)* (Washington, DC, USA, 1997), pp. 338–349
2. D. Densmore, A. Sangiovanni-Vincentelli, R. Passerone, IEEE Design & Test of Computers **23**(5), 359 (2006)
3. G. Kahn, in *Proc. IFIP Congress* (Stockholm, Sweden, 1974), pp. 471–475
4. Ptolemy Web Site. http://ptolemy.eecs.berkeley.edu
5. F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, Computer **36**(4), 45 (2003). http://dx.doi.org/10.1109/MC.2003.1193228
6. MathWorks Real-Time Workshop. http://www.mathworks.com/products/rtw/
7. NI LabVIEW Microprocessor SDK. http://www.ni.com/labview/microprocessor_sdk.htm
8. J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, M. Meredith, ACM Trans. on Design Automation of Electronic Systems **14**(1), 1:1 (2009)
9. S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, Y.P. Joo, ACM Trans. on Design Automation of Electronic Systems **12**(3), 1 (2007)
10. J. Falk, J. Keinert, C. Haubelt, J. Teich, C. Zebelein, *Integrated modeling using finite state machines and dataflow graphs*. second edition (Springer, 2012)
11. A.D. Pimentel, C. Erbas, S. Polstra, IEEE Trans. on Computers **55**(2), 99 (2006)
12. L. Thiele, I. Bacivarov, W. Haid, K. Huang, in *Proc. Int'l Conf. on Application of Concurrency to System Design (ACSD)* (Bratislava, Slovak Republic, 2007), pp. 29–40
13. H. Nikolov, T. Stefanov, E. Deprettere, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **27**(3), 542 (2008)
14. T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, ACM Trans. on Embedded Computing Systems **5**(2), 281 (2006)
15. A. Kumar, S. Fernando, Y. Ha, B. Mesman, H. Corporaal, ACM Trans. on Design Automation of Electronic Systems **31**(3), 40:1 (2008)
16. S.S. Bhattacharyya, G. Brebner, J.W. Janneck, J. Eker, C. von Platen, M. Mattavelli, M. Raulet, in *First Swedish Workshop on Multi-Core Computing (MCC)* (Uppsala, Sweden, 2008)
17. S.A. Edwards, O. Tardieu, IEEE Trans. on VLSI Systems **14**(8), 854 (2006)
18. M.I. Gordon, W. Thies, S. Amarasinghe, in *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, CA, USA, 2006), pp. 151–162
19. B. Kienhuis, E. Rijpkema, E. Deprettere, in *Proc. of the Int'l Workshop on Hardware/Software Codesign (CODES)* (San Diego, CA, USA, 2000), pp. 13–17
20. S. Verdoolaege, H. Nikolov, T. Stefanov, EURASIP Journal on Embedded Systems **2007** (2007)
21. A.D. Pimentel, Int. J. Embedded Systems **3**(3), 181 (2008)
22. J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, M. Raulet, in *IEEE Workshop on Signal Processing Systems (SiPS)* (Washington, D.C., USA, 2008), pp. 287–292
23. N. Vasudevan, S.A. Edwards, in *Proc. ACM Symposium on Applied Computing (SAC)* (Honolulu, HI, USA, 2009), pp. 1626–1631
24. D. Gelernter, N. Carriero, Commun. ACM **35**(2), 97 (1992)
25. T.M. Parks, Bounded Scheduling of Process Networks. Ph.D. thesis, University of California, Berkeley (1995)
26. IBM SDK for Multicore Acceleration. http://www-128.ibm.com/developerworks/power/cell/
27. K. Huang, I. Bacivarov, J. Liu, W. Haid, in *IEEE Symposium on Industrial Embedded Systems (SIES)* (IEEE, Ecole Polytechnique Fédérale de Lausanne, Switzerland, 2009), pp. 74–81
28. M. González Harbour, J.J. Gutiérrez García, J.C. Palencia Gutiérrez, J.M. Drake Moyano, in *Proc. Euromicro Conference on Real-Time Systems* (Delft, The Netherlands, 2001), pp. 125–134
29. S. Chakraborty, S. Künzli, L. Thiele, in *Proc. Design, Automation and Test in Europe (DATE)* (Munich, Germany, 2003), pp. 190–195

30. R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, R. Ernst, IEE Proceedings Computers and Digital Techniques **152**(2), 148 (2005)
31. R. Alur, D.L. Dill, Theoretical Computer Science **126**(2), 183 (1994)
32. M. Hendriks, M. Verhoef, in *Workshop on Parallel and Distributed Real-Time Systems* (Rhodes, Greece, 2006)
33. S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, M. González Harbour, in *Proc. Int'l Conf. on Embedded Software (EMSOFT)* (Salzburg, Austria, 2007), pp. 193–202. http://doi.acm.org/10.1145/1289927.1289959
34. E. Wandeler, L. Thiele, in *Proc. Asia and South Pacific Conf. on Design Automation (ASP-DAC)* (Yokohama, Japan, 2006), pp. 479–484
35. A. Hamann, R. Racu, R. Ernst, in *Proc. Real Time and Embedded Technology and Applications Symposium (RTAS)* (Bellevue, WA, United States, 2007), pp. 269–280
36. S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, H. Meyr, in *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (Salzburg, Austria, 2007), pp. 75–80
37. I. Bacivarov, A. Bouchhima, S. Yoo, A.A. Jerraya, International Journal of Embedded Systems (IJES) **1**(1/2), 103 (2005). http://dx.doi.org/10.1504/IJES.2005.008812
38. S. Schliecker, S. Stein, R. Ernst, in *Proc. Design, Automation and Test in Europe (DATE)* (2007), pp. 273–278
39. S. Künzli, A. Hamann, R. Ernst, L. Thiele, in *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)* (Salzburg, Austria, 2007), pp. 63–68
40. S. Künzli, F. Poletti, L. Benini, L. Thiele, in *Proc. Design, Automation and Test in Europe (DATE)* (2006), pp. 236–241
41. M. Gries, Integration, the VLSI Journal **38**(2), 131 (2004)
42. S. Ha, H. Oh, *Decidable dataflow models for signal processing: Synchronous dataflow and its extensions*. second edition (Springer, 2012)
43. K. Huang, W. Haid, I. Bacivarov, M. Keller, L. Thiele, ACM Transactions in Embedded Computing Systems (TECS) (2012)
44. Distributed application layer. http://www.tik.ee.ethz.ch/~euretile
45. M. Geilen, T. Basten, *Kahn process networks and a reactive extension*. second edition (Springer, 2012)
46. E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.Y. Brunel, W.M. Kruijtzer, P. Lieverse, K.A. Vissers, in *Proc. Design Automation Conference (DAC)* (Los Angeles, CA, USA, 2000), pp. 402–405
47. S.A. Edwards, N. Vadudevan, O. Tardieu, in *Proc. Design, Automation and Test in Europe (DATE)* (Munich, Germany, 2008), pp. 1498–1503
48. D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, K. Yazawa, IEEE Journal of Solid-State Circuits **41**(1), 179 (2006)
49. P.S. Paolucci, A.A. Jerraya, R. Leupers, L. Thiele, P. Vicini, in *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (Seoul, South Korea, 2006), pp. 167–172
50. L. Benini, D. Bertozzi, B. Alessandro, F. Menichelli, M. Olivieri, The Journal of VLSI Signal Processing **41**, 169 (2005). https://doi.org/10.1007/s11265-005-6648-1
51. T.G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, S. Dighe, in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE Computer Society, Washington, DC, USA, 2010), SC '10, pp. 1–11. https://doi.org/10.1109/SC.2010.53.
52. RTEMS Home Page. http://www.rtems.com
53. W. Haid, L. Schor, K. Huang, I. Bacivarov, L. Thiele, in *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)* (Grenoble, France, 2009), pp. 35–44
54. S. Edwards, L. Lavagno, E.A. Lee, A. Sangiovanni-Vincentelli, Proceedings of the IEEE **85**(3), 366 (1997)

55. H. Kopetz, G. Bauer, Proceedings of the IEEE **91**(1), 112 (2003)
56. T.A. Henzinger, B. Horowitz, C.M. Kirsch, Proceedings of the IEEE **91**(1), 84 (2003)
57. A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, R. De Simone, Proceedings of the IEEE **91**(1), 64 (2003)
58. E. Wandeler, L. Thiele, M. Verhoef, P. Lieverse, Int'l Journal on Software Tools for Technology Transfer (STTT) **8**(6), 649 (2006)
59. L. Thiele, S. Chakraborty, M. Naedele, in *Proc. Int'l Symposium on Circuits and Systems (ISCAS)*, vol. 4 (Geneva, Switzerland, 2000), vol. 4, pp. 101–104
60. R.L. Cruz, IEEE Trans. Inf. Theory **37**(1), 114 (1991)
61. J.Y. Le Boudec, P. Thiran, *Network Calculus — A Theory of Deterministic Queuing Systems for the Internet*, *Lecture Notes in Computer Science*, vol. 2050 (Springer Verlag, 2001)
62. E. Wandeler, L. Thiele. Real-Time Calculus (RTC) Toolbox. http://www.mpa.ethz.ch/Rtctoolbox (2006).
63. W. Haid, M. Keller, K. Huang, I. Bacivarov, L. Thiele, in *Proc. Int'l Conf. on Systems, Architectures, Modeling and Simulation (IC-SAMOS)* (Samos, Greece, 2009), pp. 92–99
64. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, ACM Trans. on Embedded Computing Systems **7**(3), 36:1 (2008)
65. A.D. Pimentel, M. Thompson, S. Polstra, C. Erbas, Journal of Signal Processing Systems **50**(2), 99 (2008)
66. D. Rai, H. Yang, I. Bacivarov, J.J. Chen, L. Thiele, (DATE11, Grenoble, France, 2011)
67. L. Schor, I. Bacivarov, H. Yang, L. Thiele, in *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (IEEE Computer, Beijing, China, 2012)
68. L. Thiele, L. Schor, H. Yang, I. Bacivarov, in *Proc. Design Automation Conference (DAC)* (ACM, San Diego, California, USA, 2011), pp. 268–273
69. E. Zitzler, L. Thiele, IEEE Trans. on Evolutionary Computation **3**(4), 257 (1999)
70. E. Zitzler, L. Thiele, J. Bader, in *Conf. on Parallel Problem Solving From Nature (PPSN)* (Dortmund, Germany, 2008), pp. 847–858
71. L. Thiele, S. Chakraborty, M. Gries, S. Künzli, in *Proc. Design Automation Conference (DAC)* (New Orleans, LA, USA, 2002), pp. 880–885
72. E. Zitzler, M. Laumanns, L. Thiele, in *Proc. Evolutionary Methods for Design, Optimisation, and Control (EUROGEN)* (Athens, Greece, 2001), pp. 95–100
73. S. Bleuler, M. Laumanns, L. Thiele, E. Zitzler, in *Int'l Conf. on Evolutionary Multi-Criterion Optimization (EMO)* (Faro, Portugal, 2003), pp. 494–508
74. G.K. Wallace, IEEE Trans. on Consumer Electronics **38**(1), 18 (1992). https://doi.org/10.1109/30.125072
75. P. Marwedel, J. Teich, G. Kouveli, I. Bacivarov, L. Thiele, S. Ha, C. Lee, Q. Xu, L. Huang, (CODES+ISSS'11, October 9–14, 2011, Taipei, Taiwan., 2011)

Intermediate Representations for Simulation and Implementation

Jerker Bengtsson

Abstract Simulation and implementation of DSP systems is often a challenge due to their complex dynamic behaviour and requirements on non functional properties. This chapter presents examples of high-level intermediate representations for implementation of design tools for parallel DSP platforms, considering modeling of non constant behaviour of programs; specialized models of computation; scheduling strategies; heterogeneous and hierarchical specifications of systems; and implementing performance analysis for design space exploration and optimization of assignments during a development process. Examples from different intermediate representations that are representative for explored techniques for simulation and implementation are presented. The basic structure and the usage of these representations are demonstrated with examples.

1 The Role of Intermediate Representations

An intermediate representation (IR) is an abstraction between the source language and the executable code for a certain target machine. In early days of compiler development, IRs were introduced to be able to create modular and retargetable implementations of compilers. A modular designed compiler makes it possible to build new back ends for an existing front end, which enables compilation portability of a source language to a different machine. Similarly, it is easier to build a new front end for translation of a different source language, and produce machine code using an existing back end. When designing an IR, the general goal is that it should be independent of details given in a particular source language and should commit to as little as possible of details describing the target machine. In a typical compile flow, the front end translates the source language, performs optimization with respect

J. Bengtsson (✉)
Saab Group, Stockholm, Sweden
e-mail: jerker.bengtsson@saabgroup.se

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_20

to source language aspects and maps the program to the IR. The back end then performs target related optimization and generates binary code for the target.

1.1 Forms of Representations

The form of an IR can be different dependently on how it is intended to be utilized. The minimum requirement is that is should capture all information needed to correctly execute the original program. It can for example be a data structure (or more typically a complex set of structures), or in the form of a language, or even an executable model. Data structure representations are commonly used in well known compiler frameworks like GCC and LLVM. Alternatively to development and compilation of applications written in C or Java, there are other forms of intermediate representations that are suitable for coarse-grained models of computation. Examples of such models are synchronous dataflow (SDF) and cyclo-static dataflow (CSDF). Besides that such models of computation constitute a good match with signal processing applications, they provide a highly suitable application representation for compilation of code to parallel hardware. Moreover, as will be explored with examples in this chapter, such well defined models enables automated analysis of static as well as dynamic execution properties.

1.2 Representation for Parallel and Distributed Hardware

Compared to compiling programs for general purpose processors, mapping digital signal processing (DSP) applications on parallel hardware, e.g. reconfigurable processors [6] and multicores [14], involves several complex steps including scheduling, synchronization and communication. Furthermore, DSP applications are typically associated with constraints of non-functional character. Some examples of such constraints are requirements on timing and power-efficiency. One classical problem is execution time analysis, which requires analysis techniques that consider variable computation and communication times. One complication is that execution times for the different components of a parallel program often are dependent on the current state of the system (i.e. the state of the hardware, the program and the I/O). The cost, e.g. in execution time, for some particular operation typically includes both a static cost and a dynamically varying cost. A static cost of an operation is a cost that is independent of at which time the particular operation is executed. For example, the cost for a processor to execute n arithmetical or logical instructions. A dynamic cost of an operation is a cost that is dependent on at which point in time the operation is executed. For example, such a cost can be a read or write blocking time when multiple cores are concurrently using shared resources (e.g. transactions over an interconnection network or writing to or reading from a global memory).

For multi-objective implementation and optimization of DSP software, the IR constitutes an important source for analysis and optimization. The process of mapping an application to a parallel target is often divided into several steps. The number of steps and the task performed at each step might be dependent on the target machine architecture and the choice of strategy used for implementation. Considering multiprocessor and multicore targets, the mapping (scheduling) process is typically performed in three steps:

1. assigning tasks to processors (*assignment*),
2. determining the execution order of tasks assigned to each processor (*ordering)*,
3. determining when each task should be executed (*timing*).

These steps can independently be performed at either compile-time or at runtime, which results in different scheduling strategies. The fundamental goal of scheduling is to find a schedule of the program that optimizes execution with respect to some performance measure. Often it is the architecture of the target hardware and the specific domain of application that determines how efficiently a certain strategy can be implemented. Table 1 shows four different classes of classical scheduling strategies using the taxonomy given by Lee and Ha [13]. This taxonomy does not provide any exact bounds between all possible scheduling strategies, but it demonstrates different typical combinations of compile time or run time decision making in the scheduling process.

The scheduling strategy is in particular important to consider when analyzing non functional-properties of a program's execution, e.g. execution times and power consumption. Naturally, the accuracy of a simulated program execution at the level of the IR is dependent on how well the IR models the execution behavior (e.g. tasks execution order, communication and timing of tasks) on the targeted hardware platform. One of the more popular and efficient scheduling strategies for the DSP domain is the self-timed scheduling strategy, in which processor assignment and task ordering is determined at compile-time, see Table 1. One of the advantages with self-timed scheduling is that execution times of the program does not have to be precisely known. Self-timed scheduling is therefore also considered to be robust since small variations in execution times will not affect the functional correctness of the execution [12].

Table 1 Scheduling taxonomy

Strategy	Assignment	Ordering	Timing
Fully dynamic	Run	Run	Run
Static-assignment	Compile	Run	Run
Self-timed	Compile	Compile	Run
Fully static	Compile	Compile	Compile

Four examples on different classes of scheduling strategies. Scheduling decisions that are made at compile time are denoted *compile* and scheduling decisions that are made during run time are denoted *run*

A schedule of a program executed in parallel is typically modeled using graph-theoretic models. One example of such graph model is the inter processor communication graph (IPC graph) G_{ipc} [18]. IPC graphs are used to co-model both computation and communication and does not assume execution using global program synchronization. Furthermore, as will be presented further on in this chapter, timing models can be added to IPC graphs to enable analysis of dynamic execution properties. Many implementations of intermediate representations for multiprocessor and multicore systems target the self-timed execution scheduling strategy.

The rest of this chapter will present examples, consisting of four different but fairly closely related types of intermediate representations, which demonstrates of state-of-the-art techniques for representation, and in particular, simulation and analysis of parallel systems. This includes both untimed and timed intermediate representations. Untimed representations are introduced in Sect. 2. Two concrete examples of such well known representations—the system property intervals (SPI) and the FunState representation are presented. The SPI representation enables representation of changing properties of programs, such as for example variable execution times. The FunState representation is an enhancement of the SPI, which unlike SPI, include more powerful means for representing different scheduling strategies.

Unlike untimed representations, timed representations include a notion of system time. Introducing the notion of system time enables representation, simulation and analysis of dynamic processing costs. In Sect. 3, timed representations for dynamic simulation and analysis of non-functional properties are presented. Two examples of such closely related intermediate representations are introduced—job configuration networks and timed configuration graphs (TCFG).

2 Untimed Representations

There are a number of representations developed for synthesis and analysis of parallel systems. This section introduces two examples of such closely related representations and how they are used. The first example, System Property Intervals (SPI), is a representation that uses the concept of constraints intervals to model non constant behavior of a system, for example variable execution times of tasks. The second example, FunState, is based on the concept of functions and state machines, which enables representation of different scheduling strategies. Thus, SPI and FunState are closely related and are both examples of representations which can be used for static (untimed) analysis of systems.

2.1 Representation of System Property Intervals

The System Property Intervals (SPI) is developed to represent and simulate the internal design of heterogeneous systems [5]. SPI is an untimed representation which can be used for static timing analysis of a system. The abstract behavior of a system can be described by means of processes communicating via channels, see Fig. 1. Each process is associated with a set of parameter intervals that abstractly represent the dynamic behavior of the system, which are of importance for analysis of a particular behavioral aspect of the application. Some typical examples are communication behavior, rules for activation of tasks and latency for different resources. The values of the parameters are constrained by intervals in the form of minimum and maximum bounds (hence the name system property intervals). As an example, execution time is one parameter that is often not constant. It can be specified by means of a best case (lower bound) and an worst case (upper bound).

An SPI model is formally a graph $G = (P, C, E)$ where

- P is the set of process nodes,
- $C = Q \cup R$ is the set of channel nodes, and
- $E = (P \times C) \cup (C \times P)$ is the set of edges.

A channel C is represented either as a FIFO (first in first out) queue $q \in Q$ or as a register $r \in R$. Processes are data driven, which means that a process is activated when all its input required for execution is available. To be able to derive rules for when processes are activated it is necessary to know the amount of data communicated. The communication in a graph is specified by assigning *data rates* for each input and output channel for all processes. The data rate specifies how many tokens that are produced or consumed through a channel during a process activation interval. A token is an abstraction for the type of data communicated on a particular channel.

Each process is associated with a set of input and output data rate constraining intervals; one constraining interval for each of the input and output channels respectively. An input data rate constraining interval is specified as $R_c = [r_{c,min}, r_{c,max}]$, where c denotes the specific input channel, $r_{c,min}$ the minimum data rate and $r_{c,max}$ the maximum data rate. Similar to the input data rate intervals, each output channel is associated with an output data rate constraining interval $S_c = [s_{c,min}, s_{c,max}]$. Note that the rate constraining intervals only specifies the maximum and minimum possible rates for a channel when execution is simulated.

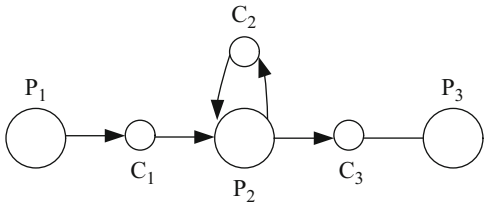


Fig. 1 Simple SPI model graph. Process nodes are denoted P and channel nodes are denoted C

Similarly to the specification of data rate intervals, each process is associated with a latency constraint interval. The latency constraint interval for a process p is described as $Lat_p = [lat_{p,min}, lat_{p,max}]$. The activation time t_{act} is the time when all data is available and the process is activated. The starting time t_{start} is the time when input data has been read and the process starts its execution. Finally, the completion time t_{comp} is the time when output data has been written and the process stops its execution. Thus, the execution time for the kth execution of process p can easily be calculated as $t_{comp,p}(k) - t_{start,p}(k)$. Obviously, the result must be within the interval given by Lat_p in order to satisfy the requirements on maximum latency for the process in question.

2.1.1 Specification of Process Mode Changes

For more complex applications it is desirable to be able describe different execution modes of processes. For example, a process might configure its next execution dependently on its current data input. This kind of process mode change can be managed using a combination of a *process mode*, a *mode tag* and an *activation function* specification. A process mode m_p is abstractly described by means of a latency interval Lat_p, an input data rate interval R_c and an output rate data interval S_c, for each of its input and output channels respectively. Each process can now be associated with a non-empty, finite set of process modes $M_p = m_{p,1}, \ldots, m_{p,n_p}$, where n_p is the number of modes for the process.

Considering the SPI graph in Fig. 2, where process P_1 has i modes, process P_2 has j modes and process P_3 has only one mode. The process modes for this graph is then described by

$$
\begin{aligned}
m_{P_1,i} &= ([lat_{P_1,min}(i), lat_{P_1,max}(i)], [r_{C_3,min}(i), r_{C_3,max}(i)], \\
&\quad [s_{C_1,min}(i), s_{C_1,max}(i)]) \\
m_{P_2,j} &= ([lat_{P_2,min}(j), lat_{P_2,max}(j)], [r_{C_1,min}(j), r_{C_1,max}(j)], \\
&\quad [s_{C_2,min}(j), s_{C_2,max}(j)]) \\
m_{P_3,1} &= ([lat_{P_3,min}(1), lat_{P_3,max}(1)], [r_{C_2,min}(1), r_{C_2,max}(1)], \\
&\quad [s_{C_3,min}(1), s_{C_3,max}(1)])
\end{aligned}
$$

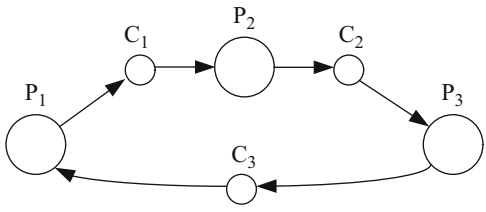


Fig. 2 Simple SPI graph with three processes. Process P_1 has i modes, process P_2 has j modes and process P_3 has only one mode

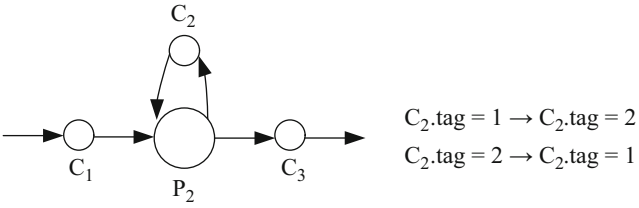


Fig. 3 An SPI model graph to the left in the figure and its associated mode tag production rules to the right. If the mode tag on channel 2 has the value 1, then a mode tag with the value of 2 is produced on channel 2. Similarly, if the mode tag has the value 2, then a mode tag with value 1 is produced on channel 2

Note that process P_1 would have i process mode descriptions and process P_2 would have j descriptions. For simplicity, here only the mode descriptions for the ith and jth mode, of process P_1 and P_2 respectively, are shown. Process P_3 has only one mode, thus only one mode description is required.

In many cases the execution mode of a process to be activated is determined by the content of the input data. That is, the input data for next execution has to be inspected. This dynamic mode change behavior is represented by *mode tags* and so called *activation functions*. Each process is associated with a set of *mode tag production rules* TP_p. Each of the mode tag production rules maps a specific input mode tag to a specific output mode tag $tp_p : I_p \rightarrow O_p$, see Fig. 3. I_p is a finite set of possible input mode tag patterns and O_p is the finite set of possible output mode tag patterns. An input mode tag pattern is a tuple having one entry for each input channel of a process. Similarly, an output mode tag pattern is a tuple having one entry for each output channel.

The role of the activation functions is to select next execution mode of a process based on the current input mode tag pattern. An activation function is a finite set of rules σ which each maps to a mode $M_\sigma \in M_p$. A rule is evaluated by means of a function of the number of available input tokens ($c.num$) and the first available input tag of some input channels of the process ($c.tag$). A process is only activated if and only if there is any rules σ evaluated to a 'true' value. To specify the cyclo-static mode change for process P_2 in Fig. 3, the set of rules are $\sigma = \{\sigma_1, \sigma_2\}$, where $\sigma_1 : (c_2.tag = 1) \rightarrow m_{P2,1}$ and $\sigma_2 : (c_2.tag = 2) \rightarrow m_{P2,2}$. Here the modes are activated on the basis of mode tags only while the number of input tokens on channel C_1 is omitted.

2.1.2 Specification of Latency Constraints

Latency constraints can be specified for paths of channels and processes. The latency constraint is the allowed time interval for which tokens are allowed to be communicated within on the path. A *path latency constraint* is a labeled path in

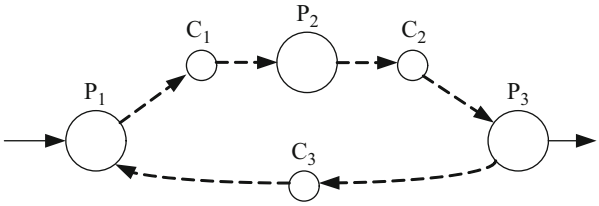


Fig. 4 Example of a specification of a cyclic latency path that starts and ends at process P_1. The dashed edges constitute the path $path = (C_1, C_2, C_3)$

the model graph. A path latency constraint involving n processes is described in the form $(p_1 \rightarrow c_1 \rightarrow p_2 \ldots \rightarrow c_{n-1} \rightarrow p_n)$. Process p_1 is the origin sending process and process p_n is the destination receiving process. A latency path including n processes is connected through $n - 1$ channels. Such a path can be described in short form as $path = (c_1, \ldots, c_n)$. Similarly to process latencies and channel data rates, path latency constraints are specified by means of intervals. A latency constraint interval is given by $LC_{path} = [t_{lat,min}, t_{lat,max}]$.

Figure 4 shows an example of an SPI graph with a cyclic latency path originating and ending at process P_1. The latency constraint for this path is given by $LC_{path} = [T_{min}, T_{max}]$.

2.1.3 Concluding Remarks on System Property Intervals

SPI models are very powerful in the sense that it is possible to describe heterogeneous compositions of models of computation. However, representations in the form of arbitrarily specified process networks graphs is most often not suitable for analysis. Instead, the typical usage of a representation like SPI is to start from reduced specifications of one or several more decidable sub-classes of process networks. For example parameterized synchronous dataflow (PSDF), cyclo static dataflow (CSDF) or synchronous dataflow (SDF). For these more restricted models there exists well-known and proven analysis techniques. Research has been demonstrated how CSDF graphs can be transformed (unfolded) to homogeneous synchronous dataflow (HSDF) graphs, which then can be mapped using the SPI representation. Further, path latency constraints for such models can be analyzed using integer linear programming techniques. Necessary and sufficient conditions for restricting SPI graphs to CSDF behavior, as well as algorithms for unfolding CSDF graphs and latency path constraints can be found in [19]. There are also other representations which similarly to SPI uses mixed communication of control information and data. Two examples are Huss' co-design model CDM [4] and Eles' conditional process graph [7].

2.2 Representation of Functions Driven by State Machines

One limitation with the previously described representation (SPI) is that it is not possible to represent scheduling strategies. The *FunState* representation is a later enhancement of SPI, which also enables verification and representation of scheduling strategies, as was discussed in Sect. 1. In difference to the SPI representation, FunState models flow of data in separate from flow of control. As was shown by examples, the execution of processes in the SPI representation were autonomously controlled in pure data flow style. A significant difference to SPI is that a FunState model is a model of functions driven by a network global state machine. Furthermore, hierarchical FunState networks and state machines can be represented. This section will describe the basics of the FunState representation, including some basic examples on how the enhancements can be used to represent different scheduling strategies.

2.2.1 Describing an Application in FunState

A FunState graph is a bipartite graph $G = (F, S, E)$ where F is the set of functions (corresponding to processes in SPI), S is a set of storage units (corresponding to channels in SPI), and E is a set of directed edges. No edges are connecting two storage units or two functions.

Figure 5 show an example of a FunState model. The upper part of the figure shows the network of functions (f_1, f_2, f_3) and its storage units (q_1, q_2, q_3, q_4), each initialized with 1, 2, 0, 3 tokens respectively. The lower part shows the finite state machine, which in this case for simplicity has only one state and three possible transitions. Note that the number of states is not identical to the number of functions in the network. Similarly to the SPI representation, there are two types of storage units: queues and registers. A queue is FIFO ordered and its length is unbounded.

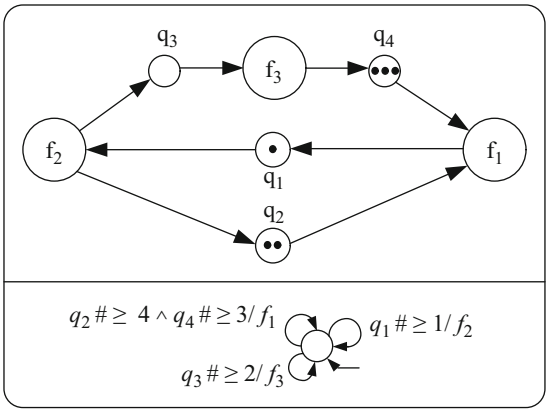


Fig. 5 The figure shows an example of a simple FunState graph. The upper part shows the network of functions and its storage units. The lower part shows the finite state machine, here having one state and three possible transitions. The number of initial tokens in the storage units are represented by filled dots. No filled dots in a storage unit means zero initial tokens

Registers are arrays of pairs $(address, value)$. A register is denoted r and the value of a register is denoted $r\$n$ where n is the id of the register. The number of tokens available in a queue is denoted $q\#$ and $q\$i$ denotes the value of the i'th token in a queue. The number of tokens and their values is a part of the system state. Similarly, registers are denoted r and their values are denoted $r\$1, r\$2 \ldots, r\$n$. However, unlike queues, register values are constants.

Functions $f \in F$ operates on tokens and values. Each input and output of a function is associated with a variable, which denotes the number of tokens consumed (c_i) or produced (p_i) on input and output channels respectively. Production and consumption rates are specified by non negative integer values. A variable evaluates to either a constant value or a random process value. Figure 6 shows a function (f_3) that consumes c tokens from the queue q_1 and 3 tokens from the register r_1. It produces some number in the interval $[1, 4]$ of tokens to queue q_2 and p tokens to register r_2.

State machines are used to specify activation of functions and states of a system. Transitions between states are specified by means of conditions and actions, see Fig. 7. It is possible to specify conditions only concerning the number of tokens in some queue, for example $q\# > u$ where u some variable. Conditions can also concern the value of some queued token, like for example $q\$2 < 1.5$. An action specifies the function that should be activated if the transition is to be taken, see Fig. 7.

When the execution of a FunState model is started the current state of the system (x_c) is always reset to its initial state (x_0). Further, all queues and register values are pre-loaded with initial tokens (queues) and values (registers). During execution of the model the following steps are repeatedly followed in order:

1. **Evaluation of conditions:** All conditions specified for transitions possible from the current state are evaluated.

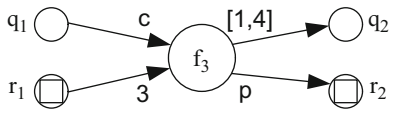


Fig. 6 Example of a function f communicating via queues q and registers r. Consumption and production rates can be specified as constants (input rate 3 from r_1), variables (input rate c from q_1 and output rate p to r_2) or by means of a random interval (output rate [1, 4] to q_2)

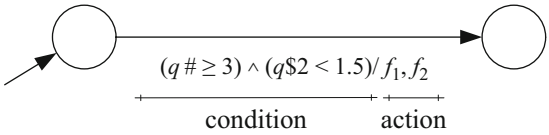


Fig. 7 Example of a specification of conditions and actions for state transitions. The activation of function f_1 and function f_2 is made only if q contains 3 or more tokens and if the value of the second token in q is less than 1.5

2. **Check possible progress of system:** If no transitions is evaluated to be enabled the execution is stopped.
3. **State machine reactions:** One non-deterministic chosen state transition (from the set of enabled transitions where the current state is the source) is chosen. All functions related to the actions associated with the transition is activated.
4. **Fire functions:** All functions that have been activated are executed in non-deterministic order. When a functions is being executed, tokens and values are removed from input queues and registers and tokens and values are added to output queues and registers respectively.

The basic model of FunState further allow construction of hierarchical models including both components and state machines. The semantics of hierarchical FunState models is out of the scope of this chapter and is not presented here. Furthermore, the underlying computational model of FunState can be described and analyzed using both static graph representation and dynamic state transition diagrams [19]. There is also extensions of FunState, including timed functions, representation of timing constraints and timing properties [20].

2.2.2 Examples of Representation of Different Models of Computation

In difference to its successor SPI, FunState enables representation of different input specifications (heterogeneous combinations of different models of computation). Communicating finite state machines are asynchronously operating finite state machines (FSMs) that communicates via FIFO ordered queues. Figure 8 shows an example of a simple communicating finite state machine model. The FSM M_1 of component C_1 can send a value to the queue q whenever it makes a transition. The transitions of the FSM M_2 can then be guarded using predicates on the value of the first element in q.

Figure 9 shows another example in which a SDF program is represented. The actors in the SDF program is in FunState represented by functions. The channels (the edges in the SDF graph) are represented by a queue and two edges (one input, a queue and one output). The number of tokens in the FunState queues are initialized according to the number of initial tokens on the corresponding SDF channels. Each

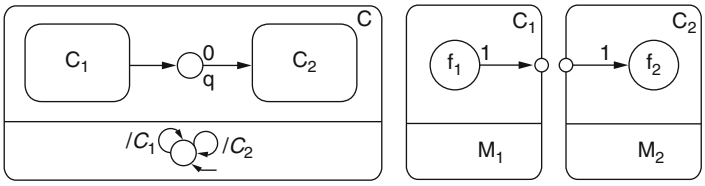


Fig. 8 Hierarchical FunState model representing a small communicating finite state machine. The highest level, shown to the left in the figure, represents FSM M_1 and FSM M_2 by means of two abstract components C_1 and C_2. To the right, the specification for the components C_1 and C_2

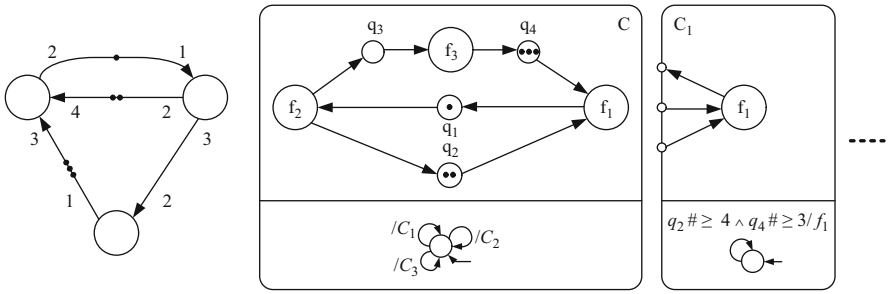


Fig. 9 A synchronous dataflow graph to the left in the figure and a hierarchical FunState representation using local control to the right. Sub model C_1 of the complete model C is shown rightmost in the figure. The state machine will make a transition to f_1 when there is three tokens available on q_4 and four tokens available on q_2

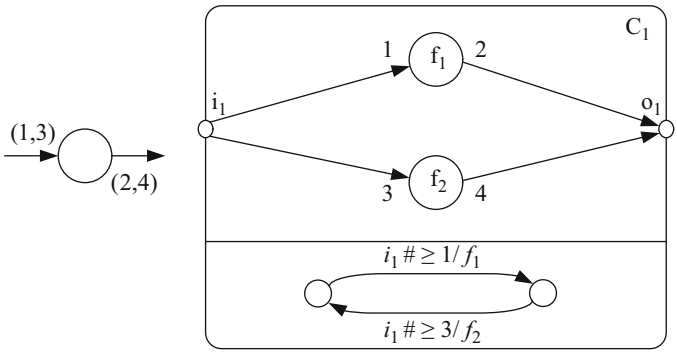


Fig. 10 A CSDF model represented in FunState. The state machine has two states, one for each actor mode, which controls the cyclic mode transitions for the CSDF actor. The initial state is here represented by the left node in the state machine (bottom of the figure)

component of C contains a function and an FSM, which has one state and one transition. The condition for the transition corresponds to the consumption rates for all input edges of the dataflow actor and the action is to activate firing of the actor.

In difference to SDF, CSDF enables periodical changes of production and consumption rates. The graph to the left in Fig. 10 shows a CSDF actor. In the FunState representation, to the right in the figure, the CSDF actor is represented using two functions (f_1, f_2). The state machine cycles through the different actor states following the at compile time specified sequence of input token rates for the CSDF graph.

The last example demonstrates representation of boolean (BDF) and dynamic dataflow (DDF) programs. BDF and DDF are two extensions of SDF, which both enables specification of data dependent dataflow. Figure 11 show switch and select actors in BDF. A select actor is used to select which input of i_1 and i_2 that should be transferred to the output when the actor is fired. The actor is activated when

there is a true or false value ($c\$1 \in true, false$) available on the control channel c. Similarly, the switch actor transfer the output from input i to either of output o_1 or o_2 dependently on the value of the control channel c ($c\$ \in true, false$).

Figure 12 shows a FunState representation of the switch and select actors from Fig. 11. The conditions for the state transitions of the select actor is defined as:

$$\begin{aligned} c_1 &: \quad i_1\# \geq 1 \wedge c\# \geq 1 \wedge c\$1 = true \\ c_2 &: \quad i_2\# \geq 1 \wedge c\# \geq 1 \wedge c\$1 = false \end{aligned}$$

Similarly, the conditions for the state transitions of the switch actor is defined as:

$$\begin{aligned} c_3 &: \quad i\# \geq 1 \wedge c\# \geq 1 \wedge c\$1 = true \\ c_4 &: \quad i\# \geq 1 \wedge c\# \geq 1 \wedge c\$1 = false \end{aligned}$$

Figure 13 shows a non-deterministic merge DDF actor and its FunState representation. The merge actor is activated for firing when there is at least one token present on either of the two input channels (i_1, i_2).

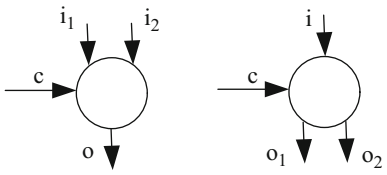


Fig. 11 A select actor to the left in the figure and a switch actor to the right. The switch and select actors chooses its input (output) dependently on the boolean value on the control channel c

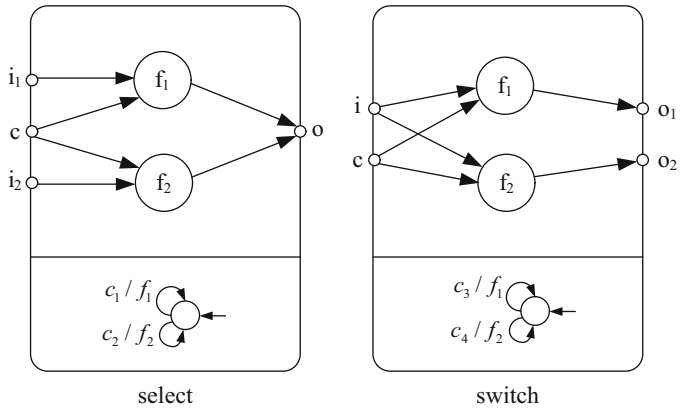


Fig. 12 Switch and select actors represented in FunState

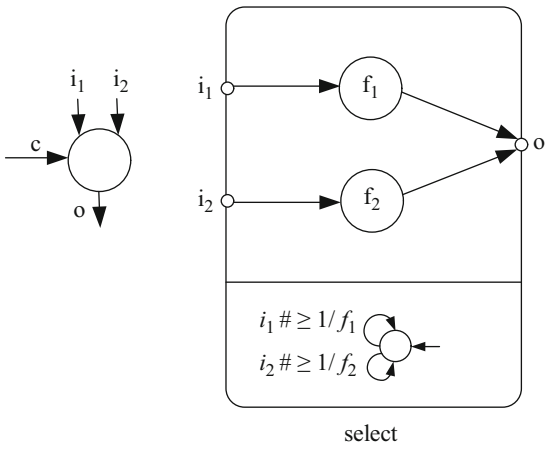


Fig. 13 A non-deterministic merge actor in DDF and its representation in FunState

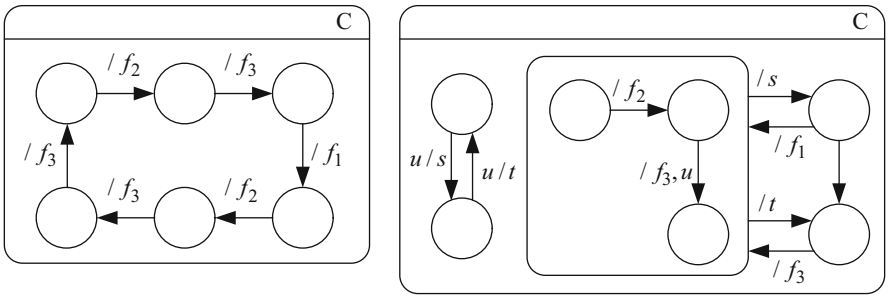


Fig. 14 Two different representations of the same static schedule. The representation to the left shows a straightforward representation of the static schedule. The representation to the right shows another representation of the same schedule but using AND compositions of FunState sub models

2.2.3 Representation of Schedules

FunState allow representation of scheduling strategies. As an example of this, consider the SDF program that was shown in Fig. 5. Only the state machine need to be replaced to specify an exact static schedule for this graph. A periodic single processor schedule for the actors f_1, f_2, f_3 in the SDF graph is $(f_2, f_3, f_1, f_2, f_3, f_3)$. Figure 14 shows two different possible representations when implementing this schedule using a static scheduling strategy. The representation to the left is a straightforward implementation, using a state machine with a state and a transition following the order of this specific schedule. However, the sub-sequence (f_2, f_3) occurs twice in the schedule. This redundancy is taken into account in the corresponding representation to the right, where AND composition between parallel state machines are used.

2.3 Concluding Remarks on Untimed Representations

Similarly to FunState, many other representations that separate data and control flow have been proposed. Some examples are SDL [17], codesign finite state machines (CFSMs) [1], combining synchronous dataflow with finite state machines (FSMs) [10, 15]. However, in comparison to FunState, many of these other approaches are limited in terms of composition, since control and data flow cannot be mixed arbitrarily in the hierarchical levels. More recent and promising work is the actor machine [11]. The actor machine is an abstract machine model for representation of dataflow actors. It has been demonstrated that it facilitates representation necessary for more complex (in terms of non-static) dataflow applications [3].

3 Timed Representations

Most embedded DSP applications have real-time constraints. To provide qualitative automated mappings with respect to performance and timing requirements, the intermediate representation must be amenable to timing analysis. This requires models representing both scheduling strategies as well as the timing properties of the target hardware. This section will present examples of timed representations and of how a machines timing properties can be modeled. The first representation example introduces timing models for analysis of inter processor communication (IPC) graphs. The second representation example includes a timing model and a machine model for a certain class of multi- and manycore processors (arrays of processing tiles).

3.1 Job Configuration Networks

In real-time analysis applications are typically modeled as a set of real-time jobs. A job in this context is considered as a task graph for a complete application. The SDF model of computation is very suitable for specification and real-time analysis of DSP applications. A job configuration network is one proposed executable intermediate representation that can be implemented using process networks [16]. Processes represent processors (cores) and channels represent communication between processors.

3.1.1 Implementation of a Job Configuration Network

Similar to the SPI and FunState representations, job configuration networks are suitably implemented using process networks, see [8]. Process networks provides

a good base for functional representation of parallel program execution on multiprocessors and multicores. Construction of a job configuration network starts from a homogeneous synchronous dataflow (HSDF) graph that describes the application. Then the HSDF graph is partitioned and mapped onto the job configuration network. Each actor in the HSDF graph is assumed to be annotated with execution times in the form of real numbers. The execution time can be a fixed value, the worst case execution time, or a variable number. A job configuration network can be described by a directed graph. The nodes of the graph are processes (denoted P), which each is assigned a subset of the mapped HSDF application graph. Thus, each process represents a processing tile and the set of computation actors mapped on it. Note that the term computation actor is here used to denote an actor that is a part of the HSDF application graph. Other types of actors are also inserted to model communication.

Channels connect actors mapped on different processes. A channel in the job configuration network models unidirectional communication by means of an input buffer, a data connection, a flow-control mechanism and an output buffer. Data from the input buffer (located at the sending side) is pumped through the data connection and is stored in the output buffer (at the receiving side). The flow-control mechanism watches the state of the output buffer in order to prevent overflow. A channel input buffer is associated with a counter counting the number of free places (credits) in the buffer at the output side. Whenever a token is placed on the input of a channel, the credit counter is decremented. The channel is blocked for writing whenever this counter reaches zero. Every time the receiving process removes a token from the channel, a credit is generated and sent back to the sender, and the credit counter is incremented. The time elapsed from departure of a token at the sending side, to the arrival of a token at the receiving side constitutes the data propagation delay. The data propagation delay is denoted δ_D. Correspondingly, the time for transferring a credit from the receiving side to the sending side is denoted δ_C. In the job configuration network, channels are implemented using specific transfer processes (denoted T). A transfer process implements the delayed transfer of tokens from the input buffer to the output buffer according to δ_D and δ_C.

3.2 IPC Graphs

In order to analyze timing properties such as job throughput, execution time and buffer sizing, the job configuration network is transformed into an IPC graph. An IPC graph models the execution of a job on a parallel processor and can be extended with timing models to be amenable for timing analysis. In difference to the application graph (HSDF), and the job configuration network (Process network), the IPC graph contains two kinds of edges: data edges and sequence edges. Data edges represents data dependencies between actors. The sequence edges represents constraints on execution order. No data is transferred on the sequence edges. The sequence edges are used to enforce some specific sequencing of actors. For each individual processing resource and the network, the sequence edges are used to

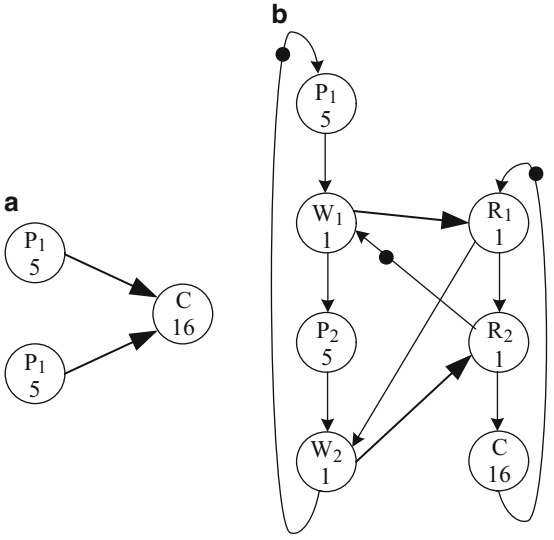


Fig. 15 A simple send receive application in the form of an HSDF graph is shown to the left (**a**). An IPC graph of the same send receive application is shown to the right (**b**). Actors P_1 and P_2 are sending data to the receiving actor C. Two write actors (W_1, W_2) models the write operations of the sending actors (P_1, P_2). Furthermore, two read actors R_1, R_2 models the required read operations of the receiving actor C

create cyclic execution paths when analyzing the timing properties of the graph. In addition to the computation actors specified by the application graph, the IPC graph contains data copy actors (read and write actors). The data copy actors represent data transaction between local memories and global memory.

IPC graphs can be implemented by means of an HSDF graph. Every process and every channel of the job configuration graph can separately be translated into an HSDF model (IPC sub graph), which then are assembled together to form the complete IPC graph for the application. Processes are modeled by means of processor cycles as illustrated by Fig. 15. It includes computation actors (P_1, P_2, C), write actors (W_1 and W_2) and read actors (R_1 and R_2) which performs the read and write operations on the channels.

Figure 16 shows two examples of a buffer model. The first example, (a), is buffer of size 2 without transfer delay. The dashed actors are mapped on processes that reads and writes to the channel. The edges drawn from the left to the right are data edges which models data dependencies. The edges drawn from right to left are sequence edges modeling the flow-control mechanism. The position of the sequence edges and the number of initial tokens on each edge are dependent on the size of the buffer. The sum of produced tokens in the graph can never exceed two. The second example, (b), shows a buffer with transfer delay and buffer size of 9. The transfer delay is implemented by splitting the data and sequence edges and inserting a delay actor in between. Delay actors on data edges are annotated with a delay of δ_D and delay actors on sequence edges are annotated with a delay of δ_C.

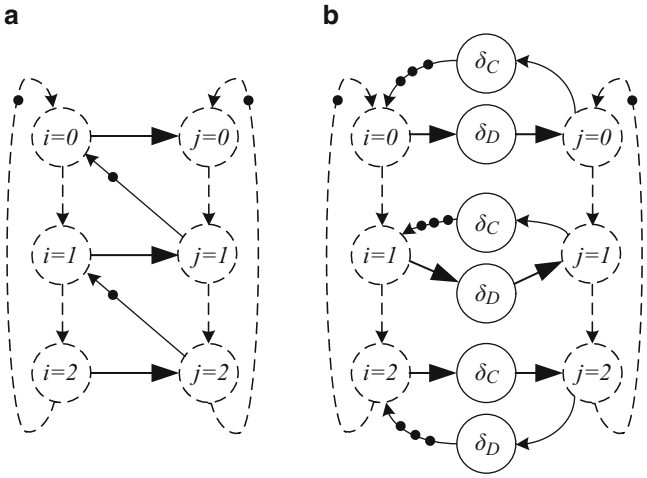


Fig. 16 Two examples of buffer models mapped on an IPC graph. The graph to the left (**a**) implements a buffer of size 2 without transfer delay. Since there is only one initial credit on the outgoing edges from $j = 1$ and $j = 2$ respectively, actor $i = 0$ will block until there are new credits on the sequence edges. Similarly, the graph to the right (**b**) implements a buffer of size 9 with transfer delays (δ_C, δ_D). Each one of the sequence edges, from right to left, have three initial credits. Thus only three iterations (nine data tokens produced) are possible until new credits have been placed on the sequence edges

3.2.1 Timing Analysis of IPC Graphs

Figure 17 shows the final IPC graph, implemented by means of an HSDF model, for the small producer consumer job shown in Fig. 15. The timing models for the IPC graph assume self-timed execution and that multiple iterations of the graph are allowed to overlap during execution. For IPC graphs with actors annotated with constant (worst-case) execution times it is possible to obtain measures on the worst-case throughput and lateness of the represented job. However, there is a set of restrictions the IPC graph must comply to:

1. The graph must be *strongly connected*. For any two actors in the graph, there must exist a directed cycle that contains both actors.
2. The graph must enforce *first-in-first-out* token production order. The output of actors must be produced in order with the actors starting order.

The second restriction is necessary when actors have variable execution times. Consider the case that the execution time of iteration $i+1$ of an actor is less than the execution time for iteration i. Then the token output of instance $i + 1$ can overtake the output of instance i. This restriction can be enforced by the usage of initial tokens on actors cycles. For each actor in the graph, it must be part of at least one cycle that has only one initial token. This restriction is enforced in the example in Fig. 15, where each actor belongs to a processor cycle and every cycle has only one initial token.

A *simple cycle* in the graph is a cyclic path that cannot contain any actor more than once. The edges of cycle can be either data edges or sequence edges. The *cycle weight* is the sum of the execution times for the actors in the cycle, which is a constant value since the actors execution times are constant. The *cycle mean* is the cycle weight divided by the sum of all initial tokens on the edges of the cycle. The cycle in the graph that has the largest cycle mean is said to be the *critical cycle*. The critical cycle constitutes the slowest part of the graph and it is the cycle that constrains the speed of computation for the whole graph.

The timing analysis demonstrated in this section makes use of three important properties of such HSDF graphs: monotonicity, periodicity and boundedness:

Periodicity For self-timed execution of an IPC graph (HSDF) $G(V, E)$ with fixed execution times, the periodicity is:

$$s(v, k + N) = s(v, k) + p \times N; v \in V, k \geq K \tag{1}$$

where the iteration interval p is the maximum cycle mean (MCM), k is the iteration index, $s(v, k)$ is the starting time of actor v, K is the number of iterations required before the self-timed execution enters the "periodic regime", and N is the number of iterations in one period. With a periodic regime, it is meant that every actor is guaranteed to start exactly N firings within any half-closed interval of length $p \times N$. Thus, by finding the actors starting times $s(v, k)$ during one period, it is possible to characterize the periodic execution of the IPC graph. If this information is available, it is also possible to obtain a measure of the graphs lateness.

Lateness The lateness σ of a graph is

$$\sigma = \max_{v \in V}(\max_{k=n..n+N-1}(s(v, k) - p \times k + t(v))) \tag{2}$$

where $t(v)$ is the fixed execution time of actor v and n is an arbitrary integer $n \geq K$. The lateness (maximum latency) is a measure on how late iteration k of the IPC graph, starting from time $p \times k$, can finish.

Boundedness The upper bound on the completion time of I iterations of the IPC graph (HSDF) is given by

$$HSDF - BOUND(I) = p \times (I - 1) + \sigma \tag{3}$$

As mentioned, the value of the iteration interval p can be found by computing the MCM (the maximum critical mean) of the IPC graph. To compute the lateness σ, it is necessary to have a sample of the starting times $s(v, k)$. This means that also that K and N have to be known. Furthermore, in case $I < K$, the HSDF $-$ BOUND(I) can be found through simulating I iterations of the graph [16].

The example in Fig. 17 shows the result after translating the configuration network for the producer—consumer example in Fig. 15 to an IPC graph. There are rules for how to position the backward edges and how to set the number of

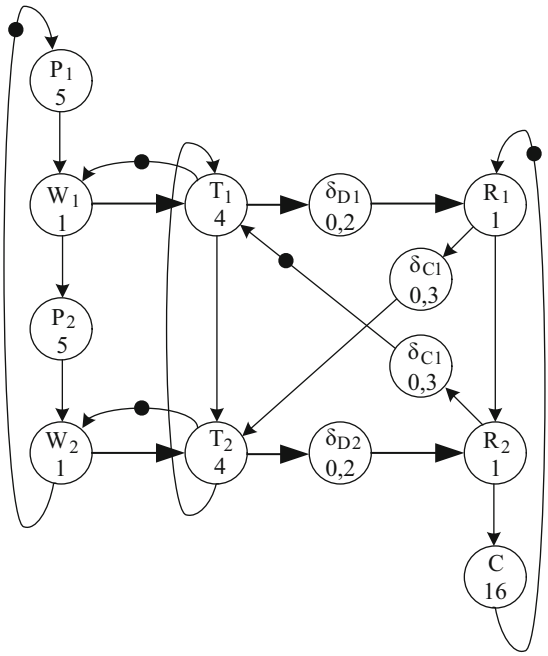


Fig. 17 Complete IPC graph for the send receive example in Fig. 15

initial tokens on these edges, but they will not be discussed further here [16]. The cycle (P_1, W_1, P_2, W_2) represents the computation of the actors mapped on the first processing tile. Similarly, the cycle (R_1, R_2, C) represents the actor C mapped on the second processing tile. The transfer actors (T_1 and T_2) and the delay actors ($\delta_{D1}, \delta_{D2}, \delta_{C1}, \delta_{C2}$) represent the channel connection between the two processing tiles.

The IPC graph in Fig. 17 is amenable for timing analysis. The average iteration interval p (MCM) can be determined by finding the critical cycle in the graph. In this case, the cycle $R_1, \delta_{C1}, T_2, \delta_{D2}, R_2, C$ is critical, resulting in an iteration interval of 22.5 time units. One way to improve the performance is to modify the size of the channels output buffer. The size of the output buffer can be increased by adding extra initial tokens. If the buffer size is increased from 2 to 3, there will be one extra token on edge (R_1, T_2). This will reduce the cycle mean. The new critical cycle will be (R_1, R_2, C). Note that the structure of the graph has not been changed.

One severe limitation with Job Configuration Networks is that the HSDF application graph tend to grow undesirable in terms of size and evaluation complexity for more complex dataflow applications. Another more recent approach that can be taken for more complex applications is to use max+ analysis [9]. The max+ analysis is based on linear-system theory and rely on SDF graphs and an FSM to represent more dynamic application scenarios.

3.3 Timed Configuration Graphs

If the goal were just to abstract away processor specific details, the intermediate representation could very well be a fairly simple graph data structure. However, in order to be able to analyze non-functional properties for a certain mapping of a graph, there is a need also to represent both time and the dynamic behaviour of the hardware. Timed configurations graphs (TCFGs) are closely related to job configurations networks. This section presents how TCFGs are constructed to represent programs mapped on tiled parallel processors. TCFGs can easily be implemented using a hierarchical heterogeneous model consisting of process networks and SDF. Programs are specified using a multi-rate SDF graph. The processing resources are implemented abstractly using process networks. Furthermore, a machine model is used to describe computational resources and performance of the processor target [2].

3.4 Set of Models

A program is specified using a multi-rate SDF graph. Each actor in the SDF graph is assumed to be associated with a tuple

$$< r_p, r_m, R_s, R_r >$$

where

- r_p is the worst case execution time, in number of operations.
- r_m is the requirement on memory, in words.
- $R_s = [r_{s_1}, r_{s_2}, \ldots, r_{s_n}]$ is a sequence where r_{s_i} is the number of words produced on output channel i each firing.
- $R_r = [r_{r_1}, r_{r_2}, \ldots, r_{r_m}]$ is a sequence where r_{r_j} is the number of words consumed on input channel j each firing.

Scheduling algorithms require cost estimates in order to find a qualitative mapping with respect to some optimization objective. For communication resources on parallel processors, the cost typically comprise a static cost for sending and receiving data and a dynamic cost determined by the resource location, the amount of data to be communicated and the current state of the processor. Thus, the accuracy of the cost estimates depends on how well the dynamic overhead associated with communication, synchronization and when accessing off-chip resources can be captured.

The machine model presented here can be used to represent a certain class of array structured multi processors and multi cores. The term processing tile is here used to refer to one core or processor in such an architecture. The machine model comprises a set of parameters describing the computational resources and a set of

abstract performance functions, which describe the performance of computations, communication and memory transactions. The processing tiles are assumed to be tightly coupled via a mesh network. Each tile is assumed to have individual instruction sequencing capability. Memory transactions between tile private and shared memory is managed in software. The resources of such an abstract tile architecture can be described using two tuples, M and F. M consists of a set of parameters describing the resources:

$$M =< (x, y), p, b_g, g_w, g_r, o, s_o, s_l, c, h_l, r_l, r_o >$$

where

- (x, y) is the number of rows and columns of cores.
- p is the processing power (instruction throughput) of each core, in *operations per clock cycle*.
- b_g is global memory bandwidth, in *words per clock cycle*
- g_w is the penalty for global memory write, in *words per clock cycle*
- g_r is the penalty for global memory read, in *words per clock cycle*
- o is software overhead for initiation of a network transfer, in *clock cycles*
- s_o is core send occupancy, in *clock cycles*, when sending a message.
- s_l is the latency for a sent message to reach the network, in *clock cycles*
- c is the bandwidth of each interconnection link, in *words per clock cycle*.
- h_l is network hop latency, in *clock cycles*.
- r_l is the latency from network to receiving core, in *clock cycles*.
- r_o is core receive occupancy, in *clock cycles*, when receiving a message

F is a set of abstract common functions describing the performance of computations, global memory transactions and local communication as functions of the resources M:

$$F(M) =< t_p, t_s, t_r, t_c, t_{gw}, t_{gr} >$$

where

- t_p is a function evaluating the time to compute a sequence of instructions
- t_s is a function evaluating the core occupancy when sending a data stream
- t_r is a function evaluating the core occupancy when receiving a data stream
- t_c is a function evaluating network propagation delay for a data stream
- t_{gw} is a function evaluating the time for writing a stream to global memory
- t_{gr} is a function evaluating the time for reading a stream from global memory

The first step when modeling a specific processor target is to set the values of the parameters of M. The second step is to define the performance functions $F(M)$.

3.4.1 Modeling a Tiled 16 Cores Processor

In this section it is demonstrated how the machine model can be configured in order to model an array structured parallel processor. It is assumed that the processor has 16 (4×4) programmable tiles. Each tile includes a single-issue core and private data and instruction memory. The tiles are tightly interconnected via a dimension-ordered (x-y), wormhole-routed network. Shared memory consist of four off-chip, individually addressed memory banks located off-chip. Transactions between tile-local and shared memory are programmed in the software, using message-passing via the on-chip interconnection network. The parameters for a processor with this configuration are can be specified with following parameter setup:

$$M =< (4, 4), 1, 1, 1, 6, 2, 5, 1, 1, 1, 1, 3 >$$

The core instruction throughput is p operations per clock cycle. Thus, for a single-issue core $p = 1$. The memory bank consisting of four shared off-chip DRAMs are connected to four separate I/O ports. The DRAMs can be accessed concurrently, each having a bandwidth of $b_g = 1$ words per clock cycle. The latency penalty for a DRAM write is $g_w = 1$ cycle and for a read the latency is $g_r = 6$ cycles. The overhead for a initiating network transfer includes sending a header and possibly an address (when addressing any of the off-chip memories). The software overhead for this initiation is here set to $o = 2$. The on-chip networks are assumed to be register mapped, meaning that after a message header has been sent, the network can be treated as destination or source operand of an instruction. Ideally, this means that the receive and send occupancy is zero. In practice, when multiple input and output channels are merged and physically mapped on a single network link, data needs to be buffered locally. Here a send and receive occupancy of $s_o = 2$ and $r_o = 2$ will be assumed. Note that this occupancy then also include the overhead for the data to be read from and written to local memory. The network hop-latency is set to $h_l = 1$ cycles per router hop and the link bandwidth is assumed to be $c = 1$. Furthermore, the send and receive latency is one clock cycle when injecting and extracting data to and from the network: $s_l = 1$ and $r_l = 1$.

After configuring the parameters of the machine model, the next step is to specify the performance functions $F(M)$:

Compute The time required to process the fire code of an SDF actor on a processing tile can be defined as

$$t_p(r_p, p) = \left\lceil \frac{r_p}{p} \right\rceil$$

which is a function of the requested number of operations r_p (worst-case execution time associated with each actor) and the core instruction throughput $p \in M$. To r_p all instructions except those related to network send and receive operations are counted.

Send The time required for a processing tile to issue a network send operation can be defined as

$$t_s(R_s, o, s_o) = \left\lceil \frac{R_s}{framesize} \right\rceil \times o + R_s \times s_o$$

Here send is defined to be a function of the requested amount of words to be sent, R_s, the software overhead $o \in M$ when initiating a network transfer, and a possible send occupancy $s_o \in M$. The framesize is a processor specific parameter that specifies the maximum length, in words, of a message. Thus, the first term of t_s captures the software overhead for the number of messages required to send the complete stream of data. For connected nodes in the SDF graph that are mapped on the same core, it is also possible to choose to represent channels mapped in local memory. In that case t_s is set to zero.

Receive Similar to send, the time required for a processing tile to issue a network receive operation can be defined as

$$t_r(R_r, o, r_o) = \left\lceil \frac{R_r}{framesize} \right\rceil \times o + R_r \times r_o$$

Network Propagation Modeling communication accurately is difficult: at the one hand high accuracy requires the use of a low machine abstraction level. At the other, the machine abstraction implemented by the model should be abstract enough to be able to model different processor targets. For simplicity, here it will be assumed that network communication is collision free. The network propagation time here consists of a possible network injection and extraction latency at the source and destination and the propagation time for one router hop on the network. The network propagation time with these assumptions can be defined as

$$t_c(R_s, x_s, y_s, x_d, y_d, s_l, h_l, r_l) = \\ s_l + d(x_s, y_s, x_d, y_d) \times h_l + n_{turns}(x_s, y_s, x_d, y_d) + r_l$$

Network injection and extraction latency are captured by s_l and r_l respectively. Further, the propagation time depends on the network hop latency h_l and the number of network hops $d(x_s, y_s, x_d, y_d)$, which is a distance function of the source and destination coordinates. Routing turns add an extra cost of one clock cycle. This is captured by the value of $n_{turns}(x_s, y_s, x_d, y_d)$ which, similar to d, is a function of the source and destination coordinates.

Shared Memory Read Each memory bank in the shared memory is assumed to be individually controlled by a memory controller. Reading from the shared memory bank requires first one send operation (the overhead which is captured by t_s), in

order to configure the memory controller and set the address of memory to be read. The second step is to issue a receive operation to receive the memory contents on that address. The propagation time when receiving a stream of data from shared memory at a processing tile is here defined as

$$t_{gr}(r_l, x_s, y_s, x_d, y_d, h_l) = \\ r_l + d(x_s, y_s, x_d, y_d) \times h_l + n_{turns}(x_s, y_s, x_d, y_d)$$

Memory read latency is not included in this expression. This needs to be accounted for in a memory model that has to be included in the intermediate representation.

Shared Memory Write Like the memory read operation, writing to global memory requires two send operations: one for configuring the memory controller (set write mode and address) and one for sending the data to be stored. The time required for streaming data from the sending core to global memory is evaluated by

$$t_{gw}(s_l, x_s, y_s, x_d, y_d, h_l) = \\ s_l + d(x_s, y_s, x_d, y_d) \times h_l + n_{turns}(x_s, y_s, x_d, y_d)$$

Like in shared memory read function, the memory write latency is included in the memory model that has to be included in the intermediate representation.

3.5 Construction of Timed Configuration Graphs

A timed configuration graph $G_M^A(V, E)$ represents a synchronous dataflow program A (with annotations $< r_p, r_m, R_s, R_r >$ as described in Sect. 3.4), mapped on an abstract machine $< M, M(F) >$. V is the set of nodes and E is the set of edges. There are two types of nodes in the graph: processor nodes, v_p, and memory nodes, v_b. A processor node represents a set of SDF sub-graphs of A mapped on a processing tile. Memory nodes represent buffers mapped in shared memory. The set of edges, E, represents the configuration for the on-chip network.

When constructing a timed configuration graph, $G_M^A(V, E)$, the SDF graph A is first clustered into sub-graphs. Each SDF sub-graph is then assigned to a processing tile in M during the scheduling step. The edges of the SDF graph that end up inside a node of type v_p will be implemented using local memory, so they do not appear as top level (visible) edges in G_M^A. The edges of the SDF that reach between pairs of nodes not belonging to the same SDF sub-graphs can be mapped in two different ways:

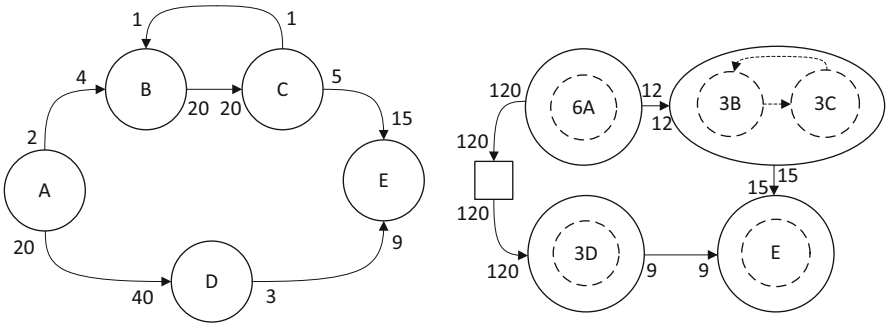


Fig. 18 The graph to the right is one possible graph G_M^A for the application graph A to the left

1. as a network connection between the two processing tiles; such a connection is represented by an edge;
2. as a buffer in global memory. In this case, a memory node is introduced.

When G_M^A has been completely constructed, each $v_p, v_b \in V$ and $e \in E$ has been assigned costs for computation, communication and memory read and writes, respectively. The costs are calculated using the parameters of M and the performance functions $F(M)$ (Sect. 3.4). These costs comprise the static part of the costs, relative to the current time and the current state of the system, when computing the total cost for executing an application.

The graph to the left in Fig. 18 shows a simple multi-rate SDF graph (A). The graph to the right in the figure, shows one possible TCFG for this same SDF graph (G_M^A). One static actor firing schedule for A in this example is $6a3b3c3de3$. Thus, actor a fires six times, actors b, c and d fire three times, and actor e one time. The firing of this schedule is repeated indefinitely. Thus, no runtime scheduling supervision is required since static code can be generated. The feedback channel from actor c to actor b is buffered in core local memory. The edge from actor a to actor d is a buffer in shared (off-chip) memory and the others are mapped as point-to-point connections on the network. The integer values represent the send and receive rates of the channels (r_s and r_r), before and after A has been clustered and transformed to G_M^A, respectively. Note that these values in G_M^A are the values in A multiplied by the number of times an actor fires, as given by the firing schedule.

3.5.1 Abstract Interpretation of TCFGs

Dynamic analysis of a timed configuration graph can be performed using abstract interpretation techniques. An interpreter can be implemented by very simple means using process networks. This section will briefly demonstrate how such an interpreter can be implemented using process networks.

The implementation of timed configuration graphs demonstrated here comprise a heterogeneous and hierarchical dataflow model. The top level is a process network. Nodes representing processing tiles are implemented using specific processor actors (processes), and memory nodes are implemented using specific memory actors (processes). The SDF program, provided as input to the tool, is transformed to a distributed (clustered) SDF model, where each cluster is embedded in a processor actor. The edges connecting the clusters in the distributed SDF graph are cut and replaced by process networks channels, which represent the inter processor communication. However, apart from this temporary cut, the SDF model is still intact.

There are different possible approaches for analysis of the dynamic behavior of a dataflow model. One is to let the model calculate resource and timing properties on itself during execution of the model. Another is to generate an abstract representation of the model. However, using abstract interpretation does not require modification of the underlying modeling architecture used. The timed configuration graph and the abstract interpreter can be added on top of an existing modeling infrastructure. Furthermore, the approach of using abstract representation of the mapped SDF graph is likely more beneficial in terms of modeling performance.

Program Abstraction The firing for each SDF actor is abstracted by means of a sequence, S, consisting of *receive*, *compute* and *send* operations:

$$S = t_{r_1}, t_{r_2} \ldots t_{r_n}, t_p, t_{s_1}, t_{s_2}, \ldots, t_{s_m}$$

The abstract operations have been bound to static costs computed using the machine model M and the defined performance functions $F(M)$, as was exemplified in Sect. 3.4.

Time There is no notion of global time in a process network. Each of the top level vertices of G_M^A (processing tiles and memories) is implemented by an individual process. Each of the processor processes has a local clock, t. The clock, t, is stepped by means of (not equal) time segments. The length of a time segment corresponds to the static cost bound to a certain operation in S and possibly a dynamic cost (blocking time) when issuing send or receive operations addressed to other cores or shared memories.

Processing Tiles and Memory A processor process (processing tile) implements a state machine. For each processor process that is firing, the current *state* is evaluated and then stored in the *history*. The *history* is a chronologically ordered list describing the *state* evolution from time $t = 0$.

Memory processes, v_m, models competing memory transactions by the first come first served policy. A read or write latency is added to the current time of a read or write operation.

States For each processor process it is recorded during which segments of time computations and communication operations were issued. For each process, the

current *state* maps to a state $type \in StateSet$, a start time, t_{start}, and a stop time, t_{stop}. The $state$ of a vertex is a tuple

$$state =< type, t_{start}, t_{stop} >$$

The *StateSet* defines the set of possible state types:

$$\text{StateSet} = \{receive,\ compute,\ send,\ receiveMem,\ sendMem\}$$

The value of t_{start} is the start of the time segment corresponding to the currently processed operation, and t_{stop} is the end of the time segment. For all *state*s, time t_{stop} corresponds to $t_{start} + \Delta$, where Δ includes the static cost bound to the particular operation. For *send*, *receive*, *receiveMem* and *sendMem* Δ also possibly includes a dynamic cost (blocking time) while issuing the operations.

Clock Synchronization The timed configuration graph is synchronized by means of discrete events. Send and receive are blocking operations. A read operation blocks until data are available on the edge, and a write operation blocks until the edge is free for writing. During a time segment, only one message can be sent over an edge. Synchronization of time between communicating processes requires two way communication. Thus, each edge in the mapped SDF graph is represented by a pair of oppositely directed edges in the implementation of G^A_M.

Network Propagation Time Channels perform no computation. Therefore delay actors are used to account for propagation times over edges. The delay actor adds the edge weight (corresponding to $t_c \in F(M)$), that has been assigned to each top level edge during the construction of G^A_M.

Program Interpretation The core interpreter makes state transitions depending on the current operation, the associated static cost of the operation and whether *send* and *receive* operations block or not (the dynamic cost). A state generating function takes timing parameters as input and returns next $state \in StateTypes$.

In this example implementation of the TCFG, the network is modeled on a relatively high level, at which communication is represented using point-to-point channels. Thus, the model does not capture message concurrency and buffer capacity of the network. This is one example of design trade-off made to keep the network abstraction at a high level for reasons of modeling performance and a higher generality of hardware representation. These are however problems that can be solved, but at the price of a lower level of implementation of the intermediate representation.

4 Chapter Summary

Implementation of DSP applications on parallel hardware is a complex task typically involving several design constraints. Many DSP systems are embedded real-time systems, which includes non-functional constraints such as for example timing. The problem of finding the best fitted parallel implementation of a program—with respect to the available processor resources and the design constraints—is a non-trivial implementation task. This chapter has presented useful techniques and different important aspects of intermediate representations aimed for not only representation, but also simulation and evaluation of systems, as well as for implementation in design and development tools. The SPI representation discussed in Sect. 2.1 allows representation of variable execution properties of programs, but does not provide means powerful enough for representation of scheduling strategies. In Sect. 2.2, it was discussed how scheduling strategies could be implemented in the IR using the FunState representation as example. Section 3.1 presented job configuration networks, which unlike SPI and FunState, is a timed executable representation that enables dynamic analysis and simulation. Finally, the concept of timed configuration graphs (TCFGs) (Sect. 3.3) was presented. TCFGs are similar to job configuration networks and have been designed to enable dynamic analysis and evaluate the execution of multi-rate SDF graphs on parallel processors by executing the IR.

References

1. Balarin, F., Chiodo, M., Giusto, P., Hsieh, H., Jurecska, A., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A., Sentovich, E., Suzuki, K., Tabbara, B.: Hardware-software co-design of embedded systems: the POLIS approach. Kluwer Academic Publishers, Norwell, MA, USA (1997)
2. Bengtsson, J., Svensson, B.: Manycore performance analysis using timed configuration graphs. In: IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, pp. 108–117. Samos, Greece (2009)
3. Cedersjo, G.: Efficient Software Implementation of Stream Programs. PhD Dissertation, Department of Computer Science, Lund University (2017)
4. Bossung, W., Huss, S.A., Klaus, S.: High-level embedded system specifications based on process activation conditions. VLSI Signal Processing Systems **21**(3), 277–291 (1999).
5. Cieslok, F., Teich, J.: Timing analysis of process models with uncertain behaviour. Tech. rep., Computer Engineering Laboratory (DATE), University of Paderborn (2000)
6. De Sutter, B., Raghavan, P., Lambrechts, A.: Coarse-grained reconfigurable array architectures. In: S.S. Bhattacharyya, E.F., Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer, New York (2018). https://doi.org/10.1007/978-3-319-91734-4_12
7. Eles, P., Kuchcinski, K., Peng, Z., Doboli, A., Pop, P.: Scheduling of conditional process graphs for the synthesis of embedded systems. Design, Automation and Test in Europe Conference and Exhibition pp. 132–139 (1998).

8. Geilen, M., Basten, T.: Kahn process networks and a reactive extension. In: S.S. Bhattacharyya, E.F., Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer, New York (2018). https://doi.org/10.1007/978-3-319-91734-4_24
9. Geilen, M., Stuijk, S.: Worst-case Performance Analysis of Synchronous Dataflow Scenarios. In: Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 125–134. Scottsdale, Arizona, USA (2010).
10. Grotker, T., Schoenen, R., Meyr, H.: PCC: a modeling technique for mixed control/data flow systems. In: Proceedings of the 1997 European conference on Design and Test, pp. 482–486. IEEE Computer Society, Washington, DC, USA (1997)
11. Janneck, J.W.: A machine model for dataflow actors and its applications In: Signals, Systems and Computers (ASILOMAR), Conference Record of the Forty Fifth Asilomar Conference on, pp. 756–760, IEEE, Pacific Grove, CA, USA (2011).
12. Lee, E.A., Goe, E.E., Heine, H., Ho, W.H., Bhattacharyya, S., Bier, J.C., Guntvedt, E.: GABRIEL: a design environment for programmable DSPs. In: Proceedings of the 26th ACM/IEEE Design Automation Conference, pp. 141–146. ACM, New York, NY, USA (1989).
13. Lee, E.A., Ha, S.: Scheduling strategies for multiprocessor real-time DSP. In: Proc of the IEEE Global Telecommunications Conference, vol. 2, pp. 1279–1283 (1989).
14. Leupers, R., Aguilar, M.A., Castrillon, J., Sheng, W.: Software compilation techniques for heterogeneous embedded multi-core systems. In: S.S. Bhattacharyya, E.F., Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer, New York (2018). https://doi.org/10.1007/978-3-319-91734-4_28
15. Pankert, M., Mauss, O., Ritz, S., Meyr, H.: Dynamic data flow and control flow in high level dsp code synthesis. In: IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 2, pp. 449–452. Adelaide, SA, Australia (1994)
16. Poplavko, P., Basten, T., Bekooij, M.J.G., Meerbergen, J.V.V., Mesman, B.: Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In: Proceedings of the International Conference on Compilers, Architecture, and Synthesis of Embedded Systems, pp. 63–72 (2003)
17. Smith, J.R.W., Reed, R.: Telecommunications Systems Engineering Using SDL. Elsevier Science Inc., New York, NY, USA (1989)
18. Sriram, S., Lee, E.A.: Determining the order of processor transactions in statically scheduled multiprocessors. VLSI Signal Processing Systems. **15**(3), 207–220 (1997).
19. Thiele, L., Strehl, K., Ziegenbein, D., Ernst, R., Teich, J.: FunState – an internal design representation for codesign. In: ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design, pp. 558–565. IEEE Press, Piscataway, NJ, USA (1999)
20. Thiele, L., Teich, J., Naedele, M., Strehl, K., Ziegenbein, D.: SCF - state machine controlled flow diagrams. Tech. Rep. TIK-33, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich (1998)

Throughput Analysis of Dataflow Graphs

Robert de Groote

Abstract Static dataflow graphs such as those presented in earlier chapters are attractive from a performance point of view, as the rate at which data is processed can be assessed beforehand. Assessing this performance involves analysing the dependency structure and the timings of the different nodes. This chapter describes different ways to approach this problem, and provides a mathematical basis from which these approaches follow. Methods for efficiently analysing the throughput are given, for single-rate (or homogeneous) graphs, synchronous dataflow graphs, and cyclo-static dataflow graphs.

1 Introduction

An attractive property of the various decidable dataflow models such as synchronous dataflow (SDF) and cyclo-static dataflow (CSDF) that are discussed in [24], is that they can be analysed *statically*. That is, *periodic schedules* for these models can be computed beforehand, and thus the rate at which data is processed can be assessed. To illustrate this, consider the SDF graph modelling a voice-band modem, taken from the seminal paper [28], and shown below in Fig. 1. Here, data is processed from left to right: input data is filtered by the front-end (FILT) and Hilbert (HIL) filters, processed by an adaptive equalizer (EQ), phase locked loop (PLL), and eventually decoded (DECO) and output. For the graph, a schedule can be constructed. Although the schedule dictates an order in which the actors fire, it does not imply timing of these firings, because the actors do not have an associated time. By providing an *upper bound* on the execution time of each actor firing (a so-called *worst-case execution time*), a *lower bound* on the maximum firing rate of each actor can be computed. As such, *real-time guarantees* with respect to timing can be given: a guaranteed lower bound on the rate at which data is processed permits designers of

R. de Groote (✉)
University of Twente, Faculty of EEMCS, Enschede, The Netherlands
e-mail: robert.degroote@utwente.nl

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_21

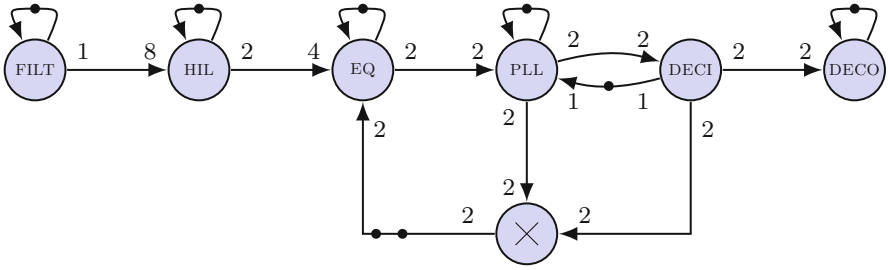


Fig. 1 An SDF graph model for a voice-band modem, taken from [28]

signal processing systems to ascertain whether the system is capable of processing the data at a given rate. This processing rate is called *throughput*.

Throughput is a general term that applies to systems that produce something in response to some input, such as communication networks. The throughput of a system says something about the maximum rate at which outputs can be produced (put through the system), and can be measured by providing a high load to the system. That is, throughput is the rate at which output is produced if input is always readily available.

For an SDF graph model of a signal processing system, the word throughput relates to the maximum rate at which nodes in the graph can fire. Throughput of a graph is limited by its *cycles*, such as the cycle connecting EQ, PLL and the multiplication ($\times$) actor in Fig. 1: because it takes time for a token to process through a cycle, each cycle limits the speed at which tokens may move through the graph. Interconnected cycles synchronise on the "slowest" cycle. Throughput analysis of dataflow graphs thus resorts to finding a graph's bottleneck cycles.

The key to understanding throughput analysis is rooted in *self-timed schedules* [30]. In a self-timed schedule, every actor fires as soon as sufficient data is available. After some time, a self-timed schedule reaches a phase in which the firing times form a complex but periodic pattern [20, 21]. The throughput of an actor is equal to the *average rate* at which an actor fires in this periodic pattern.

This chapter provides both a theoretical background on throughput analysis, using the more general framework of *max-plus algebra*, and a practical guide to how synchronous dataflow graphs may be efficiently analysed for their (maximum) throughput. Rather than giving an in-depth overview of max-plus algebra, we limit the treatment of this subject to its most relevant concepts required for a deeper understanding of the different methods.

This chapter is organized into two main parts. In the first part, we introduce throughput analysis of homogeneous SDF (HSDF) graphs. Key concepts in this first part are *maximum cycle ratio* and *strictly periodic schedules*. Several algorithms for analysing the throughput of an HSDF graph are available. We discuss their principles, and briefly describe three different approaches and their properties.

The second part of the chapter focuses on the analysis of more complex graphs: SDF and CSDF graphs. It gives two approaches for analysing the throughput of

CSDF graphs, building on the principles that underlie the analysis of HSDF graphs. Key concepts used in the second part are *single-rate approximations*, *unfolding transformations*, and *execution state*.

2 Terminology

The analysis of *throughput* is concerned with *temporal* aspects of dataflow graphs. That is, for throughput analysis we abstract from the *functional* semantics (e.g. the values of data tokens and the computations performed by actors) of dataflow graphs, and restrict our attention to those properties that impact the *times* at which firings take place. The view that we adopt in this chapter regards dataflow graphs as (mathematical) structures where *constraints* are imposed on the timing of certain *events*. This requires some terminology, which we introduce in this section. For the sake of consistency, we adopt the terms and notation introduced in earlier chapters, and avoid different interpretations of primary concepts as much as possible.

2.1 Synchronous and Cyclo-Static Dataflow Graphs

In the taxonomy of HSDF, SDF and CSDF, the latter is the most general. We therefore choose to provide the widest terminology, i.e. for CSDF, such that it applies to each of three classes.

A CSDF graph is a *directed graph*, where the *edges* are referred to as *channels*, and *nodes* as *actors* [6, 28]. Actors represent functional units that perform *computations*, by consuming data from incoming channels and producing data onto outgoing channels. These computations take time: each actor v has a periodic *execution time* sequence, denoted τ_v, where $\tau_v(k)$ denotes the time associated with the k-th firing.

Each channel vw has an integer number of *tokens*, denoted δ_{vw}. Furthermore, with each channel are associated two *periodic sequences*: a *production rate* sequence, denoted ρ^+_{vw}, which specifies the number of tokens produced onto vw by consecutive firings of v. Similarly, the *consumption rate* sequence, denoted ρ^-_{vw}, specifies the number of tokens consumed from vw by firings of w. We denote with $\rho^+_{vw}(k)$ and $\rho^-_{vw}(m)$ the number of tokens respectively produced onto vw by the k-th firing of v, and consumed from vw by the m-th firing of w. Also, we denote the number of tokens produced onto wv (consumed from uv) in a single period of actor v by σ^+_{vw} (σ^-_{uv}).

An actor v can fire as soon as sufficiently many tokens are present on each of its incoming channels, as specified by their consumption rates. An actor that can fire is said to be *enabled*. The *period* of an actor is determined by the periods of the (periodic) sequences associated with that actor, through production and consumption rate sequences bound to incoming and outgoing channels, and by the

actor's execution time sequence. We write φ_v to denote the period of an actor v. An actor with a period of n is said to have *n phases*.

An SDF graph is a CSDF graph where each sequence has a period of one. For an SDF graph, we therefore omit the parameter of τ, ρ^+ and ρ^-. Finally, a homogeneous SDF (HSDF) graph is an SDF graph in which each channel has a production and consumption rate of one.

2.1.1 Auto-Concurrency and Ordering of Firings

If sufficient tokens are available, a CSDF actor can start multiple firings simultaneously (note that the consumption (and production) of tokens from a channel is instantaneous). This is called *auto-concurrency* [21]. If each firing takes the same amount of time, as is the case for SDF graphs, then firings that have started simultaneously also complete simultaneously. Self-loops (i.e. cycles consisting of a single channel) are commonly used to explicitly limit the degree of auto-concurrency.

For CSDF actors, execution times vary cyclically, and thus a firing that has started later may be ready to produce its output tokens before earlier firings have completed. From a functional perspective, this means that the relationship between input and output values communicated over connected channels depends on the timing of the actors connecting the channels. This would break what is called *functional determinacy* [29]: if we regard the CSDF actors as mapping a stream of input values to a stream of output values, then this mapping should be independent of timing.

To ensure functional determinacy, several solutions are possible. The original semantics of CSDF, as presented in [6, 18], implicitly assumes that each actor has a self-loop, i.e. a channel with rates set to one and a single token. Such a self-loop prevents the actor to start multiple concurrent executions. This implicit assumption, however, unnecessarily limits the expressiveness of CSDF graphs. It breaks the taxonomy that is implied by the generalisation of production and consumption rate scalars to sequences, as offered by CSDF: any SDF graph that has auto-concurrent actors is, under the original definition of [6], not a CSDF graph.

Another solution is to relax the original CSDF restriction, by assuming an implicit self-loop only for those actors that have *differing* execution times [35, 42], which restores the taxonomy. In this chapter, we allow auto-concurrent actors to be included in a CSDF graph, regardless of their execution time sequences, and enforce functionally determinate execution by including dedicated constraints on the times at which actors may complete their firings, following [15]. These constraints essentially ensure that a firing delays the production of its output tokens until all firings that have started earlier have produced their output tokens (no such delay is necessary for SDF actors as all firings take the same amount of time).

2.1.2 Structural Invariants

The *repetition vector* of a CSDF graph (see [24]) is an example of a *structural invariant*. It is a vector that depends solely on the structure (topology and production/consumption rates) of the graph, not on the number of tokens in the graph. The repetition vector, which we shall, for the sake of consistency, denote q, is the smallest non-zero integer vector such that the following *balance equations* are satisfied [6, 24, 28]:

$$q_v \frac{\sigma_{vw}^{+}}{\varphi_v} = q_w \frac{\sigma_{vw}^{-}}{\varphi_w},$$

with the restriction that the repetition vector entry q_v of each actor is an integer multiple of its period φ_v [6]. A CSDF graph that admits a repetition vector is said to be *consistent*.

Vector q gives rise to the notion of a *graph iteration*: in a single graph iteration, actor v fires precisely q_v times. Note that for CSDF graph, the repetition vector entries are not necessarily relatively prime (for SDF graphs, they are).

As a dual to the repetition vector, a consistent CSDF graph has a second structural invariant, which is associated with *channels* rather than actors [16, 40]. This is the minimal integer vector s, with an entry for each channel, such that the following *flow conservation equations* hold for each pair of incoming and outgoing channels, uv and vw, of an actor v:

$$s_{uv} \frac{\sigma_{uv}^{-}}{\varphi_v} = s_{vw} \frac{\sigma_{vw}^{+}}{\varphi_v},$$

Vector s is sometimes named a P-flow [11, 40]. We refer to vector s as the graph's *normalization vector*, following the naming used in [5, 16].

The normalization vector associates every *channel* vw in the graph with a *weight* s_{vw}, such that, when multiplying the production and consumption rate of that channel, as well as its number of initial tokens, by s_{vw}, all the (production and consumption) rates associated with a single actor are invariant of the channel. As a consequence, the number of tokens on a cycle, weighted by the normalization vector, is invariant of the number of firings that each actor has completed [31, 40].

From the repetition and normalization vector, a *third* structural invariant can be derived [11, 16]. If we multiply the production and consumption rate of each channel by the associated entry in the normalization vector, then in a single graph iteration, the number of tokens produced onto (and consumed from) each channel is the same. This follows from the fact that through the normalization, firings effectively leave the number of tokens that reside on a cycle unchanged, plus the fact that in a single iteration, the number of tokens produced onto a channel equals the number of tokens consumed from it. We refer to this channel-invariant number of tokens as the *modulus* of the graph, denoted $\mathcal{N}$. In Sect. 4 we shall see how the modulus and

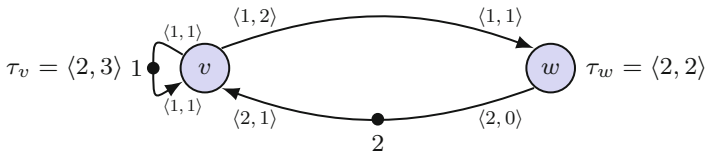


Fig. 2 An example of a consistent CSDF graph

normalization vector are used to compute strictly periodic schedules for SDF and CSDF graphs.

To illustrate the different structural invariants, consider the CSDF graph depicted in Fig. 2. Each of the two actors in the graph has a period of two: $\varphi_v = \varphi_w = 2$. The graph is consistent; its repetition vector is given by $q_v = 4$ and $q_w = 6$. In a single graph iteration, actor v completes *two*, and actor w *three* of its periods. This means that in a single iteration, v produces a total of six tokens onto channel vw, and four onto the self-loop vv. Furthermore, actor w consumes six tokens from channel vw, and produces six tokens onto wv. The smallest integer vector s that satisfies the flow conservation equations is given by $s_{vv} = 3$ and $s_{vw} = s_{wv} = 2$. If we apply these weights to their associated channels, then in a single iteration, on each channel *12* tokens are transferred from producer to consumer. As a result, the modulus of the graph is 12.

2.1.3 Self-timed Execution and Throughput

In a dataflow graph, execution is data-driven: an actor may fire as soon as it is enabled. In a *self-timed execution* of the graph, each actor fires as soon as it is enabled.

In a consistent SDF graph, a self-timed execution eventually settles in a repetitive pattern, called its *periodic phase* [20]. In the periodic phase (of a self-timed execution), each actor fires at a constant *average* rate. The ratio between the (constant) rates at which two different actors fire is given by their repetition vector entries.

The *throughput* of a consistent CSDF graph is equal to the average number of graph iterations completed per time unit, in a self-timed execution. Formally, if we let $t_v(k)$ denote the time at which an actor v completes its k-th firing, then the throughput Th of the graph satisfies:

$$Th = \lim_{k \to \infty} \frac{q_v k}{t_v(k)}.$$

In general, the self-timed execution of an arbitrary dataflow graph does not immediately enter the periodic phase [20]. The phase that precedes the periodic

phase is referred to as the *transient phase* [20, 25]. The transient phase may be very long; only very pessimistic bounds for the length of the transient phase exist [25].

Apart from reaching a periodic phase, there are two other cases to consider when simulating a self-timed execution of an SDF graph. First of all, the graph may *deadlock*: a token distribution is reached for which no actor is enabled. Second, the number of tokens on some channel in the graph may accumulate indefinitely. This can be due to either inconsistency, or to the fact that the graph is not *strongly connected* [21].

In the remainder of the chapter, we assume that CSDF graphs are both strongly connected and consistent. The throughput of a graph that is not strongly connected can be obtained by taking the minimum of the throughputs of each of the graph's strongly connected components [21].

2.2 Max-plus Algebra

The algebraic properties of the operators max and + resemble those of (respectively) addition and multiplication in conventional algebra [12]. This allows techniques that are known for (the analysis of) systems in conventional algebra to be applied to systems in max-plus algebra. To emphasise these similarities, operations $\otimes$ and $\oplus$ are commonly defined as follows, where $\otimes$ has priority over $\oplus$:

$$x \oplus y = \max(x, y), \tag{1}$$

$$x \otimes y = x + y. \tag{2}$$

The properties of these operators are similar to their counterparts, + and $\times$ in conventional algebra: they are associative and commutative, and $\otimes$ distributes over $\oplus$ [12, 25]. The respective max-plus counterparts of *one* and *zero*, which, in conventional algebra, are the unit elements of multiplication and addition, are 0 and $-\infty$:

$$x \oplus -\infty = \max(x, -\infty) = x,$$

$$x \otimes 0 = x + 0 = x.$$

Following convention, we denote $-\infty$ by ε.

The operators max and + generalise naturally to vectors and matrices. The element in row i and column j of matrix A is denoted a_{ij}, or, alternatively, $[A]_{ij}$. We denote the sum C of two matrices A and B by $C = A \oplus B$, with C defined as:

$$c_{ij} = a_{ij} \oplus b_{ij}. \tag{3}$$

The product of matrices A and B is defined as

$$[A \otimes B]_{ij} = \bigoplus_{k=1}^{m} a_{ik} \otimes b_{kj}, \tag{4}$$

where $\bigoplus$ denotes max-plus summation, analogous to the summation symbol Σ in conventional algebra.

Finally, the k-th power of a square matrix A is denoted $A^{\otimes k}$, and is recursively defined for $k \in \mathbb{N}$ by:

$$A^{\otimes 0} = E(n) \tag{5}$$

$$A^{\otimes k} = A \otimes A^{\otimes k-1}, \tag{6}$$

where E is the zero matrix, i.e., the matrix where every element equals ε.

A relevant and well-known equation in conventional algebra is the following:

$$Av = \lambda v.$$

As we shall see later, the notion of *self-timed schedule*, which is a key notion in performance analysis, can be regarded as the max-plus counterpart of eigenvalues and eigenvectors in conventional algebra.

3 Maximum Cycle Ratio Analysis

The throughput of a (C)SDF graph is defined as the average number of graph iterations completed per time unit. Computation of the throughput of a CSDF graph is analogous to computing its inverse—the average time per completed graph iteration, *iteration bound* [26] or *cycle time*. For HSDF graphs, work on computing this cycle time dates back to as early as 1968, where the term *maximum cycle ratio* was introduced in a seminal paper by Raymond Reiter [36]. The cycle ratio of a cycle in an HSDF graph (Reiter referred to the particular graphs as *computation graphs*) is the ratio between the total execution time associated with the cycle, and the number of tokens in the cycle. Formally:

$$\lambda(C) = \frac{\sum_{vw \in C} \tau_v}{\sum_{e \in C} \delta_e}. \tag{7}$$

The maximum cycle ratio is the maximum of all cycle ratios, taken over all simple cycles in the graph. Reiter did not give an efficient algorithm to compute the maximum cycle ratio of a graph, but several algorithms were developed later. For an overview, see [13].

In this section, we characterize the maximum cycle ratio both graphically and algebraically. We limit the scope of the analysis to HSDF graphs, and return to the analysis of SDF and CSDF graphs later.

The constraints imposed by HSDF channels on firing times can be represented mathematically using max-plus algebra. As we shall see, the maximum cycle ratio of the HSDF graph is equivalent to the *eigenvalue* of the corresponding max-plus system. Furthermore, the *eigenvector* of the max-plus system corresponds to a strictly periodic self-timed schedule of the HSDF graph. This mathematical characterization helps identifying the differences in the various approaches to throughput analysis, which we discuss later in Sect. 6.

3.1 Max-plus Characterization

The structure of an HSDF graph imposes an ordering on the times at which actors can fire: the time at which an actor may start (or complete) a certain firing depends on the times at which upstream actors complete their firings. Each channel vw in an HSDF graph thus imposes a *constraint* on the firing times of w, expressed in the firing times of v. If we let $t_v(k)$ denote the time at which an actor v starts its k-th firing, then these constraints have the form:

$$t_w(k) \geq t_v(k - \delta_{vw}) + \tau_v(k - \delta_{vw}). \tag{8}$$

In words, this states that actor w can not *start* its k-th firing before actor v has *completed* (at least) $k - \delta_{vc}$ firings, which occurs τ_v time units after v has *started* the last of these firings.

An actor w has a constraint of the above form for every *incoming* channel. The time that w can start its k-th firing must satisfy each of these constraints, which is captured by taking the *maximum* over all right-hand-sides of (8). Using max-plus algebra, this is expressed by:

$$t_w(k) \geq \bigoplus_{vw \in G} t_v(k - \delta_{vw}) \otimes \tau_v(k - \delta_{vw}). \tag{9}$$

As such, an HSDF graph can be represented as a set of constraints, one for each actor, formulated in max-plus algebra. If we assume that actors fire as soon as they are enabled, then the inequality in (8) is replaced by an equality sign. Using matrices and vectors for the sake of a compact notation, the set of constraints of an HSDF graph can be expressed as the following *system* of recurrences:

$$t(k) = \bigoplus_{m \in \mathbb{Z}} A_m \otimes t(k - m). \tag{10}$$

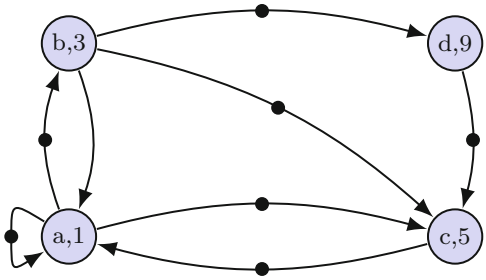


Fig. 3 An example HSDF graph

Here $t(k)$ is a *vector* of the k-th actor firing (start) times, with an entry for each actor in the graph. Matrices A_m can be regarded as *distance matrices* restricted to those edges that have precisely m tokens: entry (j, i) in matrix A_m corresponds to the presence of an edge ij having m tokens. Matrices A_m are square matrices, with number of rows and columns equal to the number of actors in the HSDF graph. Note that the above definition allows for channels to have a *negative* number of tokens.

As an example, consider the HSDF graph shown in Fig. 3. There are four actors in the graph, so the corresponding max-plus system consists of four recurrent equations:

$$\begin{aligned} t_a(k) &= t_a(k-1) \otimes 1 \oplus t_b(k) \otimes 3 \oplus t_c(k-1) \otimes 5, \\ t_b(k) &= t_a(k-1) \otimes 1, \\ t_c(k) &= t_a(k-1) \otimes 1 \oplus t_b(k-1) \otimes 3 \oplus t_d(k-1) \otimes 9, \\ t_d(k) &= t_b(k-1) \otimes 3. \end{aligned} \tag{11}$$

For a non-deadlocked HSDF graph, the max-plus system can be written as a so-called *first-order* recurrence relation [12, 25]. In a first-order recurrence relation, the dependencies between *consecutive* firings is captured in a single matrix. This means that the k-th firing times can be computed from the $(k-1)$-th firing times through a max-plus algebraic matrix multiplication. System (11), for example, can be written as:

$$t(k) = \begin{bmatrix} 4 & \varepsilon & 5 & \varepsilon \\ 1 & \varepsilon & \varepsilon & \varepsilon \\ 1 & 3 & \varepsilon & 9 \\ \varepsilon & 3 & \varepsilon & \varepsilon \end{bmatrix} \otimes t(k-1) \tag{12}$$

The max-plus system (11) thus gives the times at which the k-th firing of each HSDF actor takes place, given the previous few actor firing times, and firing each actor as soon as it is enabled. By iteratively evaluating the recurrence (11), or applying (12), starting with an arbitrary vector $t(1)$ (we choose to let the first firing be labelled one, not zero), we can thus simulate a self-timed execution. If, for example, we choose to let the first firings of the three actors take place at $t_a(1) = 0$, and $t_b(1) = t_c(1) = 1$ and $t_d(1) = 2$, then the self-timed schedule is:

$$t(1)=\begin{bmatrix}0\\1\\1\\2\end{bmatrix}\ t(2)=\begin{bmatrix}6\\1\\11\\4\end{bmatrix}\ t(3)=\begin{bmatrix}16\\7\\13\\4\end{bmatrix}\ t(4)=\begin{bmatrix}20\\17\\17\\10\end{bmatrix}\ t(5)=\begin{bmatrix}24\\21\\21\\20\end{bmatrix}\ t(6)=\begin{bmatrix}28\\25\\29\\24\end{bmatrix}\ t(7)=\begin{bmatrix}34\\29\\33\\28\end{bmatrix}$$

The process of repeatedly compute the next firing times from the previous firing times is the algebraic analogue of a self-timed execution [12, 21, 25]. After a finite number of firings, any self-timed execution will enter a periodic phase [12, 21, 25]. In the periodic phase, each actor fires in an N-periodic pattern, i.e., $t(k) = t(k - N) \otimes c$ for some $N \in \mathbb{N}$ and $c \in \mathbb{R}$. Since the throughput of an HSDF graph is defined as the (average) number of firings in the periodic phase of a self-timed execution, the periodic phase thus implicitly gives the graph's throughput.

Every strongly connected HSDF graph admits a *strictly periodic* schedule [25, 32]. That is, a schedule for which the time between two *consecutive* firings of an actor is constant. Given the analogy between matrix-vector multiplication in max-plus algebra on the one hand, and performing a self-timed execution of an HSDF graph on the other hand, the problem of finding a strictly periodic schedule can thus be formulated as the problem of finding an eigenvalue λ and eigenvector v:

$$A \otimes v = \lambda \otimes v.$$

The eigenvalue λ of matrix A is equal to the maximum cycle ratio of the corresponding HSDF graph [25]. The eigenvector, v, corresponds to the (initial) firing times from which actors immediately enter a strictly periodic schedule. Various algorithms for computing λ are available, of which the following section presents three.

3.2 Computing the Maximum Cycle Ratio

There are various methods for computing the maximum cycle ratio. An excellent overview and comparison is given in [13]. In this section, we describe three methods that often appear in literature.

3.2.1 The Power Method

The power method is a general method for computing the eigenvalue and eigenvector of a matrix. It consists of iteratively computing and normalizing vector $v_{k+1} = Av_k$, until v_{k+1} converges to an eigenvector of A. The method generalizes naturally to max-plus algebra, and as such can be used to compute the maximum cycle ratio of an HSDF graph [25].

In fact, the method was demonstrated in the evolution of the self-timed schedule for the example graph of Fig. 3. We repeatedly compute vectors $t(1), t(2), \ldots$ using the recurrence (10), until we have:

$$t(k+c) = A^{\otimes c} \otimes t(k) = t(k) \otimes d.$$

The eigenvalue is then given by $\lambda = \frac{d}{c}$ [12, 25]. Applying this to the example HSDF graph, continuing the vector sequence following from the initial vector $t = [0, 1, 1, 2]^T$ that we explored earlier, we obtain the sequence:

$$t(4)=\begin{bmatrix}20\\17\\17\\10\end{bmatrix}\ t(5)=\begin{bmatrix}24\\21\\21\\20\end{bmatrix}\ t(6)=\begin{bmatrix}28\\25\\29\\24\end{bmatrix}\ t(7)=\begin{bmatrix}34\\29\\33\\28\end{bmatrix}\ t(8)=\begin{bmatrix}38\\35\\37\\32\end{bmatrix}\ t(9)=\begin{bmatrix}42\\39\\41\\38\end{bmatrix}\ t(10)=\begin{bmatrix}46\\43\\47\\42\end{bmatrix}$$

The ninth firings take place 18 time units after the fifth firings, and the tenth firings take place 18 time units after the sixth firings. This means that we have found a 4-periodic solution:

$$A^{\otimes 4} \otimes t(5) = 18 \otimes t(5).$$

The eigenvalue is thus equal to $18/4 = 4\,1/2$, which is equal to the maximum cycle ratio of the graph of Fig. 3, attained by cycle $abdca$. The period that we have found by exploring the self-timed schedule is not *strictly* periodic: the times between two consecutive firings of the same actor varies between four and six time units. An eigenvector can be obtained from the periodic phase by taking the max-plus sum of the time-shifted vectors that make up the periodic phase [25]:

$$v = \bigoplus_{j=0}^{c-1} \lambda^{\otimes j} \otimes t(k+c-j+1). \tag{13}$$

To obtain a strictly periodic schedule for the HSDF graph of Fig. 3, we thus compute the max-plus sum of the terms $t(8)$, $t(7) \otimes \lambda$, $t(6) \otimes \lambda^{\otimes 2}$, and $t(5) \otimes \lambda^{\otimes 3}$:

$$v = \begin{bmatrix}37\frac{1}{2}\\34\frac{1}{2}\\34\frac{1}{2}\\33\frac{1}{2}\end{bmatrix} \oplus \begin{bmatrix}37\\34\\38\\33\end{bmatrix} \oplus \begin{bmatrix}38\frac{1}{2}\\33\frac{1}{2}\\37\frac{1}{2}\\32\frac{1}{2}\end{bmatrix} \oplus \begin{bmatrix}38\\35\\37\\32\end{bmatrix} = \begin{bmatrix}38\frac{1}{2}\\35\\38\\33\frac{1}{2}\end{bmatrix}.$$

Since any scalar multiple of an eigenvector is again an eigenvector, adding (or subtracting) a constant from each of the eigenvector's entries (both of which is a multiplication in max-plus algebra) yields another eigenvector. In other words, we can subtract $33\,1/2$ from each entry of the above eigenvector v, and obtain a

schedule in which all actors start their firings at a non-negative time instant. This final eigenvector is given by $v_a = 5$, $v_b = 1\frac{1}{2}$, $v_c = 4\frac{1}{2}$, and $v_d = 0$.

3.2.2 Policy Iteration

The *policy iteration* method, also known as *Howard's method* for computing the maximum cycle ratio of a marked graph is described in [10]. A more accessible version can be found in [25]. A *policy* corresponds to a *strictly periodic* schedule. That is, a schedule in which the k-th firing time of an actor v is given by $t_v(k) = t_v(1) + (k-1)\lambda$.

A policy thus assigns to every actor an initial firing time, and schedules the actors to fire in a strictly periodic fashion, every λ time units. Parameter λ is chosen to be the cycle ratio of an arbitrary cycle. This means that λ may be too optimistic: it is based on a *local* throughput, of a single cycle, but this throughput may not be admissible for the whole graph. If the schedule is not admissible, then there will be a violation of the constraints associated with the channels. If no such violation is detected, then the schedule is admissible and the maximum cycle ratio is given by the parameter λ [10, 25].

Policy iteration thus iteratively selects a cycle, and uses the cycle ratio of that cycle as its next guess for the firing period. Violation of constraints can be detected by solving a *single-source longest paths problem* using for instance the Bellman-Ford algorithm [3]. If the firing period is infeasible, then the graph has a cycle with positive weight, the cycle ratio of which is used as the next guess for the firing period. As such, the algorithm is guaranteed to find the cycle that attains maximum cycle ratio. It may, however, examine *many* simple cycles before it terminates; the number of simple cycles in a graph is, in the worst-case, exponential in the size of the graph.

An application of Howard's policy iteration algorithm to the example graph of Fig. 3 is shown in Fig. 4a, b. In these graphs, the weights associated with each edge is the *minimum time* that must separate the start times of the source and sink of that edge. Note that for edges with tokens, these weights change if the parameter λ changes.

If we initially choose cycle aca as a first guess of the critical cycle, then we find a schedule in which actors must fire every $6/2 = 3$ time units. This means that, for instance, actor b starts its k-th firing 2 time units before actor a has started its k-th firing, and that actor c and d start their k-th firings simultaneously, which violates the constraint $t_c(k) \geq t_d(k-1) + 9$. The schedule is thus found to be inadmissible, and we next substitute channel dc for ac to find a policy based on cycle $abdca$, shown in Fig. 4b.

The schedule based on cycle $abdca$ is admissible, and thus this cycle attains maximum cycle ratio, which is $18/4$. Note that the initial firing times found (indicated in the figure) are identical to those found in the previous section by the power algorithm.

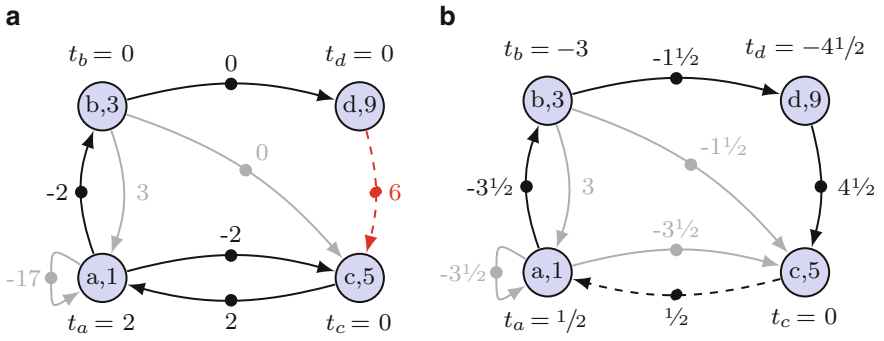


Fig. 4 (**a**) Infeasible policy based on cycle aca with $\lambda = 3$. (**b**) Feasible policy based on cycle $abdca$ with $\lambda = 4½$

3.2.3 Parametric Paths

A third approach to computing the maximum cycle ratio is similar to the policy iteration discussed earlier, in the sense that it too regards the problem as finding a feasible strictly periodic schedule. Of the three methods discussed in this section, this method is the only one that is *strongly polynomial*. That is, the number of steps necessary to compute the maximum cycle ratio is bounded by a polynomial in the size (number of nodes and edges) of the graph.

Similar to the policy iteration approach, the approach uses a weighted graph to represent the scheduling constraints. An edge e in the graph now has an associated weight *function*, y_e, which maps cycle time parameter λ to a weight: $y_e(\lambda) = w_e - \delta_e\lambda$. The (global) parameter λ is admissible if longest paths are well-defined, i.e., if the graph has no cycle of positive weight. If every cycle has a positive non-zero number of tokens, then an admissible parameter is bound to exist: one can choose λ such that every weight function maps to a non-positive value.

The key difference with the policy iteration approach lies in the update of the parameter λ. In contrast with policy iteration, where the subsequently used value for the parameter is based on an arbitrary positive cycle in the graph, this approach employs a parametric longest paths tree to bound the maximum number of parameter changes.

Once an admissible value for λ is found, the algorithm proceeds to iteratively decrease λ, changing the parametric weights of all edges. At each step, the algorithm applies the smallest decrease of λ, and updates the longest parametric paths tree.

An illustration of this approach to finding the maximum cycle ratio of the graph of Fig. 3 is shown in Fig. 5a–c. A feasible initial value for the parameter λ can be obtained by taking the sum of the execution times of all actors in the graph, which is 18. Any cycle (with at least one token) will have a cycle ratio that is at most this initial value. Choosing $\lambda = 18$ results in the longest paths tree, rooted in actor a, shown in Fig. 5a.

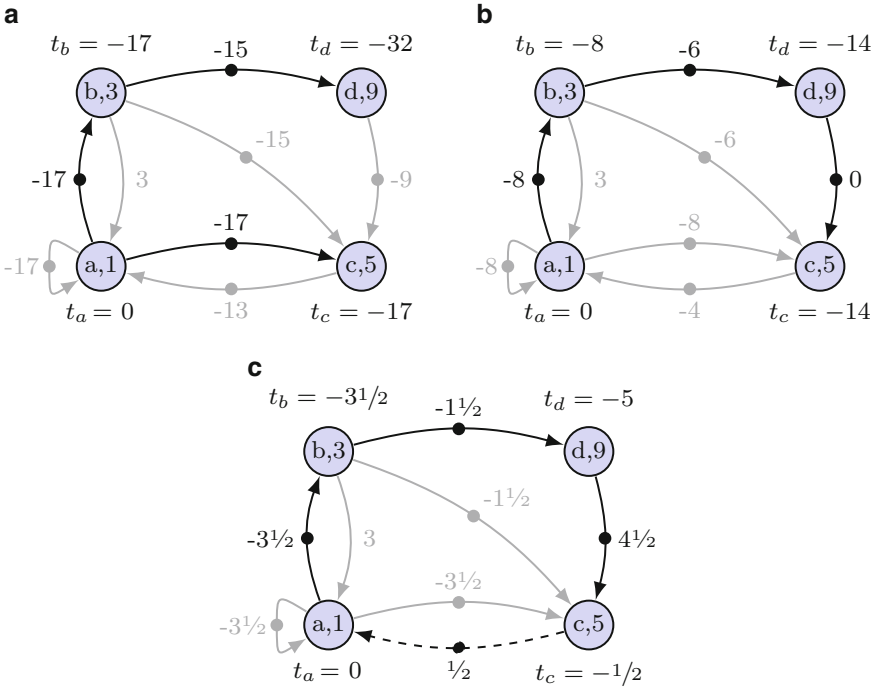


Fig. 5 (**a**) Longest paths tree, with indicated edge weights, rooted in a for $\lambda = 18$. (**b**) Longest paths tree for $\lambda = 9$. (**c**) Longest paths tree for $\lambda = 4\frac{1}{2}$

The longest paths tree does not change for $\lambda > 9$. For $\lambda < 9$, path bdc is longer than path bc, so the tree is updated into the one shown in Fig. 5b. Decreasing λ further will require a next change in the longest paths tree at $\lambda = 4\frac{1}{2}$, at which point a longer distance to actor a is found, through path $abdca$. Since this path is a cycle, its cycle ratio is the maximum cycle ratio of the graph, equal to $4\frac{1}{2}$. Similar to the policy iteration approach, the longest distances indicated in the figure denote the initial firing times of a strictly periodic schedule.

There are different ways to implement this approach. An efficient implementation maintains the different values of λ for which the parametric paths tree changes *locally*, and updates these values as the parameter changes. These so-called *keys* can be kept either for every edge or for every node. The algorithm of Karp and Orlin maintains edge keys and uses a binary heap to keep the keys ordered, resulting in an algorithm with a worst-case complexity of $O(nm \log(n))$ [27], on a graph with n nodes and m edges. In the approach described by Young, Tarjan and Orlin in [43], node keys are maintained in a Fibonacci heap, yielding a complexity of $O(nm + n^2 \log(n))$ [43]. Implementations of these two algorithms, as well as a comparison of their performance, can be found in [13].

3.3 Discussion

As we shall see in the following sections, more complex graphs such as SDF or CSDF graphs add another layer of complexity to throughput analysis. For these graphs, the fact that, for channels or actors, the rate at which tokens are produced differs from the rate at which they are consumed, implies that a strictly periodic schedule is not necessarily optimal [9]. Consequently, for these graphs the runtime of throughput analysis is not polynomially bounded in the size of the graph. However, the methods described in this section can be generalised to serve as a basis for these more complex graphs. In order to do so, several transformations can be applied to cyclo-static graphs to simplify their analysis. The next section describes these transformations in more detail.

4 Single-Rate Approximations

In the previous section we saw how an HSDF graph can be analysed for its throughput, by computing the graph's maximum cycle ratio. For more complex graphs such as SDF and CSDF, this analysis can not be applied right away. This is due to the fact that the constraints associated with channels in an HSDF graph do not translate to equally simple constraints for channels in an SDF or CSDF graph. In this section, we generalize the approach that we took in the previous section to SDF and CSDF graphs. That is, we again characterize the constraints imposed by channels on actor firing times in max-plus algebra. As we shall see, unlike HSDF channels, the constraints imposed by SDF channels on the k-th firing of an actor depends on k. It is however possible to perform series of approximations and transformation on these constraints, such that they again become independent on k. We describe two of these procedures, which effectively transform the CSDF graph into an HSDF graph. We refer to the obtained HSDF graph as *single-rate approximations*.

The transformations that we describe in this section is are *approximations*, which means that schedules that satisfy the constraints of the CSDF graph may not be valid for a single-rate approximation (and vice versa). An analysis of the *pessimistic* single-rate approximation yields a *strictly periodic* schedule that is guaranteed to be admissible for the approximated CSDF graph.

4.1 Characterization of CSDF Constraints

Each channel in a CSDF graph constrains the times at which the channel's consumer can fire, through the rate sequences and tokens associated with the channel. This constraint can be expressed as a relation between the *number* of producing and consuming firings, similar to the constraints that we derived for HSDF channels in

the previous section. Each producing and consuming firing changes the number of tokens on the channel. This can be expressed by letting $\Delta_{vw}(i, j)$ denote the number of tokens on vw after i producing and j consuming firings, as follows, where we allow the number of firings of an actor to be negative, for the sake of abstraction:

$$\Delta_{vw}(i, j) = \delta_{vw} - \sum_{l=i+1}^{0} \rho_{vw}^{+}(l) + \sum_{l=1}^{i} \rho_{vw}^{+}(l) + \sum_{l=j+1}^{0} \rho_{vw}^{-}(l) - \sum_{l=1}^{j} \rho_{vw}^{-}(l). \quad (14)$$

Here, we regard a *negative* number of firings as having an opposite effect with regard to token production or consumption. That is, a negative number of producing firings causes tokens to be "consumed" from (rather than produced onto) outgoing channels, and "produced" onto incoming channels.

We can now express the relation between producing and consuming firings in terms of the number of tokens that are on the channel. An actor can not start its next firing if there are insufficient tokens available on its incoming channels. Stated differently, an actor can complete its next firing if, after that firing, a non-negative number of tokens is left on each incoming channel. This is expressed formally as the following minimum[1]:

$$\pi_{vw}(k) = \min \{m \in \mathbb{Z} | \Delta_{vw}(m, k) \geq 0\}, \quad (15)$$

where $\pi_{vw}(k)$ gives the number of *producing* firings of actor v that must precede k consuming firings of actor w. We refer to function π_{vw} as the *predecessor function* associated with channel vw. Note that the predecessor function can equally be written as a *maximum* [16].

The predecessor function implicitly defines which schedules are feasible for a CSDF graph. If we let $t_w(k)$ denote the time at which actor w starts its k-th firing, then an admissible schedule must satisfy the following constraints for each channel vw in the graph:

$$t_w(k) \geq t_v(\pi_{vw}(k)) + \tau_v(\pi_{vw}(k)). \quad (16)$$

An obvious property of π is that it is non-decreasing, i.e. if $k \geq m$, then $\pi_{vw}(k) \geq \pi_{vw}(m)$. This is a result of the fact that firing an actor never results in the removal of tokens from its outgoing channels. Also, the average *slope* of π_{vw} follows from the fact that every q_v producing firings must be matched by q_w consuming firings. Hence, the slope of π_{vw} is equal to $\frac{q_v}{q_w}$.

The predecessor function can attain negative values: if there are many initial tokens on a channel, then it may be the case that producing firings can be "reversed" without disabling the first consuming firing. For example, consider the predecessor function depicted in Fig. 6b, associated with the CSDF channel ab of Fig. 6a. If there

[1]The notation $\min\{k \in S | P(k)\}$ denotes the minimum value of the set S that satisfy the predicate P.

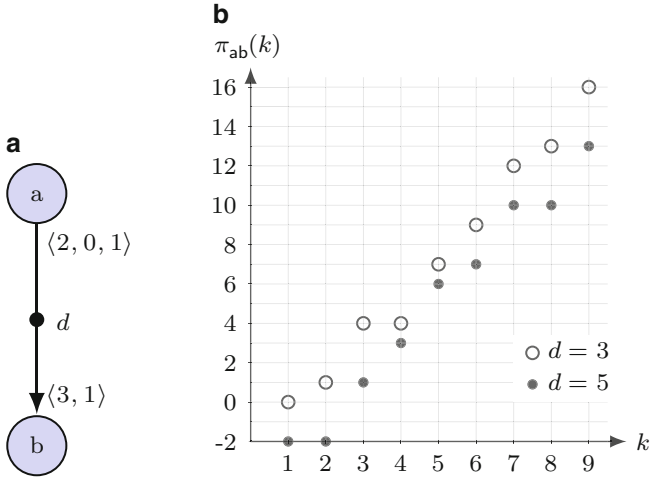


Fig. 6 (**a**) Channel. (**b**) Function

are three tokens initially present on channel ab, then no producing firings are needed to start the first firing of actor b. We thus have (for $d = 3$) $\pi_{ab}(1) = 0$. In case there are five initial tokens, then two firings (corresponding to the second and third phase) of a can be undone, reducing the number of tokens on ab, by one, from five to four. These four tokens are still sufficient to start the first two firings of a. Hence, for $d = 5, \pi_{ab}(1) = -2$.

4.2 Transforming the CSDF Constraints

The predecessor function $\pi_{vw}(k)$ gives the minimum number of producing firings of v that enable the k-th consuming firing of w. One may safely replace $\pi_{vw}(k)$ in (16) with an *upper bound* $\hat{\pi}_{vw}(k)$ for which $\hat{\pi}_{vw}(k) \geq \pi_{vw}(k)$ for all k. Such an upper bound is a *safe* in that it may overestimate but never underestimate the number of producing firings necessary to enable a given firing. By choosing $\hat{\pi}_{vw}(k)$ to have the form $\hat{\pi}_{vw}(k) = k - c$, which is precisely the predecessor function of an HSDF channel with c tokens, each CSDF channel can be replaced by an HSDF channel, transforming the CSDF graph into an *approximating* HSDF graph.

The key problem with the above approach is that the *slopes* of $\pi_{vw}(k)$ and the desired upper bound $\hat{\pi}_{vw}(k)$ are different, which means that $\hat{\pi}_{vw}(k) = k - c$ can not be an upper bound of $\pi_{vw}(k)$. In this section, we shall see how one can overcome this problem through a sequence of scaling steps and obtain an approximating HSDF graph, the admissible schedules of which map to admissible schedules for the CSDF graph.

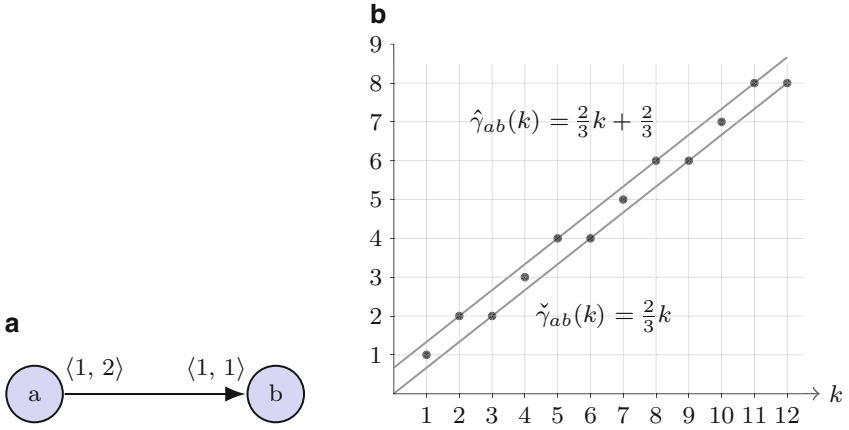


Fig. 7 (**a**) Example CSDF channel. (**b**) Linear bounds on the predecessor function of (**a**)

A tight upper *linear* bound $\hat{\gamma}_{vw}$ on the predecessor function π_{vw} is constructed by choosing an appropriate slope and intercept. The slope must be equal to the (average) slope of π_{vw}, and the intercept of $\hat{\gamma}_{vw}$ must be such that the minimum error between the predecessor function and $\hat{\gamma}_{vw}$ is zero. This is achieved by the following upper linear bound $\hat{\gamma}$:

$$\hat{\gamma}_{vw}(k) = \frac{q_v}{q_w}k - \min_{m\in\mathbb{Z}}\left\{\frac{q_v}{q_w}m - \pi_{vw}(m)\right\}. \tag{17}$$

Note that analogously, a *lower* linear bound $\check{\gamma}_{vw}$ is obtained by replacing the min operator by max. A predecessor function and its linear bounds is shown in Fig. 7.

4.2.1 Changing Counting Units

The linear bound of (17) does not have the desired form $\hat{\pi}_{vw}(k) = k - c$. We obtain this form by an appropriate *scaling* of both the domain and co-domain[2] of $\hat{\gamma}$. Downscaling the domain of $\hat{\gamma}_{vw}$ by a factor of q_w, and the co-domain by a factor q_v, gives

$$\frac{1}{q_v}\hat{\gamma}_{vw}\left(kq_w\right) = k - \min_{m\in\mathbb{Z}}\left\{\frac{m}{q_w} - \frac{\pi_{vw}(m)}{q_v}\right\}. \tag{18}$$

[2]Scaling the domain of a function $f(k)$ by a factor s gives the function $g(k) = f\left(\frac{k}{s}\right)$. Scaling the co-domain of $f(k)$ by s gives the function $h(k) = sf(k)$.

These two transformed functions are *translations*, but not integer: they map an integer to a *rational*, as the intercept (the term $\min\{\ldots\}$) is not integer. As such, they can not be translated to HSDF predecessor functions, as this implies a non-integral number of tokens. Since q_v and q_w both divide the modulus $\mathcal{N}$ of the graph, scaling the intercept by $\mathcal{N}$ results in the desired upper bound:

$$\hat{\pi}_{vw}(k) = \frac{\mathcal{N}}{q_v}\hat{\gamma}_{vw}\left(k\frac{q_w}{\mathcal{N}}\right) = k - \mathcal{N}\min_{m\in\mathbb{Z}}\left\{\frac{m}{q_w} - \frac{\pi_{vw}(m)}{q_v}\right\}. \tag{19}$$

Every CSDF channel can now be transformed into an approximating HSDF channel, by using the fact that predecessor function $\hat{\pi}_{vw}(k) = k - c$ corresponds to an HSDF channel with $\delta_{vw} = c$. Combined with a straightforward approximation of a CSDF actor v by an HSDF actor that has as execution time the maximum of the execution time sequence of v, a CSDF graph can thus be approximated by an HSDF graph.

Analogous to the derivation of a conservative (or pessimistic) approximation of a CSDF graph, one may apply the method outlined above to derive a transformed *lower linear bound* on predecessor functions, resulting in an *optimistic* approximation. The above approximation of a CSDF graph by two HSDF graphs is formalized in Algorithm 1. Through appropriate application of modular arithmetic, the intercept of the HSDF predecessor function $\hat{\gamma}_{vw}$ in (19) can be computed in time proportional to the product of φ_v and φ_w (see lines 11 and 12 in Algorithm 1).

Algorithm 1 Transforms a CSDF graph into optimistic and pessimistic single-rate approximations

input : A consistent CSDF graph $\mathcal{G}$, with flow normalisation vector s.
output: An optimistic and pessimistic single-rate approximation of $\mathcal{G}$.

1 $\mathcal{H}_{\text{opt}} \longleftarrow$ emptygraph
2 $\mathcal{H}_{\text{pess}} \longleftarrow$ emptygraph
3 **foreach** *actor* v *in* $\mathcal{G}$ **do**
4 Add actor $\hat{v}$ to $\mathcal{H}_{\text{pess}}$
5 Add actor $\check{v}$ to $\mathcal{H}_{\text{opt}}$
6 $\tau_{\hat{v}} \longleftarrow \max_{i=1}^{\varphi_v} \tau_v(i)$
7 $\tau_{\check{v}} \longleftarrow \min_{i=1}^{\varphi_v} \tau_v(i)$
8 **foreach** *channel* vw *in* $\mathcal{G}$ **do**
9 Add HSDF channel $\hat{v}\hat{w}$ to $\mathcal{H}_{\text{pess}}$
10 Add HSDF channel $\check{v}\check{w}$ to $\mathcal{H}_{\text{opt}}$
11 $g_{vw} \leftarrow \gcd(\sigma^+_{vw}, \sigma^-_{vw})$
12 $\delta_{\hat{v}\hat{w}} \longleftarrow s_{vw} \min_{\substack{i<\varphi_v \\ j<\varphi_w}} \left\{ g_{vw} \left\lceil \frac{\Delta_{vw}(i,j)+1}{g_{vw}} \right\rceil - \frac{(i+1)\sigma^+_{vw}}{\varphi_v} + \frac{j\sigma^-_{vw}}{\varphi_w} \right\}$
13 $\delta_{\check{v}\check{w}} \longleftarrow s_{vw} \max_{\substack{i<\varphi_v \\ j<\varphi_w}} \left\{ g_{vw} \left\lfloor \frac{\Delta_{vw}(i,j)}{g_{vw}} \right\rfloor - \frac{i\sigma^+_{vw}}{\varphi_v} + \frac{j\sigma^-_{vw}}{\varphi_w} \right\}$
14 **return** $\mathcal{H}_{\text{opt}}, \mathcal{H}_{\text{pess}}$

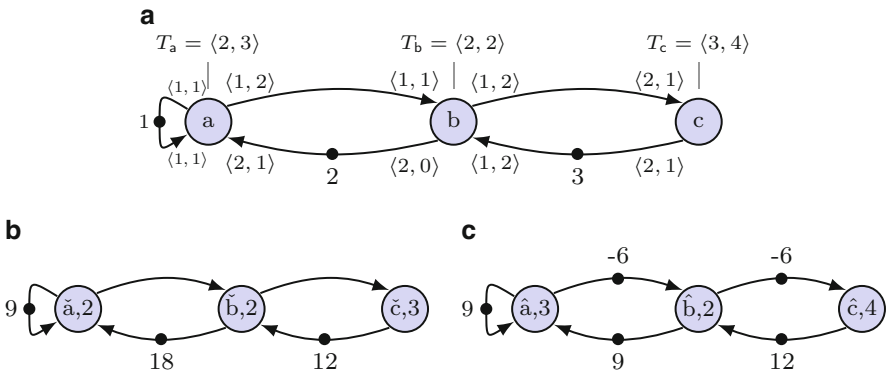


Fig. 8 (**a**) Example CSDF graph. (**b**) Optimistic single-rate approximation of the CSDF graph of (**a**), using Algorithm 1. (**c**) Pessimistic single-rate approximation of the CSDF graph of (**a**), using Algorithm 1

As an example, Fig. 8b, c show the two HSDF graphs obtained by applying Algorithm 1 to the CSDF graph of Fig. 8a. A consequence of the (pessimistic) approximation is that the number of tokens placed on an HSDF channel may be negative. This indicates that a producer must first complete several firings before the first firing of the corresponding consumer is enabled. For example, actor $\hat{a}$ in Fig. 8c must fire seven times in order to enable the first firing of actor $\hat{b}$. The following section provides more details on the precise relation between firings in the CSDF graph and its approximations.

4.3 Computing Strictly Periodic Schedules

The scaling that we performed above, using the modulus as scaling factor, is analogous to multiplying the number of tokens, on each channel in an HSDF graph, with the same factor. This means that the cycle ratio of each cycle (and thus the *maximum cycle ratio* of the graph) is scaled *down* by a factor of $\mathcal{N}$. As we shall see below, the throughput computed from an HSDF graph that is constructed from the transformed predecessor functions must be scaled by $\mathcal{N}$, to obtain a bound on the throughput of the original CSDF graph.

The single-rate approximations of a CSDF graph are HSDF graphs, and, as such, have a strictly periodic schedule that is optimal. These HSDF schedules are related to admissible CSDF schedules, in the sense that every schedule that is admissible for the pessimistic approximation can be translated to one that is admissible for the CSDF graph. Likewise, every schedule that is admissible for the CSDF graph can be translated to one that is admissible for the *optimistic* approximation. This translation is essentially a *periodic sampling* of actor firings, with a different sampling period for each actor.

In this section, we describe this translation, from a schedule that is admissible for the pessimistic approximation, to one that is admissible for the CSDF graph. For this, we need a correspondence between the firings of actors in the CSDF graph and its single-rate approximations. This correspondence follows from the construction of the single-rate approximations. Recall that two scaling steps were performed in order to change the linear bounds on the predecessor function to the integer functions of (19). The first scaling step essentially changed the counting units from individual firings to completed iterations, and the second step changed the unit at which these iterations are counted, increasing the resolution by a factor equal to the modulus of the graph. As a result, the number of firings of an actor in the single-rate approximation that corresponds to a single firing, of the same actor in the approximated CSDF graph, is given by the ratio of the modulus of the graph and the repetition vector entry of that actor. This means that if the firing times of $\hat{v}$ are given by $t_{\hat{v}}(k)$, then firing times $s_v(k)$ are given by the mapping:

$$s_v(k) = t_{\hat{v}}\left(\frac{\mathcal{N}}{q_v}k\right). \tag{20}$$

In other words, the strictly periodic schedule of the (pessimistic) single-rate approximation is mapped to a strictly periodic schedule for the CSDF graph. In the latter, each actor fires in a strictly periodic fashion, but each actor has its own firing period. Such a strictly periodic schedule is generally not optimal; higher firing rates may be attained by deviating from the strictly periodic schedule. This means that the (maximum) rate at which actors can fire in the pessimistic approximation translates to a minimum throughput of the CSDF graph.

As an example, we shall compute a periodic schedule for the CSDF graph given in Fig. 8a. An analysis of the pessimistic single-rate approximation of the graph (shown in Fig. 8c) gives that its critical cycle is aba, which attains the maximum cycle ratio of $\frac{5}{3}$. The schedule for the single-rate approximation follows from assigning actor b (or c) a start time of $t = 0$, which leads to respective start times of actors a and c of $t = -13$ and $t = 12$. Shifting these start times 13 time units forward gives the following strictly periodic self-timed schedule:

$$t_a(k) = 0 + \frac{5}{3}(k-1),$$

$$t_b(k) = 13 + \frac{5}{3}(k-1),$$

$$t_c(k) = 25 + \frac{5}{3}(k-1).$$

In order to map the above schedule from the single-rate approximation to the CSDF graph, we must compute how many HSDF actor firings make up a single CSDF actor firing. The modulus $\mathcal{N}$ of the CSDF graph is 36, and its repetition vector is given by $q_a = 4$, and $q_b = q_c = 6$. To construct an admissible schedule from the

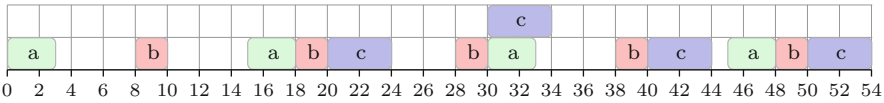


Fig. 9 A strictly periodic schedule for the CSDF graph of Fig. 8a, derived using maximum cycle ratio analysis of the single-rate approximation shown in Fig. 8c

schedule of the single-rate approximation, we must thus take every ninth firing of actor a, and every sixth firing of actors b and c. The CSDF schedule follows from the HSDF schedule using mapping (20):

$$s_a(k) = t_a(9k) = 13\frac{1}{3} + 15(k-1),$$

$$s_b(k) = t_b(6k) = 21\frac{1}{3} + 10(k-1),$$

$$s_c(k) = t_c(6k) = 33\frac{1}{3} + 10(k-1).$$

If we again shift the start times of each of the CSDF actors, backwards in time, such that actor a starts its first firing at $t = 0$, then a thus fires every 15 time units, at $t = 0, 15, 30, \ldots$. Actors b and c have the same firing period, which is ten, and start their first firing at respectively $t = 8$ and $t = 10$. The resulting schedule is depicted in Fig. 9. Note that each actor fires in a strictly periodic fashion, although the periods differ.

4.4 Discussion

This section provides a general view of several different existing approaches that all aim to achieve the same result. Shared by these existing approaches is their goal to simplify the throughput analysis of SDF and CSDF graphs, at the expense of a loss in accuracy. Here we briefly discuss different treatments of the same idea, and highlight their similarities and/or differences.

Rather than scheduling *actor firings*, one can also choose to describe the times at which individual *tokens* are produced onto and consumed from channels. In fact, for HSDF graphs, strictly periodic schedules for actor firings and token consumptions can be used interchangeably, as each actor firing corresponds to the consumption of a single token from each of its incoming channels. By assuming strictly periodic token productions or consumptions, as is done in [41, 42] and, more recently, [7], we obtain similar approximations as those shown earlier in this section. The quality of such a token-based approximation may differ from the quality of the actor-based approximation described here. For details, we refer the interested reader to [14].

The single-rate approximations constructed using Algorithm 1 are convenient in the sense that they are HSDF graphs. As such, they may be analysed by existing efficient algorithms such as those for computing maximum cycle ratio. Alternatively, the linear bounds on the predecessor function can be used to specify linear constraints between the firing times of actors [41, 42]. These linear constraints can be used to formulate the problem of computing a strictly periodic self-timed schedule as a linear program, which can be solved using a dedicated solver.

The process of transforming the predecessor function of a CSDF channel in a predecessor function of an approximating HSDF channel, through the construction of linear bounds and scaling, described in this section, is not the only way to obtain single-rate approximations. Approaches that are similar but use different strategies are described in [4, 5, 7, 8] and [9].

Finally, given the importance of periodic schedules in the analysis of hard real-time systems, several approaches to the construction of periodic schedules for acyclic SDF and CSDF graphs have been developed by the real-time systems community. For more details, we refer the interested reader to [1, 2, 23, 34].

For some live graphs, no strictly periodic schedule exists. That is, for these graphs the pessimistic single-rate approximation has one or more tokenless cycles, or cycles with a negative number of tokens. Consequently, the approximations do not have a finite maximum cycle ratio, and thus no strictly periodic schedule can be constructed. Characteristic for these graphs is that they are "close to deadlock". In the following section, we shall see how periodic (but not necessarily strictly periodic) schedules for these graphs can be constructed after applying an *unfolding* transformation to the graph.

5 Unfolding Actor Firings

In the previous section, we have seen how strictly periodic schedules can be computed for SDF and CSDF graphs. There are graphs, however, that are live but do not admit a strictly periodic schedule. This is best illustrated with an example. The SDF graph (consisting of a single cycle) depicted in Fig. 10a is live. This is easily verified by a short self-timed execution: the six tokens on channel ab are all consumed by actor b and again produced onto channel bc. Each actor subsequently consumes all six tokens on its sole incoming channel, completing as many firings as make up a single iteration in parallel. After actor d has taken its turn, a full graph iteration is completed and thus the graph is live. Clearly, actors do *not* fire in a strictly periodic fashion in this schedule, as some of their firings start in parallel.

Although the SDF graph is live, its (pessimistic) single-rate approximation given in Fig. 10b has an undefined throughput, as the total number of tokens in the cycle equals zero. Consequently, it is not possible to define a strictly periodic schedule for the graph. However, if we distinguish between odd and even firings of actor b by treating them as two groups of consecutive firings of respective actors b_1 and b_2, then we *can* construct a schedule that is strictly periodic, as illustrated in Fig. 10c.

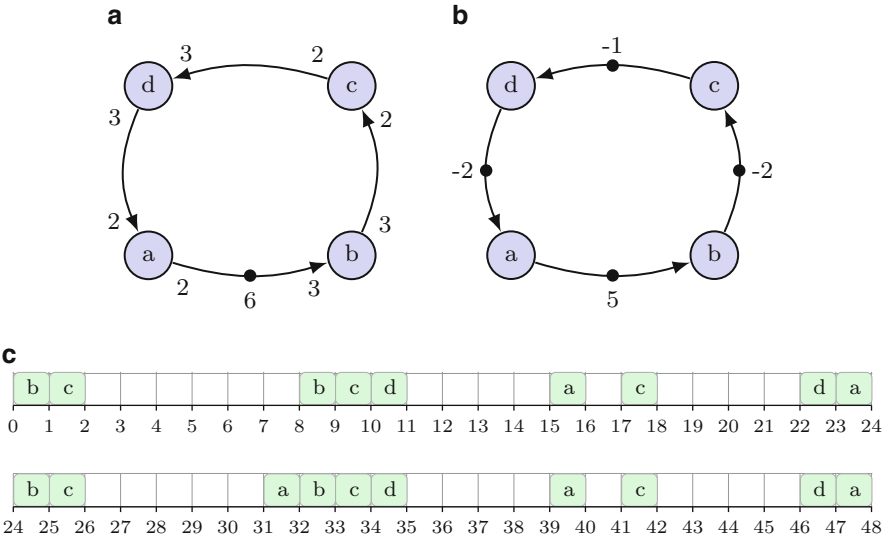


Fig. 10 (**a**) An example live SDF graph (**b**) The (pessimistic) single-rate approximation of (**a**). (**c**) Periodic schedule for the graph of (**a**): actors a, c and d fire strictly periodically, whereas actor b fires according to a 2-periodic schedule

In the depicted schedule, actors a, b, c and d start their first firing respectively at times 0, 1, 10, 15.

In this section, we show how the above example generalizes into a systematic approach to explicitly distinguish between consecutive firings of an actor, by *unfolding* these actors. This approach involves two steps, which combined transform a dataflow graph into a larger graph: a *relabelling* of actors, and *transformation* of channels.

5.1 Multi-Rate Equivalents

Similar to the transformation of an SDF graph into an equivalent HSDF graph, which is often referred to as the former's *single-rate equivalent*, the transformation of a CSDF graph into an equivalent SDF graph, which is called its *multi-rate equivalent*, involves the creation of a number of copies of each CSDF actor. Where in a single-rate equivalent the number of copies is equal to the number of firings that make up a single graph iteration, each CSDF actor is represented by as many copies in the multi-rate equivalent as it has *phases* [15].

In the example above, we constructed a schedule that is strictly periodic if we distinguish between the even and odd firings of actor b. If we consider even and odd firings of actor b as distinct *phases* then the graph of Fig. 10a changes into a CSDF

graph. The strictly periodic schedule then applies to the multi-rate equivalent of this CSDF graph.

Each (multi-rate) actor v_i in the multi-rate equivalent corresponds to phase i of actor v in the CSDF graph. The k-th firing of multi-rate actor v_i corresponds to firing $i + (k-1)\varphi_v$ of CSDF actor v. Note that this means that the first (i.e., $k = 1$) firing of v_i corresponds to the i-th firing of v. The execution time, τ_{v_i}, of multi-rate actor v_i, is a constant, equal to the execution time associated with the i-th phase of v: $\tau_{v_i} = \tau_v(i)$.

The relation between a multi-rate actor v_i and corresponding cyclo-static actor v can be used to transform the predecessor function associated with CSDF channels to the predecessor functions associated with multi-rate channels. Production and consumption rates readily follow from this transformation: if CSDF actor v fires q_v times in a single iteration of the CSDF graph, then actor v_i (i.e., the actor representing the i-th *phase* of the CSDF graph) fires q_v/φ_v times in a single iteration of the multi-rate equivalent. For the balance equations to hold, this means that the rate associated with each channel connected to v_i must be equal to the *sum* of a single period of the corresponding rate sequence of the CSDF channel connected to v.

If we consider the case where only a single CSDF actor v is unfolded, then the predecessor functions of incoming channels of v are related to those of incoming channels of the multi-rate actors v_i that represent the phases of v. For *incoming* channels of v_i, the predecessor function for a channel uv_i is obtained simply by transforming the *domain* of π_{uv}, using the mapping between firings of cyclo-static and multi-rate actors [15]:

$$\pi_{uv_i}(k) = \pi_{uv}\left(i + (k-1)\varphi_v\right). \tag{21}$$

For *outgoing* channels of multi-rate actors, the *co-domain* of the CSDF predecessor function needs to be transformed, which is slightly more involved. To understand this transformation, note that $\pi_{vw}(k)$ gives a lower bound on the number of firings of actor v that must have completed before the k-th firing of w can start. In the multi-rate equivalent, actor w has φ_v incoming channels $v_i w$, with each associated predecessor function implying a (possibly different) lower bound on the number of firings of v. The required number of firings of v that satisfies all these lower bounds is given by their maximum, which gives the following relation between π_{vw} and $\pi_{v_i w}$ [15]:

$$\pi_{vw}(k) = \max_i \left\{ i + (\pi_{v_i w}(k) - 1)\varphi_v \right\}. \tag{22}$$

If we expand both sides of the above equations using the definition of the predecessor function given earlier as (15), then the number of tokens on each incoming channel uv_i, and outgoing channel $v_i w$ can be solved for [15]:

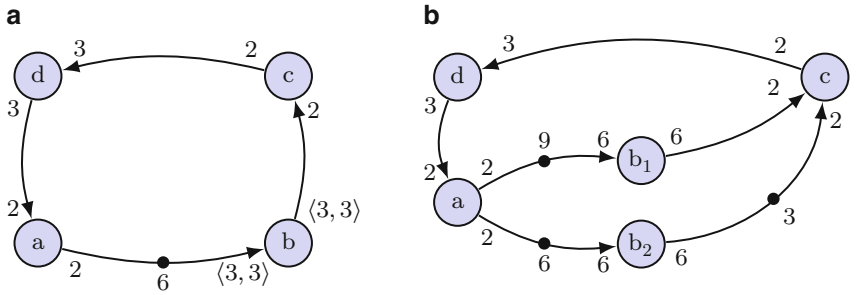


Fig. 11 (**a**) Graph of Fig. 10a, in which actor b has two distinct phases. (**b**) Multi-rate equivalent of (**b**)

$$\delta_{uv_i} = \delta_{uv} + \sum_{j=i+1}^{\varphi_v} \rho_{uv}^{-}(j) \qquad \delta_{v_i w} = \delta_{vw} + \sum_{j=1}^{i-1} \rho_{vw}^{+}(j).$$

Figure 11b depicts the result of applying the transformation to actor b in Fig. 10a. One obtains the schedule depicted earlier in Fig. 10c by computing a strictly periodic schedule, from the single-rate approximation of the multi-rate equivalent shown in Fig. 11b.

5.2 A General Transformation

When using the procedure outlined above to unfold each (C)SDF actor as many times as it fires in a single graph iteration (i.e., actor v is unfolded into q_v copies), then essentially an HSDF graph is obtained, as each multi-rate actor will have a repetition vector entry of one. In fact, a true HSDF graph (i.e., all rates are one) is obtained by a *rate-normalization* procedure, where the tokens and rates of a channel vw are divided by the greatest common divisor of ρ_{vw}^{+} and ρ_{vw}^{-} [17, 31]. In this view, the transformation of a CSDF graph into its multi-rate equivalent is a general transformation that may be used to transform a graph into its *single-rate equivalent* as a special case.

An important difference between this approach to constructing a single-rate equivalent on the one hand, and existing approaches such as described in [28], is the number of channels in the single-rate equivalent. The number of channels in the multi-rate equivalent of a CSDF graph depends on the number of phases of the CSDF actors: using the transformation outlined above, a CSDF channel vw is represented by $\varphi_v \varphi_w$ multi-rate channels. As such, the number of HSDF channels in a *single-rate* equivalent constructed in this way becomes very large, as each CSDF channel vw is represented by $q_v q_w$ HSDF channels. Several of these channels may however be identified as *redundant* in terms of the constraints that they impose [15, 38]. These redundant channels can safely be pruned from the graph,

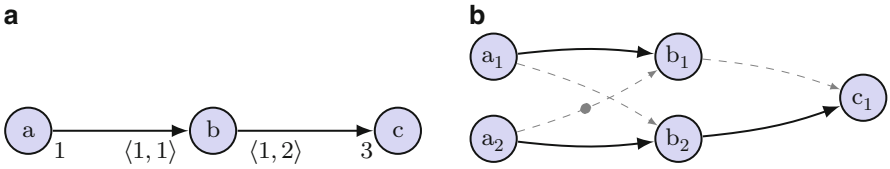


Fig. 12 (**a**) CSDF path. (**b**) Single-rate equivalent

without introducing more scheduling freedom into the graph. An example is given in Fig. 12b: the single-rate equivalent is obtained by unfolding every actor according to its repetition vector entry, after which each channel is transformed to an HSDF channel by applying the rate-normalization mentioned earlier. The dashed channels in the single-rate equivalent are redundant, which is reflected by the constraints they impose. For example, because b_2 starts its k-th firing no sooner than b_1 does, the completion of the k-th firing of actor b_2 is *sufficient* for actor c_1 to start its k-th firing, and thus channel b_1c_1 can safely be omitted. The same reasoning applies to the other dashed channels.

The identification of redundant HSDF channels can be generalised to SDF channels [14]. This gives a transformation, from CSDF, to a *sparser* multi-rate equivalent, which avoids creating redundant channels rather than pruning them. This transformation is listed as Algorithm 2.

Algorithm 2 Transforms a CSDF graph into its multi-rate equivalent, from which redundant channels are pruned

input : A CSDF graph $\mathcal{G}$.
output: An equivalent SDF graph $\mathcal{H}$ with $\sum_{v\in\mathcal{G}} \varphi_v$ actors.

$\mathcal{H} \longleftarrow$ emptygraph
foreach *actor* v *in* $\mathcal{G}$ **do**
  for $i = 1$ **to** φ_v **do**
    Add actor v_i to $\mathcal{H}$
    $\tau_{v_i} \longleftarrow \tau_v(i)$
foreach *channel* vw *in* $\mathcal{G}$ **do**
  for $j = 1$ **to** φ_w **do**
    for $i = 1$ **to** φ_v **do**
      $g_{vw} \leftarrow \gcd(\sigma^+_{vw}, \sigma^-_{vw})$
      $c_{\min} \longleftarrow \left\lfloor \frac{\Delta_{vw}(i,j)+\sigma^-_{vw}}{g_{vw}} \right\rfloor$
      $c_{\max} \longleftarrow \left\lfloor \frac{\Delta_{vw}(i-1,j)+\sigma^-_{vw}}{g_{vw}} \right\rfloor$
      if $c_{\min} > c_{\max}$ **then**
        Add channel $v_i w_j$ to $\mathcal{H}$
        $\rho^+_{v_i w_j} \longleftarrow \sigma^+_{vw}$
        $\rho^-_{v_i w_j} \longleftarrow \sigma^-_{vw}$
        $\delta_{v_i w_j} \longleftarrow \delta_{vw} + \sum_{l=1}^{i-1} \rho^+_{vw}(l) + \sum_{l=j+1}^{\varphi_w} \rho^-_{vw}(l)$
return $\mathcal{H}$

5.3 Discussion

The transformation of a CSDF graph into its multi-rate equivalent can be seen as an *unfolding* transformation, in which successive firings of an actor are explicitly represented by individual actors. An unfolding transformation for HSDF graphs is presented in [33], where several iterations of the HSDF graph are represented. The transformation of Algorithm 2 generalizes this transformation, such that it applies to CSDF graphs as well.

Note that the construction of a multi-rate equivalent does not require the targeted CSDF graph to be *consistent*. An inconsistent CSDF graph simply results in an inconsistent multi-rate equivalent. Also, consistent *subgraphs* can perfectly well be unfolded into their *single-rate* equivalent, leaving the remainder of the graph unchanged.

Finally, we remark that the notion of equivalence between a CSDF graph and its multi-rate equivalent is a different notion than discussed in [35] and [24], where an equivalent SDF actor is constructed from a CSDF actor by *fusing* together its phases into a single phase. This difference is best illustrated by the fact that the fusing operation can introduce deadlock, whereas the multi-rate equivalent of a CSDF graph can only be deadlocked if the CSDF graph is, too. In the fusing transformation, equivalence is viewed from a *functional* perspective: data that is processed *sequentially* by the different phases of a CSDF actor, is processed *monolithically* by the SDF actor. Although actors in the multi-rate equivalent of a CSDF graph do produce and consume as many tokens as their corresponding CSDF actors do in a single period, the extra tokens added to the channels ensure equivalence in terms of the schedules that both graphs admit.

6 Throughput Analysis

The previous two sections have provided the tools necessary to generalise the throughput analysis of HSDF graphs towards more complex graphs such as SDF and CSDF graphs. This section describes two approaches for the throughput analysis of CSDF graphs. The first approach is similar to the *power method* described earlier in Sect. 3. That is, we simulate a self-timed execution of the graph, and keep track of the firing times of the completed iterations, until we detect a periodic firing pattern. This method was first described in [20] and [22], and is referred to as *state-space exploration*.

The second approach is an iterative one. Starting from a strictly periodic schedule with sub-optimal throughput, we iteratively apply the unfolding transformation of Sect. 5 until we find a periodic schedule that is optimal. Every unfolding transformation allows for a greater scheduling complexity, but increases the achievable throughput monotonically as well.

Both methods have their advantages and disadvantages, which we shall explore and illustrate in Sect. 6.3.

6.1 State-Space Exploration

In Sect. 3 we described the power method for computing the eigenvalue and eigenvector of a max-plus matrix. When applied to an HSDF graph, the power method essentially performs a simulation of a self-timed execution: each actor fires as soon as it is enabled. After a finite number of firings, a *periodic phase* is entered, in which actors fire periodically (but not necessarily strictly periodically). The periodic phase is detected by examining the history of firings.

This approach can not be applied directly to a CSDF graph, since each actor fires at a different rate (the ratios of these rates are given by the graph's repetition vector). Consequently, the check for periodicity must treat graph iterations as a whole. One way to achieve this is by applying the power method to the CSDF graph's single-rate equivalent. However, the latter may be very large, prohibiting an efficient analysis.

Rather than representing each firing that occurs in a single iteration *explicitly* by means of constructing the single-rate equivalent, we may also choose for an *implicit* representation [20]. Such an implicit representation exploits the fact that the (initial) token distribution of the graph, together with all remaining firing times of active firings, determine the times at which subsequent actor firings take place. In this view, the token distribution plus, for each actor, the *multiset* (or *bag*) of time units that current firings still need to complete, together form the *state* of the graph [20].

With the above implicit representation of actor firing times in the form of a graph state, the state-space is explored by simulating a self-timed execution. Periodicity is now checked for by comparing states, which only needs to be performed once per iteration.

Similar to the power method, this method is bound to complete in a finite number of steps, having found either a periodic phase, or a deadlock. As an example, we compute the throughput of the SDF graph of Fig. 13 by exploring its state-space. Initially, only one actor is enabled: actor a can start a single firing, leaving three tokens on channel ba. The initial state for the exploration thus consists of the token distribution $\delta_{ba} = \delta_{cb} = 3$, and $\delta_{aa} = \delta_{ab} = \delta_{bc} = 0$, and the single active firing of a, with a remaining execution time of two time units.

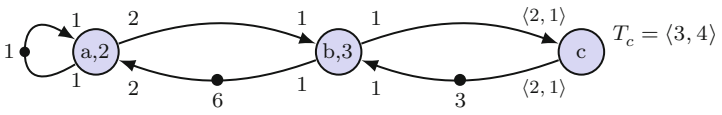


Fig. 13 Example CSDF graph, with repetition vector: $q_a = 3$, $q_b = 6$, and $q_c = 4$, normalization vector s with $s_{ab} = s_{ba} = s_{bc} = s_{cb} = 2$ and $s_{aa} = 4$. The modulus $\mathcal{N}$ of the graph equals 12

Table 1 State-space of a self-timed execution of the CSDF graph of Fig. 13

	Firings			Tokens					Firings			Tokens			
t	a	b	c	δ_{ab}	δ_{ba}	δ_{aa}	δ_{cb}	t	a	b	c	δ_{ab}	δ_{ba}	δ_{aa}	δ_{cb}
0	{2}	∅	∅	0	4	0	3	20	∅	{1, 3, 3}	∅	3	0	1	0
2	{2}	{3, 3}	∅	0	2	0	1	21	∅	{2, 2}	{4}	3	1	1	0
4	{2}	{1, 1, 3}	∅	1	0	0	0	23	{2}	∅	{2, 3}	3	1	0	0
5	{1}	{2}	{3}	1	2	0	0	25	∅	{3}	{1}	4	1	1	0
6	{2}	{1}	{2}	3	0	0	0	26	∅	{2, 3, 3}	∅	2	1	1	0
7	{1}	∅	{1, 4}	3	1	0	0	28	{2}	{1, 1}	{4}	2	0	0	0
8	∅	{3, 3}	{3}	3	1	1	0	29	{1}	∅	{3, 3}	2	2	0	0
11	{2}	{3}	{3}	2	1	0	0	30	{2}	∅	{2, 2}	4	0	0	0
13	∅	{1}	{1}	4	1	1	0	32	∅	{3, 3, 3}	∅	3	0	1	0
14	{2}	{3, 3}	{4}	2	0	0	0	35	{2}	∅	{4, 4}	3	1	0	0
16	∅	{1, 1}	{2}	4	0	1	0	37	∅	∅	{2, 2}	5	1	1	0
17	{2}	∅	{1, 3}	4	0	0	0	39	∅	{3, 3, 3}	∅	2	1	1	0
18	{1}	{3}	{2}	3	0	0	0	42	{2}	∅	{4, 4}	2	2	0	0
19	∅	{2}	{1}	5	0	1	0	44	{2}	∅	{2, 2}	4	0	0	0

After 21 steps, at $t = 30$, the state-space enters a periodic phase in which a single graph iteration is completed every 14 time units

During the self-timed execution of the graph, tokens produced onto channels bc are consumed immediately. As a result, all states in the exploration have $\delta_{bc} = 0$. The explored state space is listed in Table 1, where the column for δ_{bc} is omitted, given that $\delta_{bc} = 0$. After 21 simulation steps, a state is repeated: at $t = 30$, there is a single active firing of actor a, and two active firings of c, all of which require two more time units to complete. Channel ab has four tokens at $t = 30$, the other channels have no tokens. Six simulation steps later, at $t = 44$, the same state is reached, and thus the self-timed execution has entered a periodic phase.

The throughput of the graph can now be obtained from the periodic phase: in the periodic phase, a single graph iteration is completed every 14 time units. The throughput of the graph is thus $\frac{1}{14}$.

6.2 *Incremental Unfolding*

The state-space exploration detailed above stops as soon as either a periodic phase is found, or no actor is enabled (i.e., a deadlock is found). This process may take a considerable amount of time, particularly if the transient phase is long. Especially for graphs that have long graph iterations, the analysis may require too much time. In [9], the analysis was reported to require more than a single day of computation time on several (complex) CSDF graphs.

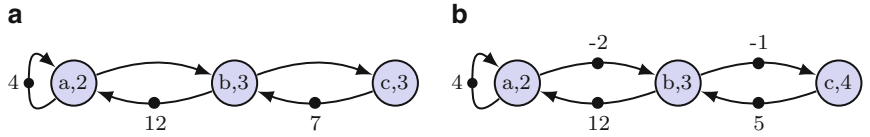


Fig. 14 (**a**) Optimistic single-rate approximation of Fig. 13. (**b**) Pessimistic single-rate approximation of Fig. 13

Rather than computing the throughput of a CSDF graph *exactly*, as is done in the state-space exploration approach, we can *approximate* it by analysing the graph's (pessimistic) single-rate approximation. As shown earlier in Sect. 4, the throughput of the single-rate approximation provides a lower bound on the throughput of the CSDF graph. The quality of the approximation, however, may be poor (see Sect. 4). By applying the unfolding transformation of Sect. 5, the approximation can be further improved. This gives rise to an iterative approach to computing the throughput: in each step, the CSDF cycle that corresponds to the critical cycle of the single-rate approximation is unfolded, using Algorithm 2, into its single-rate equivalent, leading to a larger graph with a tighter approximation.

This approach of *incrementally unfolding* the graph terminates when the critical cycle of the single-rate approximation corresponds to a CSDF cycle in which all actors have the same repetition vector. It may, however, be stopped earlier, if, for example, the upper and lower bounds on throughput are close enough.

To illustrate this incremental approach, we apply it to the CSDF graph of Fig. 13. As a first step, we construct its single-rate approximations by applying Algorithm 1. The optimistic and pessimistic approximations are shown in Fig. 14a, b.

For both HSDF graphs, maximum cycle ratio is attained by cycle bcb. The optimistic approximation has maximum cycle ratio of $6/7$, whereas the maximum cycle ratio of the pessimistic approximation is more than twice as high: $7/4$. Multiplication by the modulus of the CSDF graph, which is 12, gives that the throughput of the CSDF graph lies between $7/72$ and $1/21$. These bounds are relatively far apart: if we choose the lower bound of $1/21$, then the maximum error of this estimation is 49%.

In order to improve these bounds on throughput, we unfold the critical cycle bcb into its single-rate equivalent, using Algorithm 2. This results in the larger graph shown in Fig. 15a, of which the pessimistic single-rate approximation is shown in Fig. 15b. The latter has maximum cycle ratio of $7/3$, which is attained by cycle $b_3c_2b_3$. Since the unfolded graph has modulus $\mathcal{N} = 6$, this translates to a throughput of $1/14$; the same as found by state-space exploration.

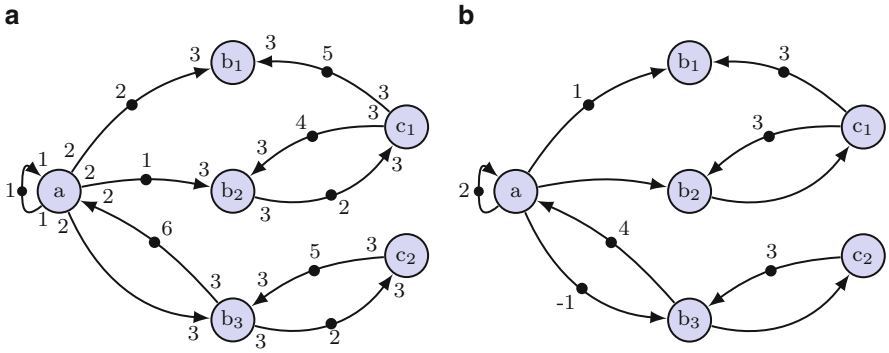


Fig. 15 (**a**) SDF graph equivalent to the CSDF graph of Fig. 13, obtained by unfolding cycle *bcb*. Execution times are omitted. (**b**) The pessimistic single-rate approximation of (**a**), with maximum cycle ratio of 7/3 attained by cycle $b_3c_2b_3$

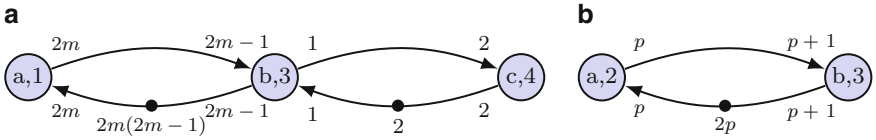


Fig. 16 (**a**) SDF graph, $5m - 1$ firings per iteration. (**b**) SDF cycle

6.3 Comparing the Two Approaches

Each of the two approaches described in the previous two sections has its advantages and disadvantages. State-space exploration is straightforward and simple to implement, however since a single graph iteration may consist of a huge number of actor firings, its runtime may not scale well in the size of the graph. Incremental unfolding, on the other hand, focuses its attention on those parts of the graph that potentially form *performance bottlenecks*. This means that it is not the size of a full graph iteration that determines the size of the graph that is eventually analysed, but rather the length of an iteration of a *subgraph*.

To illustrate both the effectiveness and difficulties of the two methods, consider Fig. 16a, b. In these figures, some properties have been parameterized to show the effect on the two methods. In the first graph, the length of a graph iteration depends on the parameter p. If we assume that p is prime, then the larger p, the larger the length of an iteration. This directly affects the runtime of state-space exploration. Incremental unfolding, on the other hand, chooses cycle *bcb* as the critical cycle. After unfolding this cycle, the incremental analysis stops since cycle *aba* is not critical.

In the other graph, the opposite conclusion is drawn. When applying an incremental analysis, the full graph is unfolded to its single-rate equivalent. The

size of this equivalent HSDF graph depends on p. When exploring the state-space of this graph, the unfolding is avoided.

To summarize the above comparison: for large graphs with large iterations, incremental analysis is likely to be more efficient if the size of the final graph is relatively small. In these cases, exploration of the (periodic) state-space might well consume too much time. The efficiency of the two methods depends largely on the size of the graph that would eventually be unfolded in the incremental approach.

6.4 Discussion

This chapter focuses primarily on throughput analysis of SDF and CSDF graphs. For more complex graphs such as scenario-aware dataflow graphs (SADF) graphs, see [19, 37].

The throughput of a graph is limited by its cycles. These cycles often arise due to the modelling of buffer capacities using reverse edges (i.e. backpressure). A problem that is closely related to throughput analysis is that of *buffer capacity optimization*: rather than computing the throughput of a graph with given buffer capacities, the goal is to minimize the (total) buffer capacity such that a certain throughput is still attained. For a more in-depth study of this subject, we encourage the reader to read [39] and [22].

References

1. Mohamed Bamakhrama and Todor Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 195–204. ACM, 2011.
2. Mohamed Bamakhrama and Todor Stefanov. On the hard-real-time scheduling of embedded streaming applications. *Design Automation for Embedded Systems*, 17(2):221–249, 2013.
3. Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.
4. Mohamed Benazouz, Olivier Marchetti, Alix Munier-Kordon, and Thierry Michel. A new method for minimizing buffer sizes for Cyclo-Static Dataflow graphs. In *8th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, pages 11–20. IEEE, Oct 2010.
5. Mohamed Benazouz, Alix Munier-Kordon, Thomas Hujsa, and Bruno Bodin. Liveness evaluation of a cyclo-static DataFlow graph. In *Proceedings of the 50th Annual Design Automation Conference on - DAC '13*, page 1, New York, New York, USA, May 2013. ACM Press.
6. G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.
7. B. Bodin, A. Munier-Kordon, and B.D. de Dinechin. K-periodic schedules for evaluating the maximum throughput of a synchronous dataflow graph. In *Proceedings of the International Conference on Embedded Computer Systems (SAMOS)*, pages 152–159, July 2012.
8. Bruno Bodin, Alix Munier-Kordon, and Benoit Dupont de Dinechin. Periodic schedules for Cyclo-Static Dataflow. In *Proceedings of the 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*, pages 105–114. IEEE, October 2013.

9. Bruno Bodin, Alix Munier-Kordon, and Benoît Dupont de Dinechin. Optimal and fast throughput evaluation of csdf. In *Proceedings of the 53rd Annual Design Automation Conference*, page 160. ACM, 2016.
10. Jean Cochet-terrasson, Guy Cohen, Stéphane Gaubert, Michael Mc Gettrick, and Jean-Pierre Quadrat. Numerical Computation of Spectral Elements in Max-Plus Algebra. In *Proceedings of the IFAC Conference on System Structure and Control*, July 1998.
11. G. Cohen, S. Gaubert, and Jean-Pierre Quadrat. Timed-event graphs with multipliers and homogeneous min-plus systems. *IEEE Transactions on Automatic Control*, 43(9):1296–1302, 1998.
12. Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and linearity: an algebra for discrete event systems*. Wiley New York, 1992.
13. Ali Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 9(4):385–418, 2004.
14. Robert de Groote. *On the Analysis of Synchronous Dataflow Graphs: a System-theoretic Perspective*. PhD thesis, University of Twente, the Netherlands, February 2016.
15. Robert de Groote, Philip K. F. Hölzenspies, Jan Kuper, and Gerard J. M. Smit. Multi-rate Equivalents of Cyclo-Static Synchronous Dataflow Graphs. In *Proceedings of the 14th International Conference on Application of Concurrency to System Design (ACSD)*, pages 62–71. IEEE Computer Society, June 2014.
16. Robert de Groote, Philip K. F. Hölzenspies, Jan Kuper, and Gerard J. M. Smit. Single-Rate Approximations of Cyclo-Static Synchronous Dataflow Graphs. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 11–20, June 2014.
17. Robert de Groote, Jan Kuper, Hajo Broersma, and Gerard J.M. Smit. Max-Plus Algebraic Throughput Analysis of Synchronous Dataflow Graphs. In *Proceedings of the 38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 29–38. IEEE, September 2012.
18. M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow: model and implementation. In *Proceedings of the 28th Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 503–507. IEEE Comput. Soc. Press, 1994.
19. Marc Geilen. Synchronous dataflow scenarios. *ACM Transactions on Embedded Computing Systems*, 10(2):1–31, December 2010.
20. A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij. Throughput Analysis of Synchronous Data Flow Graphs. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design (ACSD)*, pages 25–36. IEEE, 2006.
21. A.H. Ghamarian, M. Geilen, T. Basten, B. Theelen, M.R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In *Proceedings of the 6th conference on Formal Methods in Computer Aided Design (FMCAD)*, pages 68–75. IEEE, November 2006.
22. Amir Hossein Ghamarian. *Timing analysis of synchronous data flow graphs*. PhD thesis, Eindhoven University of Technology, The Netherlands, July 2008.
23. Steve Goddard. *On the Management of Latency in the synthesis of real-time signal processing systems from processing graphs*. PhD thesis, University of North Carolina at Chapel Hill, 1998.
24. S. Ha and H. Oh. Decidable signal processing dataflow graphs. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*. Springer, third edition, 2018.
25. B. Heidergott, Geert Jan Olsder, and Jacob van der Woude. *Max Plus at Work: modeling and analysis of synchronized systems*. Princeton University Press, 2006.
26. K. Ito and K.K. Parhi. Determining the iteration bounds of single-rate and multi-rate data-flow graphs. In *Proceedings of the 1994 Asia Pacific Conference on Circuits and Systems*, pages 163–168. IEEE, 1994.
27. Richard M. Karp and James B. Orlin. Parametric shortest path algorithms with an application to cyclic staffing. *Discrete Applied Mathematics*, 3(1):37–45, February 1981.

28. E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
29. E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.
30. R. Leupers, M. A. Aguilar, J. Castrillon, and W. Sheng. Software compilation techniques for heterogeneous embedded multi-core systems. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*. Springer, third edition, 2018.
31. Olivier Marchetti and Alix Munier-Kordon. Minimizing Place Capacities of Weighted Event Graphs for Enforcing Liveness. *Discrete Event Dynamic Systems*, 18(1):91, 2008.
32. Orlando Moreira and Henk Corporaal. *Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor*, volume 24 of *Embedded Systems*. Springer International Publishing, Cham, 2014.
33. K.K. Parhi. Algorithm transformation techniques for concurrent processors. *Proceedings of the IEEE*, 77(12):1879–1895, 1989.
34. T.M. Parks and E.A. Lee. Non-preemptive real-time scheduling of dataflow systems. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3235–3238. IEEE, 1995.
35. T.M. Parks, J.L. Pino, and E.A. Lee. A Comparison of Synchronous and Cyclo-static Dataflow. In *Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 204–210. IEEE Comput. Soc. Press, 1995.
36. Raymond Reiter. Scheduling Parallel Computations. *Journal of the ACM*, 15(4):590–599, October 1968.
37. Firew Siyoum, Marc Geilen, Orlando Moreira, and Henk Corporaal. Worst-case throughput analysis of real-time dynamic streaming applications. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES+ISSS '12*, page 463, New York, New York, USA, October 2012. ACM Press.
38. Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, February 2009.
39. Sander Stuijk, Marc Geilen, and Twan Basten. Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, October 2008.
40. Enrique Teruel, Piotr Chrzastowski-Wachtel, José Manuel Colom, and Manuel Silva. On Weighted T-Systems. *Application and Theory of Petri Nets*, pages 348–367, June 1992.
41. Maarten Wiggers, Marco Bekooij, Pierre Jansen, and Gerard Smit. Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis - CODES+ISSS '06*, page 10, New York, New York, USA, 2006. ACM Press.
42. M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs. In *Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC)*, pages 658–663. IEEE, 2007.
43. Neal E. Young, Robert E. Tarjant, and James B. Orlin. Faster parametric shortest path and minimum-balance algorithms. *Networks*, 21(2):205–221, mar 1991.

Dataflow Modeling for Reconfigurable Signal Processing Systems

Karol Desnos and Francesca Palumbo

Abstract Nowadays, adaptive signal processing systems have become a reality. Their development has been mainly driven by the need of satisfying diverging constraints and changeable user needs, like resolution and throughput versus energy consumption. System runtime tuning, based on constraints/conditions variations, can be effectively achieved by adopting reconfigurable computing infrastructures. These latter could be implemented either at the hardware or at the software level, but in any case their management and subsequent implementation is not trivial. In this chapter we present how dataflow models properties, as predictability and analyzability, can ease the development of reconfigurable signal processing systems, leading designers from modelling to physical system deployment.

1 Reconfigurable Signal Processing Systems

For many years, the design of a signal processing system was mostly driven by performance requirements. Hence, design effort was mainly focused on optimizing the throughput and latency of the designed system, while satisfying constraints of reliability and quality of service, and minimizing the system production cost. In this context, a strong predictability of system behavior is essential, especially when designing safety-critical real-time systems. Many compile-time methodologies, computer-aided design tools, and static MoCs, such as the decidable dataflow MoCs [26], have been created to assist system designers in reaching these goals.

In recent years, the ever increasing complexity of signal processing systems has lead to the emergence of new design challenges. In particular, modern signal processing systems no longer offer a fixed throughput and latency, specifically

K. Desnos (✉)
Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164, F-35000 Rennes, France
e-mail: kdesnos@insa-rennes.fr

F. Palumbo
Universita degli Studi di Sassari, Sassari, Italy
e-mail: fpalumbo@uniss.it

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_22

tuned to satisfy all timing constraints in the most demanding scenarios from the system specification. Instead, modern systems must now dynamically adapt their behavior on-the-fly to satisfy strongly varying workloads and performance objectives, while optimizing new design goals such as minimizing use of shared computational resources, or minimizing power consumption. These variations of the workload and performance objectives may be induced by functional and non-functional requirements of the system. An example of system with strongly varying functional requirements is the computing system managing a base station of the Long-Term Evolution (LTE) telecommunication network. Every millisecond, the bandwidth allocated for up to 100 active users connected to the managed antenna may change [50], with a strong impact on the amount and nature of computations performed by the system. An embedded video decoding system that lowers the quality of its output in order to augment battery life is an example of varying non-functional requirement [48, 55]. As presented in [64], increasing the dynamism of a signal processing often results in a partial loss of predictability, making it difficult, and sometimes impossible, to guarantee the real-time performance or the reliability (e.g. deadlock freedom) of a system.

The purpose of reconfigurable signal processing systems is to offer a carefully balanced trade-off between system predictability and adaptivity. This trade-off between diverging properties is essential to meet classical constraints of system performance and reliability while satisfying varying requirements of modern system. To offer a trade-off between predictability and adaptivity, reconfigurable systems rely both on design-time analysis and optimization of system behavior, that make it possible to predict the system behavior, and on runtime management technique that enable system adaptation.

The objective of this chapter is to present how the analyzability of dataflow MoCs can be exploited to ease the development of reconfigurable signal processing systems. The chapter structure is briefly summarized as follows: Sect. 2 presents how reconfigurable systems can be efficiently modeled with dedicated dataflow MoCs, and Sects. 3 and 4 presents how software and hardware techniques, respectively, can be used to implement reconfigurable applications efficiently. In more details, Sect. 2 formally introduces the concept of reconfigurable dataflow MoCs and discusses its key differences with decidable and dynamic classes of dataflow MoCs. This concept is illustrated through the semantics of several reconfigurable dataflow MoCs. In Sect. 3, software implementation techniques supporting the execution of reconfigurable dataflow application are presented. This section covers a wide range of software implementation techniques, spanning from compile-time optimizations to runtime management system for reconfigurable applications. Then, Sect. 4 presents a summary on reconfigurable computing systems, where both coarse grained and fine grained reconfiguration paradigm are addressed describing how dataflow MoCs may help in mapping and managing this kind of highly flexible computing systems.

2 Reconfigurable Dataflow Models

The high abstraction level of dataflow MoCs makes them popular models for specifying complex signal processing applications [26]. By exposing coarse-grain computational kernels,[1] the actors, and data dependencies between them, the First-In First-Out queues (FIFOs), the dataflow semantics eases the specification of parallel applications, and provides necessary formalism for many verification and optimization techniques for the design of signal processing systems.

The expressiveness of a MoC defines the range of applications behavior this MoC can describe. The expressiveness of a dataflow MoC is directly related to its firing rules, which are used to specify how and when actors produce and consume data tokens on connected FIFOs. Dataflow MoCs can be sorted into three classes depending on their expressiveness:

- **Decidable dataflow MoCs** [26]: all production and consumption rates are fixed at compile-time, either as fixed scalar, or with periodic variations. The key characteristics of decidable dataflow graphs is that, through an analysis of data rates, it is possible to derive a schedule of finite length at compile-time.
- **Dynamic dataflow MoCs** [64]: production and consumption rates can change non-deterministically at each actor firing, making these models Turing-complete ones. Hence, for most dynamic dataflow MoCs, schedulability, deadlock-freedom, real-time properties, and memory boundedness of application graphs can be verified neither at compile-time, nor at runtime. In some dynamic dataflow MoCs, this lack of analyzability is partially alleviated through specialization of the model semantics.
- **Reconfigurable dataflow MoCs**: production and consumption rates can be reconfigured (i.e. changed) non-deterministically at restricted points in application execution. These MoCs are sometimes also called parametric dataflow MoCs [11]. This restriction limits the expressiveness of reconfigurable dataflow models, but makes it possible to verify application properties, like schedulability, either at compile-time or at runtime, after a reconfiguration occurred.

The following subsections formally introduce the reconfiguration semantics behind reconfigurable dataflow MoCs, and shows how it benefits model analyzability through the presentation of several reconfigurable dataflow MoCs. Implementation optimization techniques taking advantage of the reconfiguration semantics are presented in Sects. 3 and 4.

[1] Kernels are usually intended as subparts of an application providing a specific computation. In a perspective hardware implementation, the computationally intensive ones are the part of the application that are normally delegated to specific accelerators.

2.1 Reconfiguration Semantics

The reconfiguration semantics behind reconfigurable dataflow MoCs is a mathematical model that makes it possible to detect potentially unsafe reconfigurations of an application graph [44]. A reconfiguration is said to be unsafe if it may result in an unwanted and undetected state of the application, such as a deadlock or an inconsistence in production and consumption rates.

The reconfiguration semantics for dataflow MoCs is based on the definition of hierarchical actors, parameters, and quiescent points. The reconfiguration semantics can be implemented by any dataflow MoC with atomic actor firings. This assumption guarantees the applicability of the reconfiguration semantics to a broad range of dataflow MoCs, including decidable [26], multi-dimensional [33], and some dynamic dataflow MoCs [64].

Formally, a hierarchical graph is defined as a set of actors A. An actor $a \in A$ can either be an *atomic actor*, an actor whose internal behavior is specified with host code, or a *hierarchical actor*, an actor whose internal behavior is specified with a subset of actors $A_a \subset A$. The *top-level* graph itself is considered as a hierarchical actor with no parent, and that contains all other actors in A. Each actor a is associated to a dedicated set of parameters P_a, that can influence both its production and consumption rates, and the computations it performs. At any point in execution time, each parameter $p \in P (= \bigcup_{a \in A} P_a)$ is associated to a value given by a *valuation function*: $val(p)$ whose type (e.g. integer, real, boolean, ...) depends on the underlying model semantics. The value $val(p)$ of a parameter may be independent, or may depend on the value of one or several other parameters in P. The transitive relation q depends on p, between two parameters $p \neq q$, is noted $p \rightsquigarrow q$. Reconfiguration occurs when the value of an independent parameter is modified during the execution of the application.

Figure 1 shows an example of graph specified with a synthetic dataflow MoC implementing the reconfiguration semantics. Figure 1a illustrates the semantics, which is then used to build up the example in Fig. 1b. The graph of Fig. 1b consists of five actors, including one hierarchical actor h that contains two atomic actors C and D. Each consumption and production rate is defined by a dedicated parameter whose valuation function is specified as an expression written next to the graph. In this graph, p, s, and t are independent parameters and all other parameters have dependencies; as for example $p \rightsquigarrow r$.

Reconfiguration semantics models as a *quiescent point* the state of an actor between two firings. Like actor firings, the set of quiescent points Q_a of an actor a is ordered in time according to a transitive *precedence relation*. Since firings of actors contained in a hierarchical subgraph cannot span over multiple firings of their enclosing actor, when a hierarchical actor is quiescent, all actors contained in its hierarchy must also be. A graphical representation of quiescent points for the graph of Fig. 1 is presented in Fig. 2.

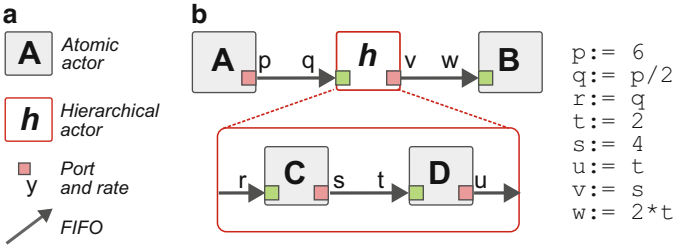


Fig. 1 Synthetic dataflow MoC implementing the reconfiguration semantics. (**a**) Semantics. (**b**) Graph and current valuation of parameters

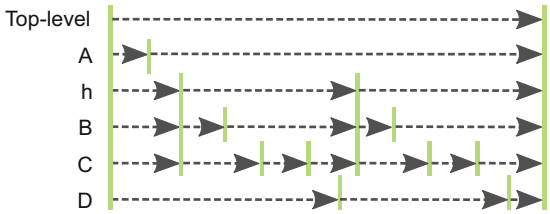


Fig. 2 Abstract representation of the quiescent points for the graph of Fig. 1. Vertical lines represent quiescent points of actors. Dotpoint arrows represent actor firings and define a partial ordering of quiescent points

The reconfiguration semantics requires reconfigurations of any independent parameter $p \in P$ to happen only during a quiescent points $q \in Q(= \bigcup_{a \in A} Q_a)$ at execution time. The set of parameters reconfigured during a quiescent point q is noted: $R(q) \subseteq P$.

When using a dataflow MoC implementing the reconfiguration semantics, a compile-time analysis of parameters and quiescent points of an application graph can be used to verify model-specific reconfiguration safety requirements [44]. Safety requirements of a dataflow MoC are expressed as statements in the form "Parameter p is constant over firings of actor a". Formally:

Definition 1 Parameter p is constant over firings of actor c if and only if $\forall a \in A, \forall q \in Q_a, \forall r \in R(q), r \rightsquigarrow p \Rightarrow q \in Q_c$.

For example, in the SDF MoC, decidability can be expressed as the following safety requirement: all parameters are required to be constant over firings of the *top-level* actor.

In [11], Bouakaz et al. present a Survey of dataflow MoCs adopting the reconfiguration semantics. Next sections present examples of MoCs implementing the reconfiguration semantics.

2.2 *Reconfigurable Dataflow Models*

2.2.1 Hierarchy-Based Reconfigurable Dataflow Meta-Models

The purpose of a dataflow meta-model is to bring new elements to the semantics of a *base dataflow MoC* in order to increase its modeling capabilities. The Parameterized and Interfaced Dataflow Meta-Model (PiMM) [17] and Parameterized Dataflow [8] are two dataflow meta-models with similar purpose: bring hierarchical graph composition and safe reconfiguration features to any decidable dataflow MoC that has a well-defined notion of graph iteration and repetition vector, such as SDF and Cyclo-Static Dataflow (CSDF) [26], or Multi-Dimensional SDF (MDSDF) [33]. A base dataflow MoC whose semantics is enriched with PiMM or with the parameterized dataflow meta-model is renamed with prefixes π- and P-, respectively. For example, πSDF and Parameterized SDF (PSDF) are the reconfigurable generalizations of the decidable SDF MoC.

In PiMM and parameterized dataflow meta-models, safe reconfiguration is expressed as a *local synchrony requirement* [8]. Intuitively, local synchrony requires the repetition vector of the subgraph of a hierarchical actor to be configured at the beginning of the execution of this subgraph, and to remain constant throughout a complete graph iteration, corresponding to a firing of the parent actor. Hence, in locally synchronous hierarchical subgraph, all parameters influencing production and consumption rates of actors must remain constant over the firing of their parent actor. Formally, with S_c, the set of direct child actors of a hierarchical actor $c \in A$, where direct child means that $\forall h \in S_c, d \in S_h \Rightarrow d \notin S_c$.

Definition 2 Subgraph S_c of actor $c \in A$, is locally synchronous if and only if: $\forall a \in S_c, \forall p \in P_a$, the requirement "$p$ is constant over firings of c" is verified.

The semantics of PiMM, which is an evolution of the parameterized dataflow semantics, is illustrated in Fig. 3. An example of πSDF graph is given in Fig. 4.

In PiMM, graph compositionality is supported by *hierarchical actors*, as defined in the reconfiguration semantics, and by *data interfaces*. The purpose of interface-

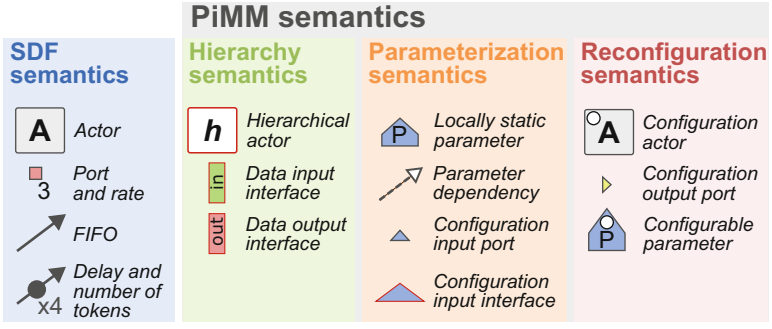


Fig. 3 Semantics of the Parameterized and Interfaced Dataflow Meta-Model (PiMM)

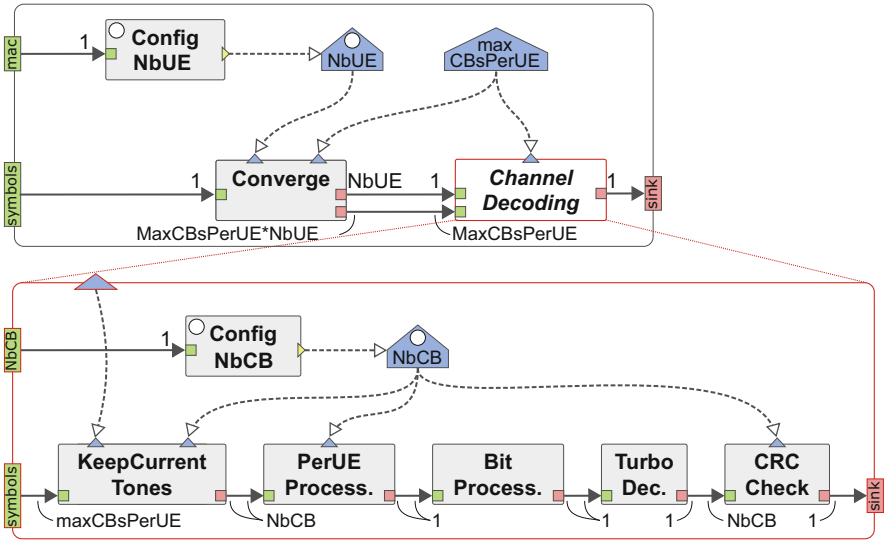


Fig. 4 Example of π SDF graph. Bit processing algorithm of the Physical Uplink Shared Channel (PUSCH) decoding of the LTE telecommunication standard [17]

based hierarchy [52] is to insulate the nested levels of hierarchy in terms of graph consistence analysis. To do so, data interfaces automatically duplicate and discard data tokens if, during a subgraph iteration, the number of tokens exchanged on FIFOs connected to interfaces is greater than the number of token produced on the corresponding data ports of the parent actor.

The parameterization semantics of PiMM consists of parameters and parameter dependencies as new graph elements, and configuration input ports and interfaces as new actor attributes. The value associated to a parameter of a graph is propagated through explicit parameter dependencies to other parameters and to actors. In the πSDF MoC, it is possible to disable all firings of an actor by setting all its production and consumption rates to zero. As illustrated in Fig. 4, parameter values can be propagated through multiple levels of hierarchy using a configuration input port on a hierarchical actor and a corresponding configuration input interface in the associated subgraph.

The reconfiguration semantics of PiMM is based on actors with special firing rules, called configuration actors. When fired, reconfiguration actors are the only actors allowed to dynamically change the value of a parameter in their graph. As a counterpart for this special ability, reconfiguration actors must be fired exactly once per firing of their parent actor, before any non-configuration actor of their subgraph. This restriction is essential to ensure the safe reconfiguration of the subgraph to which configuration actors belong. To be strictly compliant with Definition 2, configuration actors and other actors of a subgraph can be considered as two separate subgraphs, executed one after the other.

The local synchrony requirement of the πSDF MoC naturally enforces the predictability of the model. After firing all configuration actors of a subgraph, all parameters values, and hence all actor production and consumption rates of this subgraph, are known and will remain fixed for a complete subgraph iteration. Runtime analyses and optimization techniques can be used to compute the repetition vector of the subgraph, to optimize the mapping and scheduling of actors, to allocate the memory, or to verify that future real-time deadlines will be met. An important benefit of this predictability is the support for data-parallelism which is often lost in dynamic dataflow MoCs. Data parallelism consists in starting several firings of the same actor in parallel if enough data tokens are available. Data-parallelism is supported only if the next sequence of firing rates is know a priori, as is the case in πSDF graphs. In dynamic dataflow MoCs [64], the firing rules of an actor generally depend on its internal state after completion of its previous firing. This internal dependency between actor firings forces their sequential execution, thus preventing data parallelism.

Figure 4 presents a πSDF specification of the bit processing algorithm of the Physical Uplink Shared Channel (PUSCH) decoding which is part of the LTE telecommunication standard. The LTE PUSCH decoding is executed in the physical layer of an LTE base station (eNodeB). It consists of receiving multiplexed data from several User Equipments (UEs), decoding it and transmitting it to upper layers of the LTE standard. Because the number of UEs connected to an eNodeB and the rate for each UE can change every millisecond, the bit processing of PUSCH decoding is inherently dynamic and cannot be modeled with static MoCs such as SDF. Further details on this application, and specification using the parameterized dataflow meta-model, can be found in [50].

Compile-time and light-weight runtime scheduling technique for executing πSDF and PSDF graphs, are presented in Sect. 3. Combination of the parameterized dataflow semantics and the CSDF MoC is studied in [32] for the design of software-defined radio applications.

2.2.2 Statically Analyzable Reconfigurable Dataflow Models

SPDF and Boolean Parametric Dataflow (BPDF) are non-hierarchical reconfigurable generalizations of the SDF MoC that emphasize static model analyzability. In particular, the semantics of the SPDF and BPDF MoCs make it possible to verify safe reconfiguration requirements and to guarantee graph consistency and liveness at compile time. In πSDF and PSDF, although local synchrony can be checked at compile time, consistency and liveness of a subgraph can only be verified at runtime, after configuration of all the parameters contained in this subgraph.

In the SPDF MoC semantics, symbolic parameters P with integer values in $\mathbb{N}^*$ are used for parameterization. Special actors, called *modifiers*, have the ability to dynamically change the value of a symbolic parameter. To ensure safe reconfiguration, a modifier $m \in A$ will set a new value for a parameter $p \in P$ with a pre-defined change period $\alpha \in \mathbb{N}^*$. In practice, this means that the value of

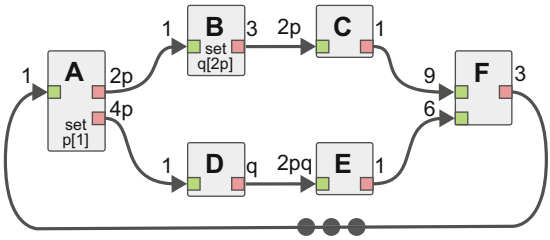


Fig. 5 Example of Schedulable Parametric Dataflow (SPDF) graph

p will be changed every α firings of m. In an SPDF graph, as shown in Fig. 5, the annotation "set $p[\alpha]$" is used to denote that an actor is a modifier of parameter p, with change period α. Change periods, and production and consumption rates of actors are specified with products $\prod_{i=0}^{n} e_i$, where e_i is either an integer in $\mathbb{N}^*$, or a symbolic parameter $p \in P$.

Using balance equations of actor production and consumption rates, similar to those used for SDF graphs [26], graph consistency can be verified, and a symbolic repetition vector can be computed. The basic operation used to find the symbolic repetition vector of an SPDF graph is the computation of the Greatest Common Divisor (GCD) of the numbers of data tokens produced and consumed on each FIFO of the graph. Using this GCD, the numbers of repetition of the producer and consumer actors, relatively to each other, can be deducted by dividing the rates of the actors by this GCD. For example, in the graph of Fig. 5, the GCD of the FIFO between actor D and E is $gcd_{DE} = gcd(q, 2pq) = q$, which means that actor E will be executed $rate_D/gcd_{DE} = 2pq/q = 2p$ times for each $rate_E/gcd_{DE} = q/q = 1$ execution of actor D. Using this principle, an algorithm detailed in [22] can be used to compute the parametric number of repetition of all actors in an SPDF graph. Using this algorithm on the SPDF graph of Fig. 5, the following symbolic repetition vector is obtained: $A^3B^{6p}C^9D^{12p}E^6F$, where X^a means that actor X is fired a times per iteration of the graph. Notation $\#X$ is used to denote the repetition count of an actor $X \in A$. Similarly, liveness (i.e. deadlock-freedom) of graphs can be verified statically with an analysis of cyclic data-paths inspired from SDF graph techniques.

Reconfiguration safety in SPDF graphs is based on the notion of parameter influence region. The influence region $R(x)$ of a parameter x is the set of: *a)* FIFOs whose rates depend on x, *b)* actors connected to these FIFOs, and *c)* actors whose numbers of repetitions depend on x. For example in Fig. 5, $R(q)$ comprises FIFO DE and actors D and E; and $R(p)$ comprises the whole graph except actor F and FIFOs connected to it.

Two safe reconfiguration requirements are used in SPDF: data and period safety. Intuitively, data safety requires that the region $R(p)$ influenced by a parameter $p \in P$ comes back to an initial state, in terms of the number of data-tokens on FIFOs, between each reconfiguration of p. In other words, data safety requires the (virtual) subgraph composed by $R(p)$ to complete a kind of local iteration and be quiescent when a reconfiguration of a parameters influencing it occurs. Formally, data safety

requires that $\forall p \in P$, with modifier $m \in A$ and change period α, all actors $a \in R(p)$ have a repetition count $\#a$ such that $gcd(\#a, \#m/\alpha) = \#m/\alpha$ (i.e. $\#a$ is a multiple of $\#m/\alpha$). For example in Fig. 5, with actor $E \in R(q)$ and modifier B annotated "set $q[2p]$", actor E is data safe since $\#E = 6$ is a multiple of $\#B/2p = 6p/2p = 3$.

Period safety restricts how often a parameter q can be reconfigured by its modifier m if $\#m$ itself depends on a parameter p. In this case, period safety ensures that q is reconfigured at least as often as the start of a new iteration of the subgraph formed by $R(p)$. Formally, $\forall p, q \in P$ with modifiers m_p and m_q, change period α_p and α_q, and $\#m_q$ depending on p, period safety requires $\#m_q/\alpha_q$ to be a multiple of $\#m_p/\alpha_p$. If this condition is not met, reconfiguration of q may happen in the middle of an iteration of the subgraph formed by $R(p)$, when it is not quiescent. For example, in Fig. 5, repetition count $\#B = 6p$ of the emitter of q depends on p, but its change period is safe since $\#B/2p = 3$ is a multiple of $\#A/1 = 3$. With the period of q set to $3p$ instead, the graph would remain data safe but would no longer be period safe, as $\#B/3p = 2$ is not a multiple of $\#A/1 = 3$. A consistent, data and period safe SPDF graph will always be schedulable in bounded memory [22].

The semantics of the BPDF MoC is closely related to the one of the SPDF MoC, and provides the same advantages in terms of static graph analyzability [6]. The main difference between the two MoCs is that reconfigurable integer parameters of SPDF are replaced with reconfigurable boolean parameters in BPDF. Through combinational logic expressions, these boolean parameters are used to change the BPDF graph topology, by enabling and disabling FIFOs. This feature is equivalent to setting data rates to 0 in the SPDF and π SDF MoCs.

Examples of SPDF and BPDF graphs of multimedia, signal processing, and software defined radio applications can be found in [6, 16, 22]. Compilation techniques for deploying SPDF graphs onto multi and many-core architecture will be presented in Sect. 3.

2.3 Dynamic Dataflow MoCs and Reconfigurability

In dynamic dataflow MoCs, as presented in [64], production and consumption rates of actors may change dynamically at each firing, depending on the firing rules specified for each actor. In most dynamic dataflow MoCs, the semantics include elements to specify explicitly the internal firing rules of each actor. A common way to characterize firing rules of a dynamic actor is to model its internal state with a Finite State Machine (FSM), or a similar model like a Markov chain, and to associate each FSM state with pre-defined production and consumption rates for each data port. These FSMs can be specified either explicitly by the application designer, using a dedicated language, as in the Scenario-Aware Dataflow (SADF) and Core-Functional Dataflow (CFDF) MoCs [40, 64], or implicitly in the language describing the internal behavior of actors, as in frameworks based on the CAL Actor Language (CAL) [13, 21, 67].

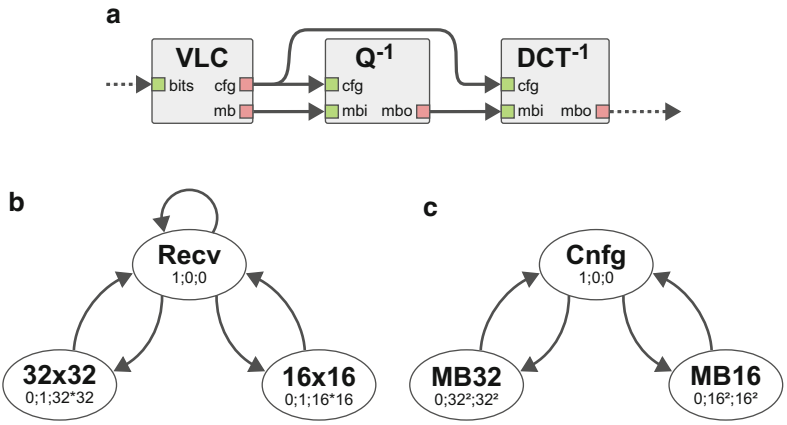


Fig. 6 Dynamic dataflow graph inspired by the residual decoding of a video decoder. (**a**) Dynamic dataflow graph. (**b**) FSM of the *VLC* actor. Prod./Cons. rates for each state are expressed in the following order `bits;cfg;mbo`. (**c**) FSM of the Q^{-1} and DCT^{-1} actors. Prod./Cons. rates for each state are expressed in the following order `cfg;mbi;mbo`

An example of dynamic dataflow graph with associated FSMs is presented in Fig. 6. This graph represents the residual decoding of macro-blocs of pixels in a video decoding application. The semantics used for the FSMs presented in Fig. 6b, c is similar to the one of the CFDF MoC. Each state of the FSMs is associated with an integer production and consumption rate for each port of the corresponding actor. When enough data tokens are available on the input FIFOs according to the current actor state, the actor is fired and one state transition is traversed, thus deciding the next firing rule.

In the graph of Fig. 6, the Variable Length Code (*VLC*) actor is responsible for decoding the information corresponding to each macroblock (i.e. square of pixel) from the input bitstream. To do so, the *VLC* actor reads the input bitstream bit-by-bit in the *Recv* state, and fills an internal buffer. When enough bits have been received, the *VLC* actor detects it, and goes to the 16×16 or 32×32 state, depending on the dynamically detected macroblock size. In the 16×16 or 32×32 states, the *VLC* actor produces a configuration data-tokens on its `cfg` port, and quantized macroblock coefficients on its `mb` port. The configuration data-token is received by the dequantizer actor Q^{-1} and the inverse discrete cosine transform actor DCT^{-1}, which successively process the quantized macroblock coefficients in the state corresponding to the dynamically detected macroblock size: 16×16 or 32×32.

Although dynamic dataflow MoCs inherently lack the predictability that characterize reconfigurable dataflow MoCs, several techniques make it possible to increase the predictability of a dynamic dataflow graph in order to enable a reconfigurable behavior. The common point between these techniques, is that they exploit the actor behavioral information specified with FSMs in order to make parts of the

application reconfigurable. Two different approaches are presented hereafter: in Sect. 2.3.1 classification techniques are used to identify reconfigurable behavior in application graphs specified with existing dynamic MoCs, and in Sect. 2.3.2, semantics of dynamic dataflow MoCs is extended to ease specification of reconfigurable behavior within dynamic applications. Further hardware reconfigurable implementation techniques based on dynamic dataflow MoCs are introduced in Sect. 4.

2.3.1 Classification of Dynamic Dataflow Graphs

The basic principle of classification techniques for dynamic dataflow graphs is to analyze the internal FSM of one or more actors in order to identify patterns that correspond to statically decidable behaviors. Here, a pattern designates a sequence of firings of one or more actors that, when started, will always be executed deterministically when running the application, despite the theoretical non-deterministic dynamism of the dataflow MoC.

In the example of Fig. 6, an analysis technique can detect two sequences of actor firings, corresponding to the processing of a macroblock of size 16×16 and 32×32, respectively. The first sequence is triggered when the *VLC* actor fires in the *16×16* state, which will always be followed by two firings of each of the Q^{-1} and DCT^{-1} actors, in the *Cnfg* and *MB16* states. The second sequence is similar for the 32×32 configuration.

In [13], Boutellier et al. propose a technique to analyze a network of actors specified with CAL, and detect static sequences of actor firings in their FSMs. Once detected, each alternative sequence of actor firings is transformed into an equivalent SDF subgraph. Similarly to what is done in reconfigurable dataflow MoCs, SDF subgraphs are connected using switch/select actors capable of triggering dynamically an iteration of a selected SDF subgraph. As shown in [13], application performance can be substantially improved by exploiting the predictability of SDF subgraphs to decrease the overhead of dynamic scheduling of actor firings.

In [21], Ersfolk et al. introduce a technique to characterize dynamic actor by identifying the control tokens of the application. A control token is defined as a data token whose value is used in actor code to dynamically decide which firing rule will be validated for the next actor firing. For example, in Fig. 6, the data tokens exchanged on the `cfg` ports are control tokens, as they decide the next firing rules for the Q^{-1} and DCT^{-1} actors. Once a control token is identified, the data and control path influencing its value is backtracked both through graph FIFOs and by applying an instruction-level dependency analysis to actor CAL code. By analyzing the datapath of control tokens, complex relations between firing rules of different actors can be revealed. As in the previous technique, these relations between firing rules of dynamic actors can be exploited to transform some part of a dynamic dataflow graph into an equivalent reconfigurable graph.

There exist several other techniques whose purpose is to detect reconfigurable or static behavior from a dynamic dataflow description. In [67], a set of rules are specified to classify the behavior of individual actors as static, cyclo-static, quasi-static, time-dependent (i.e. non-deterministic), and dynamic. These rules can be verified either by analyzing the firing rules of an actor, or by using abstract interpretation which allows verifying these rules for all possible actor states [67]. In [20], another technique based on model-checking is used to detect statically schedulable actions in a dynamic dataflow graph.

2.3.2 Reconfigurable Semantics for Dynamic Dataflow MoC

Parameterized Set of Modes (PSM) is an extension of the CFDF MoC which brings parameterization semantics on top of the dynamic semantics of the CFDF model [40]. The purpose of the PSM-CFDF MoC is to improve the analyzability of FSMs in a network of actors by explicitly specifying graph-level parameters that influence the dynamic dataflow behavior of one or more actors.

In the PSM-CFDF semantics, an actor $a \in A$ is associated to an FSM where each state, called a mode, corresponds to a fixed consumption and production rate. The set of all modes of an actor a is noted M_a. Each time a mode $m_a \in M_a$ of an actor a is fired, it selects the mode that will be used for the next firing of a. The mode selected for the next firing of an actor a depends on the parameter values, called a configuration, of a set of parameters $Param(a)$ specified at graph-level. The set of all valid configurations for an actor a is noted $DOMAIN(a)$.

Figure 7 presents an example of PSM-CFDF graph modeling part of the Orthogonal Frequency-Division Multiplexing (OFDM) demodulation of an LTE receiver, inspired by Dardaillon et al. [16] and Pelcat et al. [50]. Parameter M takes value in $\{1, 2\}$ and is used to switch the application behavior between a low-power mode $M = 1$, where a 5 MHz bandwidth is received with QPSK modulation, and a high-throughput mode $M = 2$, where a 10 MHz bandwidth is received with 16QAM modulation. Parameter B takes integer values between 1 and B^{max}, and is used to control the vectorization of computation, i.e. the number of data tokens buffered in order to be processed in a single firing of actors. In low-power mode (resp. high-throughput mode), the *FFT* actor processes 512 (resp. 1024) samples and outputs symbols for 300 (resp. 600) subcarriers, each of which is then decoded by the *Demap* actor, using QPSK (resp. 16QAM) modulation, and producing 600 (resp. 2400) bits of data. As presented in Fig. 7b, c, the fully connected FSMs of the two actors each contain $2 * B^{max}$ modes.

Building on the graph-level parameterization semantics, the basic idea of PSM-CFDF is to gather actor modes into groups of modes with similar properties (e.g. dataflow rates, mapping, ...) for a subsequent analysis or optimization of the application. These groups of modes are called the Parameterized Set of Modes (PSM) of an actor. Formally, a PSM of an actor a is defined as $\rho = (S, C, f)$, where $S \subset M_a$ is a subset of the modes of a, $C \subset DOMAIN(a)$ is a subset of the configurations of a, and $f : C \rightarrow S$ is a function giving the current mode in

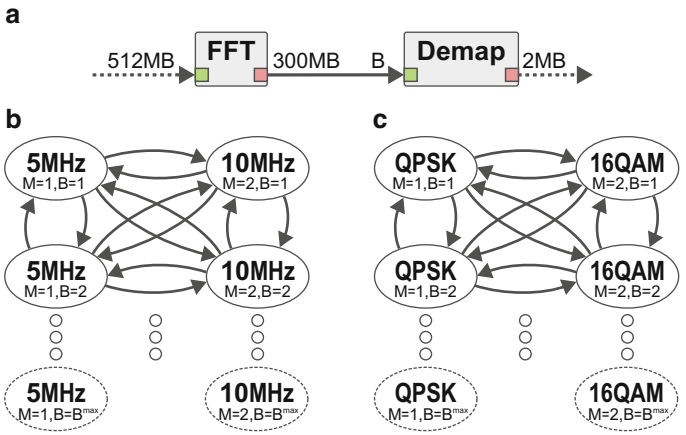


Fig. 7 PSM-CFDF graph of an OFDM demodulator used in LTE standard [16, 40, 50]. (**a**) PSM-CFDF graph. (**b**) Partial FSM of the *FFT* actor. (**c**) Partial FSM of the *Demap* actor

Fig. 8 Transition graph for the PSM-CFDF actors of Fig. 7

S depending on the current configuration in C. The set of all PSMs of an actor a, noted $PSM(a)$, must cover all modes of a; formally $\bigcup_{\rho \in PSM(a)} \rho.S = M_a$. $PSM(a)$ can be represented as a graph, called transition graph, whose vertices are the PSMs of the actor, and whose edges represent possible transitions from a PSM to another, as originally specified in the actor FSM.

PSMs of an actor can be specified explicitly by the application developer, but can also be deduced automatically from an analysis of the parameterized PSM-CFDF graph, or from execution traces [40]. Figure 8 presents the transition graph created for the FSMs in Fig. 7b, c, assuming that values of parameters B and M are changed simultaneously when no data-token is present on FIFOs of the graph of Fig. 7. This transition graph contains two PSMs $\rho 1$ and $\rho 2$, gathering modes for $M = 1$ and $M = 2$ respectively.

A smart grouping of actor modes into PSMs can be used to produce a transition graph where each PSM corresponds to a static schedule of actor firings [40, 54]. For example in Fig. 8, each of the two PSMs corresponds to a static scheduling of the PSM-CFDF graph, with FFT^1Demap^{300} for $\rho 1$, and FFT^1Demap^{600} for $\rho 2$, regardless of the value of parameter B. In such a case, changing the active PSM triggers a reconfiguration of the application that switches between pre-computed static schedules, which considerably reduces the workload of the application scheduler.

Smart PSM grouping can efficiently address many optimization objectives, such as maximizing application performance on heterogeneous platforms [40], or minimizing the allocated memory footprint for the execution of a dynamic graph [54].

3 Software Implementation Techniques for Reconfigurable Dataflow Specifications

As shown in previous section, dataflow MoCs can efficiently capture the coarse-grain reconfigurable behavior of a signal processing system. From the dataflow perspective, a reconfigurable behavior can be modeled as an explicit lightweight control-flow enabling predictable and possibly parameterized sequences of actor firings. A reconfigurable dataflow behavior can either be explicitly specified by the system developer using specialized dataflow MoCs, or be extracted automatically from a more dynamic system specification.

To execute a dataflow graph, a set of techniques must be developed to implement the theoretical MoC semantics and execution rules within diverse hardware and software environments. Implementation techniques are commonly responsible for mapping and scheduling actor firings onto available processing elements and for allocating memory and communication resources. Depending on the predictability of the implemented dataflow MoC, these implementation techniques can be part of the compilation process, or part of a runtime manager or operating system [37].

This section presents a set of software implementation techniques for dataflow specifications that exhibit a reconfigurable behavior. By taking advantage of the reconfigurable behavior of applications, the presented techniques optimize systems in terms of performance, resource usage, or energy footprint. Implementation techniques responsible for translating dataflow specifications with reconfigurable behavior into synthesizable hardware implementations are presented in Sect. 4. A more general description of software compilation techniques for other parallel programming models is presented in chapter [39].

3.1 Compile-Time Parameterized Quasi-Static Scheduling

In general, dynamic and reconfigurable dataflow MoCs are non-decidable models. Hence, contrary to decidable dataflow MoCs [26], it is not possible to determine at compile time a fixed sequence of actor firings (i.e. a schedule) that will be repeated indefinitely to execute a reconfigurable dataflow graph.

A quasi-static schedule of a dataflow graph, is a schedule where as many scheduling decisions as possible are made at compile time and only a few data-dependent decisions are left for the runtime manager. The purpose of quasi-static scheduling is generally to increase application performance by relieving the runtime manager from most of its scheduling computations overhead [9, 12].

In practice, a quasi-static schedule derived from a dataflow graph with a reconfigurable behavior is expressed as a parameterized looped schedule [9, 35]. Formally, a parameterized looped schedule S is noted $S = (I_1 I_2 ... I_n)^{\lambda}$ where λ is an integer repetition count whose expression may depend on parameter values, and instruction I_i represents either the firing of an actor $a \in A$, or a nested parameterized

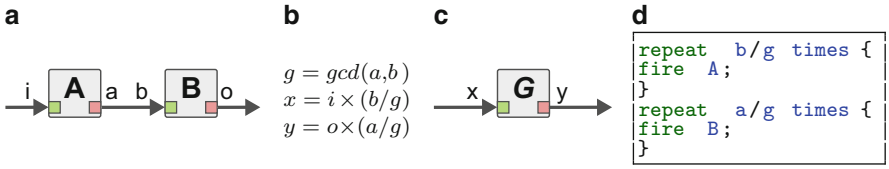


Fig. 9 Basic grouping operation used to build a quasi-static schedule. (**a**) Original pair of actors. (**b**) Equations. (**c**) Resulting actor. (**d**) Pseudo-code for G

looped schedule. The set of instructions $I_1 I_2 \ldots I_n$ of a parameterized looped schedule S is called the body of S. For example, $A(B(CD)^2E)^p$ is a parameterized loop schedule with two nested loops, where actor A is executed once, followed by p executions of the sequence of actor firings $(BCDCDE)$. A quasi-schedule is valid only if all loop iteration counts remain constant throughout firing of their associated body. When building a quasi-static schedule from a reconfigurable dataflow graph, validity of the schedule is usually enforced by the safe reconfiguration requirement of the dataflow graph (see Sect. 2.1).

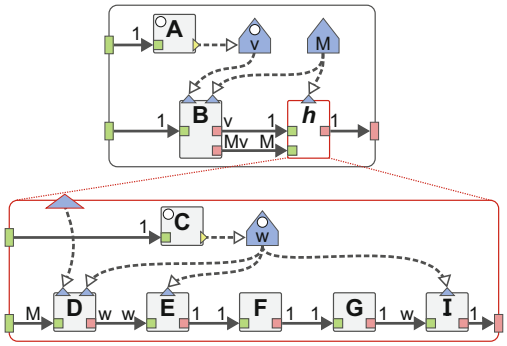
An algorithm to build a quasi-static schedule from a PSDF (or πSDF) graph is given in [9]. This algorithm, called *Parameterized Acyclic Pairwise Grouping of Adjacent Nodes (P-AGPAN)*, is an extension of a scheduling algorithm for SDF graphs whose purpose is to build a schedule minimizing both code size and memory footprint allocated for FIFOs. The basic operation of the P-AGPAN algorithm, illustrated in Fig. 9, is to select a pair of actors connected with a FIFO, and to replace them with a composite actor whose internal behavior is defined with a parameterized looped schedule. For example, the pair of actors A and B from the graph of Fig. 9a can be replaced with the composite actor G presented in Fig. 9c. Using equations from Fig. 9b, the internal behavior of actor G can be represented with the following parameterized looped schedule: $A^{b/g}B^{a/g}$. Pseudo-code corresponding to the internal behavior of the composite actor G is presented in Fig. 9d.

Applying the P-AGPAN algorithm to a PSDF or a πSDF (sub)graph consists of iteratively selecting a pair of actors and applying the pairwise grouping operation, until all actors of the considered subgraph are merged. The order in which pair of actors are selected for grouping influences code size and memory requirements of the generated quasi-static schedule. To minimize code size and memory requirements [9], priority is given to actor pairs connected with:

1. a single-rate FIFO (i.e. a FIFO with equal but possibly parameterized production and consumption rates),
2. a FIFO associated to constant rates,
3. a FIFO associated with parameterized rates whose *gcd* can be computed statically.

If a subgraph contains non-connected actors, as is the case with configuration actors and other actors of a πSDF graph, then these actors are added to the quasi-static schedule in the execution order imposed by the execution rules of the MoC. The P-AGPAN algorithm is applied in a bottom-up approach to all subgraphs of an application, starting from the innermost subgraph up to the top-level graph.

a



b

```
/* top-level graph */
fire A;    //Sets v
fire B;
repeat v times{
  /* h */
  fire C; //Sets w
  repeat w times{
    fire D;
    fire E;
    fire F;
    fire G;
  }
  fire I;
}
```

c

	Grouped actor pair:		Schedule:		Simplified schedule:	
1.	D	& E	$\rightarrow D^1E^1$	$\rightarrow DE$		
2.	F	& G	$\rightarrow F^1G^1$	$\rightarrow FG$		
3.	(DE)	& (FG)	$\rightarrow (DE)^1(FG)^1$	$\rightarrow DEFG$		
4.	$(DEFG)$	& I	$\rightarrow (DEFG)^wI^1$	$\rightarrow (DEFG)^wI$		
5.	C	& $((DEFGI)^wI)$	$\rightarrow C^1((DEFGI)^wI)^1$	$\rightarrow C(DEFGI)^wI$	$= h$	
6.	B	& h	$\rightarrow B^1h^v$	$\rightarrow Bh^v$	$= B(C(DEFGI)^wI)^v$	
7.	A	& Bh^v	$\rightarrow A^1(Bh^v)^1$	$\rightarrow ABh^v$	$= AB(C(DEFGI)^wI)^v$	

Fig. 10 Quasi-static scheduling for the LTE π SDF graph of Fig. 4. (**a**) Simplified π SDF graph with 1-letter actor names. (**b**) Quasi-static pseudo-code. (**c**) Step-by-step construction of the quasi-static schedule using algorithm from [9]

Figure 10 illustrates the application of the P-AGPAN algorithm to the LTE π SDF graph from Fig. 4. Figure 10a presents a simplified version of the π SDF graph with 1-letter actor and parameter names. Figure 10c details the step-by-step execution of the P-AGPAN algorithm for this π SDF graph. The first column of Fig. 10c presents the pair of actors selected for grouping at each step, the second column presents the internal parameterized looped schedule of the resulting composite actor, and the last column present the same schedule with simplified notations. Steps 1 to 5 correspond to the application of the P-AGPAN algorithm to the hierarchical subgraph of actor h, and steps 6 and 7 to the top-level graph. The pseudo-code corresponding to the quasi-static schedule $AB(C(DEFGI)^wI)^v$ is presented in Fig. 10b.

An algorithm for building a quasi-static schedule for a SPDF graph is given in [22]. Briefly, this algorithm first consists of computing the symbolic repetition vector of an SPDF graph, and then finding an ordering of all parameters p_i such that $\frac{\#M(p_{i+1})}{\alpha(p_{i+1})} = f_i \cdot \frac{\#M(p_i)}{\alpha(p_i)}$, where: $\#M(p_i)$ is the repetition count of the emitter of p_i, $\alpha(p_i)$ is its change period, and, finally, f_i a parametric expression. Once this order is established, the quasi-static schedule is obtained by ignoring FIFOs with delays, and constructing a schedule of actors in topological order, where each actor X is written: $(((X^{f'_{N+1}})^{f'_N})\ldots)^{f'_1}$, where f'_i are expressions depending on the N

parameters used and modified by X [22]. In the constructed quasi-static schedule, which is not a parameterized loop schedule, each parenthesis corresponds to a new value of the parameters used or set by the actor. For example, the quasi-static schedule for the SPDF graph of Fig. 5 with explicit set and get of parameter values is $(A; set\ p)^3(get\ p; (B^{2p}; set\ q))^3(get\ p; C^3)^3(get\ p; (get\ q; D^{4p}))^3(get\ p; (get\ q; E^2))^3 F$ (where all ^{1} exponent were omitted). An equivalent parameterized looped schedule can be obtained by factorizing parenthesis with equivalent exponents and matching set/get: $((AB^{2p}C^3(D^{4p}E^2))^3 F$.

Further works on quasi-static scheduling include a technique for dynamic dataflow graphs that exhibit reconfigurable behavior [12]. This technique consists of pre-computing a multicore schedule for static subparts identified in the application. The static multicore schedules are then triggered dynamically at runtime. Another interesting work on quasi-static scheduling is presented in [35], where compact representation of parameterized looped schedules is studied to speed-up execution of quasi-static schedules, and reduce their memory footprints.

3.2 Multicore Runtime for π SDFs Graphs

As presented in previous section, analysis techniques can be used to make scheduling decisions at compile-time for dataflow graphs with a reconfigurable behavior. Nevertheless, since reconfigurable dataflow MoCs are inherently non-decidable, executing them still requires making some deployment decisions, like mapping or memory allocation, at runtime. For example, when executing a quasi-static schedule, although the parameterized execution order of actor firings is known, these firings still need to be mapped on the cores of an architecture, and memory still need to be dynamically allocated for the FIFOs whose number and sizes are only known when a reconfiguration occurs. Another important task to perform at runtime is the verification that values dynamically assigned to parameters constitute a valid configuration for the application [8, 17].

A first way to provide runtime support to a reconfigurable dataflow graph is to use implementation techniques supporting dynamic dataflow MoCs [64]. The main drawback with this approach is that it does not exploit the runtime predictability of a reconfigurable MoC to make smart decisions. For example, a commonly used strategy for executing a dynamic dataflow graph is to implement each actor as an independent process that checks the content of its input and output FIFOs to decide whether it should start a new firing. Because of this limited knowledge of the graph topology and state, actors will waste a lot of processor time and memory bandwidth only to check, often unsuccessfully, their firing rules [13, 20].

In reconfigurable dataflow MoCs, predictability is achieved by exposing the parameterizable firing rules of actors as part of the graph-level semantics (Sect. 2). Building on this semantics, a runtime manager can exploit the predictability of reconfigurable graphs to make smart mapping, scheduling, and memory allocation decisions dynamically.

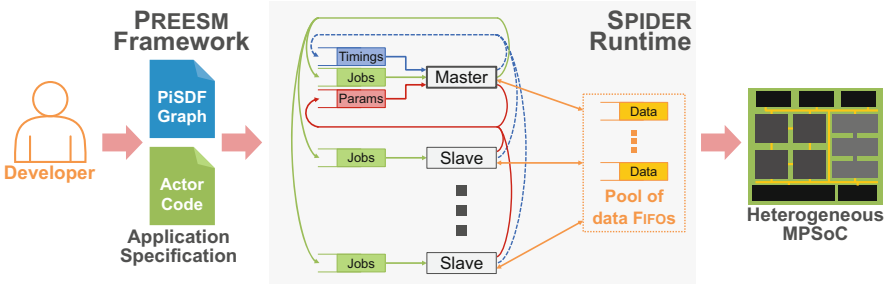


Fig. 11 Overview of the SPIDER runtime workflow and structure [29, 51]

SPIDER (*Synchronous Parameterized and Interfaced Dataflow Embedded Runtime*) is a Real-Time Operating System (RTOS) whose purpose is to manage the execution of πSDF graphs on heterogeneous Multiprocessor Systems-on-Chips (MPSoCs) [29]. As presented in Fig. 11, πSDF graphs and source code associated to actors, generally coded in C language, are designed by the application developer using the Parallel Real-time Embedded Executives Scheduling Method (PREESM) rapid prototyping framework [51]. The internal structure of the SPIDER runtime, is based on master and slave processes where a master process acts as the "brain" of the system, and distributes computations to all the slave processes. In heterogeneous architectures, the master process is generally running on a general purpose processor and slave processes are distributed on multiple types of processing elements, such as general purpose processors, digital signal processors, and hardware accelerators. As shown in Fig. 11, the communications and synchronizations between the master and slave processes are supported by a set of FIFO queues with dedicated functionality. Following πSDF execution rules, the master process maps and schedules each actor firing individually on the different processes, slave or master, by sending so-called job descriptors to them through dedicated *job queues*. A job descriptor is a structure embedding a function pointer corresponding to the fired actor, the parameters values for its firing, and references to shared *data queues* where it will consume and produce data-tokens. When a job corresponding to a configuration actor is executed by a slave or master process, it sends new parameter values to the master process through a dedicated *parameter queue*. Optionally, a *timing queue* may be used to send execution time of all completed jobs to the master process, for profiling and monitoring purposes.

Figure 12 illustrates how SPIDER dynamically manages the execution of a πSDF graph on a multicore architecture. The input πSDF graph considered in this example is depicted in Fig. 12a, b illustrates an intermediate graph resulting from graph tranformations applied at runtime by the SPIDER runtime, prior to mapping and scheduling operation. The Gantt diagram corresponding to an iteration of this graph on 2 cores is presented in Fig. 12c. The master process of the SPIDER runtime is called at the beginning of the execution, in order to map and schedule the *ConfigSize* configuration actor that is, at this point, the only executable actor

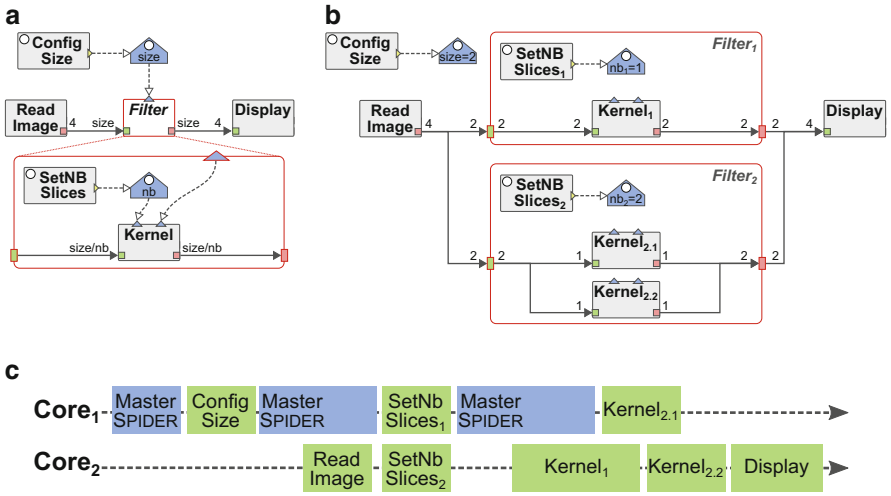


Fig. 12 Deployment process of the SPIDER runtime. (**a**) Input πSDF graph. (**b**) Intermediate single-rate graph. (**c**) Gantt diagram of one iteration of the πSDF graph with SPIDER

according to πSDF execution rules. When executed, the *ConfigSize* actor sets a new value for parameter *size* triggering a reconfiguration, and a second call to the master process. Using the new value of the *size* parameter, the master process computes the repetition vector of the top-level graph and applies a single-rate graph transformation in order to expose its data-parallelism. With $size = 2$, the single-rate transformation duplicates the *Filter* actor to make the data-parallelism of its two firings explicit. At this step, memory can be allocated for all FIFOs of the top-level graph, and the *ReadImage* actor can be scheduled. Concurrently to the execution of the *ReadImage* actor, the master process continues its execution to manage the execution of the two instances of the *Filter* hierarchical actor. The master process manages separately the two subgraphs of the *Filter* actors, and schedules a firing of the *SetNBSlices* configuration actor for each of them. During the third and final call to the master process, a new configuration of each of the two subgraphs is taken into account to compute their respective repetition vectors, to perform the single-rate graph transformation on them, to allocate all FIFOs in memory, and to schedule all remaining actor firings. The single-rate graph of Fig. 12b is the executed graph for parameter values $size = 2$, $nb_1 = 1$, and $nb_2 = 2$.

Compared to implementation strategies with no global management of applications, using a runtime manager to control the execution of a reconfigurable graph has an overhead on application performance. Indeed, such runtime manager requires processor time to compute repetition vectors, to perform graph transformations, and to map and schedule actor firings. Nevertheless, as presented in [28, 29], even with large reconfigurable graphs with several hundreds of actors, this overhead is largely compensated by the efficiency of the scheduling decisions, and generally outperforms dynamic deployment strategies with no global manager.

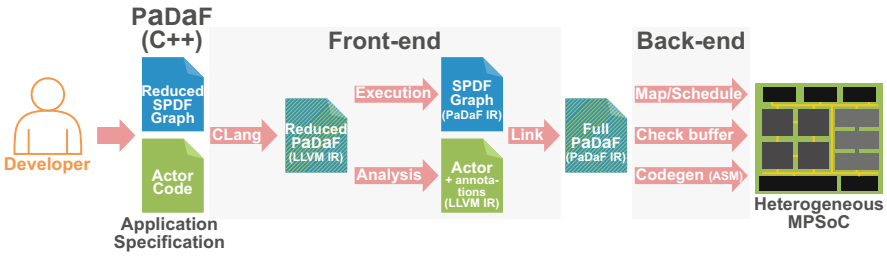
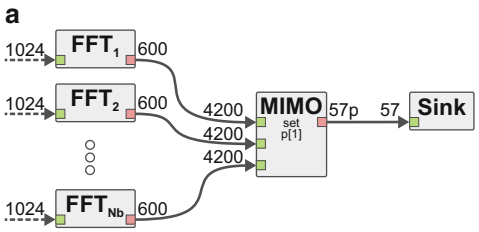


Fig. 13 Overview of the compilation flow for SPDF graphs [16]

3.3 Compilation Flow for SPDF Graphs

A compilation framework for deploying reconfigurable applications specified with the SPDF MoC onto heterogeneous MPSoCs is presented in [16]. This development flow for SPDF graphs differs from the flow based on the runtime manager presented in the previous section in that it shifts most of the deployment decisions to the compilation framework. In particular, in the SPDF compilation flow, actors are manually mapped on the cores by the application designer, and actor firings are quasi-statically scheduled based on a compile-time analysis of the graph and the behavior of actors. An overview of the different stages of the SPDF compilation flow, adopting the elements presented in Chapter [39], is illustrated in Fig. 13.

In the SPDF compilation flow, the specifications of both the SPDF graph and the internal behavior of actors are based on a hierarchy of specialized C++ classes called Parametric Dataflow Format (PaDaF). Figure 14 illustrates the syntax used to specify part of an LTE application with PaDaF [16]. The SPDF graph presented in Fig. 14a contains a parameterizable number Nb of *FFT* actors, each processing 1024 samples received from an antenna. Results of the *FFT* actors are then transmitted to a *MIMO* actor, which sets a new value for parameter p at each firing, and produces a reconfigurable number of data tokens towards the *Sink* actor. As shown in the C++ description of the LTE graph in Fig. 14b, PaDaF allows the description of SPDF graphs using `for-loop` constructs. Hence, the PaDaF syntax specifies SPDF graphs in a reduced format where a statically parameterizable number of actors and FIFOs may be instantiated and connected together. This syntax is similar to the SigmaC programming language used for the specification of decidable CSDF graphs [25]. The PaDaF code corresponding to the *MIMO* actor is presented in Fig. 14c. As can be seen in this example, actor specification is based on a hierarchy of C++ classes used to specify actors (`Actor` class), their data ports (`PortIn/PortOut` classes), and their parameters (`ParamOut` class). As in graph descriptions, control code can be used to specify a statically parameterizable number of data ports when specifying an SPDF actor with PaDaF. The internal computations performed by an actor at each firing are specified in its unique `compute()` function.



b

```
Fft fft[NB_ANT];
Mimo mimo;
Sink sink;
for(i=0; i<NB_ANT; i++){
  fft[i].out <= mimo.in[i];
}
sink.in <= mimo.out
```

c

```
class Mimo : public Actor {
  std::vector<PortIn<int>*> Iin;
  PortOut<int> Iout;
  ParamOut p;
  int nbAnt
  Mimo(int nb):  p(1), out(p*57) {
    nbAnt = nb;
    for(int i=0; i<nb; i++){
      Iin[i] = new PortIn<int>(4200);
    }
  }
  void compute();
}
```

Fig. 14 Partial LTE application specification with PaDaF (from [16]). (**a**) SPDF graph. (**b**) PaDaF graph description. (**c**) PaDaF *Mimo* actor code

The steps composing the compilation flow presented in Fig. 13 are sorted into two groups, the front-end and the back-end, presented hereafter.

- *Front-end*: The first steps of the SPDF compilation flow, called the front-end, are architecture-independent operations responsible for exposing coarse and fine grain properties of the application.
  - *Clang*: The first step of the front-end is to compile the graph and the actor PaDaF specifications of an application into an *LLVM* Intermediate Representation (IR) using the *Clang* compiler [36].
  - *Execution*: In a second step, the produced *LLVM* IR corresponding to the reduced graph description is executed in order to build the complete SPDF graph, where a parameterizable number of actors are instantiated.
  - *Analysis*: In the third step of the front-end, executed in parallel with the second, the *LLVM* IR corresponding to the internal behavior of actors is analyzed in order to detect and annotate the instructions responsible for pushing and popping data into the FIFOs connected to each actor.
  - *Link*: In the last step of the front-end, a full PaDaF IR of the application is obtained by linking the complete SPDF graph with annotated actor code, as detailed in [16].
- *Back-end*: The latter steps of the compilation flow, called the back-end, are responsible for deploying the application on a specified heterogeneous architecture.

- *Map/Schedule*: Although mapping of the actors is currently manually specified by the application developer, the back-end is still responsible for producing a quasi-static schedule of actor firings for each core.
- *Check buffer*: An analysis of the proposed mapping and scheduling is also used to check that the memory capacity of the targeted platforms are not exceeded.
- *Codegen (Assembly (ASM))*Finally, the annotated internal data access patterns of actors exposed in the PaDaF IR are used in the code generation step in order to generate calls to on-chip communication primitives, and to produce efficiently pipelined ASM code.

The efficiency of this SPDF compilation flow is demonstrated in [16] for the deployment of several Software Defined Radio (SDR) applications on a domain-specific MPSoCs. The performance of the synthesized software for evaluated SDR applications is shown to be equivalent to handwritten code.

3.4 Software Reconfiguration for Dynamic Dataflow Graphs

Applications specified with dynamic dataflow MoCs do not, in general, exhibit a safe reconfigurable behavior as defined in Sect. 2.1. Hence, software implementation techniques presented in previous sections that exploit the compile-time and runtime predictability of reconfigurable behavior can not, in general, be applied to dynamic dataflow graphs. This section presents how a design flow integrating reconfigurable software components can be used to improve the implementation of dynamic dataflow graphs. Here, a software component designates an application independent piece of software supporting the execution of a dynamic dataflow graph by managing and monitoring its deployment onto a target architecture. Reconfigurability of these software components comes from their capability to change their deployment strategies at runtime in order to impact a performance indicator (e.g. latency, energy consumption, ...), in a controlled and predictable way.

Figure 15 presents an overview of an energy-aware design flow, proposed in [55], for video decoding applications specified with dynamic dataflow graphs. The design flow is composed of three main parts:

1. a modular specification of the application based on a dynamic dataflow MoC [64],
2. reconfigurable software components controlling the energy of the deployed application
3. a hardware platform embedding energy and performance sensors

RVC-CAL [10] is a language standardized by the Motion Picture Expert Group (MPEG) committee to specify video decoders. In RVC-CAL, a set of widely used video decoding basic building blocks, like discrete cosine transforms, variable length coding algorithms, or deblocking filters, can be composed into a network

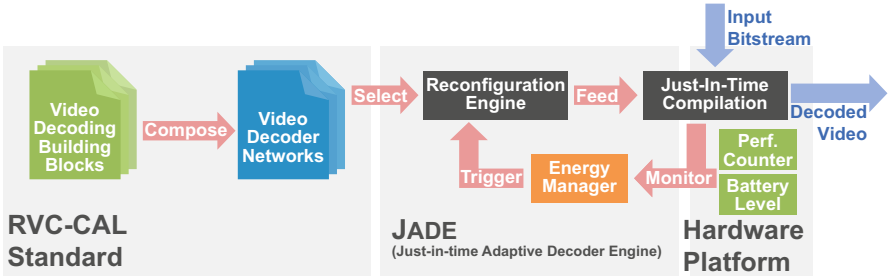


Fig. 15 Overview of an energy-aware design flow for dynamic dataflow graphs [55]

in order to specify a complete video decoder. Network specified with RVC-CAL implements the Dataflow Process Network (DPN) semantics [38], which is a non-deterministic model close to the CFDF MoC. RVC-CAL descriptions of several video decoders with diverse computational complexity are used as inputs of the design flow presented in Fig. 15.

The Just-in-time Adaptive Decoder Engine (JADE), which constitutes the second stage of the design flow presented in Fig. 15, is a software component built with *LLVM* [24]. The main purpose of JADE is to manage on-the-fly the execution of platform-independent RVC-CAL networks onto various hardware platforms. To do so, JADE translates a selected network of RVC-CAL actors into the *LLVM* IR, and feeds it to the just-in-time *LLVM* compiler and interpreter for the targeted platform.

The energy manager that was integrated within JADE in [55] is the key reconfigurable software component of the design flow presented in Fig. 15. The first objective of the energy manager is to monitor the execution of the application on the targeted platform in order to build an energy model of its energy consumption [55]. Monitoring of the application is achieved by automatically inserting calls to instrumentation functions reading performance monitoring counters of the targeted platform. Using the energy model built from monitoring information, the energy manager is able to estimate precisely the energy consumption of the different video decoders at its disposition. The second responsibility of the energy manager is to control the energy consumption of the system by triggering reconfigurations of the currently executed network. A typical scenario for reconfiguration occurs when the energy manager estimates that the remaining battery charge is insufficient to finish decoding a video stream of known length. In such a scenario, the energy manager may trigger a reconfiguration to a different network with lower computational complexity and lower quality, but which will reduce energy consumption. Practical evaluation of this energy-aware design flow [55] shows the efficiency of this approach on the latest HEVC video standard.

Further work on the use of software reconfiguration techniques for implementing dynamic dataflow MoCs is presented in [68]. In this work, a low-cost monitoring of the execution all actors of a DPN is used to obtain statistics on their execution time. Using this monitoring information, a runtime manager may trigger a reconfiguration of the mapping of the different actors on the different Processing Element (PE) of the targeted architecture.

Fig. 16 Overview of the classical computing spectrum: performance versus flexibility

4 Dataflow-Based Techniques for Hardware Reconfigurable Computing Platforms

Flexibility and adaptivity of a signal processing system at the hardware level may be achieved by means of reconfigurable computing. In recent years, reconfigurable computing has become a popular hardware design paradigm for accelerating a wide variety of applications. Hardware reconfiguration is commonly used as a way to enable kernel execution over specialized and optimized circuits, retaining much of the flexibility of a software solution.

Figure 16 depicts a very general overview of the classical computing spectrum, whose extremes are represented by Central Processing Units (CPUs)—generic and extremely flexible—and Application Specific Integrated Circuits (ASICs)—dedicated, non programmable, circuits customized and highly optimized for a given functionality. CPUs are capable of executing any type of code their compilers can translate into machine code, with average performance and very limited optimization capabilities. The flexibility of CPUs comes at the expense of—medium to highly—complex hardware micro-architectures requiring a significant amount of silicon area for their implementation to be able to serve a complete instruction set. In ASIC designs, resources are minimized since the architecture is forged accordingly to the native execution flow of the implemented application, while operating frequency, throughput and energy consumption may be optimized according to the given constraints. A counterpart of this highly optimized design is that ASICs are not flexible at all, as their hardwired datapath can execute no other function than the one they are meant for. Reconfigurable computing infrastructures lays in between, representing an appealing option since they are capable of guaranteeing a trade-off among the aforementioned extremes. In practice, reconfigurable hardware allows customizing the execution infrastructure by allowing runtime (re-)programmability of datapaths to implement application-specific datapaths, thus providing flexibility. Switching and programmability capabilities determine the type of implementable reconfiguration that, as discussed hereafter, can take place at different granularities.

Studies on the subject of reconfigurable hardware dates back to nineties and have been surveyed in several different works along time [15, 27, 34, 62, 63, 65]. Reconfigurable hardware guarantees different degrees of flexibility, being (re-)programmable over a given set of functionalities, but still offering specialization advantages. Nevertheless, as in any specialized design, programmability design does

not come for free: it requires the programmers to have a deep knowledge and understanding of the architectural details. This drawback traditionally limited the wide usage of reconfigurable computing systems. Field Programmable Gate Array (FPGA) platforms, for example, were typically considered merely as development boards for prototyping activities, rather than an actual target.

The main purpose of this section is to understand how dataflow-based specifications and design flows can be used to facilitate the design of reconfigurable computing infrastructures. Before that a bit of terminology has to be introduced. Different types of classification of reconfigurable computing systems are available. We will refer hereafter mainly to *coarse grained* (CG) and *fine grained* (FG) reconfigurable platforms. These two types of architectures differ for the size of the hardware blocks that are reconfigured.

- Coarse Grained (CG) reconfigurable computing systems involve a fixed set of—often programmable—Processing Elements (PEs) connected by means of dedicated routing blocks [15]. The basic idea behind these systems is to maximize resource re-use among different target applications. PEs are managed in a time multiplexed manner to serve different functionalities at different execution instants. The number of interchangeable scenarios is typically fixed and at runtime the system can switch from one execution to another.
- Fine Grained (FG) reconfigurable computing systems can execute a theoretically infinite number of different functions, since programmability takes place at the single bit level. An example of this kind of platforms is provided by FPGAs that are programmable both at design time and at runtime. Change of context while executing requires the support of partial dynamic reconfiguration [2, 66], meaning that part of the executed bitstream is re-loaded with a previously generated configuration(s) stored on a dedicated memory accessed from a configuration module.

4.1 Dataflow-Driven Coarse Grained Reconfiguration

Coarse grained reconfigurable systems, as already said, rely on the execution of a set of different applications on the same hardware substrate, typically composed of several PEs that are highly re-usable to implement various target specifications. CG reconfiguration strategies can be adopted both on ASIC and FPGA technologies and, being extremely modular, the instantiated PEs can be deeply optimized.

As shown in Fig. 17, the set of PEs used in a CG reconfigurable systems, can be homogeneous, which means that all PEs are identical computing blocks, or heterogeneous, which means that PEs are application specific, not identical, computing blocks. Moreover, the computing fabric may not necessarily be composed of a regular infrastructure, i.e. the communication backbone will include as many links as needed and will not be based on a fully connected grid infrastructure as in array-based systems. PEs are normally not constrained in terms of computing granularity:

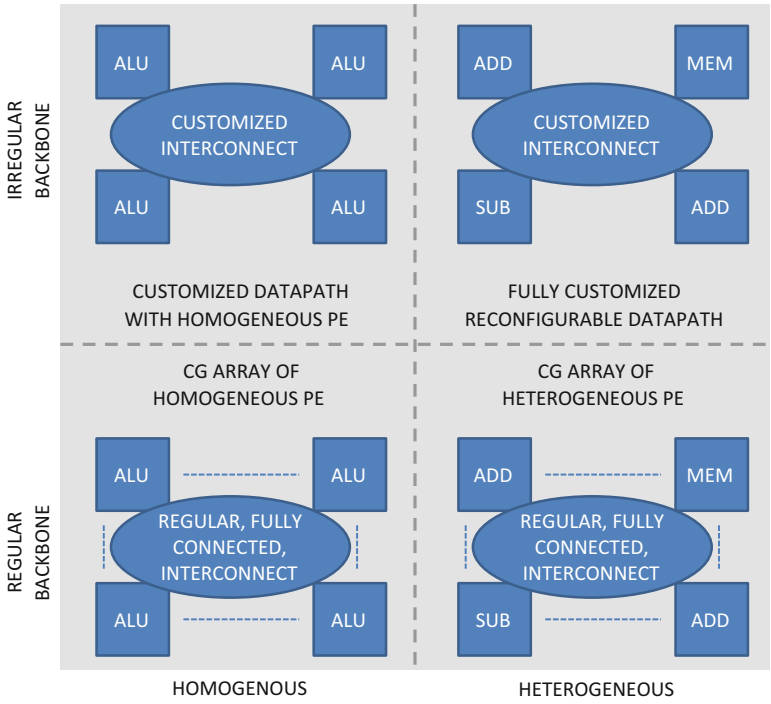


Fig. 17 CG reconfigurable architectures: classification

they can range from a simple ALU to a complex discrete cosine transform in a Video Codec platform. The work of Beaumin et al. on multi-context accelerator [5] can be considered as a first attempt to combine dataflow specifications and reconfigurable computing concepts. In the RVC-CAL context, Beaumin et al. [5] propose a reconfigurable co-processor in charge of executing different Dataflow Process Network (DPN). The design of the reconfigurable co-processor is based on a set of heterogeneous PEs, called network units, where each of network unit is capable of executing a different kernel represented by means of DPN. Each network unit instantiates as many processing units as needed to implement the CAL actors in a given input DPN; these processing units communicates together by means of communication channels, provided in hardware as FIFO queues. An example of network unit is depicted in Fig. 18. Each network unit is configurable at design time so that each processing unit can be customized to execute different actors, as well as the interconnection among them. Indeed, the interconnection infrastructure needs to enable every processing unit to be connected to every FIFO, which is achieved by leveraging on a full mesh infrastructure configured for every CAL network deployed on the co-processor. This type of reconfigurable architecture can be classified among the CG reconfigurable one, where the network units are the PEs. Moreover, it may be considered heterogeneous since different numbers of processing units and FIFOs

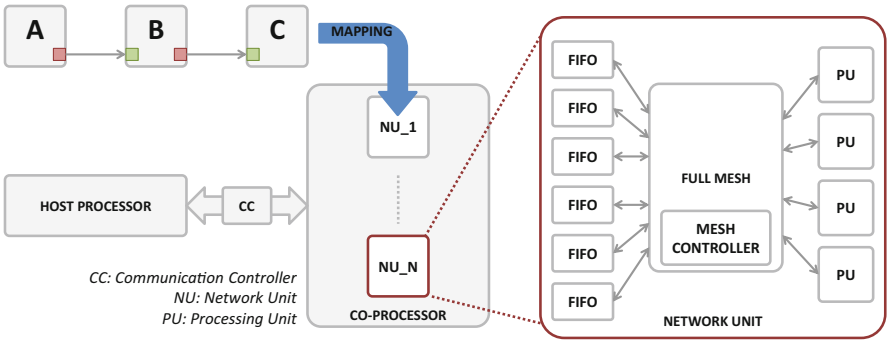


Fig. 18 Dataflow process network based co-processor [5]

can be included in the instantiated network units, according to the DPNs that they respectively implement, and each processing unit can be specialized to implement a different datapath, according to the actor mapped over it.

The more a CG reconfigurable system is customized to fit application needs, avoiding any extra PEs and unnecessary connections and adopting heterogeneous PEs, the more it is possible to maximize its efficiency in terms of power, resources usage and performance. On the other hand, besides design and debug issues,[2] the adoption of CG reconfigurable heterogeneous and irregular platforms is limited by the fact that mapping is not so straightforward. Research efforts have been undertaken to automate the application mapping process [3]. Dimensioning the underlying hardware substrate and efficiently mapping several applications over it are key challenges that can be addressed by combining the dataflow models to the CG reconfigurable approach.

4.1.1 Heterogeneous Coarse-Grained and Runtime Reconfigurable Architectures

To address the mapping problem, one possibility is provided by datapath merging (DPM) techniques, whose primary goal is to minimize the number of PEs and communication links integrated into a CG reconfigurable datapath. Given different input datapaths, described as dataflow graphs, DPM combines the graphs into a unique specification with minimal nodes and connections. Exploiting a graph-based formalism favours resource re-use, both in terms of hardware modules (representing the different nodes of the given specification) and interconnects (representing the edges among the nodes). The outcome of this procedure is a reconfigurable datapath graph that can be synthesized in hardware according to a one-to-one mapping between graph nodes and hardware modules.

[2] As for any highly specialized system designers are required to define all the micro-architecture at Register Transfer Level (RTL). All the details about the functionality-set to be implemented have to be known and a lot of effort is spent in coding and debugging.

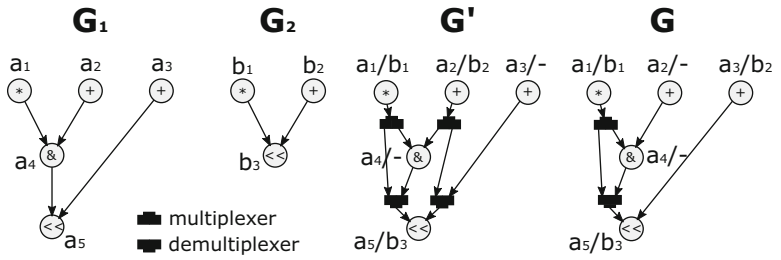


Fig. 19 The datapath merging problem [60]

Figure 19, from the work of Souza et al. [60], provides an example of DPM among two input graphs, G_1 and G_2. G and G' represent two possible solutions of the DPM problem, where the switching elements, that allows to share resources among G_1 and G_2, are indicated as multiplexers and demultiplexers. Resources are minimized both in G and G', as they present the same number of nodes and both map the couples a_1/b_1 and a_5/b_3 over the same resources, but they have a different amount of edges. In G edges are minimized since there is one more shared resource that couples a_3/b_2 and, in turn, allows sharing the link between a_3/b_2 and a_5/b_3 to connect the a_3 to a_5 (see G_1) and b_2 to b_3 (see G_2).

In the literature, several heuristic methods have been proposed to solve the DPM problem. Moreano et al. [43] solve it as a *maximum clique problem* over a *compatibility graph*. Nodes are compatible if they can be implemented by the same hardware resource. The graph gives a full overview of the compatibility among different possible mappings between the edges of the given input specifications. Solving the maximum clique problem for the compatibility graph leads to one (or more) mapping(s) capable of maximizing both resources and edges sharing that, in turn, means minimizing the PEs and the interconnections necessary to implement a datapath executing different input graph-based specifications. Two input graphs at a time are merged so, having N input specifications implies to solve the DPM problem N-1 times. The DPM problem is NP-complete, and it is currently impossible to find an optimal solution with a polynomial complexity algorithm.

A polynomial time heuristic algorithm is adopted to solve it in [4]. Another approach, proposed by Huang et al. in [31], solves the DPM problem using a bipartite matching heuristic method. Two graphs at a time are considered and all the possible mappings are weighted according to the number of sharable connections, then maximum weights drive a certain merging solution.

In the RVC-CAL context, Palumbo et al. used DPM techniques [46] to create runtime reconfigurable CG substrates, to be used as stand-alone reconfigurable systems [58] or within application-specific accelerators [57]. In those works the combination of the dataflow models and the CG reconfigurable design paradigm is quite straightforward: each actor is mapped over a single and atomic PE, and multiple input dataflows are combined together over the same substrate adopting a DPM approach. In this way different input specifications share, where convenient,

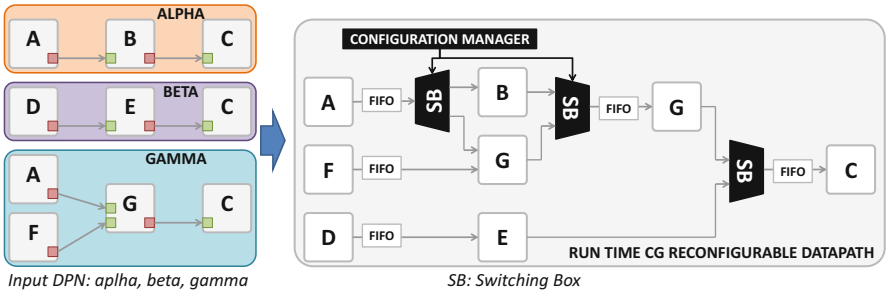


Fig. 20 Dataflow to CG reconfigurable substrate [46]

common PEs that are accessed through programmable switching elements, named *switching boxes* (SBs). SBs are placed at the crossroads among different paths of data, forking or joining the execution flow of the different input networks. Figure 20 provides an example of the CG reconfigurable datapath that may be created leveraging on such an approach. Three different input specifications are mapped over the same substrate that is capable of executing them one at a time, by switching from one configuration to another. The configuration manager drives the SBs according to the requested execution. This type of reconfigurable architecture is a CG one by definition and the constituting PEs, whose granularity depends on the actors in the given input specifications, are heterogeneous. To facilitate the automatic definition of such an architecture, starting from RVC-CAL DPN input specifications, it is possible to rely on the RVC-CAL compliant design flow presented in [58]—and depicted in Fig. 21. In this design flow, an set of tools is adopted to compose, optimize and synthesize the RTL description of the runtime reconfigurable system. The CG reconfigurable datapath is assembled using the following tools:

- the Multi-Dataflow Composer tool [47, 49]—capable of creating a multi-functional high level description of a CG reconfigurable system applying datapath merging techniques on a set of input DPNs specifications
- Xronos [7]—capable of providing High Level Synthesis (see Sect. 4.2) from CAL to RTL of each single actor of the given input DPN
- TURNUS [14]—capable of optimizing the system, by means of high-level profiling, to provide the optimal FIFO sizing (in the multi-dataflow case worst case sizing is assigned).

RVC-CAL compliant reconfigurable architectures, assembled as in [58], can be used as the CG reconfigurable processing core of a co-processing unit, as the one presented in Fig. 21 [57].

A DPM-based technique has been used also in a recent work of Edwards et al. [19]. They use the concept of *compositional hardware circuits* and exploit Kahn Networks to merge and implement in hardware different dataflow networks (Fig. 22).

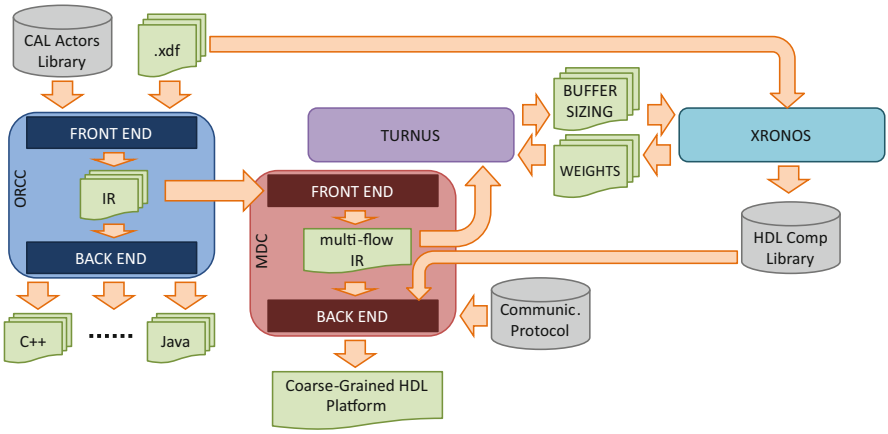


Fig. 21 Design environment for RVC-CAL compliant CG reconfigurable architectures [58]

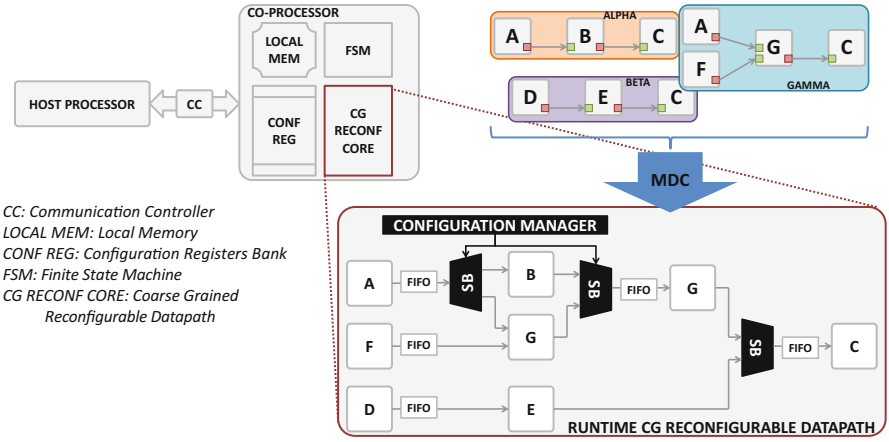


Fig. 22 Multi-Dataflow Composer (MDC) based reconfigurable co-processor [57]

4.1.2 Coarse-Grained and Runtime Reconfigurable Arrays

Another type of coarse-grain reconfigurable architecture combines a host processor controller with an array of PEs. These PEs, which are typically small and simple like ALUs, are connected together in the array with some local memories. CG reconfigurable arrays are commonly used for efficient implementations of streaming systems [1, 18, 41]. These architectures often exploit imperative programming approaches to map and control the flow of data. The examples hereafter demonstrate how dataflow models can be exploited for the same purposes.

WaveScalar, presented by Swanson et al. in [61], is a dataflow-based reconfigurable architecture that contains a pool of PEs, to which are dynamically assigned

instructions: the WaveCache loads instructions from memory and assigns them to PEs for execution. While instruction scheduling is dynamic and out-of-order, the reference application is described as a dataflow graph. Basically, the WaveScalar compiler translates the input imperative programs into a dataflow description, used as the target code. WaveScalar does not implement dataflow-driven reconfiguration, but it is certainly one of the first attempts to combine dataflow models with the CG reconfigurable paradigm.

Galanis et al. have exploited a dataflow-based approach to map different functionalities over the proposed reconfigurable computing array in [23]. In their work the host processor is connected to a co-processing unit, implemented by means of a reconfigurable datapath, where: (1) PEs are an ALU and a multiplier that can take operands from other nodes or from a register bank; and (2) the interconnection among PEs is a full crossbar (or a fat-tree network if scalability issues may arise). Sub-graphs of the parts of the application to be accelerated are mapped over the reconfigurable datapath and the control is generated by an embedded FPGA, which support the overall control flow.

Niedermeier et al. [45], targeting streaming applications, exploited dataflow principles for controlling the flow of data and configuring PEs of their CG architecture. In particular, they configure each PE, including memory blocks, by means of a finite state machine, whose stages are defined as dataflow actors with input and output token patterns. Digital Signal Processing (DSP) applications are particularly suitable to be implemented using such an approach.

Huang et al. [30] adopted dataflow-based control in order to manage complex scheduling situations throughout the propagation of control tokens along with the data to be processed. Such a self control strategy allows to relax mapping and management issues, leveraging on a distributed approach and on a dynamic dataflow control, getting rid of static scheduling. In this way, complex scheduling situations due to latency variations (e.g. when a memory access occurs) are handled transparently with token propagation.

4.2 Fine-Grained Dataflow-Driven Reconfiguration

Bit-level reconfigurability, traditionally required designers to have a deep knowledge of the hardware design flow and hardware description languages. In the last few years, High Level Synthesis (HLS) approaches became popular; they are meant to speed up both hardware and software design process [42]. In particular, from the hardware perspective, one of their advantages is relieving designers from the definition of the RTL description of the system and its components.

A popular commercial HLS synthesizer is Vivado[3] from Xilinx, which accelerates IP creation by enabling C, C++ and System C specifications to be directly

[3] https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

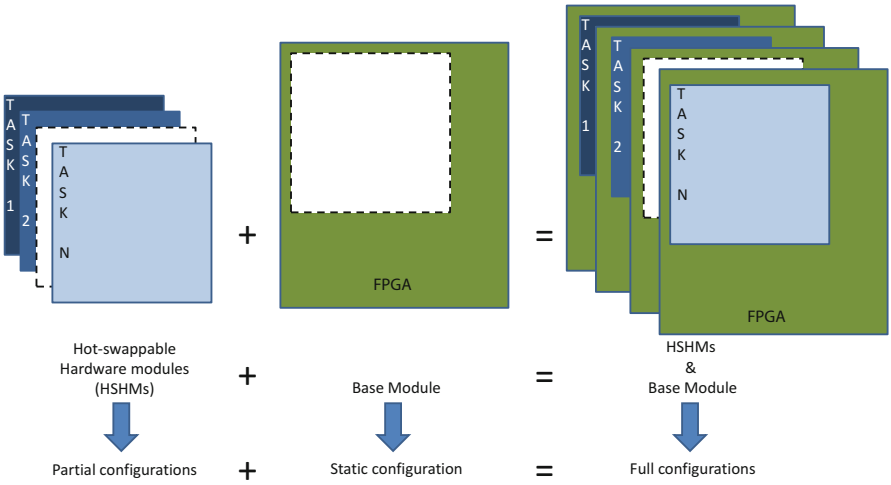


Fig. 23 Spatial and temporal partitioning of an application in a FPGA exploiting DPR [53]

implemented into Xilinx programmable devices without the need to manually create the RTL description. Similarly, but based on dataflow approaches, Xronos [7] and the Caph compiler [59] are meant to provide dataflow-to-hardware generation. Xronos, which has been developed within RVC-CAL context where dataflow networks expressed using the CAL language based on the DPN MoC, is an evolution of the CAL2HDL framework and the work done in [56]. One of the limitation of Xronos is that it generates a target dependent descriptions implementable only on Xilinx FPGA boards. This limitation is not true for the Caph approach that provides target independent RTL descriptions of dataflow compliant systems, which can be implemented both on ASIC and on FPGA.

Dynamic partial reconfiguration (DPR)[4] is defined as the ability to modify, at runtime and while the system is executing, blocks/slots of logic by downloading partial bit files. The remaining logic, i.e. those slots where reconfiguration is not applied, keeps running its execution without any interruption. As depicted in Fig. 23, DSP applications may take advantage of DPR, changing tasks in the pipeline while keeping the overall system functional. In this context, the dataflow paradigm is used to address the problem of ensuring that reconfiguration is performed at a convenient time, minimizing its impact on execution latency and memory footprint. By capturing, at compile-time, the application execution in terms of actors exchanging tokens along communications edges, dataflow networks provide a clear definition of dependences. On top of that, their predictability property, enforced by the semantics and execution rules (see Sect. 2.1), can be exploited

[4] https://www.xilinx.com/products/design-tools/vivado/implementation/partial-reconfiguration.html.

to manage in advance slots reconfiguration order. In particular, Piat et al. in [53] extended static dataflow description with additional properties to provide compile-time analysis of the DPR influence on the system, to be able to assess early in the design stage DPR time slots and memory requirements. To manage network based reconfiguration, the Parameterized SDF (PSDF) MoC is adopted, since it is capable of representing dynamic behaviors at network level. Network parameters impact on scheduling/partitioning and memory footprint changes and, therefore, perfectly matches the DPR needs that requires:

- To evaluate buffering requirements on DPR actor inputs—all the incoming edges of the DPR actor are analyzed and the memory cost of the input path is evaluated.
- To manage slots reconfiguration—a dedicated DPR layer is modelled aside the basic dataflow schedule to represent the reconfiguration instant for each DPR actor, either based on user defined reconfiguration scheme or on network tokens in the case of PSDF model.

In conclusion, as described in this Sect. 4, dataflow specifications may facilitate both mapping and synthesis of coarse and fine grained reconfigurable computing infrastructure to be used in the signal processing domain.

Acknowledgements This work was partially supported by the CERBERO (Cross-layer modEl-based fRamework for multi-oBjective dEsign of Reconfigurable systems in unceRtain hybRid envirOnments) Horizon 2020 Project, funded by the European Union Commission under Grant 732105.

References

1. Advanced Computer Architecture Group - University of California: Morphosys research project. http://gram.eng.uci.edu/morphosys/
2. Altera: Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs (2010)
3. Ansaloni, G., Tanimura, K., Pozzi, L., Dutt, N.: Integrated kernel partitioning and scheduling for coarse-grained reconfigurable arrays. IEEE Trans. on CAD of Integrated Circuits and Systems **31**(12), 1803–1816 (2012). http://dx.doi.org/10.1109/TCAD.2012.2209886
4. Battiti, R., Protasi, M.: Reactive local search for the maximum clique problem 1. Algorithmica **29**(4), 610–637 (2001)
5. Beaumin, C., Sentieys, O., Casseau, E., Carer, A.: A coarse-grain reconfigurable hardware architecture for rvc-cal-based design. In: Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on, pp. 152–159 (2010). https://doi.org/10.1109/DASIP.2010.5706259
6. Bebelis, V., Fradet, P., Girault, A., Lavigueur, B.: Bpdf: A statically analyzable dataflow model with integer and boolean parameters. In: Proceedings of the Eleventh ACM International Conference on Embedded Software, p. 3. IEEE Press (2013)
7. Bezati, E., Mattavelli, M., Janneck, J.: High-Level Synthesis of Dataflow Programs for Signal Processing Systems. In: 8th International Symposium on Image and Signal Processing and Analysis (ISPA 2013) (2013)
8. Bhattacharya, B., Bhattacharyya, S.: Parameterized dataflow modeling for dsp systems. Signal Processing, IEEE Transactions on (2001). https://doi.org/10.1109/78.950795

9. Bhattacharya, B., Bhattacharyya, S.S.: Quasi-static scheduling of reconfigurable dataflow graphs for dsp systems. In: Proceedings 11th International Workshop on Rapid System Prototyping. RSP 2000. Shortening the Path from Specification to Prototype (Cat. No.PR00668), pp. 84–89 (2000). https://doi.org/10.1109/IWRSP.2000.855200
10. Bhattacharyya, S.S., Eker, J., Janneck, J.W., Lucarz, C., Mattavelli, M., Raulet, M.: Overview of the mpeg reconfigurable video coding framework. Journal of Signal Processing Systems **63**(2), 251–263 (2011)
11. Bouakaz, A., Fradet, P., Girault, A.: A survey of parametric dataflow models of computation. ACM Trans. Des. Autom. Electron. Syst. **22**(2), 38:1–38:25 (2017). https://doi.org/10.1145/2999539. http://doi.acm.org.rproxy.insa-rennes.fr/10.1145/2999539
12. Boutellier, J., Lucarz, C., Lafond, S., Gomez, V.M., Mattavelli, M.: Quasi-static scheduling of cal actor networks for reconfigurable video coding. Journal of Signal Processing Systems **63**(2), 191–202 (2011). http://dx.doi.org/10.1007/s11265-009-0389-5
13. Boutellier, J., Sadhanala, V., Lucarz, C., Brisk, P., Mattavelli, M.: Scheduling of dataflow models within the reconfigurable video coding framework. In: 2008 IEEE Workshop on Signal Processing Systems, pp. 182–187 (2008). https://doi.org/10.1109/SIPS.2008.4671759
14. Casale-Brunet, S., Bezati, E., Mattavelli, M., Canale, M., Janneck, J.W.: Execution trace graph analysis of dataflow programs: Bounded buffer scheduling and deadlock recovery using model predictive control. In: Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on, pp. 1–6 (2014). https://doi.org/10.1109/DASIP.2014.7115623
15. Compton, K., Hauck, S.: Reconfigurable computing: A survey of systems and software. ACM Comput. Surv. **34**(2), 171–210 (2002). http://doi.acm.org/10.1145/508352.508353
16. Dardaillon, M., Marquet, K., Risset, T., Martin, J., Charles, H.P.: A new compilation flow for software-defined radio applications on heterogeneous mpsocs. ACM Transactions on Architecture and Code Optimization (TACO) **13**(2), 19 (2016)
17. Desnos, K., Pelcat, M., Nezan, J.F., Bhattacharyya, S.S., Aridhi, S.: Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration. In: Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on, pp. 41–48. IEEE (2013)
18. Dongwook Lee Manhwee Jo, K.H.K.C.: FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability. In: International Conference on Field-Programmable Technology (2009)
19. Edwards, S.A., Townsend, R., Kim, M.A.: Compositional dataflow circuits. In: Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017, pp. 175–184 (2017). http://doi.acm.org/10.1145/3127041.3127055
20. Ersfolk, J., Roquier, G., Jokhio, F., Lilius, J., Mattavelli, M.: Scheduling of dynamic dataflow programs with model checking. In: Signal Processing Systems (SiPS), 2011 IEEE Workshop on, pp. 37–42. IEEE (2011)
21. Ersfolk, J., Roquier, G., Lilius, J., Mattavelli, M.: Modeling control tokens for composition of cal actors. In: Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on, pp. 71–78 (2013)
22. Fradet, P., Girault, A., Poplavko, P.: Spdf: A schedulable parametric data-flow moc. In: 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 769–774. IEEE (2012)
23. Galanis, M.D., Dimitroulakos, G., Tragoudas, S., Goutis, C.E.: Speedups in embedded systems with a high-performance coprocessor datapath. ACM Trans. Design Autom. Electr. Syst. **12**(3), 35:1–35:22 (2007). http://doi.acm.org/10.1145/1255456.1255472
24. Gorin, J., Wipliez, M., Prêteux, F., Raulet, M.: Llvm-based and scalable mpeg-rvc decoder. Journal of Real-Time Image Processing **6**(1), 59–70 (2011)
25. Goubier, T., Sirdey, R., Louise, S., David, V.: σc: A programming model and language for embedded manycores. In: International Conference on Algorithms and Architectures for Parallel Processing, pp. 385–394. Springer (2011)

26. Ha, S., Oh, H.: Decidable signal processing dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
27. Hartenstein, R.W.: Coarse grain reconfigurable architecture (embedded tutorial). In: Proceedings of ASP-DAC 2001, Asia and South Pacific Design Automation Conference 2001, January 30-February 2, 2001, Yokohama, Japan, pp. 564–570 (2001). http://doi.acm.org/10.1145/370155.370535
28. Heulot, J., Boutellier, J., Pelcat, M., Nezan, J.F., Aridhi, S.: Applying the adaptive hybrid flow-shop scheduling method to schedule a 3gpp lte physical layer algorithm onto many-core digital signal processors. In: 2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2013), pp. 123–129 (2013). https://doi.org/10.1109/AHS.2013.6604235
29. Heulot, J., Pelcat, M., Desnos, K., Nezan, J.F., Aridhi, S.: Spider: A synchronous parameterized and interfaced dataflow-based rtos for multicore dsps. In: 2014 6th European Embedded Design in Education and Research Conference (EDERC), pp. 167–171 (2014). https://doi.org/10.1109/EDERC.2014.6924381
30. Huang, Y., Ienne, P., Temam, O., Chen, Y., Wu, C.: Elastic cgras. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, pp. 171–180. ACM, New York, NY, USA (2013). http://doi.acm.org/10.1145/2435264.2435296
31. Huang, Z., Malik, S.: Managing dynamic reconfiguration overhead in systems-on-a-chip design using reconfigurable datapaths and optimized interconnection networks. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE '01, pp. 735–. IEEE Press, Piscataway, NJ, USA (2001). http://dl.acm.org/citation.cfm?id=367072.367934
32. Kee, H., Shen, C.C., Bhattacharyya, S.S., Wong, I., Rao, Y., Kornerup, J.: Mapping parameterized cyclo-static dataflow graphs onto configurable hardware. Journal of Signal Processing Systems (2012)
33. Keinert, J., Deprettere, E.F.: Multidimensional dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)
34. Kenneth Pocek Russell Tessier, A.D.: Birth and adolescence of reconfigurable computing: a survey of the first 20 years of field-programmable custom computing machines. Field-Programmable Custom Computing Machines, Annual IEEE Symposium on **00**(undefined), 1–17 (2013). doi.ieeecomputersociety.org/10.1109/FPGA.2013.6882273
35. Ko, M.Y., Zissulescu, C., Puthenpurayil, S., Bhattacharyya, S.S., Kienhuis, B., Deprettere, E.F.: Parameterized looped schedules for compact representation of execution sequences in dsp hardware and software implementation. IEEE Transactions on Signal Processing **55**(6), 3126–3138 (2007)
36. Lattner, C.: Llvm and clang: Advancing compiler technology. Proc. of FOSDEM (2011)
37. Lee, E.A., Ha, S.: Scheduling strategies for multiprocessor real-time dsp. In: Global Telecommunications Conference and Exhibition'Communications Technology for the 1990s and Beyond'(GLOBECOM), 1989. IEEE, pp. 1279–1283. IEEE (1989)
38. Lee, E.A., Parks, T.M.: Dataflow process networks. Proceedings of the IEEE **83**(5), 773–801 (1995)
39. Leupers, R., Aguilar, M.A., Castrillon, J., Sheng, W.: Software compilation techniques for heterogeneous embedded multi-core systems. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
40. Lin, S., Wang, L.H., Vosoughi, A., Cavallaro, J.R., Juntti, M., Boutellier, J., Silvén, O., Valkama, M., Bhattacharyya, S.S.: Parameterized sets of dataflow modes and their application to implementation of cognitive radio systems. Journal of Signal Processing Systems **80**(1), 3–18 (2015). http://dx.doi.org/10.1007/s11265-014-0938-4
41. Liu, L., Wang, D., Zhu, M., Wang, Y., Yin, S., Cao, P., Yang, J., Wei, S.: An energy-efficient coarse-grained reconfigurable processing unit for multiple-standard video decoding. IEEE Trans. Multimedia **17**(10), 1706–1720 (2015)
42. Martin, G., Smith, G.: High-level synthesis: Past, present, and future. IEEE Design & Test of Computers **26**(4), 18–25 (2009). http://dx.doi.org/10.1109/MDT.2009.83

43. Moreano, N., Araujo, G., Huang, Z., Malik, S.: Datapath merging and interconnection sharing for reconfigurable architectures. In: System Synthesis, 2002. 15th International Symposium on, pp. 38–43 (2002)
44. Neuendorffer, S., Lee, E.: Hierarchical reconfiguration of dataflow models. In: MEMOCODE (2004). https://doi.org/10.1109/MEMCOD.2004.1459852
45. Niedermeier, A., Kuper, J., Smit, G.: Dataflow-based reconfigurable architecture for streaming applications. In: System on Chip (SoC), 2012 International Symposium on, pp. 1–4 (2012). https://doi.org/10.1109/ISSoC.2012.6376365
46. Palumbo, F., Carta, N., Pani, D., Meloni, P., Raffo, L.: The multi-dataflow composer tool: generation of on-the-fly reconfigurable platforms. Journal of real-time image processing **9**(1), 233–249 (2014)
47. Palumbo, F., Carta, N., Pani, D., Meloni, P., Raffo, L.: The multi-dataflow composer tool: generation of on-the-fly reconfigurable platforms. Journal of real-time image processing **9**(1), 233–249 (2014)
48. Palumbo, F., Sau, C., Evangelista, D., Meloni, P., Pelcat, M., Raffo, L.: Runtime energy versus quality tuning in motion compensation filters for hevc. IFAC-PapersOnLine **49**(25), 145–152 (2016)
49. Palumbo, F., Sau, C., Fanni, T., Meloni, P., Raffo, L.: Dataflow-based design of coarse-grained reconfigurable platforms. In: Signal Processing Systems (SiPS), 2016 IEEE International Workshop on, pp. 127–129. IEEE (2016)
50. Pelcat, M., Aridhi, S., Piat, J., Nezan, J.F.: Physical layer multi-core prototyping: a dataflow-based approach for LTE eNodeB, vol. 171. Springer Science & Business Media (2012)
51. Pelcat, M., Desnos, K., Heulot, J., Guy, C., Nezan, J.F., Aridhi, S.: Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In: Education and Research Conference (EDERC), 2014 6th European Embedded Design in, pp. 36–40 (2014). https://doi.org/10.1109/EDERC.2014.6924354
52. Piat, J., Bhattacharyya, S., Raulet, M.: Interface-based hierarchy for synchronous data-flow graphs. In: SiPS Proceedings (2009). https://doi.org/10.1109/SIPS.2009.5336240
53. Piat, J., Crenne, J.: Modeling dynamic partial reconfiguration in the dataflow paradigm. In: 2014 IEEE Workshop on Signal Processing Systems (SiPS), pp. 1–6. IEEE (2014)
54. Plishker, W., Sane, N., Bhattacharyya, S.S.: Mode grouping for more effective generalized scheduling of dynamic dataflow applications. In: Proceedings of the 46th Annual Design Automation Conference, pp. 923–926. ACM (2009)
55. Ren, R., Juarez, E., Sanz, C., Raulet, M., Pescador, F.: Energy-aware decoder management: a case study on rvc-cal specification based on just-in-time adaptive decoder engine. IEEE Transactions on Consumer Electronics **60**(3), 499–507 (2014). https://doi.org/10.1109/TCE.2014.6937336
56. Roquier, G., Bezati, E., Thavot, R., Mattavelli, M.: Hardware/software co-design of dataflow programs for reconfigurable hardware and multi-core platforms. In: 2011 Conference on Design and Architectures for Signal and Image Processing, DASIP 2011, Tampere, Finland, November 2–4, 2011, pp. 171–177 (2011). http://dx.doi.org/10.1109/DASIP.2011.6136875
57. Sau, C., Fanni, L., Meloni, P., Raffo, L., Palumbo, F.: Reconfigurable coprocessors synthesis in the mpeg-rvc domain. In: ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on, pp. 1–8. IEEE (2015)
58. Sau, C., Meloni, P., Raffo, L., Palumbo, F., Bezati, E., Casale-Brunet, S., Mattavelli, M.: Automated design flow for multi-functional dataflow-based platforms. Journal of Signal Processing Systems **85**(1), 143–165 (2016)
59. Sérot, J., Berry, F.: High-level dataflow programming for reconfigurable computing. In: Proceedings of the 2014 International Symposium on Computer Architecture and High Performance Computing Workshop, SBAC-PADW '14, pp. 72–77. IEEE Computer Society, Washington, DC, USA (2014). http://dx.doi.org/10.1109/SBAC-PADW.2014.18
60. Souza, C.C.d., Lima, A.M., Araujo, G., Moreano, N.B.: The datapath merging problem in reconfigurable systems: Complexity, dual bounds and heuristic evaluation. J. Exp. Algorithmics **10** (2005). http://doi.acm.org/10.1145/1064546.1180613

61. Swanson, S., Schwerin, A., Mercaldi, M., Petersen, A., Putnam, A., Michelson, K., Oskin, M., Eggers, S.J.: The wavescalar architecture. ACM Trans. Comput. Syst. **25**(2), 4:1–4:54 (2007). http://doi.acm.org/10.1145/1233307.1233308
62. Tessier, R., Burleson, W.: Reconfigurable computing for digital signal processing: A survey. VLSI Signal Processing **28**(1–2), 7–27 (2001). http://dx.doi.org/10.1023/A:1008155020711
63. Tessier, R., Pocek, K.L., DeHon, A.: Reconfigurable computing architectures. Proceedings of the IEEE **103**(3), 332–354 (2015). http://dx.doi.org/10.1109/JPROC.2014.2386883
64. Theelen, B.D., Deprettere, E.F., Bhattacharyya, S.S.: Dynamic dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
65. Wijtvliet, M., Waeijen, L., Corporaal, H.: Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In: International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2016, Agios Konstantinos, Samos Island, Greece, July 17–21, 2016, pp. 235–244 (2016). http://dx.doi.org/10.1109/SAMOS.2016.7818353
66. Xilinx: Partial Reconfiguration User Guide (2012)
67. Yviquel, H., Casseau, E., Wipliez, M., Gorin, J., Raulet, M.: Classification-based optimization of dynamic dataflow programs. In: Advancing Embedded Systems and Real-Time Communications with Emerging Technologies, pp. 282–301. IGI Global (2014)
68. Yviquel, H., Sanchez, A., Mickaël, R., Casseau, E.: Technical Report: Multicore Runtime for Dynamic Dataflow Video Decoders. Technical report, IETR/INSA Rennes; IRISA, Inria Rennes (2017). https://hal.archives-ouvertes.fr/hal-01503378

Integrated Modeling Using Finite State Machines and Dataflow Graphs

Joachim Falk, Kai Neubauer, Christian Haubelt, Christian Zebelein, and Jürgen Teich

Abstract In this chapter, different application modeling approaches based on the integration of finite state machines with dataflow models are reviewed. Many well-known Models of Computation (MoC) that are used in design methodologies to generate optimized hardware/software implementations from a model-based specification turn out to be special cases thereof. A particular focus is put on the analyzability of these models with respect to schedulability and the generation of efficient schedule implementations. Here, newest results on clustering methods for model refinement and schedule optimization by means of quasi-static scheduling are presented.

1 Intro

Dataflow graphs are widely accepted for modeling DSP algorithms, e.g., multimedia and signal processing applications. For *static Dataflow Graphs (DFGs)*, efficient techniques for analyzing liveness, boundedness and throughput properties do exist. However, modeling complex multimedia applications using only static dataflow models is a challenging task. Often, dynamic dataflow models are applied that are able to also model control flow. Indeed, these turn out to be a good fit for multimedia and signal processing applications. However, the introduction of control flow in dynamic dataflow actors has lead to problems concerning adequate information

J. Falk · J. Teich
University of Erlangen-Nuremberg, Hardware-Software-Co-Design, Erlangen, Germany
e-mail: joachim.falk@fau.de; jurgen.teich@fau.de

K. Neubauer · C. Haubelt (✉)
University of Rostock, Applied Microelectronics and Computer Engineering, Rostock-Warnemünde, Germany
e-mail: kai.neubauer@uni-rostock.de; christian.haubelt@uni-rostock.de

C. Zebelein
Valeo Siemens eAutomotive Germany GmbH, Erlangen, Germany
e-mail: christian.zebelein.jv@valeo-siemens.com

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_23

extraction from these models to serve as a basis for the analysis of the previously mentioned properties. Hence, neither static nor dynamic DFGs seem to be the best choice to establish a design methodology upon it. As an alternative, different modeling approaches integrating *FSMs* with DFGs have been proposed in the past. However, this often comes with the drawback of decreased analysis capabilities. Nevertheless, several successful academic system-level design methodologies have emerged recently based on such integrated modeling approaches.

In this chapter, we will present different integrated modeling approaches (cf. Sect. 2). Next, in Sect. 3, various ways to represent schedules for DFGs will be discussed. Section 4 is devoted to the problem of clustering subgraphs of static actors and computing *Quasi-Static Schedules (QSSs)* for these clusters to improve scheduling efficiency. Subsequently, the computations of QSSs will be generalized to DFGs with bounded channel capacities in Sect. 5. Finally, in Sect. 6, we recap the important points of this chapter.

2 Modeling Approaches

In this section, we will discuss modeling approaches combining DFGs with FSMs. Starting with the recapitulation of some fundamental dataflow models like Synchronous Dataflow (SDF) [26] and Dynamic Dataflow (DDF), the **charts* (pronounced "star charts") [21] formalism is introduced. The **charts* formalism was one of the first integrated modeling approaches implemented in the Ptolemy project [11]. Later on, Extended Codesign Finite State Machines (ECFSMs) [31] and SystemC Models of Computation (SysteMoC) [13] will be discussed.

2.1 Dataflow Graphs

DFGs are directed graphs of nodes (actors) and edges (channels). Actors have a notion of firing that is an atomic computation that consumes a number of tokens from each incoming edge (input channel) and produces a number of tokens on each outgoing edge (output channel). More formally, a DFG is given as follows:

Definition 1 (DFG) A DFG is a directed graph $g = (A, C)$, where the set of vertices A represents the *actors* and the set of edges $C \subseteq A \times A$ represents the *channels*. Additionally, a *delay function* $\mathbf{delay} : C \to \mathcal{V}^*$ is given.[1] It assigns to

[1]The '*'-operator is used to denote Kleene closure of a value set. It generates the set of all possible finite sequences of values from a value set, that is $X^* = \cup n \in \mathbb{N}_0 : X^n$. As is customary, $\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$ denotes the set of non-negative integers.

each channel $(a_{\text{src}}, a_{\text{snk}}) = c \in C$ a (possibly empty) sequence of initial tokens.[2] The set $\mathcal{V}$ is the set of data values which can be carried by a token.

Moreover, if a DFG has limited channel capacities, then the function $\textbf{size} : C \rightarrow \mathbb{N} \cup \{\infty\}$ is used to specify this.[3] In the following, it is assumed that all channels have unlimited capacities if not otherwise mentioned.

In SDF [26] graphs, the firing of an actor is an atomic computation that consumes a fixed number of tokens from each incoming edge and produces a fixed number of tokens on each outgoing edge. One subclass of SDF [26] graphs are Homogeneous Synchronous Dataflow (HSDF) graphs, where each actor exactly consumes and produces a single token from each incoming and on each outgoing edge, respectively. Edges are conceptionally unbounded First In First Out (FIFO) queues representing a possibly infinite stream of data. The consumption and production rates can be used to unambiguously define a so-called *iteration*, which is a firing sequence that returns the queues to their original state. The number of firings of each actor in an iteration may be found by solving the balance equation $\eta_{a_i} \textbf{prod}(a_i, a_j) = \eta_{a_j} \textbf{cons}(a_i, a_j)$ where the edge (a_i, a_j) is directed from actor a_i to actor a_j. Moreover, $\textbf{prod}(a_i, a_j)$ is the number of tokens produced by actor a_i onto this particular edge, whereas $\textbf{cons}(a_i, a_j)$ is the number of tokens consumed by actor a_j. η_{a_i} and η_{a_j} denote the number of actor firings of actor a_i and a_j, respectively, to return the edge (a_i, a_j) to its original state. Beside the trivial solution $\eta_{a_i} = \eta_{a_j} = 0$, one is typically interested in a minimal, but non-zero vector of actor firings. Considering a network of $|A|$ actors with $|C|$ channels, there will be $|C|$ equations with $|A|$ unknowns. For connected graphs, there exists either the only solution $\eta_a = 0, \forall a \in A$ or a unique smallest solution, called the *repetition vector* $\boldsymbol{\eta}^{\text{rep}} = (\eta_{a_1}, \eta_{a_2}, \ldots \eta_{a_{|A|}})$, with $\eta_a > 0, \forall a \in A$. A more thorough treatment of the SDF Model of Computation (MoC) can be found in chapter "Decidable Signal Processing Dataflow Graphs" [22].

In DDF, actors consume and produce a variable number of tokens on each firing. It is assumed that a DDF actor must assert, prior to any firing, the required number of tokens on each input. If the requirements are met, the DDF actor can fire. A detailed discussion of the DDF MoC can be found in chapter "Dynamic Dataflow Graphs" [33].

2.2 **charts*

One of the first modeling approaches integrating FSMs with dataflow models is **charts* (pronounced "star charts") [21]. The concurrency model in *charts is not restricted to be dataflow: Other choices are discrete event models or synchronous/re-

[2] In some dataflow models that abstract from token values, the delay function may only return a non-negative integer that denotes the number of initial tokens on the channel. In such a context, the number of initial tokens may also be called the *delay* of a channel.

[3] As is customary, $\mathbb{N} = \{1, 2, 3, \ldots\}$ denotes the set of positive integers.

active models. However, in the scope of this chapter, we will limit our discussion to the FSM/dataflow integration.

We start by formally defining deterministic finite state machines:

Definition 2 (FSM) A deterministic FSM is a five tuple $(Q, \Sigma, \Delta, \sigma, q_0)$, where Q is a finite set of states, Σ is a set of symbols denoting the possible inputs, Δ is a set of symbols denoting possible outputs, $\sigma : Q \times \Sigma \rightarrow Q \times \Delta$ is the transition mapping function, and $q_0 \in Q$ is the initial state.

In one reaction, an FSM changes from its current state $q_{\text{src}} \in Q$ given an input symbol $\kappa \in \Sigma$ to a unique next state $q_{\text{dst}} \in Q$ and outputs the output symbol $\delta \in \Delta$, where $(q_{\text{dst}}, \delta) = \sigma(q_{\text{src}}, \kappa)$. Given a sequence of input symbols, an FSM performs a sequence of reactions starting in its initial state q_0. Thereby, a sequence of output symbols in Δ is produced.

FSMs are often represented by *state transition diagrams*. In a *state transition diagram*, vertices correspond to states and edges model state transitions, see Fig. 1. Edges are labeled by $guard/action$ pairs, where $guard \in \Sigma$ specifies the input symbol triggering the corresponding state transition and $action \in \Delta$ specifies the output symbol to be produced by the FSM reaction. The edge without source state points to the initial state.

One reaction of the FSM consists in taking a single enabled state transition. An enabled state transition is an outgoing transition from the current state where the guard matches the current input symbol. Thereby, the FSM changes to the destination state of the transition and produces the output symbol specified by the action.

The *charts approach allows nesting of DFGs and FSMs by allowing the refinement of a dataflow actor via FSMs and allowing an FSM state to be refined via a DFG. We first discuss how FSMs are used to refine dataflow actors.

2.2.1 Refining Dataflow Actors via FSMs

To use an FSM as a refinement for a dataflow actor, the consumption and production rates have to be determined from the FSM for the refined actor. However, the derivation of the consumption and production rates depends on the MoC of the DFG containing the actor. The tokens on the channels connected to the refined dataflow actor are mapped to the input alphabet Σ of the refining FSM. In the guards of the corresponding FSM, we will use the notation $i[n]$ to refer to the value, e.g., Booleans for the examples given in Figs. 1 and 2, of the nth token on the channel connected to the input port i. In the action of the FSM, we will use the notation $o[n] = v$ to give the value v of the nth token to be produced on the channel connected to the output port o. Each $i[n]$ has a symbol subdomain $\Sigma_{i,n}$ associated with it and the cross product of these subdomains form the input alphabet to the FSM. The same is true for $o[n]$ and the output alphabet Δ, thus perfectly matching the FSM model of Definition 2. In each reaction of the FSM, the corresponding action might emit a symbol from the output alphabet of the FSM, which can be used to derive a token value to be produced on each outgoing edge of the actor.

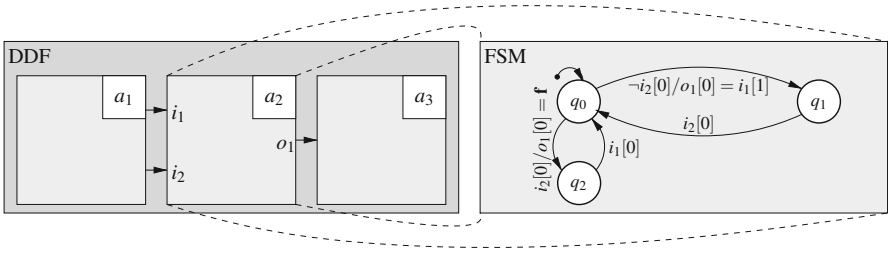


Fig. 1 Refinement of a DDF actor by an FSM in *charts

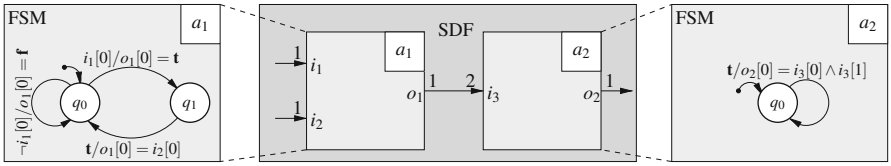


Fig. 2 Refinement of SDF actors by FSMs in *charts

FSM in Dynamic Dataflow [11] If an FSM is used to define the firing semantics of a DDF actor, then each state of the FSM is associated with a number of tokens that will be consumed by the next firing of the actor. The semantics is as follows: For a given state q of the FSM, the number of consumed tokens is determined by examining each outgoing transition of the state. For each input port i of the actor, the maximum index n is determined with which the input port i occurs in any of the transitions leaving the state q. If an input port i does not occur in any transition, an index of -1 is assumed for this port. Then, for the given state q the consumption rate for each input port i is the maximum index n incremented by one. Note that this derivation of the consumption rates implicitly produces actors which are continuous in Kahn's sense [24]. An introduction to Kahn Process Networks (KPNs) can be found in chapter "Kahn Process Networks and a Reactive Extension" [19].

To illustrate the above concept, consider state q_0 in the FSM depicted in Fig. 1 refining the dynamic actor a_2. In this case, regardless if the transition to q_1 or q_2 is taken, the actor will consume exactly two tokens from i_1 and one token from i_2.

FSM in Synchronous Dataflow [11] When an FSM refines an SDF actor, it must externally obey SDF semantics, i.e., the actor must consume and produce a fixed number of tokens upon each firing.

Consider Fig. 2 as an example. The shown SDF model consists of two actors connected by a single edge. The minimal repetition vector is $\eta_{a_1} = 2$ and $\eta_{a_2} = 1$. Since the edge (a_1, a_2) contains no initial tokens, each iteration consists of two firings of actor a_1 before actor a_2 is fired once. First, actor a_1 fires if at least one token is available on each input edge. After the second firing of actor a_1, actor a_2 is enabled.

We now consider the description of SDF actors by FSMs. In the simplest case, the consumption and production rate of each state of the FSM are equal. Then, the FSM can be used unmodified as an SDF actor where the consumption and production rate of the actor are rates which hold in each state of the FSM. An example of such an FSM can be seen for the refinement of the SDF actor a_2 in Fig. 2. The FSM neatly corresponds to the SDF model consuming two tokens and producing one token in each reaction.

However, the FSM describing the firing semantics of actor a_1 in Fig. 2 does not correspond to the simple case. In this case, the consumption and production rates of the refined SDF actor are defined as the maximum of the consumption and production rates of each state of the FSM. If consumed tokens are not present in the guards leaving a state, they are simply ignored. If values of produced tokens are not specified in the action of an enabled state transition, the values are interpreted as ϵ values indicating the absence of values. Note that even an ϵ value produces a token in the enclosing DFG. This solution is a little unsatisfactory and indeed there is another approach to handle this case. The FSM can be embedded into a *heterochronous dataflow graph*.

FSM in Heterochronous Dataflow The idea of Heterochronous Dataflow (HDF) is similar to parameterized dataflow modeling [5], that is, dynamic behavior is allowed and is represented via FSMs. However, all FSMs in the *heterochronous dataflow graph* are forced to only change state once the *heterochronous dataflow graph* has executed a full iteration. This constraint ensures that the consumption and production rates of the HDF actor does not change while the HDF graph executes its iteration. However, after the iteration is finished, the HDF actors are free to update their state leading to new consumption and production rates for the HDF actors in the system. With these new consumption and production rates, a new balance equation is solved and a new repetition vector calculated which is executed in the next iteration. For the duration of this next iteration, all HDF actors have to keep their consumption and production rates unmodified.

Let us consider again Fig. 2. But now instead of an SDF domain, we use an HDF domain. In the case that actor a_1 is in state q_0, to execute a full iteration of the HDF graph, the actor a_1 is executed twice—consuming two tokens from i_1—and the actor a_2 is executed exactly once. Note that while actor a_1 is executed, it remains in state q_0 regardless of the value $i_1[0]$ of the first token on input port i_1. After the full iteration of the SDF graph has finished, the FSM of actor a_1 may change its state to q_1 depending on the value $i_1[0]$ of the first token on input port i_1. In the case that actor a_1 is now in state q_1, a full iteration of the HDF graph corresponds again to the actor firing sequence $\langle a_1, a_1, a_2 \rangle$. However, two tokens from input port i_2 will be consumed and none from i_1. After the full iteration of the SDF graph has finished, the FSM of actor a_1 will change its state to q_0.

2.2.2 Refining FSM States via Dataflow Graphs

Previously, we have seen how an FSM can be used to refine a dataflow actor. On the opposite side, an FSM can be used to coordinate between multiple DFGs. This coordination is achieved by refining FSM states by DFGs. The DFG is composed into a single actor which is executed if the FSM is in the refined state. To refine a state by a DFG, a notion of *iteration* is necessary as the execution of one reaction of the FSM has to terminate. An *iteration* has been chosen as a natural boundary to stop the execution of the embedded DFG. However, the existence of a finite iteration is undecidable for general DFGs. Hence, the application of refinements of states to DFGs is restricted in *charts to certain subclasses of dataflow, e.g., SDF [26] and Cyclo-Static Dataflow (CSDF) [7] graphs, which provide such a notion of iteration naturally. Moreover, combining the actors in a subgraph of a DFG into a single actor, which will execute an iteration for the subgraph, is not always possible. The problem is treated in detail in Sect. 4.

As an example, consider Fig. 3a, which is taken from [21]. The FSM of actor a_1 consists of two states q_0 and q_1. If the current state of the FSM is not refined, e.g., q_0, the FSM reacts like an ordinary FSM. State q_1 is refined by an SDF graph consisting of two actors a_2 and a_3 which are connected by a single edge. The minimal repetition vector may be determined as $\eta_{a_2} = 2, \eta_{a_3} = 1$, i.e., the repetition vector is satisfied by two firings of actor a_2 followed by one firing of actor a_3 implementing one iteration of the SDF graph. Assuming the above behavior, the SDF graph can be substituted by a single SDF actor as shown in Fig. 3b. If a DFG refines a state of an FSM, the firing rules of the DFG are exported to the environment of the FSM, i.e., if the FSM is in the state which is refined by the DFG, then for a reaction of the FSM exactly as many tokens will be required as for the execution of the *iteration* of the embedded DFG.

After the embedded DFG has finished its iteration, the FSM will execute a transition. If the FSM is embedded in the DDF or SDF domain, then the FSM

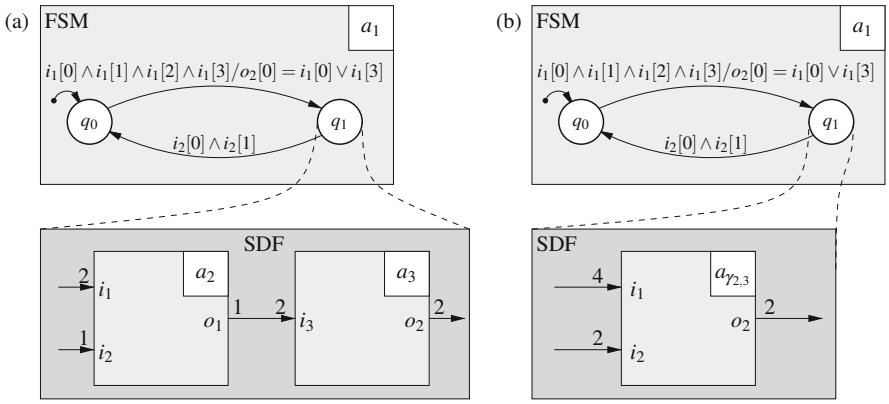


Fig. 3 (**a**) SDF graph refining a state. (**b**) SDF graph is substituted by a single SDF actor

executes a transition after the *corresponding DFG* has finished its iteration. If the FSM is embedded in the HDF domain, then this transition is delayed until all parent graphs have finished their iteration.

Assuming that FSM a_1 is in state q_1, the FSM is not embedded in an HDF graph, and the tokens $i_2[0]$ and $i_2[1]$ have both the presence value (note that tokens carrying ϵ values are still tokens and not absence of tokens), then the next activation of actor a_1 will execute the embedded SDF graph for one iteration followed by a state transition to state q_0.

2.3 Extended Codesign Finite State Machines

The Extended Codesign Finite State Machine (ECFSM) MoC has been presented in [31]. As will be seen, the original Codesign Finite State Machine (CFSM) MoC used in POLIS [3] is a special case of the ECFSM MoC. Both models, however, are refinements of the Abstract Codesign Finite State Machine (ACFSM) Model, also presented in [31].

In this section, we will therefore first introduce the ACFSM Model of Computation, followed by the discussion of the refinement step which transforms a given ACFSM into an ECFSM. An ACFSM is a formal model consisting of a network of FSMs (in the following called actors) connected via unbounded *FIFO channels*. Tokens (also called *events*) are transmitted across the channels carrying data values of some data type. Figure 4 shows an example of such a dataflow network consisting of several ACFSMs.

Briefly, a single ACFSM consists of an FSM controlling the communication behavior of the actor, transforming a finite sequence of input tokens into a finite sequence of output tokens.

In order to formally describe the behavior of a single ACFSM, we will first look at the behavior of the `Filter` actor from Fig. 4. Its task is to filter a sequence of pixels from the `Producer` actor by multiplying all pixels of a frame by a coefficient (provided by the `Controller` actor). At the beginning of each frame, the `Producer` actor writes two tokens onto the output `out` containing the number of lines per frame and the number of pixels per line, respectively. These two values

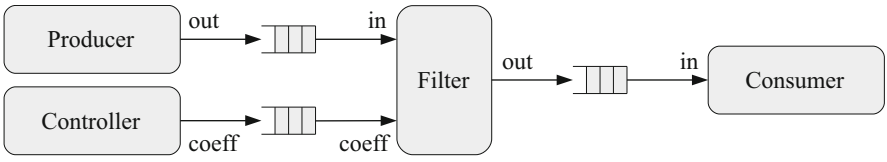


Fig. 4 DFG of the filter example: The ACFSMs (`Producer`, `Controller`, `Filter` and `Consumer`) are connected via FIFO channels. The `Filter` actor has two inputs (`in` and `coeff`) from which tokens are consumed and one output (`out`) onto which tokens are produced

```
1 module Filter {
2   input byte in, coeff;
3   output byte out;
4   int nLines, nPix, line, pix;
5   byte b;
6   forever {
7     (nLines, nPix) = read(in, 2);
8     b = read(coeff, 1);
9     for(line = 0; line < nLines; ++line) {
10      if(present(coeff, 1))
11        b = read(coeff, 1);
12      for(pix = 0; pix < nPix; ++pix) {
13        write(out, read(in, 1) * b, 1);
14      }
15    }
16  }
17 }
```

Fig. 5 Behavior of the `Filter` actor shown in Fig. 4

are read by the `Filter` actor from the input `in`, whereas the initial multiplication coefficient is read from the input `coeff`. For each line, the coefficient values may change, depending on whether or not a token is available on the input `coeff`. Subsequently, the filtered pixels are written onto output `out`.

The behavior of the `Filter` actor can be represented by C-like pseudocode as shown in Fig. 5. There exist three primitives which can be used in order to access the FIFO channels connected to the inputs and outputs of the actor: `read(in, n)` and `write(out, data, n)` consume and produce `n` tokens from input `in` and output `out`, respectively. Note that whereas `read` blocks until enough tokens are available, `write` never blocks, as the FIFO channels are unbounded. The third primitive `present(in, n)` returns true if at least `n` tokens are available on the input `in`.

In order to transform the `Filter` pseudocode into an ACFSM, we will now discuss the formal definition of an ACFSM:

Definition 3 (ACFSM) An ACFSM is a triple $a = (I, O, T)$ consisting of a finite set of *inputs* $I = \{i_1, i_2, \ldots, i_n\}$, a finite set of *outputs* $O = \{o_1, o_2, \ldots, o_m\}$, and a finite set of *transitions* T. In the following, we will use $i[n]$ to indicate the token on the nth position on the FIFO channel connected to input $i \in I$ (as seen from the ACFSM). Analogously, we will use $o[m]$ to indicate the token produced by a transition onto the mth position on the FIFO channel connected to output $o \in O$.

Definition 4 A transition of an ACFSM is a tuple $T = (\mathbf{req}, \mathbf{cons}, \mathbf{prod}, f_{\text{guard}}, f_{\text{action}})$: The *input enabling rate* $\mathbf{req} : I \to \mathbb{N}_0$ maps each input $i \in I$ to the number of tokens which must be available on the channel connected to i in order to enable the transition. The *input consumption rate* $\mathbf{cons} : I \to \mathbb{N}_0 \cup \{ALL\}$ specifies for each input $i \in I$ the number of tokens which will be consumed from the channel connected to i when the transition is executed. Specifying $\mathbf{cons}(i) = ALL$ for a

given input i *resets* the channel connected to i, i.e., all tokens currently stored in the channel are consumed. Otherwise, a transition is not allowed to consume more tokens than requested, i.e., $\mathbf{cons}(i) \leq \mathbf{req}(i)$. Analogously, the *output production rate* $\mathbf{prod} : O \to \mathbb{N}_0$ specifies for each output $o \in O$ the number of tokens which will be produced on the channel connected to o when the transition is executed. The *guard function* f_{guard} is a Boolean-valued expression over the values of the tokens required on the inputs, i.e., $\{i[n] \mid i \in I \wedge 0 \leq n < \mathbf{req}(i)\}$. Note that only if both the input enabling rate **req** and the guard function f_{guard} are satisfied, a transition can be executed. Finally, the *action function* f_{action} determines the values of the tokens which are to be produced on the outputs, i.e., $\{o[m] \mid o \in O \wedge 0 \leq m < \mathbf{prod}(o)\}$, when the transition is executed.

Moreover, ACFSMs cannot have explicit state variables. Nevertheless, these can be modeled by *state feedback channels*, i.e., by an input/output pair of an ACFSM which is connected to the same channel containing a certain state variable. If a transition wants to update the state variable, it must consume the token containing the old value, and produce a token containing the new value.

The execution semantics of an ACFSM can be described as follows: Initially, an ACFSM is *idle*, i.e., waiting for input tokens. An ACFSM is *ready* if at least one transition $t \in T$ is enabled, i.e., if both the input consumption rate $t.\mathbf{req}$ and the guard function $t.f_{\text{guard}}$ are satisfied. Subsequently, a single enabled transition t will be chosen, transitioning the ACFSM into the *executing* state. During the *executing* state, the ACFSM *atomically* consumes tokens from the inputs according to $t.\mathbf{cons}$, performs the action function $t.f_{\text{action}}$, and produces tokens on the outputs according to $t.\mathbf{prod}$.

It should be noted that due to the non-blocking reads which can be implemented by ACFSMs, they are not continuous in Kahn's sense [24], i.e., the arrival of tokens at different times may change the behavior of the DFG.

In [31], the authors claim that "ACFSM transitions [...] can be conditioned to the *absence* of an event over a signal". In principle, there are two possibilities how this could be modeled:

- Provide a special syntax for the input enabling rate such that also checks for empty channels can be expressed. However, the input enabling rate as given in Definition 4 only permits the expression of *minimum conditions*. Therefore, an input i with $\mathbf{req}(i) = 0$ could lead to the execution of the corresponding transition even if the channel connected to i is *not* empty.
- Specify priorities for the transitions of an ACFSM such that transitions requiring more tokens can have a higher priority than transitions requiring less tokens. However, in the original paper [31] it is also stated that "if several transitions can be executed [...], the ACFSM is non-deterministic and can execute any of the matching transitions", i.e., there seems to be no concept of priorities for transitions.

For example, lines 10–11 in Fig. 5 cannot be adequately modeled without checking for the absence of tokens: One transition is needed which checks if at

least one token is available on input `coeff`, i.e., **req**(coeff) = 1. However, as we do not want to block execution if there is no token available, a second transition is needed with **req**(coeff) = 0. As discussed above, the second transition could be executed even if the corresponding channel is not empty. Without the means of expressing a check for an empty channel, we can only assume that the first transition has a higher priority than the second, allowing us to check for the absence of a new multiplication coefficient.

The behavior of the `Filter` actor given in Fig. 5 can be transformed into an ACFSM as follows: First, feedback channels are added for the state variables `nLines`, `nPix`, `line`, `pix` and `b`. Additionally, a feedback channel is needed for storing the current state of the ACFSM, `state`. The corresponding input/output pairs will be named accordingly, resulting in

- the set of inputs $I = \{\text{in, coeff, nLines, nPix, line, pix, b, state}\}$ and
- the set of outputs $O = \{\text{out, nLines, nPix, line, pix, b, state}\}$.

We assume that an initial token with a value of 0 is placed on each state feedback channel.

The resulting transitions are shown in Table 1. For example, the first transition can be enabled if two tokens are available on input `in`, one token on `coeff`, and one token on each input connected to a state feedback channel (except `pix`). Additionally, the current state of the ACFSM must be the initial state, i.e., the value of the (single) token on input `state` must represent the initial state, i.e., `state[0] = 0`. If these requirements are satisfied, the transition will be executed, and the specified output tokens produced: For example, the value of the first token consumed from input `in` is used to produce the single token on output `nLines`. Note that the value of the current state is set to 1, allowing other transitions to be executed.

In conclusion, ACFSMs specify the *topology* of the DFG and the *functional* behavior of each module. However, the communication over unbounded channels with FIFO semantics needs to be refined in order to be implementable with a finite amount of resources. This refinement from infinite-sized queues to implementable finite-sized queues is exactly the transformation from an Abstract CFSM into an Extended CFSM, and will be briefly discussed in the following.

Write operations carried out by the transitions of an ACFSM are always *non-blocking*, as the corresponding channels have an infinite capacity. For a given finite-sized channel, however, if write operations should be non-blocking and the channel in question is full, new data arriving on the channel will overwrite old data stored in the channel. If this behavior is not desired, different approaches may be applied. First, it is often sufficient to increase the channel's capacity. This approach fails, however, if the average production rate of the producer is greater then the average consumption rate of the consumer. In general, calculating sufficient channel capacities is undecidable. Therefore, ECFSMs introduce blocking behavior to prevent data loss. Finally, traditional *CFSMs* are a special case of ECFSMs, where all channels have a capacity of one.

Table 1 Corresponding ACFSM of the `Filter` actor shown in Fig. 5 consisting of six transitions

$(\mathbf{req}(i), \mathbf{cons}(i))/f_{\text{guard}}$								$\mathbf{prod}(o)/f_{\text{action}}$						
in	coeff	nLines	nPix	line	pix	b	state	out	nLines	nPix	line	pix	b	state
2	1	1	1	1	0	1	1	0	1	1	1	0	1	1
state[0] = 0								nLines[0] ⇐ in[0] nPix[0] ⇐ in[1] line[0] ⇐ 0 b[0] ⇐ coeff[0] state[0] ⇐ 1						
0	0	(1,0)	0	(1,0)	0	0	1	0	0	0	0	0	0	1
state[0] = 1 ∧ line[0] = nLines[0]								state[0] ⇐ 0						
0	1	(1,0)	0	(1,0)	1	1	1	0	0	0	0	1	1	1
state[0] = 1 ∧ line[0] < nLines[0]								b[0] ⇐ coeff[0] pix[0] ⇐ 0 state[0] ⇐ 2						
0	0	(1,0)	0	(1,0)	1	0	1	0	0	0	0	1	0	1
state[0] = 1 ∧ line[0] < nLines[0]								pix[0] ⇐ 0 state[0] ⇐ 2						
0	0	0	(1,0)	1	(1,0)	0	1	0	0	0	1	0	0	1
state[0] = 2 ∧ pix[0] = nPix[0]								line[0] ⇐ line[0] + 1 state[0] ⇐ 1						
1	0	0	(1,0)	0	1	(1,0)	(1,0)	1	0	0	0	1	0	0
state[0] = 2 ∧ pix[0] < nPix[0]								out[0] ⇐ in[0] ∗ b[0] pix[0] ⇐ pix[0] + 1						

In order to adequately model the multiplication coefficient update (Lines 10–11 in Fig. 5), we assume that the transitions are assigned priorities according to their ordering, with higher priorities first. Note also that for reasons of readability, $\mathbf{cons}(i)$ is omitted if $\mathbf{req}(i) = \mathbf{cons}(i)$.

2.4 SysteMoC

SysteMoC [13] extends the notion of ECFSM by actor states and hierarchy (see Sect. 3 for details). At a glance, the notation extends conventional dataflow models by finite state machines controlling the consumption and production of tokens by actors. SysteMoC itself is an extension library for *SystemC* [2], which is itself a C++ class library. In the following, we will introduce how DFGs are composed from SysteMoC actors. The vertices of the graph correspond to actors, that are more formally defined below:

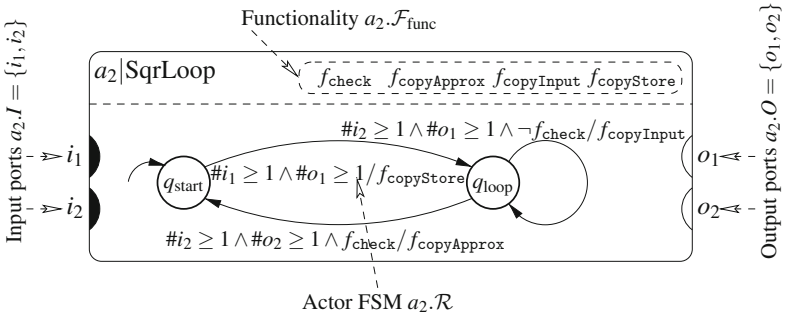


Fig. 6 Visual representation of a `SqrLoop` actor a_2 as used in the network graph g_{sqr} displayed in Fig. 7. The `SqrLoop` actor is composed of *input ports* I and *output ports* O, its *functionality* $\mathcal{F}_{\text{func}}$, and the *actor FSM* $\mathcal{R}$ determining the communication behavior of the actor. The *input predicate* $\#i_x \geq n$ and the *output predicate* $\#o_y \geq m$ on a *transition*, respectively, test the availability of at least n tokens on the actor input port i_x and at least m free places on the actor output port o_y

Definition 5 (Actor) An actor is a tuple $a = (I, O, \mathcal{F}_{\text{actions}}, \mathcal{F}_{\text{guards}}, \mathcal{R})$ containing *actor ports* partitioned into the set of *actor input ports* I and *actor output ports* O, the *actor functionality* $\mathcal{F}_{\text{func}} = \mathcal{F}_{\text{actions}} \cup \mathcal{F}_{\text{guards}}$ partitioned into a set of actions and a set of guards, as well as the *actor FSM* $\mathcal{R}$.[4]

In contrast to ECFSMs, SysteMoC actors can also have a functionality state that can only be manipulated by the actions $f_{\text{action}} \in \mathcal{F}_{\text{actions}}$ and used—but not updated—by the guards $f_{\text{guard}} \in \mathcal{F}_{\text{guards}}$. Finally, the *actor FSM state* contained in $\mathcal{R}$, as known from ECFSMs, is fully controlled by the *actor FSM* and cannot be manipulated by the evaluation of actions or guards. This distinction enables the mathematical expression of the separation of the actor data manipulation calculating the token values produced, e.g., $f_{\texttt{copyApprox}}$ as shown in Fig. 6, and the *communication behavior* controlling the amount of tokens consumed and produced. More intuitively, one can think of the state of the actor functionality as state variables of the actor, which are primarily modified by the data path of the actor modeled by actions, e.g., $f_{\texttt{copyApprox}} \in \mathcal{F}_{\text{actions}}$.

Furthermore, we have extended the well-known DFG definition given in Definition 1 with a notion of ports, i.e., channels no longer connect only actors but actor ports as depicted in Fig. 7. We will call such an extended representation a *network graph*, formally defined below:

[4]Note that in contrast to previous editions of this handbook, the separation of functionality into actions and guards has been explicitly introduced in the definition.

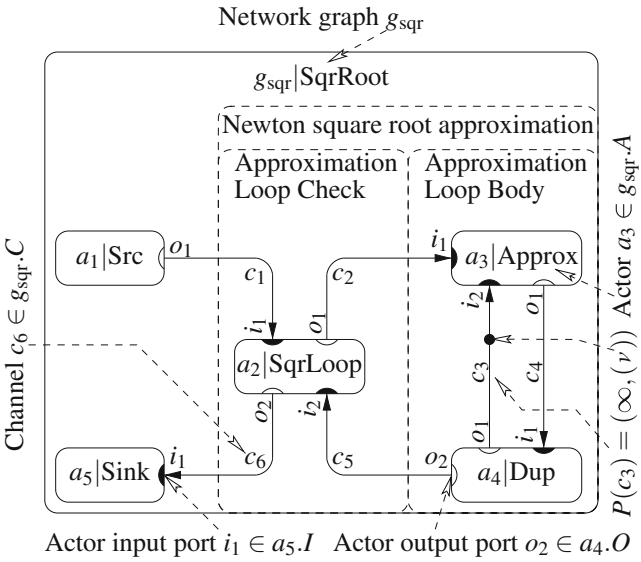


Fig. 7 The *network graphs* displayed above implement Newton's iterative algorithm for calculating the square roots of an infinite input sequence of numbers generated by the `Src` actor a_1. The square root values are generated by Newton's iterative algorithm `SqrLoop` actor a_2 for the error bound checking and a_3–a_4 to perform an approximation step. After satisfying the error bound, the result is transported to the `Sink` actor a_5

Definition 6 (Network Graph) A *network graph* is a directed graph $g_n = (A, C)$ containing a set of *actors* A and a set of *channels* $C \subseteq A.O \times A.I$.[5] Furthermore, we require that each actor port $p \in A.I \cup A.O$ is connected to exactly one channel c.

The execution of a SysteMoC model can be divided into three phases: (i) checking for enabled transitions for each actor, i.e., sufficient tokens and free places on input and output ports and guard functions evaluating to `true`, (ii) selecting and executing one enabled transition per actor, and (iii) consuming and producing tokens as specified by the transition. Note that step (iii) might enable new transitions. Hence, SysteMoC is similar in this regard to ECFSMs where actors are blocked as long as the output buffers cannot store the results. More formally, the *actor FSM* is defined as follows:

Definition 7 (Actor FSM) The *actor FSM* of an actor is a tuple $\mathcal{R} = (T, Q, q_0)$ containing a set of *transitions* T, a set of *states* Q and an *initial state* $q_0 \in Q$.

[5] We use the '.'-operator, e.g., $a.I$, for member access of tuples whose members have been explicitly named in their definition, e.g., member I of actor $a \in A$ from Definition 5. Moreover, this member access operator has a trivial extension to sets of tuples, e.g., $A.I = \bigcup_{a \in A} a.I$, which is also used throughout this document.

Furthermore, a *transition* is specified by a tuple $t = (q_{\text{src}}, k, f_{\text{action}}, q_{\text{dst}}) \in T$ containing source (q_{src}) and destination (q_{dst}) of the transition, a *transition guard* k that must evaluate to true if the transition is taken, and an *action* $f_{\text{action}} \in \mathcal{F}_{\text{actions}}$ from the actor functionality that is executed if the transition is taken.[6]

Here, we assume that the transition guard k is a logical conjunction of (i) an *input/output guard* k^{io} that checks the availability of tokens and free places on the input and output ports of the actor and (ii) a *functionality guard* that is allowed to be an arbitrary Boolean expression consisting of guard functions, e.g., $\neg f_{\texttt{check}}$ as seen in Fig. 6. The input/output guard k^{io} must be a logical conjunction that can only contain *input predicates* ($\#i_x \geq n$) and *output predicates* ($\#o_y \geq m$), e.g., $\#i_1 \geq 1 \wedge \#o_1 \geq 1$. Thus, the *enabling rate* **req** of a transition can be derived from the input/output guard k^{io} contained in the transition guard k. Moreover, input consumption rate and the output production rate can be derived from the number of tokens consumed and produced by the action of a transition. Furthermore, a transition is not allowed to consume more tokens than requested, i.e., $\mathbf{cons}(i) \leq \mathbf{req}(i)$, or produce more tokens than reserved, i.e., $\mathbf{prod}(o) \leq \mathbf{req}(o)$. Finally, if not otherwise stated, it is assumed that the action of a transition produces and consumed exactly as many tokens as requested, i.e., $\mathbf{cons}(i) = \mathbf{req}(i)\, \forall i \in I$ and $\mathbf{prod}(o) = \mathbf{req}(o)\, \forall o \in O$.

2.5 Further Approaches

A number of further approaches exist that integrate finite state machines into the modeling of dataflow graphs.

Enable-Invoke Dataflow (EIDF) [30] derives its name from the requirement that each actor is separated into an *enable function* ν and an *invoke function* κ. Those functions are responsible for testing if enough tokens are available on the input channels for enabling the invoke function which in turn represents the actor firing. In order to allow modeling a dynamic behavior, both enable and invoke functions are dependent on modes of the DFG that are represented by an FSM.

The *California Actor Language (CAL)* [11] is a programming language which was first designed to describe actors and their behavior only. Later on, the language was extended to allow the specification of DFGs containing these actors (see also chapter "MPEG Reconfigurable Video Coding" [27]).

Scenario-Aware Dataflow (SADF) was first developed by Theelen et al. [34] to model dynamic aspects of streaming applications that originate from different modes of operation (scenarios). While static Synchronous Dataflow (SDF) is highly analyzable with respect to liveness, boundedness and throughput properties but

[6] Notice that the *enabling rate* **req** as well as the *input consumption rate and the output production rate* functions **cons** and **prod** are no longer explicitly given but can be derived from the *transition guard* and the action, respectively.

lacks support of any form of dynamism, Kahn Process Networks (KPN) [24] allow modeling of many dynamic aspects but lack from undecidability of correctness and performance evaluation in the general case. SADF fills the gap between the aforementioned approaches. Internally, each scenario is modeled as an SDF graph. Dynamism is supported through the definition of modes of operation. The FSM-Based SADF (FSM–SADF) graph consists of a set of scenarios and a non-deterministic scenario (FSM) [20]. The FSM represents the possible order in which the active scenarios can occur.

Detailed descriptions of EIDF, CAL, and FSM–SADF are given in chapter "Dynamic Dataflow Graphs" [33].

3 Scheduling Dataflow Graphs

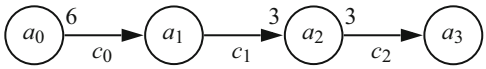
When implementing dataflow MoCs on multi-processor architectures, it is necessary to not only determine actor mappings (see chapter "Methods and Tools for Mapping Process Networks onto Multi-Processor Systems-on-Chip" [1]) but also the execution order of actor firings. To this end, novel techniques for representing such schedules within dataflow graphs are discussed in this section.

3.1 Modeling Static-Order Schedules

Synchronous Dataflow Graphs (SDFGs) are predestined for modeling of streaming applications present in digital signal processing and multimedia systems. Such static models offer high analyzability in terms of timing (i.e. throughput and latency) and memory (i.e. buffer size) requirements (see chapter "Throughput Analysis of Dataflow Graphs" [10]). However, as embedded systems rely increasingly on multi-processor architectures, additional constraints like inter-processor communication complicate analysis techniques in contrast to single-processor architectures. For such multi-processor systems, usually *Periodic Static-Order Schedules (PSOSs)* are constructed for each processor. Instead of the specific start times of each actor as determined by fully static schedules, PSOSs specify the order of actor firings for each processor individually. In the following, two techniques will be presented that model PSOSs into SDFGs, consequently allowing utilizing standard timing and memory usage analysis techniques on schedule-extended SDFGs.

The first technique [4] requires a conversion of an SDFG into a homogeneous SDFG (HSDFG) where all production and consumption rates equal to one. Given a PSOS $s = \langle a_0, a_1, \ldots, a_n \rangle$, for each pair of adjacent actors (a_i, a_{i+1}) in the PSOS, a channel is added to the HSDFG. Finally, an additional channel with one initial token is placed between the last and first (a_n, a_0) actor in the PSOS. Exemplary, consider the SDFG in Fig. 8 with four actors $A = \{a_0, a_1, a_2, a_3\}$ and three channels $C = \{c_0, c_1, c_2\}$ [9]. The equivalent HSDFG is given in Fig. 9 (without

Fig. 8 Example SDFG [9]



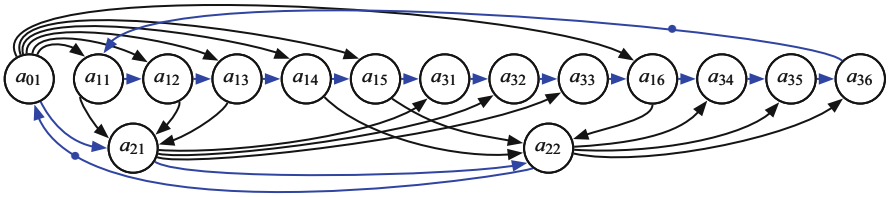


Fig. 9 SDFG from Fig. 8 converted into an HSDFG (black edges) and extended with scheduling decisions (blue edges) with the technique from [4]

blue edges) and consists of 15 actors as well as 18 channels. Assume the subsets of actors $\{a_0, a_2\}$ and $\{a_1, a_3\}$ to be mapped onto processors $CPU1$ and $CPU2$, respectively and corresponding PSOSs to be constructed as $s_0 = \langle a_0(a_2)^2 \rangle$ and $s_1 = \langle (a_1)^5(a_3)^3 a_1(a_3)^3 \rangle$. The blue edges in Fig. 9 show the additional channels, that are added to guaranty the sequential order specified in schedules s_0 and s_1. Initial tokens are placed on the channels between the second firing of a_2 and the first firing of a_0 (cf. s_0) and sixth firing of a_3 and the first firing of a_1 (cf. s_1).

Whenever a conversion into an HSDFG is needed for a code generation step, this technique can be used with only a small overhead. However, as the run times of many SDFG timing and memory analysis methods depend heavily on the graph size, the potentially exponential growth converting an SDFG to an HSDFG may become a problem for large graphs. For example, the buffer analysis of an $H.263$ decoder [32] requires less than 1 ms when directly applied to the SDFG (four actors) and more than 1330 ms when applied to an equivalent HSDFG (200 actors) [9]. Furthermore, buffer sizing techniques often fail to determine minimal buffer sizes when applied to HSDFGs.

Therefore, a different approach is proposed in [9]. Here, no conversion into an HSDFG is necessary and consequently the increase in size of the schedule-extended SDFG is minimized. The technique is called *Decision State Modeling* (DSM) and consists of three steps necessary to integrate PSOSs into an SDFG. As shown in Fig. 10 for the running example, only four additional actors and twelve additional channels have to be added in order to construct the schedule-extended graph.

First, auto-concurrency edges (red) are added that prevent simultaneous execution of equal actors when sufficient token were available on the inputs to allow firing the actor two or more times. Second, inter-iteration edges (blue) prevent activated actors of the next iteration from firing until the given PSOS has finished one period. As seen in Fig. 10, between the last a_l and first a_f actor of a PSOS s_i, an additional actor a_{i-end} and two channels $c_{i-pre} = (a_l, a_{i-end})$ and $c_{i-pro} = (a_{i-end}, a_f)$ are added. Production and consumption rates as well as initial tokens are derived from the quantity of corresponding actors (i.e. $CNT(a)$) in the PSOS such that the production and consumption rate equal to $\mathbf{prod}(a_{i-end}) = CNT(a_f)$ and

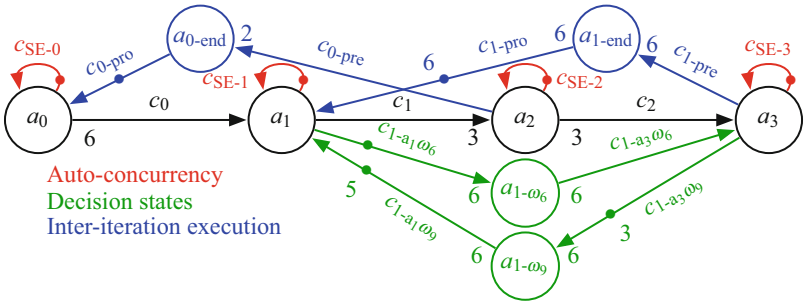


Fig. 10 Modeling the PSOSs s_0 and s_1 within the SDFG from Fig. 8 using Decision State Modeling [9, Fig. 3]

cons$(a_{i-end}) = CNT(a_l)$, respectively. The number of initial tokens is determined analogously: $init(c_{i-pro}) = CNT(a_f)$. Note that, the inter-iteration edges have the same purpose as the additional edges in Fig. 9. Last, *decision states* have to be determined and treated. Basically, a decision state is a state of the SDFG in which more than one actor is activated and thus is able to fire. In order to prevent an execution order other than specified in the PSOS, additional actors and edges are added that steer the selection of the *actor of choice*. In Fig. 10, these additional elements are depicted in green. For example, consider PSOS s_1. The five initial tokens in channel $c_{1-a_1\omega_9}$ allow firing of a_1 exactly five times. Subsequently, actor $a_{1-\omega_6}$ can be executed which in turn activates the execution of a_3. After firing three times, actor $a_{1-\omega_9}$ is executed and a_1 can fire one final time. For a more detailed description on how to construct the decision states, refer to [9].

3.2 Quasi-Static and Dynamic Schedule Modeling

While Decision State Modeling (cf. Sect. 3.1) offers an efficient way to model PSOSs within SDFGs, it cannot be used to represent scheduling decisions for dynamic applications (i.e. dataflow graphs where a PSOS cannot be created). To this end, a technique is presented in [37] that permits modeling schedules for such dataflow graphs. In fact, it can be used to represent the whole model-based transformation from a behavioral description into a partitioned and scheduled implementation. The design flow is depicted exemplary in Fig. 11 [38] which will be detailed in the following section.

The idea behind the technique is an extension of dataflow actors through guarded actions with a static communication behavior which allows a hierarchical actor description. Basically, an actor can be defined hierarchically and may contain a set of *child actors* A and a set of *transitions* T represented as an FSM. In order to model several aspects of the design flow, actors are organized into three different types such that simple actors as defined in Definition 5 have no child actors whereas actors

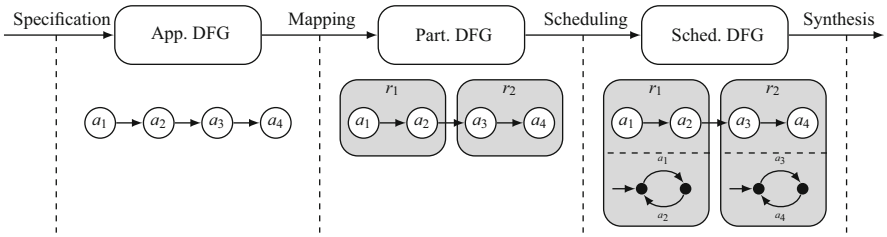


Fig. 11 Design flow for modeling dynamic schedules with the technique of [38]

with child actors are called clusters. Depending on the set of transitions T, clusters are in turn partitioned into *structural* ($T = \emptyset$) and *functional* ($T \neq \emptyset$) clusters. Actors are used to model the functionality of the application as shown in Fig. 11 "App. DFG" whereas structural clusters are used to model mapping decisions without any behavioral description (cf. "Part. DFG" in Fig. 11). Finally, functional clusters include schedules into partitioned applications as shown in "Sched. DFG" in Fig. 11. We will see how such schedules may be described by the FSMs.

3.2.1 Actor Execution Model

In the following, the representation of mapping and scheduling decisions is discussed in detail. First, actors are considered which represent the functionality of the application. Unlike clusters, they are not composed of child actors and thus do not require the scheduling mechanism of the full cluster model.

The actor FSM (cf. Definition 7) of an actor can be used in conjunction with actions $f_{\text{action}} \in \mathcal{F}_{\text{actions}}$ and guard functions $f_{\text{guard}} \in \mathcal{F}_{\text{guards}}$ to model dynamic dataflow behavior. To this end, a guard function evaluates to true whenever a transition of the current actor state q_{cur} is activated. Subsequently, the corresponding action function is executed and actor state switches to the successor state q_{dst}. A guard function must not consume or produce any token from the connected channels nor change internal state variables. On the other hand, action functions are allowed to produce and consume tokens as well as change internal states. In the model, action functions are restricted to static communication behavior. That is, the production (**prod**) and consumption (**cons**) rates derived from k^{io} of a transition $t \in T$ are fixed and specify the number of tokens produced and consumed when excuting the corresponding action function.

The process of executing a transition consists of two steps. In the *evaluation phase*, an actor is checked whether it is enabled or not. That is, a transition t is enabled whenever the current actor mode q_{cur} corresponds to transitions mode $t.q_{src}$, the transition guard $t.k$ evaluates to $true$, and sufficient tokens and space are available on the input and output ports, respectively. The *execution phase* initiates the actual firing by calling the action function. As a consequence, tokens are consumed from the input and produced on the output ports as specified by **cons** and

prod respectively, and state variables may be modified. After the execution phase, the current actor mode q_{cur} changes to $t.q_{dst}$.

3.2.2 Cluster Execution Model

As indicated in the outline, clusters are introduced to model several aspects in the design flow implementing dataflow graphs. Namely, they can represent mapping and scheduling decisions directly in the graph. In addition to actors, clusters are hierarchically composed of child elements that can be actors and again clusters and contain additional channels that connect child elements. Formally, they are defined as follows.

Definition 8 (Cluster) A cluster is a graph $g_\gamma = (I, O, A, C, \mathcal{F}_{\text{actions}}, \mathcal{F}_{\text{guards}}, \mathcal{R})$ containing cluster ports partitioned into cluster input ports $I \subseteq A.I$ and cluster output ports $O \subseteq A.O$, a set of child actors A, a set of channels $C \subseteq A.O \times A.I$, the *cluster functionality* $\mathcal{F}_{\text{func}} = \mathcal{F}_{\text{actions}} \cup \mathcal{F}_{\text{guards}}$ partitioned into a set of actions and a set of guards, as well as the *cluster FSM* $\mathcal{R}$.

Note that a cluster with an empty set of child actors A (and thus an empty set of channels C), results in an actor as specified in Definition 5. In consideration of representing scheduling decisions, the cluster FSM $\mathcal{R}$ is defined as follows.

Definition 9 (Cluster FSM) A cluster FSM $\mathcal{R} = (Q_\gamma, q_0, T)$ is a tuple consisting of a set of states Q, an initial state q_0, and a set of transitions. A transition is again a tuple $T \ni t = (q_{src}, k, \mathcal{F}, q_{dst}, \prec, r)$ consisting of a source q_{src} and destination q_{dst} state of the transition, a transition guard k, a set of actions $\mathcal{F} \subseteq \mathcal{F}_{\text{actions}} \cup \mathcal{P}(A.T)$, a strict partial order "$\prec$" over $\mathcal{F}$, and a flag $r \in \{true, false\}$ which determines if token and space availability checks should be performed by transitions of child actors for ports bound to channels of the cluster.[7]

A partial order $f_i \prec f_j$ specifies that action $f_i \in \mathcal{F}$ has to be executed before action $f_j \in \mathcal{F}$ can be activated. As the set of functions $\mathcal{F}$ is composed of action function $f_a \in \mathcal{F}_{\text{actions}}$ and child transitions $A.T$, evaluation and execution phase differ in contrast to actor transitions. Whenever $f \in \mathcal{F}_{\text{actions}}$ is an action function, it is processed analogously to actor transitions. However, if $f \in \mathcal{P}(A.T)$ is a set of child transitions, a single transition has to be selected and executed. For the latter case, the evaluation phase becomes more complex and is outlined below.

In the first step of the evaluation phase, as discussed for actor transitions, the current actor mode q_{cur} is checked if it conforms to the mode $t.q_{src}$ in which transition t is active. In difference to actor transitions, however, cluster transitions consist of a set of actions. Hence, in order to perform token and space availability checks as explained for actor transitions, values of **cons** and **prod** must

[7] The definition of a cluster FSM (Definition 9) has been extended compared to previous editions to allow for the representation of more general schedules within the cluster FSM.

be accumulated first for all actions in $\mathcal{F}$, i.e. $\mathbf{prod}(\mathcal{F}, p) = \sum_{f \in \mathcal{F}} \mathbf{prod}(f, p)$ and $\mathbf{cons}(\mathcal{F}, p) = \sum_{f \in \mathcal{F}} \mathbf{cons}(f, p)$. While production and consumption rates of action functions are statically defined by the user, for a transition set $f \in \mathcal{P}(A.T)$ it is unclear at compile time which transactions are executed at run time. In order to keep a static communication behavior, the modeling approach requires that all transitions in f have the same consumption and production rates w.r.t. the ports of the composite actor. Formally, this is ensured by the following constraint: $\forall t_i, t_j \in f : \forall p \in I : \mathbf{cons}(t_i.\mathcal{F}, p) = \mathbf{cons}(t_j.\mathcal{F}, p) \wedge \forall p \in O : \mathbf{prod}(t_i.\mathcal{F}, p) = \mathbf{prod}(t_j.\mathcal{F}, p)$. Note that if $\mathcal{F}$ contains only a single action function $f_a \in \mathcal{F}_{\text{actions}}$, the cluster transition conforms to an actor transition.

The flag r of a cluster transition determines if space and token availability checks have to be performed by transitions of child actors. It can only be set to $false$ if it is statically known that enough tokens are available on all ports of child actor transitions. Otherwise, it has to be set to $true$.

After space and token availability checks have been performed, the transition guards have to be evaluated. If they return $true$, the evaluation phase for the set of actions without predecessor $\mathcal{F}' = \{f_j \in \mathcal{F} \mid \nexists f_i \in \mathcal{F} : f_i \prec f_j\}$ is started. Obviously, for the subset of action functions $f_a \in \mathcal{F}' \cap \mathcal{F}_{\text{actions}}$ this is already done by evaluating the guard function of the clusters. However, child actor transitions have to be evaluated as explained above, resulting in a recursive evaluation. If each action $f \in \mathcal{F}'$ evaluates to $true$, the composite actor is activated.

Finally, in the execution phase, the actions $t.\mathcal{F}$ of a transition t are executed according to the partial order specified by "$\prec$". As in the evaluation phase only actions without predecessors have been evaluated, actions with predecessors are evaluated as soon as all predecessor actions have been executed. After the execution phase, the current actor mode q_{cur} is set to $t.q_{dst}$.

3.2.3 Scheduling Examples

The first example in Fig. 12 implements a quasi-static schedule (QSS). In short, a QSS is a schedule in which most scheduling decisions can be made statically at

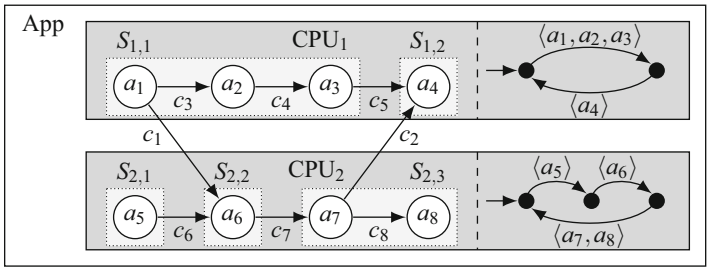


Fig. 12 Quasi-Static Scheduling (QSS) example

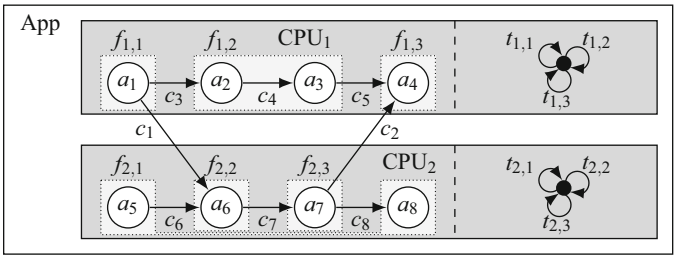


Fig. 13 Dynamic Scheduling example

compile time but some have to be decided by the run-time environment. Considering CPU$_1$, it consists of two partial PSOSs $S_{1,1}$ and $S_{1,2}$, one for each transition that can be described by two transitions according to Def. 9 as

$$t_{1,1} = (q_0, k^{io}_{1,1} \wedge k_{true}, \{a_1, a_2, a_3\}, q_1, \{a_1 \prec a_2 \prec a_3\}, false) \text{ and}$$

$$t_{1,2} = (q_1, k^{io}_{1,2} \wedge k_{true}, \{a_4\}, q_0, \emptyset, false).$$

At the beginning, the current actor mode q_{cur} is equal to q_0 and thus space and token availability checks have to be performed for PSOS $S_{1,1}$ according to $k^{io}_{1,1}$, which return $true$. As a_1-a_3 represent static actors, token and space availability checks are unnecessary for child actors (i.e. $r = false$). The transition guard k_{true} always evaluates to $true$. Thus, transition $t_{1,1}$ is activated and a_1-a_3 are executed sequentially as specified by "$\prec$". The remaining schedules are evaluated analogously.

Figure 13 implements a dynamic schedule. Child actors having the same communication behavior w.r.t. composite ports are grouped (e.g. actors a_5 and a_8). The resulting transitions for CPU$_2$ are outlined below:

$$t_{2,1} = (q_0, k^{io}_{2,1} \wedge k_{true}, \{a_5, a_8\}, q_0, \emptyset, true)$$

$$t_{2,2} = (q_0, k^{io}_{2,2} \wedge k_{true}, \{a_6\}, q_0, \emptyset, true)$$

$$t_{2,3} = (q_0, k^{io}_{2,1} \wedge k_{true}, \{a_7\}, q_0, \emptyset, true).$$

After evaluating the current actor mode, space and token checks, and the guard function, in contrast to the QSS scheduling scheme, token and space availability checks have to be performed for child actors (i.e. $r = true$). For example, if transition $t_{2,1}$ is activated, one transition of child actors a_5 or a_8 is selected according to space and token availability checks for the corresponding child actor transitions.

As the last example, consider Fig. 14 representing the partially-ordered transitions schedule (POTS) $S_{par} = \{a_1 \prec a_2 \prec a_3 \prec a_4, a_5 \prec a_6 \prec a_7 \prec a_8, a_1 \prec a_6, a_7 \prec a_4\}$. In contrast to the QSS schedule, the partial PSOSs are split after and before inter-processor communication occurs. However, the transition order is given

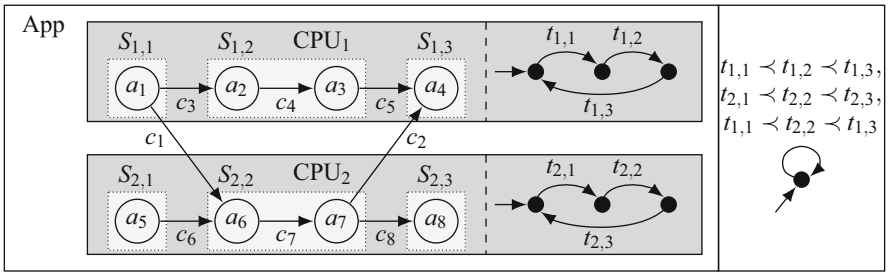


Fig. 14 Partially-ordered Transitions Scheduling (POTS) example

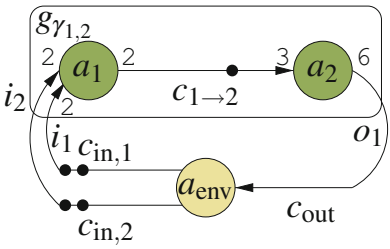


Fig. 15 Example of the DFG $a_{\gamma_{1,2}}$ containing a cluster $g_{\gamma_{1,2}}$ with two SDF actors a_1 and a_2 (green) connected to a cluster environment consisting of the sole Kahn actor a_{env} (yellow)

within the "App" cluster and thus token and space availability can be guaranteed statically (i.e. $r = false$). That is, a transition can directly be executed as soon as all predecessors have been executed without the need for checking the availability of tokens.

4 Exploiting Static MoCs for Scheduling

Generating *program code* from a DFG for microprocessor target requires a scheduling strategy for the actors mapped to this microprocessor. *Static scheduling* [6, 23] can be used for models with limited expressiveness, e.g., SDF and CSDF DFGs. However, real-world designs also contain dynamic actors, e.g., as seen in Fig. 15 where two static actors a_1 and a_2 are connected to the dynamic actor a_{env}. Unfortunately, existing algorithms [26, 29] might result in infeasible schedules or greatly restrict the clustering design space by considering only SDF subgraphs that can be clustered into monolithic SDF actors without introducing deadlocks. To exemplify, consider converting the cluster $g_{\gamma_{1,2}}$ into an SDF actor. This would deadlock the DFG g_{ex_1} as neither the environment actor a_{env} could fire (no tokens on its input channel c_{out}) nor the SDF actor that would have been derived from $g_{\gamma_{1,2}}$ (insufficient tokens, i.e., only 2 instead of the 6 required, on its input channels $c_{\text{in},1}$ and $c_{\text{in},2}$).

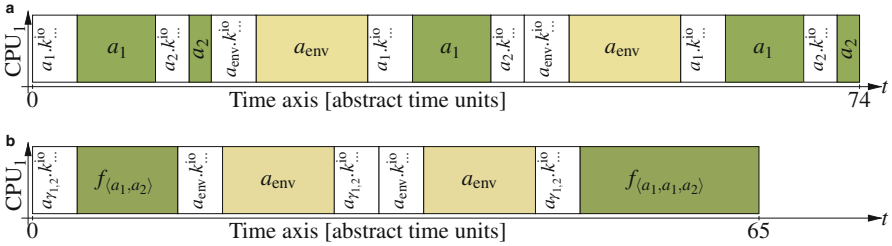


Fig. 16 Depicted above are two Gantt charts that could be encountered during scheduling of the DFG g_{ex_1} on the single resource CPU_1. Here, white boxes represent scheduling overhead that is encountered during the check if an actor can be fired. Conversely, the colored, i.e., green and yellow, boxes represent the actual useful computational payload of the actors a_1, a_2, and a_{env}. (**a**) Dynamic scheduling of the DFG. (**b**) Schedule overhead reduction due to employing a QSS

4.1 Scheduling Overhead

The most basic scheduling strategy is to postpone all scheduling decisions to run time, e.g., performing *dynamic scheduling* as depicted in Fig. 16a, with the resulting significant scheduling overhead and, hence, a reduced system performance. However, this strategy is suboptimal if it is known that some of the actors exhibit regular communication behavior like SDF and CSDF actors. The scheduling overhead problem can be mended by coding the actions of an actor at an appropriate level of granularity, i.e., combining as much functionality into one action such that the computation costs dominate the scheduling overhead, i.e., as is done in Fig. 16b where the composite actor $a_{\gamma_{1,2}}$ has the two actions $f_{\langle a_1,a_2\rangle}$ and $f_{\langle a_1,a_1,a_2\rangle}$ that combine the actor firings a_1 and a_2 as well as the actor firings a_1, a_1, and a_2, respectively.

However, as can be seen in Fig. 17, the appropriate level of granularity is dependent on the underlying architecture on which a DFG is scheduled for execution. Thus, if the distribution of actors to resources itself is part of the synthesis step—as it is in the case of design space exploration—an appropriate level of granularity can no longer be chosen in advance by the designer. Hence, computing QSSs must become part of the synthesis flow and should not be performed in advance. In the following section, we will demonstrate how QSSs can be derived for a cluster as given by Definition 8 and the derived schedule represented by a cluster FSM as defined in Definition 9.

4.2 Cluster FSM Computation for QSS

The key idea for producing a deadlock-free QSS is to assume the so-called "worst case" environment for the cluster. This is the case when each output port $o \in g_\gamma.O$

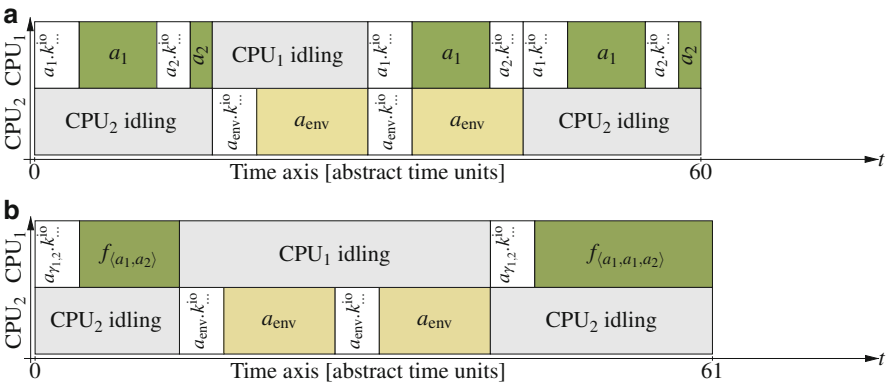


Fig. 17 Here, two Gantt charts demonstrate that quasi-static scheduling might trade scheduling efficiency, e.g., less overall computation in (**b**) as compared to (**a**), for increased latency, e.g., as seen for the second firing of actor a_1 that is delayed in (**b**) as compared to (**a**). Moreover, as exemplified above, this increased latency might also reduce the overall system throughput if multiple computational resources are available for execution of the DFG, e.g., the processors CPU_1 and CPU_2. (**a**) Dynamic scheduling of the DFG. (**b**) Example that a QSS may not always increase the system throughput

is part of a feedback loop to each input port $i \in g_\gamma.I$ and any produced token on an output port $o \in g_\gamma.O$ is needed for the activation of an actor $a \in g_\gamma.A$ in the same cluster through these feedback loops, e.g., as seen in Fig. 15. In particular, postponing the production of an output token results in a deadlock of the entire system. Thus, the cluster states $\mathbf{q} \in Q_\gamma \subset \mathbb{N}_0{}^{g_\gamma.A}$ are states of the cluster where a maximal number of output tokens have been produced from consumption of a minimal number of input tokens.[8] For the cluster $g_{\gamma_{1,2}}$, these states can be seen in Fig. 18. Briefly, the cluster state space is derived by starting with the initial state space set $\{\,\mathbf{q}_0\,\}$ that contains the initial state $\mathbf{q}_0 = (0, 0)$ that denotes that neither the actor a_1 nor the actor a_2 have fired previously. Next, for each cluster output port, e.g., only the port o_1 in case of the cluster $g_{\gamma_{1,2}}$, we demand successively more tokens on this port until the encountered cluster state $\mathbf{q}$ exceeds the repetition vector of the cluster, e.g., $\boldsymbol{\eta}^{\text{rep}}_{\gamma_{1,2}} = (3, 2)$ for cluster $g_{\gamma_{1,2}}$. For this purpose, we repeat the following three steps until the repetition vector of the cluster is exceeded: (i) To require the minimal number of input tokens, we search for the minimal number of actor firings that satisfy the production of one more token on the selected output port. Subsequently, (ii) to maximize the number of produced output tokens, we prohibit

[8]The initial state $\mathbf{q}_0$ might be an exception to this observation as the cluster might have an initial state where tokens can be produced without consuming additional inputs. In contrast to previous editions of this handbook and to ease the visualization, these cluster states are represented by the number of firings of each actor in the cluster instead of the number of tokens consumed and produced on each of cluster input and output ports. Both representations are equivalent and can be transformed into each other.

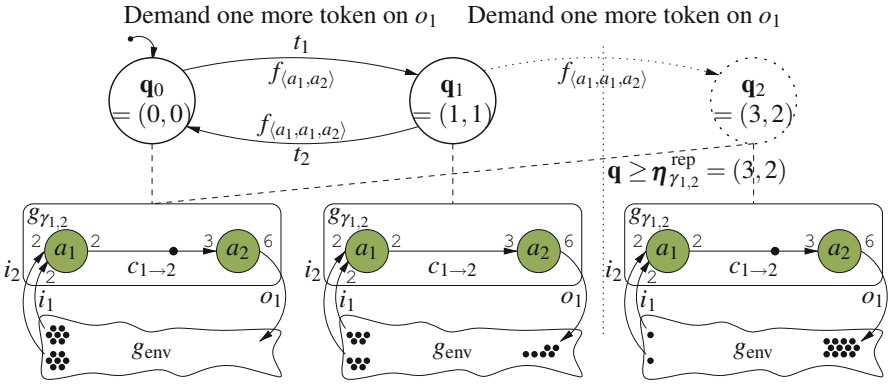


Fig. 18 Computing the cluster state space for the cluster $g_{\gamma_{1,2}}$

any more token consumptions on the cluster input ports and try to fire as many actors as is still possible under this constraints. Finally, (iii) we add the resulting state to the cluster state space. To exemplify, consider cluster $g_{\gamma_{1,2}}$, start state $\mathbf{q}_0$, and output port o_1. One more token on o_1 necessitates firing of actor a_2, which in turn demands the firing of actor a_1. No further firings are possible without consuming more inputs. Hence, we add $\mathbf{q}_1 = (1, 1)$ to the cluster state space. Next, we again require one more token and, thus, fire the actor sequence $\langle a_1, a_1, a_2\rangle$. As previously, no further firings are possible without consuming more inputs and, thus, we add $\mathbf{q}_2 = (3, 2)$ to the cluster state space. This terminates the loop.

To continue, we exploit the observation that for each cluster state that exceeds the repetition vector of the cluster, there exists an equivalent cluster state that does not exceed the vector that can be derived by successively subtracting the repetition vector, e.g., the equivalent state to $\mathbf{q}_2$ is $\mathbf{q}_0$. Thus, the cluster state space is finite if for all cluster outputs the produced output tokens eventually depend on tokens consumed on all cluster input ports. More formally, this means that there is a directed path in the cluster from each cluster input port to each cluster output port.

Finally, we derive the transitions of the cluster FSM by inserting transitions between each pair of cluster states that do not bypass another cluster state, e.g., if there would be another state $\mathbf{q}_{0'} = (1, 0)$ in Fig. 18, then transition t_1 would be replaced by a transition from $\mathbf{q}_0$ to $\mathbf{q}'_0$and a transition from $\mathbf{q}_{0'}$ to $\mathbf{q}_1$. For a more formal explanation of this condition and the algorithm to derive the *cluster FSM*, we refer the interested reader to [12, 14].

5 Quasi-Static Scheduling in the Presence of Bounded Channels

The QSS methodology presented in Sect. 4.2 does assume FIFOs with unbounded capacities. Hence, it is unsuitable for code generation for embedded system targets supporting only static memory allocation where FIFOs have to be preallocated with a fixed capacity.

To exemplify the problem, we consider Figs. 19 and 20. Here, the DFG g_{ex_1} is depicted without QSS (see Fig. 19) and with QSS for the cluster $g_{\gamma_{1,2}}$ (see Fig. 20). Moreover, the FIFO channels have been given channel capacities as depicted by the square boxes on the edges (channels) connecting the three actors. As can be seen by contrasting Figs. 19c and 20c, simply using the QSS previously computed for $g_{\gamma_{1,2}}$ without any modification to the capacities of the adjacent channels can cause deadlocks, e.g., here do to the inability of the composite actor $a_{\gamma_{1,2}}$ to fill channel $c_{1\to 2}$ to capacity. In the examples used here, the cluster environment is represented by a single actor named a_{env}. In reality, however, it represents the arbitrary complex subgraph of the DFG g that represents the DFG without its contained cluster.

5.1 *Channel Capacity Adjustment Problem*

Based on a QSS given as a cluster FSM (cf. Definition 9)—that is applicable to refine a DFG with unbounded channel capacities—we will define the formal problem of channel capacity adjustment that has to be solved in order to make the same QSS applicable for platforms only supporting DFGs with fixed channel capacities. For the work at hand, we assume that such a cluster FSM is computed for each cluster by methods known from literature [14, 16–18, 29, 35, 36]. Note that the determination of limited channel capacities that do not introduce any deadlock for a KPN is in general undecidable [8, 28]. Moreover, the channel capacities for the

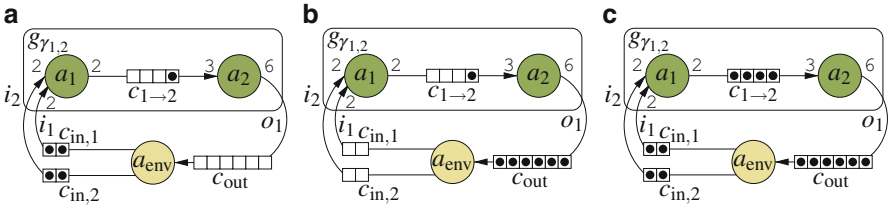


Fig. 19 Given above is the DFG g_{ex_1} with the cluster $g_{\gamma_{1,2}}$ that contains the two SDF (green) actors a_1 and a_2. The cluster environment contains the Kahn actor a_{env} (yellow) that for the first three actor firings is assumed to not consume anything and produce two tokens each on the channels $c_{\mathrm{in},1}$ and $c_{\mathrm{in},2}$. The three steps from (**a**) to (**c**) depict a sequence of actor firings that fills all FIFO channels to capacity. (**a**) Initial state. (**b**) After actor firing sequence $\langle a_1, a_2\rangle$. (**c**) After additional actor firings $\langle a_{\mathrm{env}}, a_1, a_{\mathrm{env}}, a_1, a_{\mathrm{env}}\rangle$

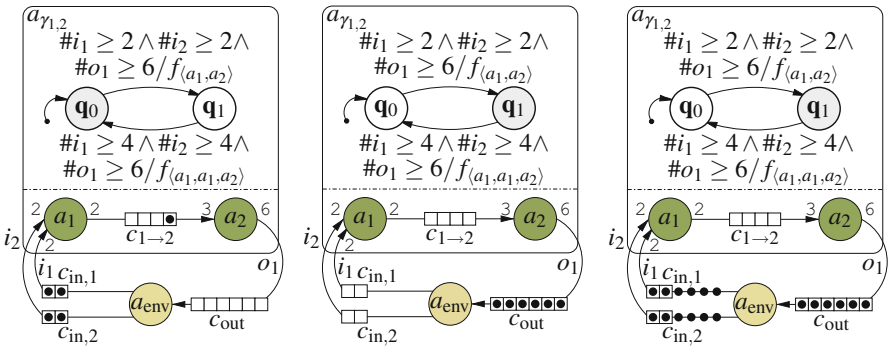


Fig. 20 Given above is the refined DFG $\widetilde{g_{\text{ex}_1}}$ that results from replacing the cluster $g_{\gamma_{1,2}}$ with the composite actor $a_{\gamma_{1,2}}$ implementing a QSS for the cluster. However, the refined graph cannot mirror the behavior of the original graph shown in Fig. 19. In particular, a deadlock/channel capacity overflow results in (**c**) while trying to reproduce the three firings of the actor a_{env} from Fig. 19c. (**a**) Initial state; Current state of $\mathcal{R}$ is $\mathbf{q}_0$ (gray). (**b**) After firing action $f_{\langle a_1,a_2\rangle}$; State change to $\mathbf{q}_1$ (gray). (**c**) Trying to reproduce the actor firings from Fig. 19c

FIFO channels connected to the cluster input and output ports of a cluster cannot be used unmodified from the original unrefined DFG for the refined DFG containing composite actors that implement a QSS. Thus, the channel capacities of the cluster input and output channels cannot be recomputed from scratch, but must be derived by computing an adjustment [12, 15] for the worst case that enlarges the channel capacities of the cluster input and output channels of the original unrefined DFG, e.g., the channel capacities $\mathbf{size}(c_{\text{in},1})$, $\mathbf{size}(c_{\text{in},2})$, and $\mathbf{size}(c_{\text{out}})$ might have to be increased in order to make the refinement of the DFG g_{ex_1} into the DFG $\widetilde{g_{\text{ex}_1}}$ via application of a QSS for the actors a_1 and a_2 feasible. More formally, this can be defined as follows:

Definition 10 (FIFO Channel Capacity Adjustment Vector) The *FIFO channel capacity adjustment vector* $\mathbf{adj} \in \mathbb{N}_0^{|I \cup O|}$[9] specifies a *sufficient increase* in channel capacities for each cluster input and output channel such that given *any deadlock free DFG g exhibiting KPN semantics* and containing the cluster g_γ, then replacing the cluster with its corresponding composite actor a_γ and increasing the channel capacities of the channels connected to this composite actor by the amount given by the adjustment vector results in a refined DFG $\widetilde{g}$ that is also free of any deadlock.

For the cluster $g_{\gamma_{1,2}}$ shown in Fig. 20, the adjustment vector is $g_{\gamma_{1,2}}.\mathbf{adj} = (n_{c_{\text{in},1}}, n_{c_{\text{in},2}}, n_{c_{\text{out}}}) = (4, 4, 0)$. Application of this adjustment vector to resolve the deadlock results in the DFG $\widetilde{g_{\text{ex}_1}}'$ shown in Fig. 21.

[9] In the following, we will drop the 'g_γ.'-prefix from various notations when the cluster is clear from context, e.g., I instead of $g_\gamma.I$.

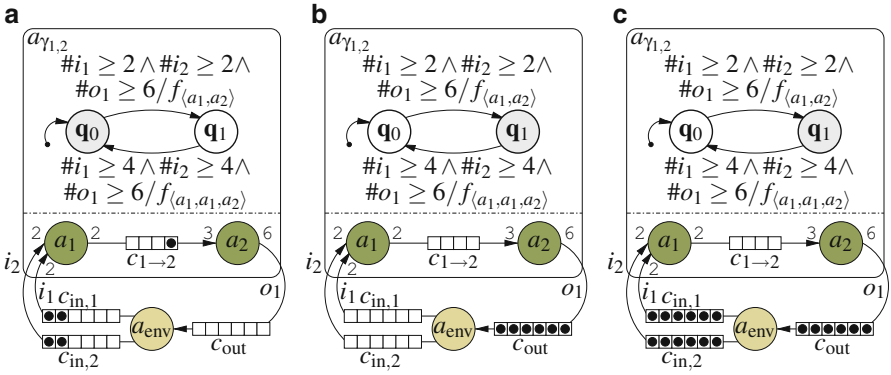


Fig. 21 Refined DFG $\widetilde{g_{\text{ex}_1}}'$ after channel adjustments to the adjacent channels of the composite actor $a_{\gamma_{1,2}}$ have been applied. (**a**) Initial state with adjusted channel capacities. (**b**) After firing action $f_{\langle a_1,a_2\rangle}$. (**c**) After actor firing sequence $\langle a_{\text{env}}, a_{\text{env}}, a_{\text{env}}\rangle$

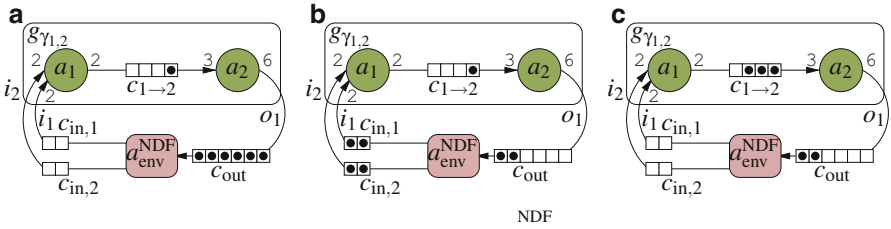


Fig. 22 Given above is the DFG g_{ex_2} with the cluster $g_{\gamma_{1,2}}$ that contains the two SDF actors a_1 and a_2 (green). The cluster environment contains the NDF actor $a_{\text{env}}^{\text{NDF}}$ (red) that in each firing forwards one token from the input channel to one of the output channels. If both output channels have at least one free place available, then one of them is randomly selected. (**a**) Initial state. (**b**) Four firings of $a_{\text{env}}^{\text{NDF}}$ enable actor a_1. (**c**) After firing of actor a_1

The strategy to enlarge the channel capacities of channels adjacent to a composite actor is applicable for DFGs where the increase in channel capacity will never introduce deadlocks. DFGs only containing Kahn actors, i.e., actors that do not change their behavior depending on when tokens or free places appear on their input or output channels, have this property. However, as soon as Non-determinate Dataflow (NDF) [25] is considered, this property no longer holds. To exemplify, we consider the identical cluster $g_{\gamma_{1,2}}$ in a different cluster environment, i.e., the DFG g_{ex_2} shown in Fig. 22 containing an NDF environment actor $a_{\text{env}}^{\text{NDF}}$ (red).

The three steps from Fig. 22a–c depict a sequence of actor firings that enable one firing of the actor a_1. The actor a_1 will be enabled no matter in which order actor $a_{\text{env}}^{\text{NDF}}$ selects the output channels for token forwarding as both channels $c_{\text{in},1}$ and $c_{\text{in},2}$ need to be filled to capacity before actor a_1 can fire. However, if the channel capacity of $c_{\text{in},1}$ or $c_{\text{in},2}$ is enlarged sufficiently – as is the case in Fig. 23, then the

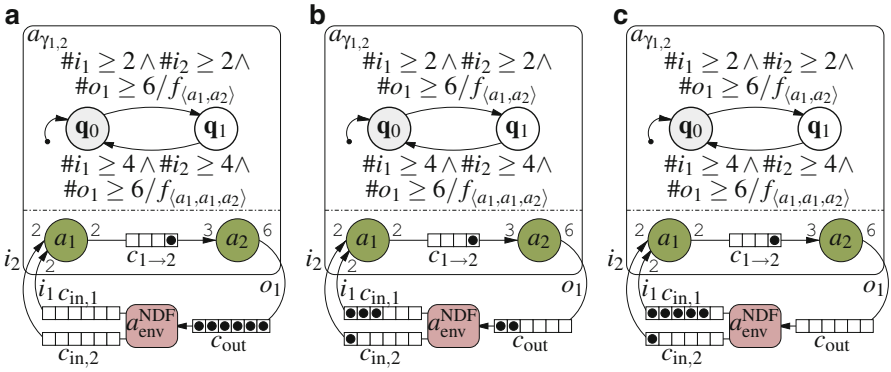


Fig. 23 Depicted above is the refined DFG $\widetilde{g_{\text{ex}_2}}'$ that has been obtained by substituting the cluster $g_{\gamma_{1,2}}$ by its composite actor $a_{\gamma_{1,2}}$ and applying the channel adjustments to the adjacent channels of the composite actor $a_{\gamma_{1,2}}$. However, in this case, the increase in the channel capacities of the channels $c_{\text{in},1}$ and $c_{\text{in},2}$ does introduce an artificial deadlock into $\widetilde{g_{\text{ex}_2}}'$. (**a**) Initial state. (**b**) After four firings of a_{env}. (**c**) Deadlock after two more firings of a_{env}

NDF actor $a_{\text{env}}^{\text{NDF}}$ might distribute the six tokens from c_{out} to the channels $c_{\text{in},1}$ and $c_{\text{in},2}$ in such a way that actor a_1 will not be enabled. Thus, resulting in the deadlock depicted in Fig. 23c.

5.2 *Channel Capacity Adjustment Algorithm*

In the following, the reasons for the introduction of artificial deadlocks due to limited channel capacities will be detailed and a FIFO channel capacity adjustment algorithm [12, 15] will be introduced. This adjustment algorithm ensures that cluster refinement will not introduce an artificial deadlock into the refined DFG if the original unrefined DFG is a KPN. Note that the presented algorithm *does not require any knowledge—indeed, the key aspect of the problem at hand is exactly the lack of such knowledge—of the cluster environment*. Nevertheless, for illustration purposes, cluster environments will be modeled by SDF or CSDF actors in the examples presented in this section.

In general, a QSS ensures that there are no artificial deadlocks caused by missing tokens on feedback loops from the cluster output ports to the cluster input ports. If the channel capacities are limited, however, clustering might introduce artificial deadlocks due to back pressure. Back pressure induced deadlock denotes the fact that there exists a (reverse) feedback loop of channels and actors where all actors in the loop cannot fire because free places are missing on their output ports connected to the channels contained in the loop. Three reasons can be identified for the introduction of artificial deadlocks due to limited capacities. These reasons are caused by back pressure from

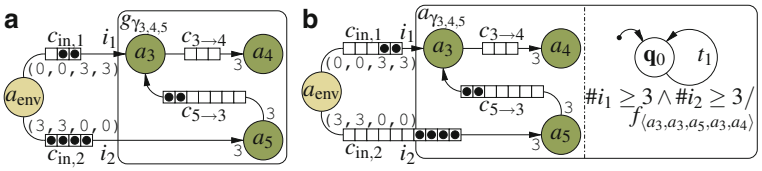


Fig. 24 Here, the DFG g_{ex_3} is used to generalize the input-to-input back pressure problem to multiple input ports. (**a**) Original unrefined DFG g_{ex_3} with its cluster $g_{\gamma_{3,4,5}}$. (**b**) Refined DFG $\widetilde{g_{\text{ex}_3}}$ containing the composite actor $a_{\gamma_{3,4,5}}$

(A) missing free places on feedback loops from one cluster input port to another cluster input port, i.e., *input-to-input back pressure*, and
(B) missing free places on feedback loops from one cluster output port to another cluster output port, i.e., *output-to-output back pressure*.
(C) missing free places on feedback loops from the cluster input ports to the cluster output ports, i.e., *input-to-output back pressure*,

First, in Sects. 5.2.1 and 5.2.2, the solutions given by the channel capacity adjustments $\mathbf{adj}_{\text{i2i}}$ and $\mathbf{adj}_{\text{o2o}}$ for the reasons (A) and (B) will be discussed. Then, in Sect. 5.2.3, we will discuss the channel capacity adjustments $\mathbf{adj}_{\text{i2o}}$ required to solve reason (C) concerned with deadlocks due to back pressure from a cluster input port to a cluster output port.

5.2.1 Input-to-Input Back Pressure

To exemplify, we will use the DFG g_{ex_3} shown in Fig. 24 that exhibits the input-to-input back pressure problem in isolation. That is to say, input-to-output and output-to-output back pressure do not pose a problem for this example as the cluster $g_{\gamma_{3,4,5}}$ has no output ports.

In the following, we will present a critical scenario that leads to an artificial deadlock due to reason (A). As can be seen in Fig. 24a, the cluster environment a_{env} tries to produce six tokens onto the cluster input channel $c_{\text{in},2}$ by producing two batches of three tokens each. In the unrefined case, there are sufficient channel capacities inside the cluster to facilitate the consumption of six tokens from the cluster input channel $c_{\text{in},2}$, i.e., fire actor a_5 once, consuming the first batch of three tokens from cluster input channel $c_{\text{in},2}$, fire actor a_3 once, consuming one of the initial tokens provided in the cluster input channel $c_{\text{in},1}$, and fire actor a_5 once more, consuming the second batch of three tokens from cluster input channel $c_{\text{in},2}$.

On the other hand, after $g_{\gamma_{3,4,5}}$ has been refined into the composite actor $a_{\gamma_{3,4,5}}$ (see Fig. 24b), the transition t_1 requires three tokens on the cluster input ports i_1 and i_2 before it can be taken. Moreover, as the cluster environment a_{env} does not provide any additional tokens on the cluster input channel $c_{\text{in},1}$, the transition t_1 cannot be taken. Hence, assuming that the channel capacity of the cluster input channel $c_{\text{in},2}$ is still four tokens, an artificial deadlock results due to reason (A). To compensate,

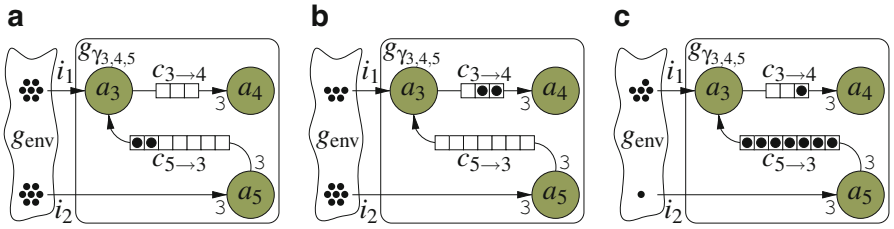


Fig. 25 In the above given example, a cluster environment g_{env} that provides an infinite number of tokens is assumed. For illustration purpose, seven tokens are chosen for each cluster input port. Three situations distinguished by their corresponding state of the cluster are shown. The original situation ($\mathbf{q}_0$) is given in (**a**). The situation ($\mathbf{q}_{\text{dyn},1}$) where the cluster consumes the maximal number of tokens (here two tokens) from input port i_1 while still not activating transition t_1 is given in (**b**). Conversely, the situation ($\mathbf{q}_{\text{dyn},2}$) where the cluster consumes the maximal number of tokens (here six tokens) from input port i_2 while still not activating transition t_1 is shown in (**c**)

the channel capacities must be adjusted as follows:

$$\widetilde{g_{\text{ex}_3}}.\,\mathbf{size}(c_{\text{in},1}) = g_{\text{ex}_3}.\,\mathbf{size}(c_{\text{in},1}) + n_{c_{\text{in},1}} = 3 + 2 = 5$$
$$\widetilde{g_{\text{ex}_3}}.\,\mathbf{size}(c_{\text{in},2}) = g_{\text{ex}_3}.\,\mathbf{size}(c_{\text{in},2}) + n_{c_{\text{in},2}} = 4 + 6 = 10$$

These adjustments correspond to the number of tokens that can be consumed by the cluster on its two different input ports while still consuming less tokens on both input ports than would be necessary to activate the transition t_1. For a better depiction of the number of consumed tokens, we start from the situation as shown in Fig. 25a. As can be seen in Fig. 25c, the cluster $g_{\gamma_{3,4,5}}$ can consume up to six tokens on its input port i_2 (fire actor a_3 once and actor a_5 twice) without enabling the transition t_1. Thus, the channel capacity of the cluster input channel $c_{\text{in},2}$ must be increased by six tokens, i.e., $n_{c_{\text{in},2}} = 6$. Consumption of more than six tokens would require nine tokens on input port i_2 and four tokens on input port i_1. Hence, enabling the input/output guard $k^{\text{io}} = \#i_1 \geq 3 \wedge \#i_2 \geq 3$ of the transition t_1. An equivalent observation (see Fig. 25b) can be made for the input port i_1, which can consume up to two tokens without enabling the transition t_1. For a more formal definition of the $\mathbf{adj}_{\text{i2i}}$ function, we refer the reader to [12, 15].

5.2.2 Output-to-Output Back Pressure

The problem of output-to-output back pressure can even occur if the cluster is a pure source for the DFG. We will use this to consider the problem in isolation by analyzing the output-to-output back pressure problem for the cluster $g_{\gamma_{7,4,6}}$ shown in Fig. 26. As this cluster has no input ports, input-to-output and input-to-input back pressure does not occur.

As in the previous case, we will start with the description of a critical scenario for the output-to-output back pressure problem in the DFG g_{ex_4}. In this scenario, the

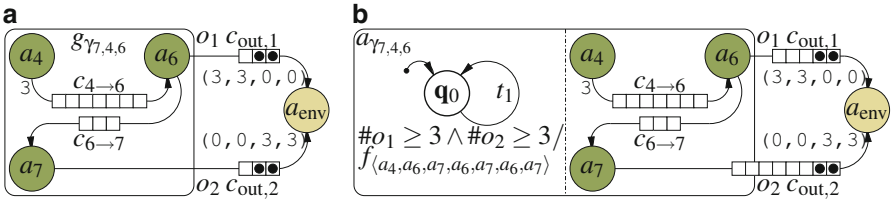


Fig. 26 For illustration of how we generalize the solution of the output-to-output pack pressure problem from one to multiple output ports the above given DFG g_{ex_4} is employed. (**a**) Original unrefined DFG g_{ex_4} with its cluster $g_{\gamma_{7,4,6}}$. (**b**) Refined DFG $\widetilde{g_{\text{ex}_4}}$ containing the composite actor $a_{\gamma_{7,4,6}}$

cluster environment a_{env} pulls six tokens from the cluster output channel $c_{\text{out},1}$ by twice consuming three tokens from the channel. This can be accommodated by the unrefined DFG g_{ex_4} by firing both actors a_6 and a_7 once, a consumption of the first batch of three tokens by the cluster environment a_{env}, three more firings of actor a_6, and the consumption of the second batch of three tokens by the cluster environment a_{env}. After this sequence of actor firings, the channel $c_{6\to7}$ is filled to capacity.

On the other hand, the composite actor $a_{\gamma_{7,4,6}}$ is always producing the maximum number of output tokens, thus always producing all tokens remaining in channel $c_{6\to7}$ to the cluster output channel $c_{\text{out},2}$. Hence, assuming that the channel capacity of the cluster output channel $c_{\text{in},2}$ is still three tokens, an artificial deadlock results due to reason (B). To compensate, the channel capacities must be adjusted as follows:

$$\widetilde{g_{\text{ex}_4}}.\,\textbf{size}(c_{\text{out},1}) = g_{\text{ex}_4}.\,\textbf{size}(c_{\text{out},1}) + n_{c_{\text{out},1}} = 3 + 2 = 5$$
$$\widetilde{g_{\text{ex}_4}}.\,\textbf{size}(c_{\text{out},2}) = g_{\text{ex}_4}.\,\textbf{size}(c_{\text{out},2}) + n_{c_{\text{out},2}} = 3 + 5 = 8$$

To compute these adjustment values, we determine for the transition t_1 and each output port the maximal number of token productions that can be undone[10] on this port while still not having undone all produced tokens of transition t_1. To enable a graphical illustration of undoing token productions, we assume the original situation given in Fig. 27a where the cluster environment has consumed seven tokens from each cluster output port.

As can be seen in Fig. 27c, five token productions can be undone on the cluster output port o_2 while still allowing the production of at least one token that would also have been produced by the transition t_1, e.g., the token produced by the first firing of actor a_6 of the transition t_1. Hence, the channel capacity of the cluster

[10]In reality, the cluster will not undo any actor firing and this should be thought of as delayed production of tokens by a dynamically scheduled cluster in contrast to a composite actor implementing a QSS. A QSS is required to always produce the maximal number of output tokens from a minimal number of input tokens and, hence, will never delay the production of tokens on its output ports.

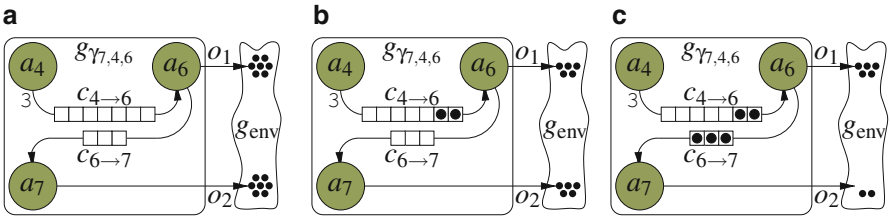


Fig. 27 To illustrate the undoing of actor firings of the cluster $g_{\gamma_{7,4,6}}$, we assume the situation in (**a**) where the cluster environment g_{env} has consumed seven tokens provided on each cluster output port. Moreover, two situations resulting from undoing actor firings starting from (**a**) are shown in (**b**) and (**c**). These situations correspond to the dynamic cluster states $\mathbf{q}_{\text{dyn},1}$ and $\mathbf{q}_{\text{dyn},2}$, respectively. In particular, state $\mathbf{q}_{\text{dyn},1}$ corresponds to the situation where the cluster has undone the maximal number of token productions (here two tokens) on the output port o_1 while still not having undone all produced tokens of transition t_1. Likewise, state $\mathbf{q}_{\text{dyn},2}$ has undone the maximal number of token productions (here five tokens) on the output port o_2

output channel $c_{\text{out},2}$ has been increased by five tokens, i.e., $n_{c_{\text{out},2}} = 5$. An equivalent observation (see Fig. 27b) can be made for the output port o_1. In this case, two firings of the actor a_6 can be undone while still producing at least one token, i.e., the token produced by the first firing of actor a_6. Thus, the channel capacity of the cluster output channel $c_{\text{out},1}$ has been increased by two tokens, i.e., $n_{c_{\text{out},1}} = 2$. If the first firing of actor a_6 is also undone, then the three firings of actor a_7 have also to be reversed. Hence, all of the tokens produced by the transition t_1 have been undone. For a more formal definition of the $\mathbf{adj}_{\text{o2o}}$ function, we refer the reader to [12, 15].

5.2.3 Input-to-Output Back Pressure

While reasons (A) and (B) can occur even if the composite actor is only a sink or source actor, reason (C) can only occur if the cluster has both inputs and outputs. To exemplify, we again consider DFG $\widetilde{g_{\text{ex}_1}}$ with its cluster $g_{\gamma_{1,2}}$ and analyse it in detail. In Fig. 20, a scenario is presented that is deadlocking due to reasons (A) and (C). Obviously (see Fig. 19), when the unrefined DFG g_{ex_1} is dynamically scheduled, the specified FIFO channel capacities are sufficient. After this scenario has finished, all channels $c_{\text{in},1}$, $c_{\text{in},2}$, c_{out}, and $c_{1\to2}$ are filled to capacity.

Note that the cluster state space may contain more than one state and that the *cluster environment* is not aware of the current state of a cluster. Moreover, a deadlock-free execution of the DFG must be guaranteed for each state. Thus, the analyses of the required channel capacity adjustments given by the three functions $\mathbf{adj}_{\text{i2o}}$, $\mathbf{adj}_{\text{i2i}}$ and $\mathbf{adj}_{\text{o2o}}$ are performed for each state of the cluster state space and a *pointwise maximum* of the adjustments required by the individual states is computed. For this purpose, the notation $\max \boldsymbol{X}$ is used to denote the pointwise maximum of a set of vectors, e.g., $\max\{(1,0),(0,1)\} = (1,1)$. As an example, the

Table 2 The three parts of the channel capacity adjustments for the cluster $g_{\gamma_{1,2}}$

	$\mathbf{adj}_{\text{o2o}}(\mathbf{q})$ $(n_{c_{\text{out}}})$	$\mathbf{adj}_{\text{i2i}}(\mathbf{q})$ $(n_{c_{\text{in},1}}, n_{c_{\text{in},2}})$	$\mathbf{adj}_{\text{i2o}}(\mathbf{q}, (0))$ $(n_{c_{\text{in},1}}, n_{c_{\text{in},2}})$
$\mathbf{q} = \mathbf{q}_0$	(0)	(0, 0)	(2, 2)
$\mathbf{q} = \mathbf{q}_1$	(0)	(2, 2)	(4, 4)
$\max\{\ldots \mid \mathbf{q} \in Q_\gamma\}$	(0)	(2, 2)	(4, 4)

results of the three different analyses for the cluster $g_{\gamma_{1,2}}$ are given in Table 2. Given these three adjustment functions for $g_{\gamma_{1,2}}$, the FIFO channel capacity adjustment vector $g_{\gamma_{1,2}}.\mathbf{adj}$ can be computed as follows:

$$\pi_O(\mathbf{adj}) = \max\{\, \mathbf{adj}_{\text{o2o}}(\mathbf{q}) \mid \mathbf{q} \in Q_\gamma \,\} \tag{1}$$

$$\boldsymbol{n}_{\text{i2i}} = \max\{\, \mathbf{adj}_{\text{i2i}}(\mathbf{q}) \mid \mathbf{q} \in Q_\gamma \,\} \tag{2}$$

$$\boldsymbol{n}_{\text{i2o}} = \max\{\, \mathbf{adj}_{\text{i2o}}(\mathbf{q}, \pi_O(\mathbf{adj})) \mid \mathbf{q} \in Q_\gamma \,\} \tag{3}$$

$$\pi_I(\mathbf{adj}) = \begin{cases} \max\{\, \boldsymbol{n}_{\text{i2i}}, \boldsymbol{n}_{\text{i2o}} \,\} & \text{if } O \neq \emptyset \\ \boldsymbol{n}_{\text{i2i}} & \text{otherwise} \end{cases} \tag{4}$$

First, in Eq. (1), the output-to-output back pressure problem is considered. As can be seen in Table 2, reason (B) is not a problem for the cluster $g_{\gamma_{1,2}}$, i.e., $\mathbf{adj}_{\text{o2o}}(\mathbf{q}) = (0)\ \forall \mathbf{q} \in Q_\gamma$. This can be easily deduced as reason (B) appears when a transition can be partially undone from the perspective of tokens production by a transition. However, both transitions in Fig. 20 have only a single output actor firing and, thus, there is no possibility for a partial undo from the perspective of token production.

Next, in Eq. (2), the input-to-input back pressure problem is considered. Reason (A) appears when the cluster can consume tokens but no transition is enabled from the perspective of consuming tokens. Considering $g_{\gamma_{1,2}}$ again, we notice that it can only consume tokens if actor a_1 is fired. In state $\mathbf{q}_0$ the outgoing transition contains a single firing of a_1 and, thus, there is no input-to-input back pressure problem for state $\mathbf{q}_0$, i.e., $\mathbf{adj}_{\text{i2i}}(\mathbf{q}_0) = (0, 0)$. However, in state $\mathbf{q}_1$, the sole outgoing transitions contains two firings of a_1 and, thus, the cluster $g_{\gamma_{1,2}}$ can consume tokens without enabling this transition by firing actor a_1 only once. To compensate, the channel capacities of the two input channels $c_{\text{in},1}$ and $c_{\text{in},2}$ must be increased by two free places each, i.e., $\mathbf{adj}_{\text{i2i}}(\mathbf{q}_1) = (2, 2)$. However (see Fig. 20c), this is still insufficient to prevent artificial deadlock due to reason (C).

Subsequently, in Eq. (3), the input-to-output back pressure problem is handled. This problem only appears if the cluster has both input as well as output ports. Here, the fact that the cluster output channel capacities will be enlarged by the vector specified by $\pi_O(\mathbf{adj})$—not the case for cluster $g_{\gamma_{1,2}}$ where $\pi_O(\mathbf{adj}) = (0)$—can be used to minimize the required adjustments on the cluster input channels. To compute $\mathbf{adj}_{\text{i2o}}$, a cluster state $\mathbf{q}_{\text{dyn}} \in Q_\gamma^{\text{dyn}}$ that could have been encountered during dynamic scheduling of the cluster is searched for. This cluster state $\mathbf{q}_{\text{dyn}}$ must have the property that it stores a local maximum of tokens inside the cluster. The number

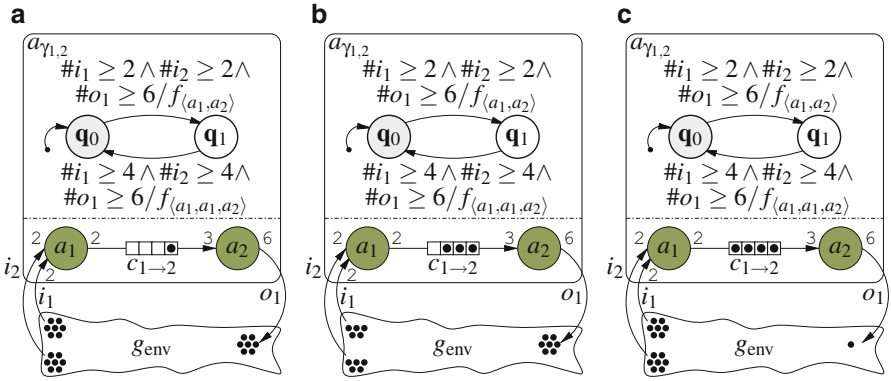


Fig. 28 Analysis of the input-to-output back pressure problem starting from cluster state $\mathbf{q}_0$. (**a**) Original situation in cluster state $\mathbf{q}_0$. (**b**) Dynamic cluster state $\mathbf{q}_{\text{dyn},1}$ reachable by firing actor a_1 once. (**c**) Dynamic cluster state $\mathbf{q}_{\text{dyn},2}$ reachable by undoing one firings of the actor a_2

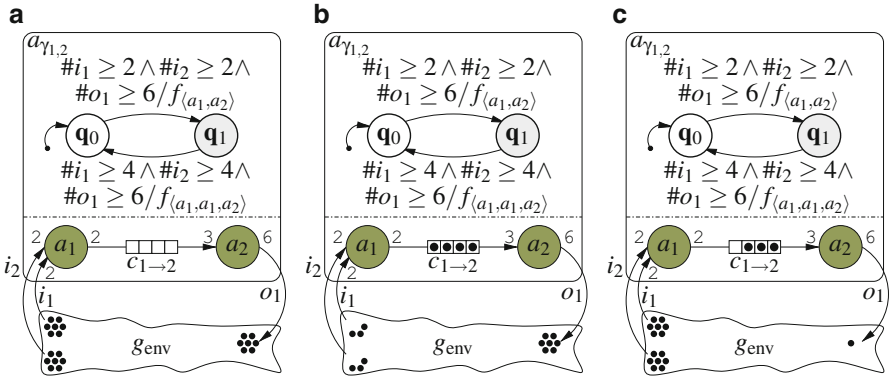


Fig. 29 Analysis of the input-to-output back pressure problem starting from cluster state $\mathbf{q}_1$. (**a**) Original situation in cluster state $\mathbf{q}_1$. (**b**) Dynamic cluster state $\mathbf{q}_{\text{dyn},1}$ reachable by firing actor a_1 twice. (**c**) Dynamic cluster state $\mathbf{q}_{\text{dyn},2}$ reachable by undoing one firing of the actor a_2

of tokens stored inside a cluster is maximized by starting from the cluster state $\mathbf{q} \in Q_\gamma$ consuming tokens from the cluster inputs and undoing the production of produced tokens on the cluster output ports. In case of cluster $g_{\gamma_{1,2}}$, the state where a local—even a global—maximum of tokens is stored inside the cluster is given when the channel $c_{1\to2}$ is filled to capacity. The two cluster states $\mathbf{q}_0$ and $\mathbf{q}_1$ of cluster $g_{\gamma_{1,2}}$ are analysed in Figs. 28 and 29, respectively.

As can be seen in Fig. 28, two dynamic cluster states $\mathbf{q}_{\text{dyn},1}$ and $\mathbf{q}_{\text{dyn},2}$ of a local—or even global—maximum token storage are derived from the original state $\mathbf{q}_0$. The first dynamic cluster state $\mathbf{q}_{\text{dyn},1}$ (see Fig. 28b) is reached from the original state $\mathbf{q}_0$ by firing actor a_1 once, thus consuming two tokens from each of the input ports i_1 and i_2, i.e., $\mathbf{adj}_{\text{i2o}}(\mathbf{q}_0, (0)) = (2, 2)$. The second dynamic cluster state $\mathbf{q}_{\text{dyn},2}$

(see Fig. 28c) is reachable by undoing one firing of the actor a_2, thus undoing the production of six tokens from port o_1, i.e., $\mathbf{adj}_{\mathrm{i2o}}(\mathbf{q}_0, (6)) = (0, 0)$. Both situations represent local maxima of stored tokens inside $g_{\gamma_{1,2}}$ as no additional tokens can be stored in the cluster without also removing some tokens from the cluster.

The same analysis (see Fig. 29) must also be performed starting from cluster state $\mathbf{q}_1$. Here, we have $\mathbf{adj}_{\mathrm{i2o}}(\mathbf{q}_1, (0)) = (4, 4)$ or alternatively $\mathbf{adj}_{\mathrm{i2o}}(\mathbf{q}_1, (6)) = (0, 0)$. As we do not enlarge the output channels, the values from $\mathbf{adj}_{\mathrm{i2o}}(\mathbf{q}_0, (0)) = (2, 2)$ and $\mathbf{adj}_{\mathrm{i2o}}(\mathbf{q}_1, (0)) = (4, 4)$ have been selected to solve the input-to-output back pressure problem. Thus, the values for $\mathbf{adj}_{\mathrm{i2o}}$ in Table 2 are explained and the updated channel capacities given below can be computed according to Eqs. (1) to (4). Finally, the channels can be adjusted as follows leading to the refined system shown in Fig. 21.

$$\widetilde{g_{\mathrm{ex}_1}}.\mathbf{size}(c_{\mathrm{in},1}) = g_{\mathrm{ex}_1}.\mathbf{size}(c_{\mathrm{in},1}) + \pi_{i_1}(g_{\gamma_{1,2}}.\mathbf{adj}) = 2 + 4 = 6$$
$$\widetilde{g_{\mathrm{ex}_1}}.\mathbf{size}(c_{\mathrm{in},2}) = g_{\mathrm{ex}_1}.\mathbf{size}(c_{\mathrm{in},1}) + \pi_{i_2}(g_{\gamma_{1,2}}.\mathbf{adj}) = 2 + 4 = 6$$
$$\widetilde{g_{\mathrm{ex}_1}}.\mathbf{size}(c_{\mathrm{out}}) = g_{\mathrm{ex}_1}.\mathbf{size}(c_{\mathrm{in},2}) + \pi_{o_1}(g_{\gamma_{1,2}}.\mathbf{adj}) = 6 + 0 = 6$$

For a more formal definition of the $\mathbf{adj}_{\mathrm{i2o}}$ function, we again refer the reader to [12, 15].

6 Conclusions

The complexity of many applications requires dataflow modeling approaches to reflect their dynamic behavior. Still, these applications very often contain large isles of static actors, which are amenable to compile-time analysis. In this chapter, we presented several models of computation, which are able to integrate static dataflow with a dynamic dataflow environment while still preserving the analyzability of the static parts of a given dataflow graph. The incorporation of *finite state machines* has proven invaluable for the coordination between the static and the dynamic parts within DFGs. Apart from introducing and comparing recent modeling approaches that are combining dataflow models with FSMs, we present results on *quasi-static scheduling*, which reduces the scheduling overhead when mapping networks of dataflow actors to modern multiprocessor, i.e., MPSoC targets. It is also shown that such scheduling decisions may be presented as well by FSMs. Moreover, quasi-static scheduling may be incorporated as a model transformation to reflect structural and mapping decisions at design time in order to optimize for throughput, or to minimize buffering requirements.

References

1. Bacivarov, I., Haid, W., Huang, K., Thiele, L.: Methods and tools for mapping process networks onto multi-processor systems-on-chip. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
2. Baird, M. (ed.): IEEE Standard 1666–2005 SystemC Language Reference Manual. IEEE Standards Association, New Jersey, USA (2005)
3. Balarin, F., Giusto, P., Jurecska, A., Passerone, C., Sentovich, E., Tabbara, B., Chiodo, M., Hsieh, H., Lavagno, L., Sangiovanni-Vincentelli, A., Suzuki, K.: Hardware-Software Co-Design of Embedded Systems: The POLIS Approach. Kluwer Academic Publishers (1997)
4. Bambha, N., Kianzad, V., Khandelia, M., Bhattacharyya, S.S.: Intermediate representations for design automation of multiprocessor dsp systems. Design Automation for Embedded Systems **7**(4), 307–323 (2002). https://doi.org/10.1023/A:1020307222052
5. Bhattacharya, B., Bhattacharyya, S.: Parameterized dataflow modeling for DSP systems. Signal Processing, IEEE Transactions on **49**(10), 2408–2421 (2001)
6. Bhattacharyya, S.S., Buck, J.T., Ha, S., Lee, E.A.: Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms. IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications **42**(3), 138–150 (1995)
7. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.: Cyclo-Static Dataflow. IEEE Transaction on Signal Processing **44**(2), 397–408 (1996)
8. Buck, J.T.: Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model. Tech. rep., Dept. of EECS, UC Berkeley, Berkeley, CA 94720, U.S.A. (1993). Technical Report UCB/ERL 93/69, Ph.D dissertation
9. Damavandpeyma, M., Stuijk, S., Basten, T., Geilen, M., Corporaal, H.: Modeling static-order schedules in synchronous dataflow graphs. In: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 775–780. EDA Consortium (2012)
10. de Groote, R.: Throughput analysis of dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
11. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity - the ptolemy approach. Proceedings of the IEEE **91**(1), 127–144 (2003)
12. Falk, J.: A Clustering-Based MPSoC Design Flow for Data Flow-Oriented Applications. Dr. Hut, Sternstr. 18, München, Germany (2015). Dissertation, Computer Science Department, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
13. Falk, J., Haubelt, C., Teich, J.: Efficient Representation and Simulation of Model-Based Designs in SystemC. In: Proc. Forum on Specification & Design Languages, FDL'06, pp. 129–134 (2006)
14. Falk, J., Keinert, J., Haubelt, C., Teich, J., Bhattacharyya, S.S.: A Generalized Static Data Flow Clustering Algorithm for MPSoC Scheduling of Multimedia Applications. In: Proc. 8th ACM international conference on Embedded software, EMSOFT'08, pp. 189–198. ACM, New York, NY, USA (2008). http://doi.acm.org/10.1145/1450058.1450084
15. Falk, J., Schwarzer, T., Glaß, M., Teich, J., Haubelt, C.: Quasi-Static Scheduling of Data Flow Graphs in the Presence of Limited Channel Capacities. In: Proc. of the 13th IEEE Symposium on Embedded Systems for Real-time Multimedia, ESTIMEDIA'15, p. 10 (2015)
16. Falk, J., Zebelein, C., Haubelt, C., Teich, J.: A Rule-Based Static Dataflow Clustering Algorithm for Efficient Embedded Software Synthesis. In: Proc. Design, Automation and Test in Europe, DATE'11, pp. 521–526. IEEE (2011)
17. Falk, J., Zebelein, C., Haubelt, C., Teich, J.: A Rule-Based Quasi-Static Scheduling Approach for Static Islands in Dynamic Dataflow Graphs. ACM Trans. Embedded Comput. Syst. **12**(3), 74:1–74:31 (2013)
18. Falk, J., Zebelein, C., Keinert, J., Haubelt, C., Teich, J., Bhattacharyya, S.S.: Analysis of SystemC Actor Networks for Efficient Synthesis. ACM Trans. Embedded Comput. Syst. **10**(2), 18:1–18:34 (2011). http://doi.acm.org/10.1145/1880050.1880054

19. Geilen, M., Basten, T.: Kahn process networks and a reactive extension. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
20. Geilen, M., Stuijk, S.: Worst-case performance analysis of synchronous dataflow scenarios. In: Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pp. 125–134. ACM (2010)
21. Girault, A., Lee, B., Lee, E.: Hierarchical finite state machines with multiple concurrency models. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **18**(6), 742–760 (1999)
22. Ha, S., Oh, H.: Decidable signal processing dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
23. Hsu, C.J., Bhattacharyya, S.S.: Cycle-Breaking Techniques for Scheduling Synchronous Dataflow Graphs. Tech. Rep. UMIACS-TR-2007-12, Institute for Advanced Computer Studies, University of Maryland at College Park (2007). URL http://hdl.handle.net/1903/4328
24. Kahn, G.: The Semantics of a Simple Language for Parallel Programming. In: IFIP Congress, pp. 471–475 (1974)
25. Kosinski, P.R.: A Straightforward Denotational Semantics for Non-determinate Data Flow Programs. In: Proc. 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL'78, pp. 214–221. ACM, New York, NY, USA (1978). https://doi.org/10.1145/512760.512783
26. Lee, E.A., Messerschmitt, D.G.: Synchronous Data Flow. Proc. of the IEEE **75**(9), 1235–1245 (1987)
27. Mattavelli, M., Janneck, J.W., Raulet, M.: MPEG reconfigurable video coding. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
28. Parks, T.M.: Bounded Scheduling of Process Networks. Tech. rep., Dept. of EECS, UC Berkeley, Berkeley, CA 94720, U.S.A. (1995). URL http://www.eecs.berkeley.edu/Pubs/TechRpts/1995/2926.html. Technical Report UCB/ERL M95/105, Ph.D dissertation
29. Pino, J.L., Bhattacharyya, S.S., Lee, E.: A Hierarchical Multiprocessor Scheduling System for DSP Applications. In: Proc. Asilomar Conference on Signals, Systems, and Computers, vol. 1, pp. 122–126 (1995). http://dx.doi.org/10.1109/ACSSC.1995.540525
30. Plishker, W., Sane, N., Kiemb, M., Bhattacharyya, S.S.: Heterogeneous design in functional DIF. In: Proceedings of the 8th international workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS '08, pp. 157–166. Springer-Verlag, Berlin, Heidelberg (2008)
31. Sangiovanni-Vincentelli, A.L., Sgroi, M., Lavagno, L.: Formal models for communication-based design. In: Proceedings of the 11th International Conference on Concurrency Theory, CONCUR '00, pp. 29–47. Springer-Verlag, London, UK (2000)
32. Stuijk, S., Geilen, M., Basten, T.: Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. IEEE Transactions on Computers **57**(10), 1331–1345 (2008). https://doi.org/10.1109/TC.2008.58
33. Theelen, B.D., Deprettere, E.F., Bhattacharyya, S.S.: Dynamic dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
34. Theelen, B.D., Geilen, M.C.W., Basten, T., Voeten, J.P.M., Gheorghita, S.V., Stuijk, S.: A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In: Proceedings of International Conference on Formal Methods and Models for Co-Design, pp. 185–194 (2006). https://doi.org/10.1109/MEMCOD.2006.1695924
35. Tripakis, S., Bui, D.N., Geilen, M., Rodiers, B., Lee, E.A.: Compositionality in Synchronous Data Flow: Modular Code Generation from Hierarchical SDF Graphs. ACM Trans. Embedded Comput. Syst. **12**(3), 83:1–83:26 (2013). http://dx.doi.org/10.1145/2442116.2442133

36. Tripakis, S., Bui, D.N., Rodiers, B., Lee, E.A.: Compositionality in Synchronous Data Flow: Modular Code Generation from Hierarchical SDF Graphs. In: J. Sztipanovits, R. Rajkumar (eds.) ACM/IEEE 1st International Conference on Cyber-Physical Systems, ICCPS'10, p. 199. ACM (2010). https://doi.org/10.1145/1795194.1795223
37. Zebelein, C., Haubelt, C., Falk, J., Schwarzer, T., Teich, J.: Representing mapping and scheduling decisions within dataflow graphs. In: Proceedings of the 2013 Forum on specification and Design Languages (FDL), pp. 1–8 (2013)
38. Zebelein, C., Haubelt, C., Falk, J., Teich, J.: Model-based representation of schedules for dataflow graphs. In: 16. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV 2013), pp. 105–116 (2013)

Kahn Process Networks and a Reactive Extension

Marc Geilen and Twan Basten

Abstract Kahn and MacQueen have introduced a generic class of determinate asynchronous data-flow applications, called Kahn Process Networks (KPNs) with an elegant mathematical model and semantics in terms of Scott-continuous functions on data streams together with an implementation model of independent asynchronous sequential programs communicating through FIFO buffers with blocking read and non-blocking write operations. The two are related by the Kahn Principle which states that a realization according to the implementation model behaves as predicted by the mathematical function. Additional steps are required to arrive at an actual implementation of a KPN to take care of scheduling of independent processes on a single processor and to manage communication buffers. Because of the expressiveness of the KPN model, buffer sizes and schedules cannot be determined at design time in general and require dynamic run-time system support. Constraints are discussed that need to be placed on such system support so as to maintain the Kahn Principle. We then discuss a possible extension of the KPN model to include the possibility for sporadic, reactive behavior which is not possible in the standard model. The extended model is called Reactive Process Networks. We introduce its semantics, look at analyzability and at more constrained data-flow models combined with reactive behavior.

M. Geilen (✉)
Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: m.c.w.geilen@tue.nl

T. Basten
Embedded Systems Innovation by TNO and Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: a.a.basten@tue.nl

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_24

1 Introduction

1.1 Motivation

Process networks are a popular model to express behavior of data-flow and streaming nature. This includes audio, video and 3D multimedia applications such as encoding and decoding of MPEG video streams. Using process networks, an application is modeled as a collection of concurrent processes communicating streams of data through FIFO channels. Process networks make task-level parallelism and communication explicit, have a simple semantics, are compositional and allow for efficient implementations without time-consuming synchronizations. There are several variants of process networks. One of the most general forms are Kahn process networks [34, 35], where the nodes are arbitrary sequential programs, that communicate only via channels of the process network with blocking read and non-blocking write operations. Although harder to analyze than more restricted models, such as Synchronous Data Flow networks [30, 39, 64], the added flexibility makes KPNs a popular programming model. Where synchronous data-flow models can be statically scheduled at compile time, KPNs must be scheduled dynamically in general, because their expressive power does not allow them to be statically analyzed. A run-time system is required to schedule the execution of processes and to manage memory usage for the channels. On a heterogeneous implementation platform, a KPN may be distributed over several components with individual scheduling and memory management domains. In this case, the execution of the process network has to be coordinated in a distributed fashion.

A process network is *determinate* if its input/output behavior can be expressed as a function. Kahn Process Networks represent the *largest class* of determinate data-flow process networks if we compare them based on the input/output functions they can express. The model abstracts from temporal behavior and focuses on functional input/output behavior of a network of parallel processes.

In this chapter we discuss the syntax and operational semantics of the model, a denotational semantics and the Kahn Principle, which relates them. The denotational semantics of KPNs of [34] is attractive from a mathematical point of view, because of its abstractness and compositionality. In a realization of a process network, FIFO sizes and contents play an important role and are influenced by a run-time environment that governs the execution of the network. It is for this reason that we present a simple operational semantics of process networks, similar to [21, 56]. We study and prove properties of the resulting transition system and we illustrate a simple proof of the Kahn Principle. Not because these are new results, but we believe that they provide insight in the fundamental properties and limitation of the Kahn model. We look at methods and requirements for implementation of Kahn Process Networks and we look at possible extensions of Kahn Process Networks, in particular with the ability to express reactive behavior.

1.2 Example

Figure 1 shows an example of a Kahn Process Network. It consists of four *processes* which compute deterministic functions. Processes have *inputs* and *outputs*. A KPN has *unbounded* FIFO channels connecting an output of a process to an input of a process. Unconnected channels represent inputs or outputs of the KPN. The network in Fig. 1 has no inputs and one output (d_k). The process Add repeatedly reads one integer number from each input (c_k and f_k, $k \geq 0$) and writes the sum of both numbers to its output ($a_k = c_k + f_k$). The processes Delay n first write the value n to their output and subsequently start copying their input to their output. The fourth process, Split, copies its input to three separate outputs, $c_k = d_k = e_k = b_k$. If the network executes, Delay0 and Delay1 write respectively a value 0 and 1 to their outputs. The value 1 is copied by the Split actor giving $d_0 = 1$. Now the Add process has the values 0 and 1 on its inputs, adds them up and writes the value 1 to its output. This value is copied again by Delay1 and Split and the value 1 ($= d_1$) is produced again on the output. It is easy to see that $d_k = a_{k-1}$ for $k \geq 1$ and $f_k = d_{k-1} = a_{k-2}$ for $k \geq 2$. Thus, $d_k = d_{k-1} + d_{k-2}$ with $d_0 = d_1 = 1$ and the recurrence equation corresponds to the Fibonacci sequence which the KPN produces on its output.

A more practical example of a KPN is shown in Fig. 2. It shows a part of a pipeline of a JPEG decoder. Input stream to the network is a stream of compressed data, the Variable Length Decoder function turns it into a stream of macro-blocks of 8 by 8 pixels. Those blocks are subsequently passed through three functions transforming those blocks, undoing the quantization, the zig-zag reordering of data and the discrete cosine transformation respectively. The corresponding mathematical functions are in practice specified by code as illustrated for the Inverse Zig Zag function (see Sect. 6). In this case the function is an infinite loop because the function operates block by block on a stream of blocks. It uses a `read` operation to read a macro block from its input `in`, then does some processing on it and at the end uses a `write` operation to write the result to its output.

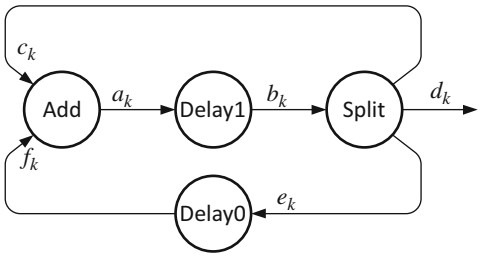


Fig. 1 Example: a Kahn Process Network computing the Fibonacci sequence

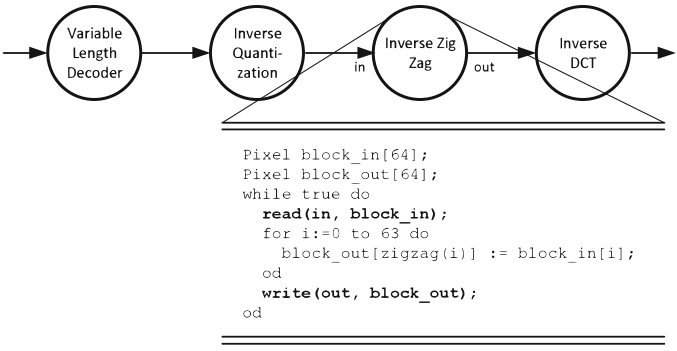


Fig. 2 KPN of JPEG decoder

1.3 Preliminaries

We introduce some mathematical notation and preliminaries. We assume the reader is familiar with the concept of a complete partial order (CPO) (see for instance [17]). We use $(X, \sqsubseteq)$ to denote a CPO on the set X with partial order relation $\sqsubseteq \subseteq X \times X$ to denote the corresponding partial order relation. We use $\sqcup D$ to denote the least upper bound of a directed subset D of X. For convenience, we assume a universal, countable, set *Chan* of channels and for every channel $c \in Chan$ a corresponding finite channel alphabet Σ_c. We use Σ to denote the union of all channel alphabets, and A^* (A^ω) to denote the set of all finite (and infinite) strings over alphabet A and $A^{*,\omega} = A^* \cup A^\omega$. $\sqsubseteq$ denotes the prefix relation on strings (a complete partial order, see for instance [17]). If σ and τ are strings, σ finite, then $\sigma \cdot \tau$ denotes the usual concatenation of the strings. If $\sigma \sqsubseteq \tau$, and σ is finite, then $\tau - \sigma$ denotes the string τ without its prefix σ.

A *history* of a channel denotes the sequence of data elements communicated along a channel [53], for instance the sequence of Fibonacci numbers. A history h of a set C of channels is a mapping from channels $c \in C$ to strings over Σ_c. The set of all histories of C is denoted as $H(C)$. If $h \in H(C)$ and $D \subseteq C$, then $h|D$ denotes the history obtained from h by restricting the domain to D. If h_1 is a history of C_1 and h_2 is a history of C_2, we write $h_1 \sqsubseteq h_2$ if $C_1 \subseteq C_2$ and for every $c \in C_1$, $h_1(c) \sqsubseteq h_2(c)$. The set of histories together with the relation $\sqsubseteq$ on histories form a complete partial order with as bottom element the empty history $\emptyset$. If $h_1 \sqsubseteq h_2$, then $h_2 - h_1$ denotes the history which maps a channel $c \in C_1$ to $h_2(c) - h_1(c)$. Histories h_1 and h_2 are called *consistent* if they share an upper bound, i.e., if there exists some history h_3 such that $h_1 \sqsubseteq h_3$ and $h_2 \sqsubseteq h_3$. If $h_1, h_2 \in H(C)$, then the concatenation $h_1 \cdot h_2$ is the history such that $h_1 \cdot h_2(c) = h_1(c) \cdot h_2(c)$ for all $c \in C$. A history is called finite if it maps every channel in its domain to a finite string and only a finite number of channels to a non-empty string. A finite history is a finite element of the CPO of histories. (Recall from [17] that an element $k \in X$ of a CPO $(X, \sqsubseteq)$ is a finite element if for every chain $D \subseteq X$, if $k \sqsubseteq \sqcup D$ then there is some $d \in D$ such that $k \sqsubseteq d$.)

In this chapter we frequently use functions $f : X \to X$ on some CPO $(X, \sqsubseteq)$ that are *monotone*. This means that applying the function to ordered elements of the CPO yields again ordered elements: if $x \subseteq y$, then $f(x) \sqsubseteq f(y)$. *(Scott-) continuity* is another common property of a function on a CPO. It states that the function preserves least upper bounds. A function is Scott-continuous iff $f(\sqcup D) = \sqcup\{f(d) \mid d \in D\}$.

2 Denotational Semantics

We discuss the formal definition of the semantics, i.e., the behavior, of a KPN. Traditionally, the focus is on the functional input/output behavior of the network, abstracting from the timing. More precisely, we are interested in a *function* relating the output produced by the network to the input provided to the network. For the JPEG decoder for instance, the sequence of decoded output blocks is a function of the compressed input data. We first concentrate on a *denotational* semantics, focusing on *what* input/output relation a KPN computes and later, in Sect. 3 on an *operational* semantics, which also captures *how* that output is computed.

The denotational semantics of KPNs [34] defines the behavior of a KPN as a *Scott-continuous input/output function* on the input or output stream histories. Assuming that the (Scott-continuous) input/output functions of the individual processes are known, then the input/output function of the KPN as a whole can be defined as the least fixed-point of a collection of equations specifying the relations between channel histories based on the processes. For the JPEG example, the overall function is straightforwardly obtained by function composition. A more challenging situation arises when there is feedback, such as in the Fibonacci example.

A KPN *process* with m inputs and n outputs is a Scott-continuous function $f : (\Sigma^{*,\omega})^m \to (\Sigma^{*,\omega})^n$. This implies that such a process is *monotone*: if $i \sqsubseteq j$ then $f(i) \sqsubseteq f(j)$; when additional input is provided, additional output may be produced, but existing output cannot be changed.

Because processes may exhibit memory (the current output may depend on input from the past), the function is defined in terms of the sequences representing the entire input *history* of the channel.

The functions of the individual processes of the Fibonacci example can be defined as follows. $\mathsf{Delay}_n : \mathbb{N}^{*,\omega} \to \mathbb{N}^{*,\omega}; i \in \mathbb{N}^{*,\omega}; n \in \{0, 1\}$:

$$\mathsf{Delay}_n(i) = n \cdot i \tag{1}$$

$\mathsf{Add} : (\mathbb{N}^{*,\omega})^2 \to \mathbb{N}^{*,\omega}$ is defined inductively by the following equations ($i, j \in \mathbb{N}^{*,\omega}$; $x, y \in \mathbb{N}$, ϵ is the empty sequence):

$$\begin{cases} \mathsf{Add}(i, \epsilon) = \epsilon \\ \mathsf{Add}(\epsilon, i) = \epsilon \\ \mathsf{Add}(x \cdot i, y \cdot j) = (x + y) \cdot \mathsf{Add}(i, j) \end{cases} \tag{2}$$

The FIFO channels in a KPN connect output streams of processes to input streams of other processes, creating relationships between the functions of the processes. In the example, the streams a, b, c, d, e and f are governed by the joint equations of the processes:

$$\begin{cases} b = \mathsf{Delay1}(a) \\ f = \mathsf{Delay0}(e) \\ a = \mathsf{Add}(c, f) \\ c = b, \quad d = b, \quad e = b \end{cases} \tag{3}$$

One can show that this set of equations has a single, unique, solution, namely the Fibonacci numbers, for its output sequence d.

In general, the semantics of a KPN are defined in a similar way as illustrated for the Fibonacci example above. In [34], Kahn presented the denotational semantics of process networks as the solution to a set of equations that capture the input/output relations of the individual processes and the way they are connected in the network. These equations are called the *network equations*.

A KPN consists of a set P of processes and a set S of streams, partitioned into *inputs* I, *outputs* O and *internal channels* C.

The processes p, with inputs I_p and outputs O_p, that constitute the network are described by (Scott-continuous) functions $f_p : H(I_p) \to H(O_p)$ that map input histories to output histories of the processes. The network as a whole can be described as a function, defined by Kahn's network equations as follows. For $h \in H(C \cup I \cup O)$ to be a valid history describing a behavior of the whole KPN (by describing the data communicated on each of its channels), it must satisfy the *network equations* derived from the processes $p \in P$:

$$h|O_p = f_p(h|I_p) \text{for all} p \in P$$

It simply expresses that if we take from h the channels that are the inputs and outputs of p, then these must be related corresponding to the function f_p.

To characterize the behavior of the KPN as a function f that maps an input history i to a history of *all channels* (including the output channels, which we are after) in the KPN, $f : H(I) \to H(I \cup C \cup O)$, we can derive the following equations (substituting $f(i)$ for h and adding an equation for the input channels).

$$\begin{cases} f(i)|I = i \\ f(i)|O_p = f_p(f(i)|I_p) \text{for all} p \in P \end{cases} \tag{4}$$

Note that every internal or output channel of the KPN is an output channel of one of the processes of the KPN. Hence, all channels are defined by these equations. Equation (4) is a recursive equation and in general, there may be many functions f that satisfy this equation, but only one that corresponds to an actual behavior respecting *causality* (one where, intuitively speaking, symbols are produced before they are consumed). The causal solution is the *smallest* function f that satisfies the

Table 1 Kleene iteration of the Fibonacci example

k	0	1	2	3	4	5	6	7	8	9	10	11	12
$a(k)$	ϵ	ϵ	ϵ	1	1	1	1.2	1.2	1.2	1.2.3	1.2.3	1.2.3	1.2.3.5
$b(k)$	ϵ	1	1	1	1.1	1.1	1.1	1.1.2	1.1.2	1.1.2	1.1.2.3	1.1.2.3	1.1.2.3
$c(k)$	ϵ	ϵ	1	1	1	1.1	1.1	1.1	1.1.2	1.1.2	1.1.2	1.1.2.3	1.1.2.3
$d(k)$	ϵ	ϵ	1	1	1	1.1	1.1	1.1	1.1.2	1.1.2	1.1.2	1.1.2.3	1.1.2.3
$e(k)$	ϵ	ϵ	1	1	1	1.1	1.1	1.1	1.1.2	1.1.2	1.1.2	1.1.2.3	1.1.2.3
$f(k)$	ϵ	0	0	0.1	0.1	0.1	0.1.1	0.1.1	0.1.1	0.1.1.2	0.1.1.2	0.1.1.2	0.1.1.2.3

network equations. Technically, this solution can be obtained as the least fixed-point of an appropriate functional. From Eq. (4) we derive the functional

$$\Phi : (H(I) \to H(I \cup C \cup O)) \to (H(I) \to H(I \cup C \cup O))$$

defined as (compare Eq. (4)):

$$\begin{cases} \Phi(f)(i)|I = i \\ \Phi(f)(i)|O_p = f_p(f(i)|I_p) \text{for all} p \in P \end{cases} \tag{5}$$

Clearly, a fixed-point of Φ satisfies Eq. (4). Moreover, Φ is a continuous function on a CPO and according to Kleene's fixed-point theorem [17] it has a least-fixed point, the limit of the ascending Kleene chain (Kleene iteration), $\bigsqcup\{\Phi^k(h_\epsilon) \mid k \geq 0\}$, where h_ϵ denotes the empty history associating an empty string with every channel.Therefore, Kleene iteration provides a way to constructively compute the network behavior, by updating the output of a process whenever its input has changed. If this procedure is repeated, then either it stops and the complete output has been computed or it continues forever and the infinite output equals the limit of the sequence of outputs computed during the procedure. Kleene iteration for the Fibonacci example (up to 12 steps, although the process continues ad infinitum) is shown in Table 1 and illustrates how it converges to the infinite Fibonacci sequence.

3 Operational Semantics

The KPN denotational semantics specifies the input/output behavior of a KPN in an elegant, abstract way as a mathematical function. This description however is far away from the actual operation or implementation of a KPN. Formal reasoning about implementations or run-time systems for KPNs can be easier based on an *operational* semantics. For instance, the denotational semantics provides no information to reason about buffer sizes required for the FIFO communication, or reasoning about potential deadlocks. In this section, we present a compositional operational semantics to hierarchical KPNs.

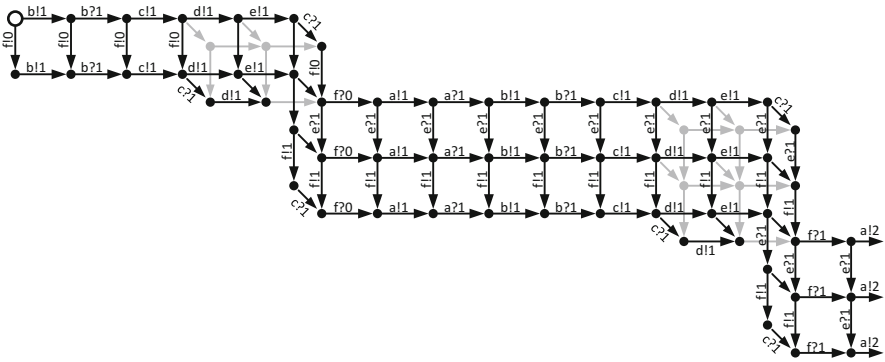


Fig. 3 Operational semantics of the Fibonacci example

3.1 Labeled Transition Systems

3.1.1 Semantics

We give an operational semantics to KPNs in the form of a labeled transition system (LTS). We use, more specifically, an LTS with an initial state, with designated input and output actions in the form of reads and writes of symbols on channels, as well as internal actions. We illustrate this with the LTS of the Fibonacci example (Fig. 3).

Definition 1 (LTS) An LTS is a tuple $(S, s_0, I, O, Act, \longrightarrow)$ consisting of a (countable) set S of states, an initial state $s_0 \in S$, a set $I \subseteq Chan$ of input channels, a set $O \subseteq Chan$ (disjoint from I) of output channels, a set Act of actions consisting of input (read) actions $\{c?a \mid c \in I, a \in \Sigma_c\} \subseteq Act$, output (write) actions $\{c!a \mid c \in O, a \in \Sigma_c\} \subseteq Act$ and (possibly) internal actions (all other actions), and a labeled transition relation $\longrightarrow \subseteq S \times Act \times S$ describing possible transitions between states.

Thus, $c!a$ is a write action to channel c with symbol a; $c?a$ models passing of a symbol from input channel c to the LTS. The initial state in Fig. 3 is indicated with the larger open circle. From this state, two write actions are possible, $b!1$ and $f!0$, leading to different, new states. We write $s_1 \xrightarrow{\alpha} s_2$ if $(s_1, \alpha, s_2) \in \longrightarrow$ and $s_1 \xrightarrow{\alpha}$ if there is some $s_2 \in S$ such that $s_1 \xrightarrow{\alpha} s_2$.

With a write operation, the symbol on the output channel is determined by the LTS. With a read operation, the symbol that appears on the input channel is determined by the environment of the LTS. Therefore, a read operation is modeled with a set of input actions that provides a transition for every possible symbol of the alphabet. (Note that the Fibonacci example KPN has no inputs to the network, only an output d.)

If Act is a set of actions and $C \subseteq Chan$ a set of channels, we write $Act|C$ to denote $\{c!a, c?a \in Act \mid c \in C\}$, i.e., actions of Act on channels in C. An execution

σ is a path through the transition system starting from the initial state, a sequence $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ of states $s_i \in S$ and actions $\alpha_i \in Act$, such that $s_i \xrightarrow{\alpha_i} s_{i+1}$ for all $i \geq 0$ (up to the length of the execution). If σ is such an execution, then we use $|\sigma| \in \mathbb{N} \cup \{\infty\}$ to denote the length of the execution. For $k \leq |\sigma|$, we use σ^k to denote the prefix of the execution up to and including state k. If $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} s_n$, we write $s_0 \xrightarrow{a} s_n$, where $a = \alpha_0 \cdot \alpha_1 \cdot \ldots \cdot \alpha_n$. From a given execution σ with actions $a = \alpha_0 \cdot \alpha_1 \cdot \ldots$, we extract the consumed input and the produced output on a set $D \subseteq Chan$ of channels as follows. For a channel $c \in D$, $a?c$ is a (finite or infinite) string over Σ_c that results from projecting a onto *read* actions on c. $a!c$ is a (finite or infinite) string over Σ_c that results from projecting a onto *write* actions on c. Input history $a?D = \{(c, a?c) \mid c \in D\}$ and output history $a!D = \{(c, a!c) \mid c \in D\}$.

Furthermore, we use the same notation for executions σ with actions a: $\sigma?c = a?c$, $\sigma!c = a!c$, $\sigma?D = a?D$ and $\sigma!D = a!D$. Thus $\sigma?I$ denotes the input consumed by the network in execution σ and $\sigma!O$ denotes the output produced by the network. The *I/O-history* $h(\sigma)$ of an execution σ is $\sigma?I \cup \sigma!O$. To reason about the input *offered* to the network (consumed or not consumed), we say that σ is an execution with input $i : I \to \Sigma^{*,\omega}$ if $\sigma?I \sqsubseteq i$ (the consumed data is consistent with i, but need not be all of i). If $i \sqsubseteq j$ then σ is also an execution with input j.

Executions in general may be only partially completed or they may be unrealistic or wrong because certain actions are systematically being ignored. To be able to exclude such executions when necessary, we need the notions of *maximality* and *fairness*. Let $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ be an execution of the LTS.

- (Maximality) Execution σ with input i is called maximal if it does not stop prematurely, if it is infinite or in its last state only read actions on input channels from which all input of i has been consumed are possible, i.e., if $|\sigma| = n$ and $s_n \xrightarrow{\alpha}$ then $\alpha = c?a$ for some $c \in I$ and $a \in \Sigma_c$ and $\sigma?c = i(c)$.
- (Fairness) Execution σ is fair with input i if it is finite, or it is infinite and if at some point an action is enabled, it is eventually executed or disabled (the latter does not occur for KPNs), i.e.,
  - if for some $n \in \mathbb{N}$ and internal or output action α, $s_n \xrightarrow{\alpha}$, then there is some $k \geq n$ such that $\alpha_k = \alpha$ or $s_k \nrightarrow \alpha$;
  - if for some $n \in \mathbb{N}$, $c \in I$ and $a \in \Sigma_c$, $s_n \xrightarrow{c?a}$, and $i(c) = (\sigma^n?c)a\tau$ for some $\tau \in \Sigma_c^{*,\omega}$, then there is some $k \geq n$ such that $\alpha_k = c?a$ or $s_k \nrightarrow c?a$;

Every finite execution with input i of an LTS can be extended to a fair and maximal execution with input i. We can describe the externally observable behavior of a labeled transition system by relating the output actions the LTS produces with the input actions provided to the network. In general this gives a relation between input histories and output histories. We restrict the attention to 'proper' executions in the sense that only maximal and fair executions are taken into account.

Definition 2 (Input/Output Relation) The input/output relation IO of an LTS is the relation $\{(i, \sigma!O) \mid \sigma$ is a maximal and fair execution with input $i\}$.

3.1.2 Determinacy

In general, the input/output relation is too abstract to adequately characterize the behavior of an LTS. If it is non-deterministic, the order in which input is consumed or output is produced can be relevant (see Sect. 7). Kahn Process Networks do not exhibit such non-determinism. Transitions are deterministic and if multiple actions are available at the same time, then they are independent (i.e., they can be executed in any order with an identical result). This leads to a special type of LTS, which we call *determinate*. Recall the LTS in Fig. 3, showing the beginning of the LTS of the Fibonacci example. Although there are very many different paths, there is only very limited actual choice in choosing a path from the initial top-left state. Any path we choose executes exactly the same actions in a slightly modified order caused by concurrency in the process network. We give a precise definition of this type of LTS and summarize their properties.

Definition 3 (Determinacy) LTS $(S, s_0, I, O, Act, \longrightarrow)$ is determinate if for any $s, s_1, s_2 \in S, \alpha_1, \alpha_2 \in Act$, if $s \xrightarrow{\alpha_1} s_1$ and $s \xrightarrow{\alpha_2} s_2$, the following hold:

1. (Determinism) if $\alpha_1 = \alpha_2$ is some input or output action, then $s_1 = s_2$, i.e., executing a particular action has a unique deterministic result;
2. (Confluence) if α_1 and α_2 are not two input actions on the same channel (i.e., instances of the same read operation), then there is some s_3 such that $s_1 \xrightarrow{\alpha_2} s_3$ and $s_2 \xrightarrow{\alpha_1} s_3$. Figure 3 shows many instances of this structure, where if from some state multiple transitions are possible, the actions that are not chosen remain enabled and taking them in different orders leads to the same state.
3. (Input Completeness) if $\alpha_1 = c?a$ for some $c \in I$, then for every $a' \in \Sigma_c$, $s \xrightarrow{c?a'}$, i.e., input symbols are completely defined by the environment, the LTS cannot be selective in the choice of symbols it accepts;
4. (Output Uniqueness) if $\alpha_1 = c!a$ and $\alpha_2 = c!a'$ for some $c \in O$, then $a = a'$, i.e., output symbols are completely determined by the transition system. Here, the environment cannot be selective in its choice of symbol to receive.

A *sequential* LTS is a determinate LTS with the additional property that

5. (Sequentiality) if $\alpha_1 = c\sharp a$ ($\sharp \in \{!, ?\}$) (some read or write operation), then $\alpha_2 = c\sharp a'$ for some $a' \in \Sigma_c$ and $c \in I \cup O$, i.e., the LTS accepts at most one input/output operation at any point in time and no other (for instance internal) actions.

Although a determinate LTS may have multiple actions enabled at the same time. The order in which they are taken has no influence on the consumed input or produced output of the network in a fair and maximal execution. Two different

executions are essentially the same, because the actions of one can be reordered without changing the input or output histories, to obtain the other execution (they are equivalent Mazurkiewicz traces).

Proposition 1 *The input/output relation of a determinate labeled transition system is a continuous function.*

A detailed proof can be found in [22]. If λ is a labeled transition system, we use f_λ to denote its I/O relation. In particular, if λ is determinate, this is an I/O *function*.

An important property in process networks is a deadlock condition. We can define a deadlock as the possibility to reach a state from which no further transitions are possible, a finite maximal execution. It is usually called a deadlock only if this is an undesirable situation. An important corollary from the analysis above is that if a determinate LTS has some execution which leads to a deadlock state, then *all of its executions* lead to the same deadlock state. In other words, the deadlock cannot be avoided by a smarter scheduling strategy, it is inherent to the determinate LTS specification and as we will see, therefore also holds for KPN.

3.2 Operational Semantics

We can now formalize an operational semantics of a KPN as a determinate LTS.

Definition 4 (Kahn Process Network) A Kahn Process Network is a tuple $(P, C, I, O, Act, \{\lambda_p \mid p \in P\})$ that consists of the following elements.

- A finite set P of *processes*.
- A finite set $C \subseteq Chan$ of *internal channels*, a finite set $I \subseteq Chan$ of input channels and a finite set $O \subseteq Chan$ of output channels, all distinct.
- Every constituent process $p \in P$ is itself defined by a determinate labeled transition system $\lambda_p = (S_p, s_{p,0}, I_p, O_p, Act_p, \overrightarrow{p})$, with $I_p \subseteq I \cup C$ and $O_p \subseteq O \cup C$. The sets $Act_p \backslash (Act_p | (I_p \cup O_p))$ of internal actions of the processes are disjoint.
- The set Act of *actions* consisting of the actions of the constituent processes: $Act = \bigcup_{p \in P} Act_p$.
- For every channel $c \in C \cup I$, there is exactly one process $p \in P$ that reads from it ($c \in I_p$) and for every channel $c \in C \cup O$, there is exactly one process $p \in P$ that writes to it ($c \in O_p$).

To define the operational semantics of a KPN, we need a notion of *global state* of the network; this state is composed of the individual states of the processes and the current contents of the internal channels. A configuration of the process network is a pair (π, γ) consisting of a process state π and a channel state γ, where

- a *process state* $\pi : P \to S = \bigcup_{p \in P} S_p$ is a function that maps every process $p \in P$ on a local state $\pi(p) \in S_p$ of its transition system;
- a *channel state* $\gamma : C \to \Sigma^*$ is a history function that maps every internal channel $c \in C$ on a *finite* string $\gamma(c)$ over Σ_c.

The set of all configurations is denoted by *Confs* and there is a designated initial configuration $c_0 = (\pi_0, \gamma_0)$, where π_0 maps every process $p \in P$ to its initial state $s_{p,0}$ and γ_0 maps every channel $c \in C$ to the empty string ϵ. We assign to a KPN $\kappa = (P, C, I, O, Act, \{\lambda_p \mid p \in P\})$, an operational semantics in the form of an LTS $(Confs, c_0, I, O, Act, \longrightarrow)$. The labeled transition relation $\longrightarrow$ is inductively defined by the following five rules (given in Plotkin style [50] inference rules; if the condition above the rule is satisfied, the conclusion below is also valid). For reading from and writing to internal channels by processes we have the following two rules respectively:

$$\frac{\pi(p) \xrightarrow[p]{c?a} s, \gamma(c) = a\sigma, c \in C}{(\pi, \gamma) \xrightarrow{c?a} (\pi\{s/p\}, \gamma\{\sigma/c\})} \qquad \frac{\pi(p) \xrightarrow[p]{c!a} s, \gamma(c) = \sigma, c \in C}{(\pi, \gamma) \xrightarrow{c!a} (\pi\{s/p\}, \gamma\{\sigma \cdot a/c\})}$$

Input channels and output channels are open to the environment:

$$\frac{\pi(p) \xrightarrow[p]{c?a} s, c \in I}{(\pi, \gamma) \xrightarrow{c?a} (\pi\{s/p\}, \gamma)} \qquad \frac{\pi(p) \xrightarrow[p]{c!a} s, c \in O}{(\pi, \gamma) \xrightarrow{c!a} (\pi\{s/p\}, \gamma)}$$

Individual processes may perform internal actions:

$$\frac{\pi(p) \xrightarrow[p]{\alpha} s, \alpha \notin Act_p | (I_p \cup O_p)}{(\pi, \gamma) \xrightarrow{\alpha} (\pi\{s/p\}, \gamma)}$$

This LTS is denoted as $\Lambda(\kappa)$.

The labeled transition system of a KPN is determinate. We check the four properties of a determinate LTS.

- Determinism follows from determinism of the process that accepts or produces the input or output action respectively.
- Confluence. If both actions originate from different processes, it can be checked that they cannot disable each other. If they originate from the same process, it follows from confluence of that process.
- Input completeness follows immediately from input completeness of the constituent processes.
- Output uniqueness similarly follows directly from output uniqueness of the process delivering the output.

The LTS of the KPN has the same property (determinacy) as the individual processes. Therefore, the presented semantics of KPN is *compositional*, a determinate process network is constructed from individual determinate processes, and this means it can itself be used as a process in a larger KPN. This way, we can hierarchically construct larger KPNs. At the lowest level we can start with primitive processes, for instance sequential processes which are typically implemented by sequential programs with read and write operations.

If κ is a KPN, then we use f_κ to denote the I/O function realized by that KPN. In the remainder, we assume that $(P, C, I, O, Act, \{\lambda_p \mid p \in P\})$ is a Kahn Process Network with LTS $(Confs, c_0, I, O, Act, \longrightarrow)$.

Using the operational semantics of KPN, we can now reason about resources such as scheduling on processors or buffer capacities for the FIFOs connecting the processes. We can also determine whether finite buffer capacities are sufficient. In the Fibonacci example of Fig. 3, there exist executions which require a buffer size of 2 on channel e, for instance if we take the path according to the upper envelope of the picture, two write actions occur in the channel e before the first read action. The path along the bottom envelope of the picture requires only a buffer size of 1 for channel e because the first read occurs before the second write. We have seen that all paths have the same functional behavior, but they may be different in terms of the required resources! An execution σ is *bounded* if there exists a mapping $B : C \to \mathbb{N}$ such that at any state (π, γ) of σ, $|\gamma(c)| \leq B(c)$ for all $c \in C$. Not every KPN allows a bounded execution. Some KPNs may have both bounded and unbounded executions. This is relevant for realizations of KPNs, as discussed in Sect. 6.

4 The Kahn Principle

The operational semantics given in the previous section is a model closer to a realization of a KPN than the denotational semantics of Sect. 2. The denotational semantics defines behavior as the least solution to a set of network equations [34]. The correspondence between both semantics, demonstrating that they are consistent, is referred to as the *Kahn Principle*. It was stated convincingly, but without proof, by Kahn in [34] and was later proved by Faustini [21] for an operational model of process networks, in [56] for an operational model of concurrent transition systems and in [44] for an operational characterization using I/O automata. Based on the operational semantics, a determinate LTS, a functional relation, is obtained between inputs and outputs. This function is shown to correspond to the least solution of Kahn's network equations.

The proof presented here is similar to the proof of the Kahn Principle for I/O automata of [44]. We reproduce it here in outline, because it illustrates the connections between denotational and operational semantics and the essential properties of the KPN model. We first show that if the operational behaviors of individual processes of the KPN respect their functional specifications, then so does the KPN as a whole. For an execution σ with input i, we use the notation $h(\sigma, i)$ to denote the history identical to $h(\sigma)$ except for the input channels, which are mapped according to i, since some of the input of i offered to the network may not (yet) have been consumed. Thus $h(\sigma, i)$ is equal to the mapping $i \cup \sigma!(C \cup O)$. We can derive from an execution of the overall KPN how individual processes have contributed to that execution. $\sigma|p$ denotes execution σ projected on process p. If

$\sigma = (\pi_0, \gamma_0) \xrightarrow{\alpha_0} (\pi_1, \gamma_1) \xrightarrow{\alpha_1} (\pi_2, \gamma_2) \xrightarrow{\alpha_2} \ldots$. Then $\sigma|p = \pi_{n_0}(p) \xrightarrow{\alpha_{n_0}} \pi_{n_1}(p) \xrightarrow{\alpha_{n_1}} \pi_{n_2}(p) \xrightarrow{\alpha_{n_2}} \ldots$ where n_0, n_1 etcetera are such that $n_0 < n_1 < \ldots$ and α_{n_0}, $\alpha_{n_1}, \ldots$ are precisely the actions from process p.

Lemma 1 *If σ is a fair and maximal execution of a KPN with input i and p is a process of the KPN, then $\sigma|p$ is a fair and maximal execution of p with input $\sigma!I_p$.*

Proof That $\sigma|p$ is an execution of p follows from the fact that if the KPN executes an action not from p, then the configuration does not change w.r.t. the state of p. If α does originate from p, then from $(\pi, \gamma) \xrightarrow{\alpha} (\pi', \gamma')$, it follows that $\pi(p) \xrightarrow[p]{\alpha} \pi'(p)$. Fairness follows from the fact that an enabled read or write operation of the process induces an enabled action of the KPN. Fairness of the execution σ of the KPN prescribes that the action is executed at some point in σ and hence also in $\sigma|p$. Similarly, maximality is obtained from maximality of the execution of the network.

Lemma 2 *For every fair and maximal execution σ with input i, the history $h(\sigma, i)$ satisfies the network equations.*

Proof Let σ be a fair and maximal execution with input i. It follows using Lemma 1 that $h(\sigma, i)|O_p = f_p(h(\sigma, i)|I_p)$. Thus, $h(\sigma, i)$ satisfies the network equations.

Lemma 3 *A history corresponding to a fair and maximal execution of the KPN with input i corresponds to the smallest solution to the network equations with input i.*

Proof The history is unique (i.e., independent of the execution) by Proposition 1. In Lemma 2 we proved that it is a solution to the network equations. We have to prove that every solution to the network equations is an upper bound of the history of the execution. Let σ be a fair and maximal execution of the KPN with input i and let h be any history satisfying the network equations such that $h|I = i$. It suffices to prove that h is an upper bound of the history of every finite prefix σ' of the execution, $h(\sigma', i) \sqsubseteq h$, proved by induction on the length of the execution. This is trivial for the empty execution (π_0, γ_0); we proceed with the induction step. Let $\sigma' = \sigma'' \longrightarrow \alpha(\pi_n, \gamma_n)$,

- If α is internal to one of the processes or an input action of one of the processes, then $h(\sigma', i) = h(\sigma'', i)$ and the result follows by the induction hypothesis.
- If α is an output action of some process p, then by the induction hypothesis, $h(\sigma'', i)|I_p \sqsubseteq h|I_p$. By monotonicity of f_p and the fact that h satisfies the network equations, it follows that $f_p(h(\sigma'', i)|I_p) \sqsubseteq f_p(h|I_p) = h|O_p$. By monotonicity of f_p and $\sigma''?I_p \sqsubseteq h(\sigma'', i)|I_p$ we also have $f_p(\sigma''?I_p) \sqsubseteq f_p(h(\sigma'', i)|I_p)$. Combining everything, we then have that $h(\sigma', i)|O_p \sqsubseteq f_p(\sigma'?I_p) = f_p(\sigma''?I_p) \sqsubseteq f_p(h(\sigma'', i)|I_p) \sqsubseteq h|O_p$. Hence, it follows that $h(\sigma', i) \sqsubseteq h$.

Theorem 1 (Kahn Principle) *The I/O relation of a KPN, derived from the operational semantics is a continuous function which corresponds to the denotational*

semantics of the KPN, i.e., to the least fixed point of the functional Φ defined in Sect. 2.

Proof It follows from Lemma 3 that for every input i, a fair and maximal execution of the KPN with input i yields the smallest channel history that satisfies the network equations. The least fixed point of Φ is the function that assigns to any input i precisely that smallest history.

5 Analyzability Results

Kahn Process Networks are a very expressive model of computation, despite the fact that it only allows specification of functional, determinate systems. In particular, compared to more restricted data-flow models such as Synchronous Data Flow or Cyclo-Static Data Flow [30], it allows that the rates at which a process communicates on its outputs are data dependent. Combined with unlimited storage capacity in its unbounded FIFO buffers, this makes KPN an expressive model. Expressiveness is usually in direct conflict with analyzability, the ability to (statically, off-line) analyze an application for its properties, such as deadlock-freedom (for buffers of a given, possibly unbounded, capacity), equivalence, minimum throughput (when adding timing information), static scheduling, and so on.

It is known for instance that the problem to decide for a given KPN (with unbounded buffer capacities) whether it is deadlock-free is undecidable. The proof of this fact [13, 49] relies on a reduction from the Halting Problem of Turing Machines. It is shown that a Boolean Dataflow (BDF) Graph (and therefore also a Kahn Process Network) can simulate a Universal Turing Machine; in other words KPN is *Turing Complete*. This allows the Halting Problem for Turing Machines to be reduced to deadlock-freedom of KPNs. Because the former is known to be an undecidable problem, so must the latter problem be undecidable. A sketch of the translation from Turing Machines to BDF is given in [13].

We can formalize the question for deadlock-free execution in the semantic framework of this chapter as follows.

Definition 5 (Deadlock) Given a KPN. Is there some input i, such that the KPN permits a finite, maximal execution with input i?

Note that in this case any maximal execution with input i is finite. Moreover, note that this does not always indicate a problem. Some KPNs may be designed to have only finite executions. The boundedness question can be formalized as follows.

Definition 6 (Boundedness) Given a KPN with an input/output function f and channels C and input channels I. Do there exist finite channel capacities $B : C \to \mathbb{N}$, such that for every input history i, there is an execution σ_i such that $h(\sigma_i, i)|O = f(i)$ and at any state (π, γ) of σ_i, $|\gamma(c)| \leq B(c)$ for all $c \in C$?

A slightly weaker form of boundedness can also be defined in which for every input i there is a bounded execution, but there need not necessarily be a single bound for all input. It would however not be strong enough to guarantee that a deadlock free implementation with fixed, bounded FIFOs exists to handle all possible inputs. A corresponding optimization problem, the *buffer sizing problem*, would be to try to find minimal such channel capacities. These capacities are not computable however, because of the undecidability of deadlock-freedom.

In the same way, *most non-trivial questions about KPNs are undecidable*. What is the minimum throughput or maximum latency of a given KPN extended with timing information? What are sufficient FIFO buffer capacities to execute with a guaranteed minimal throughput? Is a particular channel or process ever used or activated? Some of these questions have to be answered to arrive at an implementation of a KPN. We discuss this further in the next section.

6 Implementing Kahn Process Networks

KPNs are a mathematical model of computation with conceptually unbounded FIFO buffers. KPNs are a very expressive model and we have seen that many of its properties are not statically decidable. Therefore, in some cases subclasses of KPN are used as a starting point for an (automated) synthesis trajectory. However, in many cases the expressiveness of the model is exploited and KPNs are used directly as the basis for synthesis, or for simulation, in which case conceptually the same problems need to be solved.

Besides the infinite buffer capacities, another issue to be addressed is that the semantics involves potentially infinite behaviors. The Fibonacci example if Fig. 1 represents a network that produces an infinite stream of numbers, the Fibonacci sequence. A real, physical implementation however will never be able to produce an infinite sequence in a finite amount of time. A judgement whether an implementation is correct should be based on its behavior in finite time to be of practical relevance. Section 9 discusses existing implementations of KPN.

6.1 Implementing Atomic Processes

The semantics of KPN assumes atomic processes which implement elementary continuous functions. Kahn and MacQueen suggest [35] that the atomic processes can be implemented with sequential program code which does not use any global variables shared with other processes and with explicit read and write operations added for reading symbols from input and writing symbols to output channels. An example is the code of the Inverse Zig Zag process in Fig. 2.

If several of such processes are required to run on a single processor, then typically a multi-threaded (light-weight) OS is used which spawns a single thread

for every process. Determinacy of KPN guarantees that synchronization between these threads is only needed for communication on the FIFOs. When a read operation is executed on an empty FIFO, the reading process thread should stall until new symbols are written to the channel. (It is not allowed to do anything else than wait, because that would break determinacy, since scheduling order may have an impact on the outcome.) Similarly, if FIFO buffers of limited capacity are used in the implementation, a writing process thread may need to stall until sufficient space is available in the channel to complete the write operation. Threads can be, but need not be, scheduled in a preemptive manner.

6.2 Correctness Criteria

An implementation of a KPN should respect the formal (denotational and operational) semantics. It is not entirely trivial how to define correctness because the semantics talks about infinite executions and infinite streams as a convenient abstraction of streaming computation. However, in the real-world we will never be able to observe any actual infinitely long executions. We break down correctness of a KPN implementation into three aspects: *soundness*, *completeness* and *boundedness*. Soundness is the most basic requirement and it states that the KPN implementation should never produce any output that contains symbols *different* from the output predicted by the semantics, nor should it produce *more output* than predicted. For instance, if the semantics says that the Fibonacci KPN produces the outputs: 1, 1, 2, 3, etcetera, then the implementation should not produce: 1, 2, ..., and the JPEG decode should not produce any wrongly decoded blocks.

Definition 7 (Soundness) An implementation strategy for KPN is *sound* if for every KPN with input/output function f and every behavior the strategy may produce, if the strategy consumes input i and produces output o, then $o \sqsubseteq f(i)$.

Secondly, output should be complete. Intuitively this means that the implementation should produce *all* output predicted by the semantics. But of course we cannot expect an implementation to produce an infinite amount of output in a finite amount of time. However, every individual part (symbol) of the infinite output stream occurs after a finite amount of output produced before it. Hence, we do require from a good implementation that every bit of the predicted output is *eventually* (meaning after a *finite* amount of time) produced by the implementation. In other words, if we sample at finite time intervals the input consumed and output produced by the KPN implementation as a sequence of finite executions σ_k, then the limit of that sequence (its least upper bound) should be the *entire infinite* execution. If the JPEG decoder would be used ad infinitum, with an infinite stream of images to be decoded, it should not come to a complete halt after a finite amount of time.

Definition 8 (Completeness) An implementation strategy for KPN is *complete* if for every KPN and for every input i offered to this KPN and for every behavior the

strategy may produce, if it is sampled at regular time intervals t_k having produced the finite executions σ_k, then $\sigma = \bigsqcup\{\sigma_k \mid k \geq 0\}$ is a maximal and fair execution with input i that is part of the operational semantics.

In particular, this implies that (a) any new amount of progress is made in a finite amount of time, and (b) no channels are excluded from making progress in finite time, there must be no starvation of parts of the network. We illustrate the importance of this constraint when we discuss run-time scheduling and buffer management below.

Thirdly, it is important that this is achieved within bounded memory *whenever possible* (we know that it is not always possible). This amounts to keeping the FIFOs bounded, also in (conceptually) infinite computations.

Definition 9 (Boundedness) An implementation strategy for KPN is *bounded* if for every KPN and for every input offered to this KPN, if there exists a bounded execution according to the operational semantics, then this strategy will produce a bounded execution.

Not every KPN allows for bounded executions. In such a case, the strategy is allowed to produce an unbounded execution, but rather one should perhaps decide not to implement such a KPN, although one cannot, in general, automatically decide whether this will be the case!

6.3 Run-Time Scheduling and Buffer Management

Conceptually, the FIFO communication channels of a KPN have an unbounded capacity. A realization of a process network has to run within a finite amount of memory. An unbounded capacity can be mimicked using dynamic allocation of memory for a write operation, as suggested in [35], but rather than that, it is for reasons of efficiency better to allocate fixed amounts of memory to channels and change this amount only sporadically, if necessary. An added advantage of the fixed capacity of channels is that one can use an execution scheme introduced by Parks [49], where a write action on a full FIFO channel of limited capacity blocks until there is room freed up in the FIFO. Note that this behavior can be modeled within the KPN model using additional channels in opposite directions, similar to the well-known trick with Synchronous Data Flow graphs [49]. This importantly demonstrates that this does not impact determinacy of the model. This gives an efficient mixed form of data-driven and demand-driven scheduling [3, 49], illustrated in Fig. 4, in which a process can produce output (in a data-driven way) until the channel it is writing to is full and/or the channel it is consuming from is empty. Then it blocks and other processes take over, effectively regulating the relative speeds of different processes. As discussed previously, it is undecidable in general, how much buffer capacity is needed in every channel [13]. If buffers are chosen too large, memory is wasted. If buffers are chosen too small, so-called

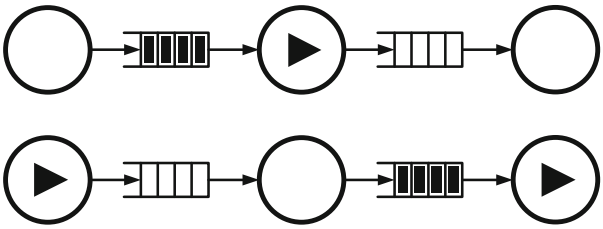


Fig. 4 Parks' scheduling method

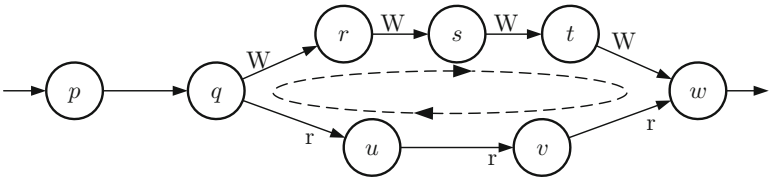


Fig. 5 An artificial deadlock

artificial deadlocks may occur that are not present in the original KPN, when processes are being permanently blocked because of one or more full channels, an event which cannot occur in the original KPN. Therefore, a scheduler is needed that determines the order of execution of processes and manages buffer sizes at run-time.

Figure 5 shows a process network in an artificial deadlock situation. Process w cannot continue because its input channel to process v is empty; it is blocked on a read action on the channel to v, denoted by the 'r' in the figure. The required input should be provided by v, but this is in turn waiting for input from u. u is waiting for q. Process q is blocked, because it is trying to output a token on the full channel to r; the block on the write action is denoted by a 'w'. Similarly processes r, s and t are blocked by a full channel. Only w could start emptying these channels, but w is blocked. The processes are in artificial deadlock (q, r, s and t would not be blocked in the original KPN) and can only continue if the capacity of one of the full channels is increased. Note that a blocked process is dependent on a unique channel on which it is blocked either for reading or for writing and hence it depends on a unique other process that is also connected to that channel. This gives rise to a chain of dependencies and if this chain is cyclic, it is a deadlock. A *real* (non-artificial) deadlock, in contrast, is entirely due to the KPN specification and cannot be avoided by buffer capacity selection. Because the KPN with bounded FIFOs is also determinate we can conclude that the buffer management is independent from scheduling; an artificial deadlock, like a real deadlock, cannot be avoided by a different scheduling (see [24] for more details and a proof.)

Thus, an important aspect of a run-time scheduler for KPNs is dealing with artificial deadlocks. We can discern different kinds of deadlocks. A process network is in *global deadlock* if none of the processes can make progress. In contrast, a *local*

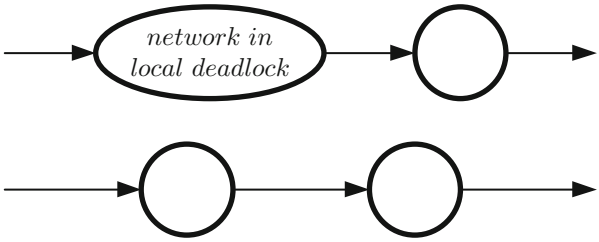


Fig. 6 A local deadlock

deadlock arises if some of the processes in the network cannot progress and their further progress cannot be initiated by the rest of the network.

Parks' strategy [49] for dealing with scheduling, artificial deadlocks and buffer management is the following procedure. First, select some arbitrary, fixed initial buffer capacities. Then, execute the KPN, with blocking read and blocking write operations. If and when a global deadlock occurs and it is an artificial deadlock, then increase the size of the smallest buffer and continue. (One can try to be more efficient in the selection of the buffer that needs to be enlarged [3, 24], but this does not fundamentally change the strategy.) Such a run-time scheduler thus needs to detect a global deadlock. In a single processor multi-threaded implementation, this is typically achieved by having a lowest priority thread of execution that becomes active only when all other threads are blocked, indicating a global deadlock. It then tests whether the deadlock is artificial and if so, increases the size of a selected buffer, ultimately enabling one of the blocked processes again. In terms of the formulated correctness criteria, one can show this method to be sound and bounded, but not complete.

Proposition 2 *The scheduling strategy of [49] for KPNs is sound and bounded, but not complete.*

Proof Soundness is straightforward. The appropriate code is executed and nothing else. The strategy of selecting the smallest buffer to increase guarantees that after a finite number of resolved artificial deadlocks, the buffer capacities must have grown beyond the bounds needed when a bounded execution exists. Incompleteness of the scheduling strategy follows from the counter example of Fig. 6, discussed below.

The strategy chosen in [3] is in this respect similar to the one of [49] and also leads to a bounded execution if one exists.

To guarantee the correct output of a network, a run-time scheduler must detect and respond to artificial deadlocks. Parks proposes to respond to global artificial deadlocks. In implementations of this strategy [1, 29, 36, 57, 63], global deadlock detection is realized by detecting that all processes are blocked, some of which by writing on a full FIFO. Although this guarantees that execution of the process network never terminates because of an artificial deadlock, it does not guarantee the production of *all* output required by the KPN semantics; output may not be

complete. For example, in the network of Fig. 6, if the upper part reaches a local artificial deadlock, then the lower, independent part is not affected. Processes may not all come to a halt and the local deadlock is not detected and not resolved. The upper part may not produce the required output. Such situations exist in realistic networks. A particular example is the case when multiple KPNs are controlled by a single run-time scheduler, one entire process network may get stuck in a deadlock. Hence, to achieve completeness, a deadlock detection scheme has to detect *local* deadlocks as well.

It is shown in Sect. 4 that the Kahn Principle hinges on *fair* scheduling of processes [11, 34, 56]. Fairness means that all processes that can make progress should make progress at some point. This is often a tacit but valid assumption if the underlying realization is truly concurrent, or fairly scheduled. However, in the context of bounded FIFO channels where processes appear to be inactive while they are blocked for writing, fairness of a schedule is no longer evident. This issue is neglected if one responds to global deadlocks only, leading to a discrepancy with the behavior of the conceptual KPN.

It is proved in [24] and illustrated below, that a perfect scheduler for KPN, satisfying all three correctness criteria for any KPN cannot exist.

Theorem 2 ([24]) *A scheduling strategy for KPNs satisfying soundness, completeness and boundedness does not exist.*

The incompleteness of the scheduling method of [49] can have rather severe consequences, leading to starvation of parts of the network. The way to resolve this is to not wait until a global deadlock occurs before taking actions, but act (in finite time) when a local artificial deadlock occurs and resolve it. This is the adaptation that the strategy proposed by Geilen and Basten [24] makes to the original strategy of [49].

A problem for an implementation strategy for KPN is posed by the production of data that is never used. This is illustrated with Fig. 7. Process p writes n data elements (tokens) on channel c connecting p to process q; after that, it writes tokens to output channel a forever. q never reads tokens from c and outputs tokens to channel b forever. If the capacity of c is insufficient for n tokens, then output a will never be written to unless the capacity of c is increased. q doesn't halt and execution according to Parks' algorithm does not produce output on channel a, violating completeness, because in the KPN, infinite output is produced on both channels. The above suggests that a good scheduler should eventually increase the capacity of channel c so that it can contain all n tokens. However, such a scheduler fails to correctly schedule another KPN. Consider a process network with the same structure as the one of Fig. 7, but this time, p continuously writes tokens on output a, mixed with infinitely many tokens to channel c. q writes infinitely many tokens to b and reads infinitely many tokens from c, but at a different rate than p writes tokens to c. Note that a bounded execution exists; a capacity of one token suffices for channel c. If process p writes to c faster than q reads, channel c may fill up and the scheduler, not knowing if tokens on channel c will ever be read, decides to increase channel capacity. A process q exists that always postpones the read

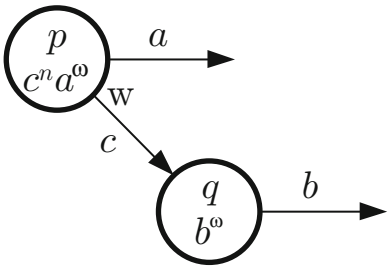


Fig. 7 Non effective network

actions until after the scheduler decides to increase the capacity; the execution will be unbounded, although a bounded execution exists.

The way out of this dilemma taken by Geilen and Basten [24] is to assume that in a reasonable KPN, every token that is written to a channel, is eventually also read. Such KPNs are called *effective*.

The scheduling algorithm is based on the use of blocking write operations to full channels as in [49]. In order to define local deadlocks, it builds upon the notion of causal chains as introduced in [3]. Any blocked process depends for its further progress on a unique other process that must fill or empty the appropriate channel. These dependencies give rise to chains of dependencies. If such a chain of dependencies is cyclic, it indicates a local deadlock; no further progress can be made without external help. In Fig. 5, such a causal chain is indicated by the dashed ellipse indicating the cyclic mutual dependencies of the processes q, r, s, t, u, v and w, which form a local deadlock.

The scheduling strategy of [24] is identical to the scheduling strategy of [49] except that the search for an artificial deadlock does not occur only when the network is in global deadlock, but the scheduler needs to monitor the process network for the occurrence of local artificial deadlocks and resolve those in finite time. The behavior of this scheduling strategy has the following characteristic property, proved in [24].

Theorem 3 *The scheduling strategy of [24] for KPNs is sound, complete and bounded for effective KPNs.*

7 Extensions of KPN

KPN is an expressive model that allows the description of a wide class of data-flow applications. Its expressiveness leads to undecidability of various elementary questions about KPNs (Sect. 5). Yet still, there is sometimes the desire to extend the KPN model with additional features. The most prominent ones are *time* and *events* or *reactive behavior*. KPN describes the functional behavior of a process network, but makes no statements about the timing with which these outputs are produced, or the latency or throughput that can or should be attained. Another important element

on the wish list is the ability to deal with sporadic messages or events. Often both are added at the same time to save determinacy of the model.

7.1 Events

The desire to deal with sporadic events is illustrated by the addition of the `select` statement in the KPN programming library Yapi [36]. This statement allows a process to see which of a number of input channels has data available and to read from that channel first. Similarly, [45] describes an extension of the data-flow model with the ability to probe a channel for the presence or absence of data. While both extensions clearly enhance the expressiveness of the model, they also destroy the property of determinacy, of independence of any concrete schedule or scheduling strategy, the denotational semantics of KPN and the Kahn Principle.

From a theoretical perspective, the essential ingredient to add to KPNs to deal with sporadic or reactive behavior is a *merge* process. A merge process has two inputs and one output and it copies data arriving on any of its inputs to the single output, in an order which is based on the arrival order of data, for instance in the order in which they arrive on the inputs. In this case however, the input/output relation of the process is no longer a function; the same inputs can lead to different outputs, depending on the order in which they arrive. KPN denotational semantics captures behavior as continuous input/output functions and thus no longer works. It has been attempted to capture KPN with merge processes by input/output *relations*. It turns out however that this is not possible. This is commonly known as the Brock-Ackerman anomaly [10]. For non-determinate processes, their input/output relation does not sufficiently characterize the system's behavior to use it as a basis for a semantic framework. The counter example is reproduced in Fig. 8. It has a 'merge' process as the only indeterminate (non-KPN) process in the network, a process 'first' which passes on only the first symbol, a process 'split' which splits the stream in two exact copies and a process 'inc' which passes on all symbols after incrementing them by one. The network is provided with the input consisting of the single symbol 5. The merge process passes it on to its output and the processes

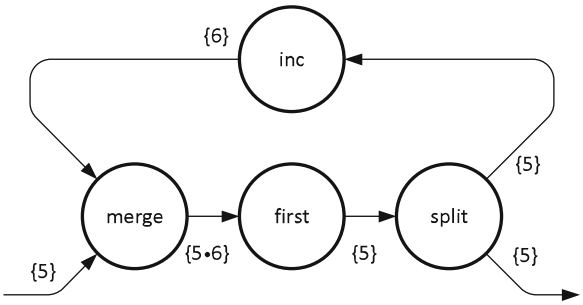


Fig. 8 Brock-Ackerman counter example (from [10])

'first' and 'split' pass it on as well. 'inc' turns it into 6 and passes it on to 'merge'. *Operationally speaking*, merge now passes the 6 to its output and because the token 6 is *causally dependent* on the token 5 passing to the output of the merge *first*, the only possible output of the merge process is $5 \cdot 6$. On the other hand, the denotational semantics of the merge process specifies that for the inputs $\{(5, 6)\}$, the possible outputs are $\{6 \cdot 5, 5 \cdot 6\}$, but the $6 \cdot 5$ is not possible in any causal reality. The causality *information* which is necessary for providing a correct semantics to indeterminate processes is lost in the input/output relation.

Brock and Ackerman showed [10] that I/O relations do not provide a good semantics for the so-called *fair merge* [48] that merges streams in such a ways that all input tokens on either input are *eventually* produced at the output. Subtly different and weaker versions of merge processes (but with strictly different expressiveness) can be defined, *fair merge*, *infinity fair merge*, *angelic merge*, but it was later shown (see [53] for an overview), that they all suffer the same limitation, that input/output relations are insufficient. Only merge processes which interleave symbols from the inputs in a fixed, a priori determined order, for instance alternatingly, are determinate.

As an alternative to a denotational semantics, a trace-based semantics has been found to be able to serve as a fully abstract semantics [33], i.e., a semantic model which contains *enough* information, but *no redundant* information to represent behavior. The use of a totally ordered trace indicates the loss of independence of scheduling of concurrent processes associated with the introduction of a merge construct, which is one of the strongest points of KPN. A theoretical analysis of relational models for indeterminate dataflow models is [62].

7.2 Time

A timed process network model is a model which not only describes the data transformations of the network, but also the timing of such a process [12, 65]. Such a model is typically made by labeling symbols on streams with time-stamps or tags from a particular time domain of choice (for instance non-negative integers or reals) or, more general, according to the tagged-signal model of [40], allowing for instance also partially ordered or super-dense time domains common in hardware description languages [43]. Alternatively, a stream can then be described as a mapping from the time-domain to the channel alphabet. If this function is total, it may also specify the absence of data at certain points in time, which can then be deterministically exploited by processes. Time-stamps may be interpreted as the exact timing of the production of tokens or as a specification of a deadline for the production.

From a semantics perspective, time-stamps can be exploited to 'rescue' the merge process from indeterminacy and in general to make reactive behavior deterministic. Using time information, decisions can be taken in a deterministic way, based on the time-stamps of data, for instance to describe a merge process which merges symbols in the order in which they arrive (with some deterministic provision, for instance

fixed priority, when tokens arrive at the same time) [12, 43, 65]. Alternative network equations can be formulated and different fixed-point theorems (such as Banach's) can be used to show them to have a unique solution. The (big) price one has to pay for salvaging determinacy of the model is that it requires a global notion of synchronized (logical) time, which puts constraints on implementation and requires additional synchronization.

By adopting such a notion of synchronized global logical time, we end up close to another end of the spectrum of data-flow languages, the domain of synchronous languages such as Esterel [6], Signal [4] or Lustre [31], where the execution of a network needs to be globally synchronized. This can be a strong disadvantage to a distributed implementation. Later work [5, 14] in the synchronous language domain searches for conditions to relax the global synchrony constraints towards so-called GALS (globally asynchronous, locally synchronous) implementations, where particularly the most costly global synchronization may be eliminated.

8 Reactive Process Networks

The Reactive Process Networks (RPN) model of computation intends to provide a semantic framework and implementation model for data-flow networks with sporadic events, in the same way KPN is a reference for determinate data-flow models of computation. RPN integrates control and event processing with stream processing in a unifying model of computation with a compositional operational semantics. The model tries to find a balance in a trade-off between expressiveness, determinism and predictability, and implementability.

8.1 Introduction

We illustrate the Reactive Process Networks model by looking at the domain of multimedia applications, working with information streams such as audio, video or graphics. With modern applications, these streams and their encodings can be very dynamic. Smart compression, encoding and scalability features make these streams less regular than they used to be.

Streams are typically parts of larger applications. Other parts of these applications tend to be control-oriented and event-driven and interact with the streaming components. Modern (embedded) multimedia applications can often be seen as instances of the structure depicted in Fig. 9. At the heart of the application, computationally intensive data operations have to be performed in streams of for instance pixels, audio samples or video frames. Input and output of these processes are highly regular patterns of data. These data processing activities can often be statically analyzed and scheduled on efficient processing units. At a higher level, modern multimedia streams show a lot of dynamism. Object-based video (de)coders

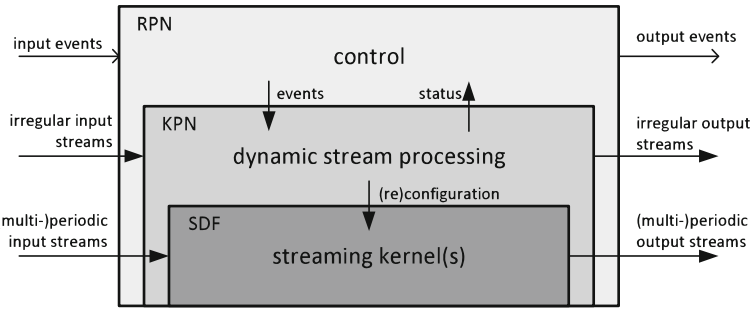


Fig. 9 Embedding of types of streams

for instance work with dynamic numbers of objects that enter or leave a scene. Decoding of the individual objects themselves uses the static data processing functions, but they may need to be added, removed or adapted dynamically, for instance encoding modes or frame types in MPEG audio or video streams. These dynamic streams still compute functions and processing is determinate, i.e., the functional result is independent of the order in which operations are executed. In turn, the processing of these dynamic data streams is governed by control oriented components. This may for instance be used to convey user interactions to the streaming application or to respond to changing network conditions.

The three levels of an application require typical modeling and implementation techniques. A good candidate for instance to describe static computation kernels could be Synchronous Data Flow (SDF) [39], discussed in [30]. Dynamic stream processing can perhaps be best most generally described using KPNs. They are capable of showing data dependent behavior and dynamic changes in their processing, but they are still determinate and can be executed fully asynchronously. If static analyzability is required more restricted dynamic models such as SADF can be employed [9]. To specify the control dominated parts of an application, there are many techniques, such as state machines and event-driven software models [20].

RPN defines a model and its formal operational semantics that allows for an integrated description and analysis of an application consisting of these three levels of computation. It is a unified model for streaming and control, that is also hierarchical and compositional. The goal is to be also able to incorporate more analyzable, less expressive models in the same way SDF fits within the KPN model, but still combining data-flow and control oriented behavior. For instance, a combination of SDF with finite state machines such as HDF or SADF [9, 26–28, 58, 59], which yields analyzable models, would fit within the framework. This is illustrated in Fig. 10. Vertically, it shows the trade-off between expressiveness and analyzability, with a border of decidability. Horizontally, it shows streaming vs. control oriented models.

Figure 11 shows the compositional integration of state machines with process networks. A process network is a component with stream input(s) (i in Fig. 11a),

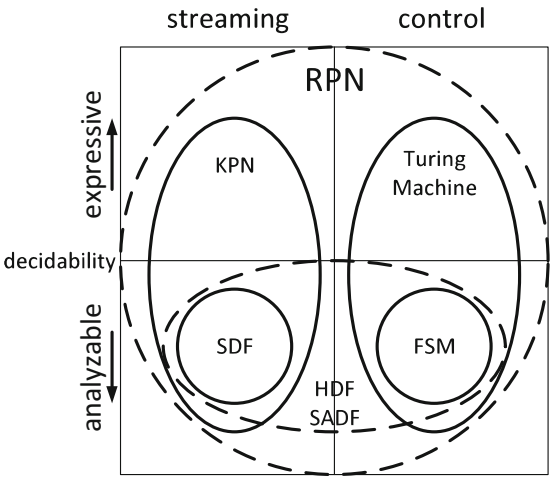


Fig. 10 Classification of models of computation

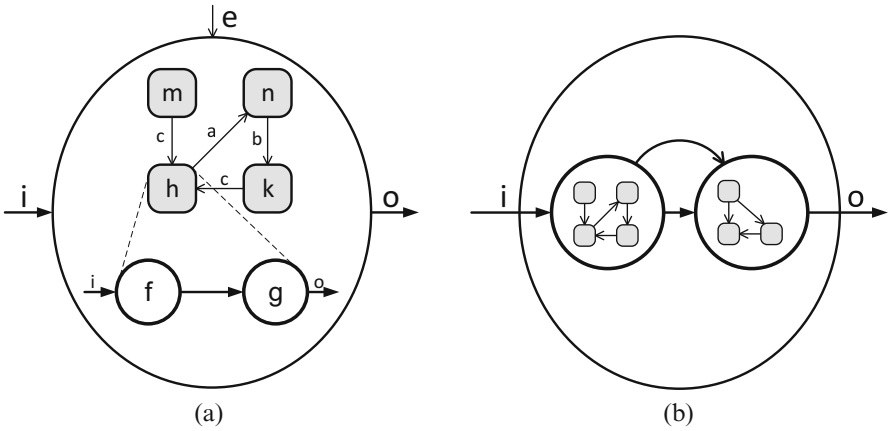


Fig. 11 Mixing state machines with process networks

stream output(s) (o) and event input(s) (e). At any point in time, the network operates in a mode that implements a particular streaming function, for instance mode h in Fig. 11a, implemented as the network drawn below it. At some later time, because of the occurrence of some event a, the function of the network needs to change to a mode n having a different streaming function. One could think for instance of a video system where a user changes settings or turns on or off special image processing features. One could view this as a (not necessarily finite) state machine where in every state, the network implements a particular function and external events force the state machine to move from one state to another. In every state, a particular process network performs operations on data streams. Similarly, such state machines can be embedded in components of a reactive process network as shown in Fig. 11b. Events that are communicated to such components

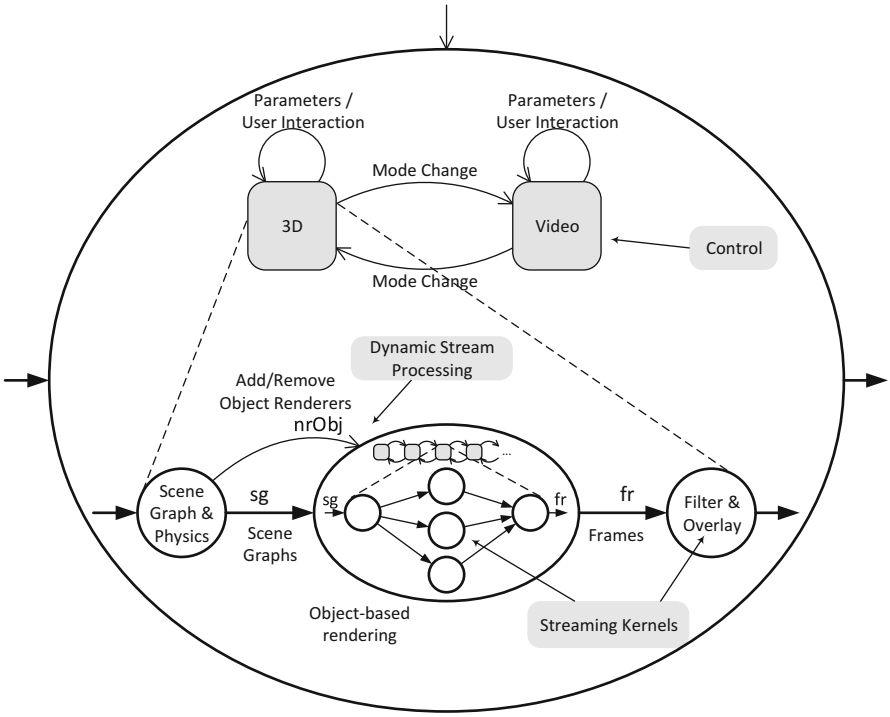


Fig. 12 An interactive 3D game

can be generated from an output port of another component. Process networks can be hierarchical entities, we would like such a construct to be compositional and applicable at different levels of a network hierarchy. Note that this is the conceptual behavior of an RPN, but that does not mean that this is exactly how it is implemented. In many cases the events cause relatively small changes to the current streaming function, changing of parameters or activation or deactivation of individual functions.

8.2 A Reactive Process Network Example

An illustrative example of the type of application we are considering is shown in Fig. 12. It depicts an imaginary game, which includes modes of 3-dimensional game play with streaming video based modes. The rendering pipeline, used in the 3D mode, is a dynamic streaming application. Characters or objects may enter or leave the scene because of player interaction, rendering parameters may be adapted to achieve the required frame rates based on a performance monitoring feedback loop. Overlayed graphics (for instance text or scores) may change. This happens

under control of the event-driven game control logic. At the processing core of the application, the streaming kernels, a lot of intensive pixel based operations are required to perform the various texture mapping or video filtering operations. Special hardware or processors may be available to execute these operations, which can be scheduled off-line, very efficiently.

The model is organized around the two main modes (3D graphics, video). In these modes, the game dynamics and mode changes are influenced or initiated by user interaction, game play and performance feedback. This is the control oriented part of the game depicted as the automaton at the top of Fig. 12. The self-loops on these states denote changes where the streaming network essentially stays the same, but its parameters may be changed.

In the 3D graphics mode (enlarged at the bottom of the figure), a scene graph, describing all entities and their positions in 3D space, is rendered to a 2-dimensional view on the scene on some output device. The first process communicates the scene graphs with objects to the rendering component. The rendering component transforms the scene graphs into 2-dimensional frames. The last process adds 2-dimensional video processing such as filtering, overlays, and so forth. The output is shown by the display device.

Notice that the game as a whole has different modes of streaming (3D graphics, video), but similarly, components in the stream processing part have different modes or states of streaming execution. The object renderer for instance can be reconfigured to different modes depending on the number of objects that need to be rendered (using the event channel *nrObj* controlled by the scene graph process). This illustrates the need for a hierarchical, compositional approach to combining state/event based models with streaming and data-flow based models, as realized by the RPN model.

8.3 Design Considerations of RPN

We discuss the main concepts that have had an impact on the design of the model of Reactive Process Networks.

8.3.1 Streams, Events and Time

Streaming applications represent functions or data transformations. They absorb input and they produce the corresponding output. There is often no inherent notion of time, except for the ordering of tokens in the individual data streams. (We are aiming for an untimed model similar to KPN.) Tokens in different streams have no relation in time, except for causal relationships implicitly defined by the way processes operate on the tokens. These process networks are ideally determinate, i.e., the order and time in which processes execute is irrelevant for the functional result. The output of a process network is completely determined as soon as the

input is known. The actual computation of this output introduces a certain latency in the reaction. This latency is not part of the functional specification, but merely a consequence of the computation process. It may be subject to constraints, such as a maximum latency. Time is sometimes implicitly present in the intention of streams. A stream may carry for instance, a sequence of samples of an audio signal that are 1/44100th of a second apart, or video frames of which there are 25 or 30 in every second. Such streams are called *periodic*. For final realizations, time-related notions such as throughput, latency and jitter are of course important.

Events, have a somewhat different relationship to time. An event is unpredictable and the moment when it arrives, in relation to the streams, is significant, but unknown in advance. In many event-based models, the *synchrony hypothesis* applies, which states that the response to an event can be completed before the following event arrives or is taken into account. This simplifies specifying how a system responds to events. A classical model for event-based systems are state machines, where events make it change from one state into another. Prominent characteristics are non-determinism and a total ordering of events. If events come from outside and are not predictable a priori, then the system evolves in a non-deterministic way under the influence of the events, even if the response to a particular event is deterministic.

Discrete changes due to events are alternated with stable periods of streaming. Conceptually, a discrete change occurs at a well defined point within the stream. Because of pipelining implementation of the stream processing however, there is not necessarily a point in time where the change can be applied instantaneously to the whole network. A video decoder for instance may be processing video frame by video frame in a pipelined fashion. A discrete change occurs between two frames such that the new frames are decoded according to new parameter settings. However, when the first new frame enters into the pipeline, there are still old frames in the pipeline ahead of it. The StreamIt language [60], which employs a fairly synchronized model of streaming, allows a mechanism of delivering asynchronous messages (events) to processes in accordance with the 'information wavefront', i.e., in a pipelined fashion. In general, an important aspect of dealing with streaming computation and events is to coordinate their execution to implement a smooth transition.

There is a trade-off between predictability and synchronization overhead. Predictability of processing (non-deterministic) events is improved by added control over the moment when and the way how the event is processed relative to the streaming activities. Increased predictability requires more synchronization between processes and hence additional overhead. Such overhead is undesirable, especially if events occur only sporadically.

8.3.2 Semantic Model

A denotational semantics is often preferred to capture the intended functionality of a process network or to define the functional semantics of a system or programming

language implementing process networks, without specifying unnecessary implementation details. The operational semantics on the other hand allows reasoning about implementation details, such as artificial deadlocks [24] or required buffer capacities [3, 8]. For RPN, a denotational semantics in terms of input/output relations is not possible (Sect. 7) and one based on sequential execution traces is possible, but hides the parallelism of the model. The detailed operational semantics of RPN can be found in [25]. In the next section, we discuss the main concepts.

8.3.3 Communicating Events

Processes or actors in data-flow graphs communicate via FIFO channels. We want to add communication of events and we have to decide what communication mechanism is used for events. It is often the case that what is perceived by a lower level process as an event, is considered to be part of streaming by higher level processes. For instance, a video decoder is decoding a stream of video frames and header information for every frame is an integral part of the data stream. For the lower level frame decoder processes, the frame header information is seen as an event that initializes the component to deal with the specific parameters of the following frame. For this reason, we use the same infrastructure for communicating streams also for events. For the sending process, there is no difference at all; at the receiving side, we distinguish *stream input* ports and *event input* ports. The former are used in ordinary streaming activity; tokens arriving on the latter will trigger discrete events.

8.4 Operational Semantics of RPN

The operational semantics of reactive process networks associates RPNs with a corresponding labeled transition system. An RPN, like a KPN, consists of processes, which may in turn be other RPNs, or primitive processes defined through other means, such as sequential code segments, and the LTS is constructed compositionally, similar to the operational semantics of KPN in Sect. 3. A detailed description of the operational semantics of RPN can be found in [25]; here we concentrate on the important concepts.

In general, two different types of things may happen to the network: data can be streaming through it, or it can encounter events that need to be processed. The events introduce non-determinism. The result should still be as predictable as possible. In particular, we want to guarantee that input consumed before the arrival of a new event will lead to the required output, also if the output has not been completed when the event arrives. In implementations, this is achieved at the expense of additional synchronization or coordination. In specific subclasses of the RPN model, this synchronization may be realized without much overhead. There are very regular subclasses of RPN, such as the SDF-based models of Scenario

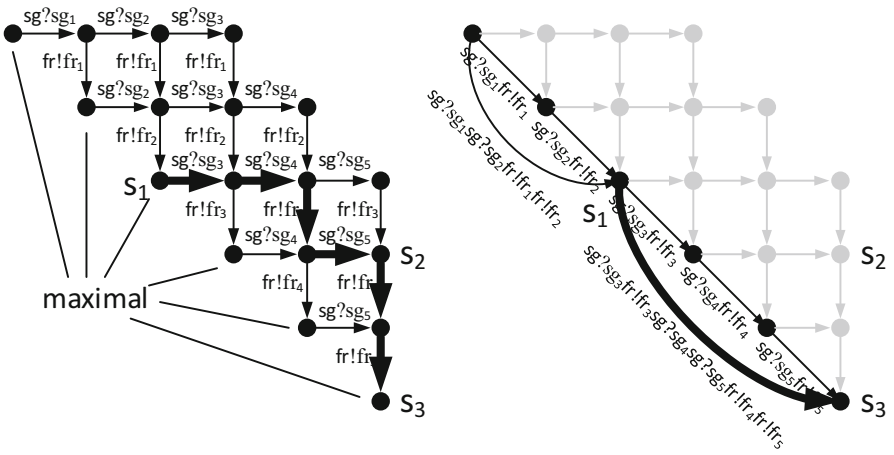


Fig. 13 Streaming transactions

Aware Data Flow (SADF) [59] or Heterochronous Data Flow (HDF) [28]. SDF graphs execute periodic behavior, often called iterations. In-between such iterations is typically a good moment for reconfiguration. In [46] such moments are referred to as *quiescent states*, but as explained earlier, in a pipelined execution, such states may not naturally occur and may need to be enforced by stalling the pipeline.

The operational semantics of RPN models streaming as a sequence of individual read and write actions of the processes involved in the computation of the output, comparable to the semantics of KPN of Sect. 3. Since, conceptually, the reaction of a process network to incoming data is immediately determined, it may not be disturbed by the processing of events. To this end, in RPN semantics, these sequences of actions resulting from a data input are grouped together and represented as single, atomic transitions of the labeled transition system. Effectively, this gives internal actions (i.e., completing the reaction to already received input) priority over processing of events.

This leads to the concept of so-called *streaming transactions*, as illustrated in Fig. 13. The picture on the left shows the states and transitions of a process network performing individual input and output actions. Input actions are shown as horizontal arrows, output actions as vertical arrows. Because of pipelining, the network can perform several input actions before the corresponding output actions are produced. Events should only be accepted in those states where no internal or output actions are available. In Fig. 13, these are all states on the lower-left boundary, such as states s_1 and s_3. In state s_2, some remaining output is still pending and an event cannot be processed yet. Sequences of actions starting from a state without enabled output or internal actions and going back to such a state represent complete reactions of the network to input stimuli. These sequences of actions are grouped together to form atomic transitions, the *streaming transactions*, that end in states where events can safely be processed. For example, the sequence of transitions

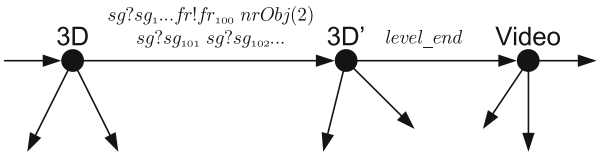


Fig. 14 Execution of the 3D game RPN

with thick arrows in the picture from state s_1 to s_3 forms a streaming transaction; from the end state, only reading of new input is possible, all received input has been fully processed. (These states correspond to end-points of maximal executions for a particular finite input, and closely resemble the quiescent states of [46].) Some of the possible streaming transactions are shown in the picture on the right. The bold one corresponds to the bold path on the left. In this new LTS, the state space consist of only quiescent states.

The streaming transactions of an RPN are determined in two steps. Individual streaming transactions of the constituent processes are determined as well as the processing of events by these processes. Executions of these actions are then taken together to form maximal streaming transactions of the whole network. Such streaming transactions may not consume *input events*. However, they may include occurrences of internal events and hence they can be indeterminate.

Figure 14 shows a part of a possible execution of the 3D game example. The first transition is a streaming transaction, consisting of streaming actions of the processes, but also internal events (*nrObj*(2)). Note that we have abstracted from internal actions of for instance the rendering component, which might also be visible in these transactions. The second transition is an event; the player has reached the end of a level, and the game is reconfigured from 3D mode to the video mode.

Such event transitions of an RPN are directly determined by the reception of input events, followed by the corresponding network transformation. The RPN semantics associates with every event an abstract transformation of the network structure. How such reconfigurations are precisely specified is left up to the concrete instances of models or languages based on this model. In *charts [28], which can be seen as an RPN instance, specification is done by direct refinement of FSM states by data-flow graphs. In SADF [59] actors or processes read control tokens sent to them by the controlling FSM and adapt or suspend their behavior accordingly.

The behavior of the network as a whole is formed by interleaving transitions of both kinds as in the example. An RPN thus has a labeled transition system with executions interleaving events and streaming transactions. In contrast with KPN, this LTS is indeterminate. With this LTS semantics, an RPN can be directly used as a process within a larger RPN and this ways supports hierarchical and compositional specification of RPNs.

8.5 Implementation Issues

The operational semantics of RPN defines the boundaries of the behavior that correct implementations of RPNs should adhere to. Within these boundaries there is still some room for making specific implementation decisions, depending on the application and the context. In this section, we briefly discuss some of these considerations.

8.5.1 Coordinating Streaming and Events

One of the most powerful aspects of KPNs is that execution can take place fully asynchronously. Processes need not synchronize and determinacy of the output is automatically guaranteed. The (deliberate) drawback of the generalization to RPN is that this advantage is (partially) lost. Before processing an input event, the streaming input to the network—conceptually—needs to be frozen and all data must be processed internally. Only when all data has been processed, the event can be applied and the data-flow can be continued. If implemented in this way, the pipelining of the data-flow may be disrupted and deadlines could potentially be missed because of this disruption if the nature of the application doesn't allow this.

In practice, one can do better for many classes of systems. Instead of processing an event for the whole process network at once, it may in some cases be possible to make the changes along with the 'information flow'. In particular, if the response of a network to an event is the forwarding of the event to one or more of its sub-processes, then this forwarding can be synchronized with the flow of data such that pipelining need not be interrupted. This is done for instance for the asynchronous messages following the stream wave-front in StreamIt [60].

The discussed operational semantics suggests that events must always be accepted by any process. In practice, it can be useful to allow a process some control w.r.t. the moment when events are accepted. For example, to allow it to accept events only at moments when the corresponding transformation is most easy to do, because the process is in a well-defined state, e.g. at frame boundaries.

Such an approach can for example be easily implemented if the underlying process network is an SDF graph that can be statically scheduled. It is then possible to define a cyclic schedule in such a way that one iteration of the cycle constitutes a single streaming transaction. Then, a test for newly arrived events can be inserted at the beginning of the cycle and event transitions can be safely executed.

8.5.2 Deadlock Detection and Resolution

The correct execution of KPNs using bounded FIFO implementations depends on the run-time environment to deal with artificial deadlock situations (see Sect. 6, [24, 49]). The same situation may arise in reactive process networks. The dependencies

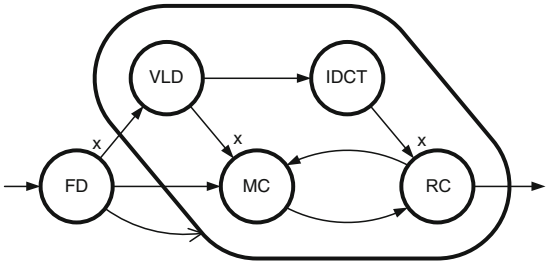


Fig. 15 SADF model of an MPEG-4 SP decoder

may now also include event channels. One can deal with these artificial deadlocks in a similar manner as for ordinary KPNs. Solving an artificial deadlock may be needed for completing the maximal transaction before processing an event.

8.6 Analyzable Models Embedded in RPN

KPN is a model of determinate stream processing, but due to its expressiveness has many undecidable aspects. In practice often Synchronous Data Flow Graphs (SDF) or Cyclo-Static Data Flow Graphs (CSDF) are used as restricted but analyzable subsets of KPN. Obviously, as an extension of KPN, RPN also has many undecidable aspects.

The SADF model of computation uses Synchronous Data Flow graphs as the streaming model, extended with time according to [55] to capture performance aspects. For the control aspects, it uses Markov chains which can be seen as finite state machines decorated with transition probabilities. These probabilities intend to capture the typical behavior of the application in a particular use case. As such it is comparable to the earlier Heterochronous Data Flow [28] model which also makes a combination of SDF and FSMs, but without time and probabilities.

Figure 15 shows an example of an SADF graph of an MPEG-4 Simple Profile decoder. An input stream is analyzed by the FD (frame detector) process which can observe the frame type of the next incoming frame. Different frames may employ a different number of macro blocks. In the figure, this number is denoted by x. For every given number x, the sub-network of the other processes is an SDF graph with fixed rates, which allows a static schedule and the performance of which can be analyzed off-line. For every frame, the frame detector sends an event message to the rest of the network indicating the proper frame type and the sub-network makes the necessary changes and invokes the appropriate schedule.

The Markov model of the transitions between frame types can be used to study the expected performance of the given system [59]. Alternatively, a worst-case analysis can be done to establish a guaranteed performance using techniques such as introduced in [23, 26, 51].

Another example of an efficient combination of (restricted structures of) SDF with events is StreamIt [60] which employs a concept called *teleport messaging* [61]

for sending sporadic messages along the information wavefront. Upstream actors or processes can send event messages to downstream actors to arrive at a specified iteration distance to the iteration at which the first data arrives which is dependent on the output currently being produced. In this case, the compiler and scheduler can automatically take care of the required synchronization.

9 Bibliography

Kahn Process Networks have been introduced by Kahn in [34]. Kahn and McQueen presented a programming/implementation paradigm for KPNs as the behavior of a model of (sequential) programs reading and writing tokens on channels. The Kahn Principle, stating that the operational behavior of such an implementation model conforms to the denotational semantics of [34], was introduced, but not proved, by Kahn in [34]. It was proved [21, 56] for an operational model of transition systems and in [44] for I/O automata. The operational semantics in terms of I/O automata is very much like the semantics in Sect. 3, except that in the I/O-automata semantics, FIFOs are not modeled explicitly; it is assumed that they are implicit in the transition system of the processes at the lowest level. This means that the processes will accept any input at any given time. Composition of networks is then achieved with synchronous communication. By making channels and their FIFOs more explicit one can reason about realizations including memory management (FIFO capacities).

KPN is often informally regarded as the upper bound of a hierarchy of data-flow models, although technically speaking it is not entirely obvious how to compare data-flow processes based on firing rules with either the denotational or operational semantics of KPN. The relationship between both types of models is elaborated in [41].

Scheduling and resource management, possibly in a distributed fashion are important subjects for realizing KPN implementations or simulators. Scheduling process networks using statically bounded channels is an important contribution towards this aim by Parks [49], introducing an algorithm that uses bounded memory if possible. Based on this scheduling policy, a number of tools and libraries have been developed for executing KPNs. Yapi [36] is a C++ library for designing stream-processing applications. Ptolemy II [38] is a framework for codesign using mixed models of computation. The process-network domain is described in [29]. The Distributed Process Networks of [63] form the computational back end of the Jade/PAGIS system for processing digital satellite images. Stevens et al. [57] covers an implementation of process networks in Java. Allen et al. [1] is another implementation for digital signal processing. Common among all these implementations is a multi-threading environment in which processes of the KPN execute in their own thread of control and channels are allocated a fixed capacity. Semaphores control access to channels and block the thread when reading from an empty or writing to a full channel. This raises the possibility of a deadlock when one or more processes are permanently blocked on full channels. A special thread

(preempted by the other threads) is used to detect a deadlock and initiate a deadlock resolution procedure when necessary. This essentially realizes the scheduling policy of [49]. The algorithm of Parks leaves some room for optimization of memory usage by careful selection of initial channel capacities (using profiling) and clever selection of channels when the capacity needs to be increased; see [3]. Basten and Hoogerbrugge [3] also introduces causal chains, also used in this chapter to define deadlocks.

It is argued in [24] as discussed in Sect. 6 that a run-time scheduler for KPN should include local deadlock detection. Such deadlock detection has subsequently been implemented in a number of KPN implementations [2, 18, 32, 47]. To optimize the process it can be organized in a distributed fashion [2] or in a hierarchical way [32]. Cheng and Wawrzynek [16] employs deadlock detection techniques for buffer management in hardware implementations generated from software process network specifications. Alternatively, one could implement dedicated solutions for special classes of dynamic process networks that allow for static buffer allocation solutions. For instance, in [42], filters with data dependent rates are possible, but the rates are made static in the realization using empty or 'dummy' data tokens. Schor et al. [54] describes a dynamic scheduling environment where process network applications are scheduled according to dynamic resource variations. Castrillon et al. [15] presents another run-time for process networks in which the performance of the communication over an interconnect network is explicitly considered to find good mappings of processes to processing elements.

The desire to express non-deterministic behavior and event-based communication has led to additions to pure data-flow models. Examples are the probes of [45] and [36]. Martin [45] describes an extension of the data-flow model with so-called probes, the possibility to test whether a channel has data available for reading. This can be a powerful construct, but it destroys the property of determinacy; the behavior is no longer independent of the concrete scheduling. This probe construct inspired the designers of Yapi [36] to introduce the `select` statement, having the same disadvantage.

In Ptolemy II [19, 29, 38], a framework is defined to connect multiple models of computation, including data-flow and event-based ones. The combination of the reactive and process network domains however, induces too much synchronization overhead to be directly used for implementation.

Many of the combinations of data-flow and reactive behavior are based on a combination of the Synchronous Data Flow model together with some form of reactive behavior [28, 37, 58–60]. The use of an analyzable model such as SDF is natural because it allows for a predictable and determinate combination. The reactive part is frequently specified using (hierarchical) state machines. Lee [37] describes a combination of hierarchical state machines and SDF models, where a complete iteration of the SDF is taken as an action of the state machine. A state machine inside an SDF is required to adhere to the SDF firing characteristics. FunState [58] defines a model, described as 'functions driven by state machines', which deliberately tries to separate functional behavior from control. For FunState, the control part also extends to the coordination of the processes that form a data-

flow graph and the functions are comparable to separate KPN processes. *charts [28] separates hierarchical finite state machine models from concurrency models. In the data-flow domain, a combination of Hierarchical Finite State Machines with Synchronous Data Flow is presented, called Heterochronous Data Flow (HDF). To combine the finite state transitions of an FSM with the typically infinite behavior associated with concurrent models of computations, the infinite behavior is split into finite pieces. For SDF, separate iterations are used. Similarly, FunState uses individual functions for that. For KPN inside RPN, we have used complete reactions in Sect. 8. Scenario Aware Data Flow (SADF) [23, 59] models a system as a finite state machine (decorated with transition probabilities to a Markov Chain) of SDF graphs and is similar to HDF, although primarily with the intention to capture dynamic variation within a determinate execution rather than to capture indeterminate behavior. SADF focuses less on functional specification, but rather on performance modeling by including temporal behavior and stochastic abstraction of the automaton behavior. Moreover, SADF deals explicitly with the effects of pipelining different iterations of the SDF graphs and the intermediate automaton transitions. StreamIt [60] employs an SDF-like model of computation. It allows sending sporadic event between processes in the network by a construct called teleport messaging [61]. The scheduling and compilation framework automatically takes care that all data in the pipeline between the sender and receiver is processed before the message is delivered to the receiving process, similar to the RPN model. Rai et al. [52] deals with the problem of determining suitable distributed states in a streaming applications to process events, in their case reconfigurations and process migrations. A model is provided that describes the timing of bringing the application into such a stable state.

Bhattacharya and Bhattacharyya [7] describes parameterizable SDF models, allowing dynamic reconfiguration during run-time. Processes can change their data-flow behavior depending on parameter settings. In a given SDF configuration, actor executions are characterized by iterations, which fire sub-processes in a particular order that returns the internal buffer states in their original configuration. Such a process can (only) be reconfigured between such iterations and if the data-flow behavior has changed, a new schedule is determined (at run-time).

Similar to RPN in generality, is an approach of Neuendorffer and Lee [46]. It focuses on reconfiguration as a particular kind of event handling or change of model parameters. It defines *quiescent states* as the states where reconfigurations are allowed. These quiescent states are strongly related to the maximal streaming transactions. It also proposes to use FIFO (First In First Out) channel communication also for events or parameters and to divide input ports in streaming input ports and parameter input ports. In contrast with [46], RPN considers more general event handling and reconfiguration than changing parameters. It also focuses on formalizing dependencies between parameters and quiescent points at different levels of a system hierarchy.

The operational semantics of RPN connects easily to implementations of the model. This operational semantics is however not fully abstract, i.e., it contains

more details than strictly necessary. Russell [53] discusses fully abstract semantics of indeterminate data-flow models.

The traditionally separated branch of synchronous data-flow languages such as Lustre [31] or Signal [4] treat sporadic events in a completely different way. Because these models are based on a global synchrony hypothesis (all system components execute conceptually at the pace of a single global clock), absence of an event in a particular clock cycle is easily, deterministically detected. In this case, sporadic events do not necessarily lead to a non-deterministic model. The overhead of the globally synchronous clock may impact efficiency however.

Acknowledgements This work is supported in part by the EC through FP7 IST project 216224, MNEMEE and by the Netherlands Ministry of Economic Affairs under the Senter TS program in the Octopus project.

References

1. Allen G, Evans B, Schanbacher D (1998) Real-time sonar beamforming on a UNIX workstation using process networks and POSIX threads. In: Proc. of the 32nd Asilomar Conference on Signals, Systems and Computers, IEEE Computer Society, pp 1725–1729
2. Allen G, Zucknick P, Evans B (2007) A distributed deadlock detection and resolution algorithm for process networks. In: Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on, vol 2, pp II–33–II–36, https://doi.org/10.1109/ICASSP.2007.366165
3. Basten T, Hoogerbrugge J (2001) Efficient execution of process networks. In: Chalmers A, Mirmehdi M, Muller H (eds) Proc. of Communicating Process Architectures 2001, Bristol, UK, September 2001, IOS Press, pp 1–14
4. Benveniste A, Guemic PL (1990) Hybrid dynamical systems theory and the signal language. IEEE Trans Automat Contr 35:535–546
5. Benveniste A, Caillaud B, Carloni LP, Caspi P, Sangiovanni-Vincentelli AL (2008) Composing heterogeneous reactive systems. ACM Trans Embed Comput Syst 7(4):1–36
6. Berry G, Gonthier G (1992) The Esterel synchronous programming language: Design, semantics, implementation. Sci Comput Program 19:87–152
7. Bhattacharya B, Bhattacharyya S (2001) Parameterized dataflow modeling for DSP systems. IEEE Transactions on Signal Processing 49(10):2408–2421
8. Bhattacharyya S, Murthy P, Lee E (1999) Synthesis of embedded software from synchronous dataflow specifications. J VLSI Signal Process Syst 21(2):151–166
9. Bhattacharyya SS, Deprettere EF, Theelen BD (2013) Dynamic Dataflow Graphs, Springer New York, New York, NY, pp 905–944. https://doi.org/10.1007/978-1-4614-6859-2_28, URL http://dx.doi.org/10.1007/978-1-4614-6859-2_28
10. Brock J, Ackerman W (1981) Scenarios: A model of non-determinate computation. In: Díaz J, Ramos I (eds) Formalization of Programming Concepts, International Colloquium, Peniscola, Spain, April 19–25, 1981, Proceedings, LNCS Vol. 107, Springer Verlag, Berlin, pp 252–259
11. Brookes S (1998) On the Kahn principle and fair networks. Tech. Rep. CMU-CS-98-156, School of Computer Science, Carnegie Mellon University
12. Broy M, Dendorfer C (1992) Modelling operating system structures by timed stream processing functions. Journal of Functional Programming 2(1):1–21, URL citeseer.nj.nec.com/broy92modelling.html
13. Buck J (1993) Scheduling dynamic dataflow graphs with bounded memory using the token flow model. PhD thesis, University of California, EECS Dept., Berkeley, CA

14. Carloni LP, Sangiovanni-Vincentelli AL (2006) A framework for modeling the distributed deployment of synchronous designs. Form Methods Syst Des 28:93–110
15. Castrillon J, Tretter A, Leupers R, Ascheid G (2012) Communication-aware mapping of kpn applications onto heterogeneous mpsocs. In: Proceedings of the 49th Annual Design Automation Conference, ACM, New York, NY, USA, DAC '12, pp 1266–1271, https://doi.org/10.1145/2228360.2228597, URL http://doi.acm.org/10.1145/2228360.2228597
16. Cheng S, Wawrzynek J (2016) Synthesis of statically analyzable accelerator networks from sequential programs. In: Proceedings of the 35th International Conference on Computer-Aided Design, ACM, New York, NY, USA, ICCAD '16, pp 126:1–126:8, https://doi.org/10.1145/2966986.2967077, URL http://doi.acm.org/10.1145/2966986.2967077
17. Davey BA, Priestley HA (1990) Introduction to Lattices and Order. Cambridge University Press, Cambridge, UK
18. Dulloo J, Marquet P (2004) Design of a real-time scheduler for Kahn Process Networks on multiprocessor systems. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA, pp 271–277
19. Eker J, Janneck J, Lee EA, Liu J, Liu X, Ludvig J, Sachs S, Xiong Y (2003) Taming heterogeneity - the ptolemy approach. Proceedings of the IEEE 91(1):127–144, URL http://chess.eecs.berkeley.edu/pubs/488.html
20. Falk J, Haubelt C, Zebelein C, Teich J (2013) Integrated Modeling Using Finite State Machines and Dataflow Graphs, Springer New York, New York, NY, pp 975–1013. https://doi.org/10.1007/978-1-4614-6859-2_30, URL http://dx.doi.org/10.1007/978-1-4614-6859-2_30
21. Faustini A (1982) An operational semantics for pure dataflow. In: Nielsen M, Schmidt EM (eds) Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12–16, 1982, Proceedings, LNCS Vol. 140, Springer Verlag, Berlin, pp 212–224
22. Geilen M (2009) An hierarchical compositional operational semantics of Kahn Process Networks and its Kahn Principle. Tech. rep., Electronic Systems Group, Dept. of Electrical Engineering, Eindhoven University of Technology
23. Geilen M (2011) Synchronous data flow scenarios. Transactions on Embedded Computing Systems 10(2):16:1–16:31
24. Geilen M, Basten T (2003) Requirements on the execution of Kahn process networks. In: Degano P (ed) Proc. Of the 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7–11, 2003. LNCS Vol.2618, Springer Verlag, Berlin
25. Geilen M, Basten T (2004) Reactive process networks. In: EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software, ACM, New York, NY, USA, pp 137–146, http://doi.acm.org/10.1145/1017753.1017778
26. Geilen M, Stuijk S (2010) Worst-case performance analysis of synchronous dataflow scenarios. In: International Conference on Hardware-Software Codesign and System Synthesis, CODES+ISSS 10, Proc., Scottsdale, Az, USA, 24–29 October, 2010, pp 125–134
27. Geilen M, Falk J, Haubelt C, Basten T, Theelen B, Stuijk S (2017) Performance analysis of weakly-consistent scenario-aware dataflow graphs. Journal of Signal Processing Systems 87(1):157–175, https://doi.org/10.1007/s11265-016-1193-7, URL http://dx.doi.org/10.1007/s11265-016-1193-7
28. Girault A, Lee B, Lee E (1999) Hierarchical finite state machines with multiple concurrency models. IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems 18(6):742–760
29. Goel M (1998) Process networks in Ptolemy II. Technical Memorandum UCB/ERL No. M98/69, University of California, EECS Dept., Berkeley, CA
30. Ha S, Oh H (2013) Decidable Dataflow Models for Signal Processing: Synchronous Dataflow and Its Extensions, Springer New York, New York, NY, pp 1083–1109. https://doi.org/10.1007/978-1-4614-6859-2_33, URL http://dx.doi.org/10.1007/978-1-4614-6859-2_33
31. Halbwachs N, Caspi P, Raymond P, Pilaud D (1991) The synchronous programming language LUSTRE. Proceedings of the IEEE 79:1305–1319

32. Jiang B, Deprettere E, Kienhuis B (2008) Hierarchical run time deadlock detection in process networks. In: Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on, pp 239–244, https://doi.org/10.1109/SIPS.2008.4671769
33. Jonsson B (1989) A fully abstract trace model for dataflow networks. In: POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, pp 155–165
34. Kahn G (1974) The semantics of a simple language for parallel programming. In: Rosenfeld J (ed) Information Processing 74: Proceedings of the IFIP Congress 74, Stockholm, Sweden, August 1974, North-Holland, Amsterdam, Netherlands, pp 471–475
35. Kahn G, MacQueen D (1977) Coroutines and networks of parallel programming. In: Gilchrist B (ed) Information Processing 77: Proceedings of the IFIP Congress 77, Toronto, Canada, August 8–12, 1977, North-Holland, pp 993–998
36. Kock, de et al E (2000) YAPI: Application modeling for signal processing systems. In: Proc. of the 37th. Design Automation Conference, Los Angeles, CA, June 2000, IEEE, pp 402–405
37. Lee B (2000) Specification and design of reactive systems. PhD thesis, Electronics Research Laboratory, University of California, EECS Dept., Berkeley, CA, memorandum UCB/ERL M00/29
38. Lee E (2001) Overview of the Ptolemy project. Technical Memorandum UCB/ERL No. M01/11, University of California, EECS Dept., Berkeley, CA
39. Lee E, Messerschmitt D (1987) Synchronous data flow. IEEE Proceedings 75(9):1235–1245
40. Lee E, Sangiovanni-Vincentelli A (Dec 1998) A framework for comparing models of computation. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 17(12):1217–1229, https://doi.org/10.1109/43.736561
41. Lee EA, Matsikoudis E (2007) The Semantics of Dataflow with Firing, Cambridge University Press. URL http://chess.eecs.berkeley.edu/pubs/428.html, chapter from "From Semantics to Computer Science: Essays in memory of Gilles Kahn"
42. Li P, Agrawal K, Buhler J, Chamberlain RD (2010) Deadlock avoidance for streaming computations with filtering. In: Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures, ACM, New York, NY, USA, SPAA '10, pp 243–252, https://doi.org/10.1145/1810479.1810526, URL http://doi.acm.org/10.1145/1810479.1810526
43. Liu X, Lee EA (2008) Cpo semantics of timed interactive actor networks. Theor Comput Sci 409(1):110–125, http://dx.doi.org/10.1016/j.tcs.2008.08.044
44. Lynch N, Stark E (1989) A proof of the Kahn principle for Input/Output automata. Information and Computation 82(1):81–92, URL citeseer.nj.nec.com/lynch89proof.html
45. Martin A (1985) The probe: An addition to communication primitives. Information Processing Letters 20(3):125–130
46. Neuendorffer S, Lee EA (2004) Hierarchical reconfiguration of dataflow models. In: Proc. Second ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2004), IEEE Computer Society Press
47. Olson A, Evans B (2005) Deadlock detection for distributed process networks. In: Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on, vol 5, pp v/73–v/76 Vol. 5, https://doi.org/10.1109/ICASSP.2005.1416243
48. Park D (1979) On the semantics of fair parallelism. In: Abstract Software Specifications, Volume 86 of Lecture Notes in Computer Science, Springer Verlag, Berlin
49. Parks T (1995) Bounded Scheduling of Process Networks. PhD thesis, University of California, EECS Dept., Berkeley, CA
50. Plotkin G (1981) A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Århus University, Computer Science Department, Århus, Denmark
51. Poplavko P, Basten T, van Meerbergen J (2007) Execution-time prediction for dynamic streaming applications with task-level parallelism. In: DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, IEEE Computer Society, Washington, DC, USA, pp 228–235, http://dx.doi.org/10.1109/DSD.2007.52

52. Rai D, Schor L, Stoimenov N, Thiele L (2013) Distributed stable states for process networks - algorithm, analysis, and experiments on intel scc. In: 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), pp 1–10
53. Russell J (1989) Full abstraction for nondeterministic dataflow networks. Symposium on Foundations of Computer Science 0:170–175, http://doi.ieeecomputersociety.org/10.1109/SFCS.1989.63474
54. Schor L, Bacivarov I, Yang H, Thiele L (2014) Adapnet: Adapting process networks in response to resource variations. In: Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, ACM, New York, NY, USA, CASES '14, pp 22:1–22:10, https://doi.org/10.1145/2656106.2656112, URL http://doi.acm.org/10.1145/2656106.2656112
55. Sriram S, Bhattacharyya SS (2000) Embedded Multiprocessors: Scheduling and Synchronization. Marcel Dekker, Inc., New York, NY, USA
56. Stark E (1987) Concurrent transition system semantics of process networks. In: Proc. of the 1987 SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Munich, Germany, January 1987, ACM Press, pp 199–210
57. Stevens R, Wan M, Laramie P, Parks T, Lee E (1997) Implementation of process networks in Java. Technical Memorandum UCB/ERL No. M97/84, University of California, EECS Dept., Berkeley, CA
58. Strehl K, Thiele L, Gries M, Ziegenbein D, Ernst R, Teich J (2001) FunState - an internal design representation for codesign. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9(4):524–544, URL citeseer.nj.nec.com/strehl01funstate.html
59. Theelen BD, Geilen M, Basten T, Voeten J, Gheorghita SV, Stuijk S (2006) A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In: Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design 2006 (MEMOCODE '06), pp 185–194
60. Thies W, Karczmarek M, Amarasinghe S (2002) StreamIt: A language for streaming applications. In: Horspool RN (ed) Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8–12, 2002, Proceedings, LNCS Vol. 2306, Springer Verlag, Berlin, pp 179–196
61. Thies W, Karczmarek M, Sermulins J, Rabbah R, Amarasinghe S (2005) Teleport messaging for distributed stream programs. In: PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM, New York, NY, USA, pp 224–235, http://doi.acm.org/10.1145/1065944.1065975
62. Thomas T Hildebrandt GW Prakash Panangaden (2004) A relational model of non-deterministic dataflow. Mathematical Structures in Computer Science pp 613–649
63. Vayssière J, Webb D, Wendelborn A (1999) Distributed process networks. Tech. Rep. TR 99-03, University of Adelaide, Department of Computer Science, South Australia 5005, Australia
64. Verdoolaege S (2013) Polyhedral Process Networks, Springer New York, New York, NY, pp 1335–1375. https://doi.org/10.1007/978-1-4614-6859-2_41
65. Yates RK (1993) Networks of real-time processes. In: Best E (ed) CONCUR'93: Proc. of the 4th International Conference on Concurrency Theory, Springer Verlag, Berlin, Heidelberg, pp 384–397

Decidable Signal Processing Dataflow Graphs

Soonhoi Ha and Hyunok Oh

Abstract Digital signal processing algorithms can be naturally represented by a dataflow graph where nodes represent function blocks and arcs represent the data dependency between nodes. Among various dataflow models, decidable dataflow models have restricted semantics so that we can determine the execution order of nodes at compile-time and decide if the program has the possibility of buffer overflow or deadlock. In this chapter, we explain the synchronous dataflow (SDF) model as the pioneering and representative decidable dataflow model and its decidability focusing on how the static scheduling decision can be made. Through static scheduling, we can estimate the performance and resource requirement of an SDF graph on a multiprocessor system. In addition the cyclo-static dataflow model and a few other extended models are briefly introduced to show how they overcome the limitations of the SDF model.

1 Introduction

Digital signal processing (DSP) algorithms are often informally, but intuitively, described by block diagrams in which a block represents a function block and an arc or edge represents a dependency between function blocks. While a block diagram is not a programming model, it resembles a formal dataflow graph in appearance. Figure 1 shows a block-diagram representation of a simple DSP algorithm, which can also be regarded as a dataflow graph of the algorithm.

A dataflow graph is a graphical representation of a dataflow model of computation in which a node, or an *actor*, represents a function block that can be executed, or *fired*, when enough input data are available. An arc is a FIFO channel that delivers

S. Ha (✉)
Seoul National University, Seoul, Republic of Korea
e-mail: sha@snu.ac.kr

H. Oh
Hanyang University, Seoul, Republic of Korea
e-mail: hoh@hanyang.ac.kr

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_25

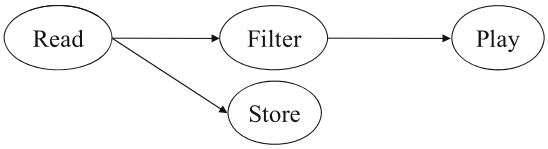


Fig. 1 Dataflow graph of a simple DSP algorithm

data samples, also called *tokens*, from an output port of the source node to an input port of the destination node. If a node has no input port, the node becomes a source node that is always executable. In DSP algorithms, a source node may represent an interface block that receives triggering data from an outside source. The "Read" block in Fig. 1 is a source block that reads audio data samples from an outside source. A dataflow graph is usually assumed to be executed iteratively as long as the source blocks produce samples on the output ports.

The dataflow model of computation was first introduced as a parallel programming model for the associated computer architecture called dataflow machines [8]. While the granularity of a node is assumed as fine as a machine instruction in dataflow machine research, the node granularity can be as large as a well-defined function block such as a filter or an FFT unit in a DSP algorithm representation. The main advantage of the dataflow model as a programming model is that it specifies only the true dependency between nodes, revealing the function-level parallelism explicitly. There are many ways of executing a dataflow graph as long as data dependencies between the nodes are preserved. For example, blocks "Filter" and "Store" in Fig. 1 can be executed in any order after they receive data samples from the "Read" block. They can be executed concurrently in a parallel processing system.

To execute a dataflow graph on a target architecture, we have to determine where and when to execute the nodes, which is called *scheduling*. Scheduling decision can be made only at run-time for general dataflow graphs. A dynamic scheduler monitors the input arcs of each node to check if it is executable, and schedules the executable nodes on the appropriate processing elements. Thus dynamic scheduling incurs run-time overhead of managing the ready nodes to schedule in terms of both space and time. Another concern in executing a dataflow graph is resource management. While a dataflow graph assumes an infinite FIFO queue on each arc, a target architecture has a limited size of memory. Dynamic scheduling of nodes may incur buffer overflow or a deadlock situation if buffers are not carefully managed. A dataflow graph itself may have errors to induce deadlock or buffer overflow errors. It is not decidable for a general dataflow program whether it can be executed without buffer overflow or a deadlock problem.

On the other hand, some dataflow models have restricted semantics so that the scheduling decision can be made at compile-time. If the execution order of nodes is determined statically at compile-time, we can decide before running the program if the program has the possibility of buffer overflow or deadlock. Such dataflow graphs are called *decidable dataflow graphs*. More precisely, a dataflow is decidable if and only if a schedule of which length is finite can be constructed statically. Hence,

in a decidable dataflow graph, the invocation number of each node is finite and computable at compile time. The *SDF (synchronous dataflow) model* proposed by Lee in [19], is a pioneering decidable model that has been widely used for DSP algorithm specification in many design environments including Ptolemy [7] and Grape II [17].

Subsequently, a number of generalizations of the SDF model have been proposed to extend the expression capability of the SDF model. The most popular extension is CSDF (cyclo-static dataflow) [5]. Three other extensions that aim to produce better software synthesis results will also be introduced in this chapter. In this chapter, we explain these decidable dataflow models and focus on their characteristics of decidability. For decidable dataflow graphs, the most important issue is to determine an optimal static schedule with respect to certain objectives since there are numerous ways to schedule the nodes.

2 SDF (Synchronous Dataflow)

In a dataflow graph, the number of tokens produced (or consumed) per node firing is called the output (or the input) sample rate of the output (or the input) port. The simplest dataflow model is the single-rate dataflow (SRDF) in which all sample rates are unity. When a port may consume or produce multiple tokens, we call it multi-rate dataflow (MRDF). Among multi-rate dataflow graphs, synchronous dataflow (SDF) has a restriction that the sample rates of all ports are fixed integer values and do not vary at run time. Note that the SRDF is called sometimes the homogeneous SDF.

Figure 2a shows an SDF graph, which has the same topology as Fig. 1, where each arc is annotated with the number of samples produced and consumed by the incident nodes. There may be delay samples associated with an arc. The sample delay is represented as initial samples that are queued on the arc buffer from the beginning, and denoted as xD where x represents the number of initial samples, as shown on arc AD in Fig. 2a.

From the sample rate information on each arc, we can determine the relative execution rate of two end nodes of the arc. In order not to accumulate tokens unboundedly on an arc, the number of samples produced from the source node should be equal to the number of samples consumed by the destination node in the long run. In the example of Fig. 2a, the execution rate of node C should be

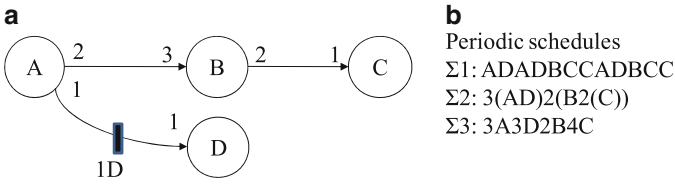


Fig. 2 (**a**) An SDF graph and (**b**) some periodic schedules for the SDF graph

twice as fast as the execution rate of node B on average. Based on this pair-wise information on the execution rates, we can determine the ratio of execution rates among all nodes. The resultant ratio of execution rates among nodes A, B, C and D in Fig. 2a becomes 3:2:4:3.

2.1 Static Analysis

The key analytical property of the SDF model is that the node execution schedule can be constructed at compile time. The number of executions of node A within a schedule is called the repetition count $x(A)$ of the node. A *valid* schedule is a finite schedule that does not reach deadlock and produces no net change in the number of samples accumulated on each arc. In a valid schedule, the ratio of repetition counts is equal to the ratio of execution rates among the nodes so that one iteration of a valid schedule does not increase the samples queued on all arcs. If there exists a valid schedule, the SDF graph is said *consistent* . We represent the repetition counts of nodes in a consistent SDF graph G by vector q_G. For the graph of Fig. 2a,

$$q_G = (x(A), x(B), x(C), x(D)) = (3, 2, 4, 3) \tag{1}$$

Since an SDF graph imposes only partial ordering constraints between the nodes, the order of node invocations can be determined in various ways. Figure 2b shows three possible valid schedules of Fig. 2a graph. In Fig. 2b, each parenthesized term $n(X_1X_2 \ldots X_m)$ represents n successive executions of the sequence $X_1X_2 \ldots X_m$, which is called a looped schedule. If every block appears exactly once in the schedule such as $\Sigma 2$ and $\Sigma 3$ in Fig. 2b, the schedule is called a *single appearance (SA) schedule*. An SA-schedule that has no nested loop is called a *flat SA-schedule*. $\Sigma 3$ of Fig. 2b is a flat SA-schedule, while $\Sigma 2$ is not. Consistency analysis of an SDF graph is performed by constructing a valid schedule; no valid schedule can be found for an erroneous SDF graph.

To construct a valid schedule, we first compute the repetition counts of all nodes. For a given arc e, we denote the source node as $src(e)$ and the destination node as $snk(e)$. The output sample rate of $src(e)$ onto the arc is denoted as $prod(e)$ and the input sample rate of $snk(e)$ as $cons(e)$. Then, the following equation, called a *balance* equation, should be held for a consistent SDF graph.

$$x(src(e))prod(e) = x(snk(e))cons(e) \ for\ each\ e. \tag{2}$$

We can formulate the balance equations for all arcs compactly with the following matrix equation.

$$\Gamma q_G^T = 0 \tag{3}$$

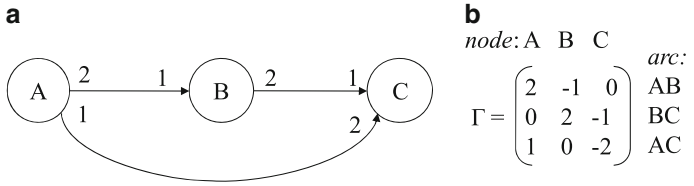


Fig. 3 (**a**) An SDF graph that is sample rate inconsistent and (**b**) the associated topology matrix

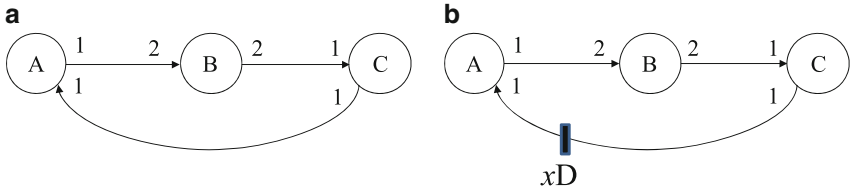


Fig. 4 (**a**) An SDF graph that is deadlocked, and (**b**) the modified graph with initial samples on the feedback arc

where Γ, called the topology matrix of G, is a matrix of which rows are indexed by the arcs in G and columns are indexed by the nodes in G, An entry of the topology matrix is defined by

$$\Gamma(e, A) = \begin{cases} prod(e), & \text{if } A = src(e) \\ -cons(e), & \text{if } A = snk(e) \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

A valid schedule exists only if Eq. (3) has a non-zero solution of repetition vector q_G. Mathematically, this condition is satisfied when the rank of the topology matrix Γ is $n - 1$ [19], where n is the number of nodes in G. In case no non-zero solution exists, the SDF graph is called *sample rate inconsistent*. Figure 3a shows a simple SDF graph that is sample rate inconsistent, and its associated topology matrix. Note that the rank of the topology matrix is 3, not 2.

Sample rate consistency does not guarantee that a valid schedule exists. A sample rate consistent SDF graph can be deadlocked as illustrated in Fig. 4a if the SDF graph has a cycle with insufficient amount of initial samples. The repetition vector, $q_G = (x(A), x(B), x(C))$, is (2,1,2). However, there is no fireable node since all nodes wait for input samples from each other. So we modify the graph by adding initial samples on arc CA in Fig. 4b. Suppose that there is an initial sample on arc CA, or x =1. Then node A is fireable initially. After node A is fired, one sample is produced and queued into the FIFO channel of arc AB. But the graph is deadlocked again since no node becomes fireable afterwards. In this example, the minimum number of initial samples is two in order to rescue the graph from the deadlock condition.

The simplest method to detect deadlock is to construct a static SDF schedule by simulating the SDF graph as follows:

1. At first, make an empty schedule list that will contain the execution sequence of nodes, and initialize the set of fireable nodes.
2. Select one of the fireable nodes and put it in the schedule list. If the set of fireable nodes is empty, exit the procedure.
3. Simulate the execution of the selected node by consuming the input samples from the input arcs and producing the output samples to the output arcs.
4. Examine each destination node of the output arcs, and add it to the set of fireable nodes only if it becomes fireable and its execution count during the simulation is smaller than its repetition count.
5. Go back to step 2 to repeat this procedure.

When we complete this procedure, we can determine if the graph is deadlocked by examining the schedule list. If there is any node that is scheduled fewer times than its repetition count in the schedule list, the graph is deadlocked. Otherwise, the graph is deadlock-free. In summary, an SDF graph is consistent if it is sample rate consistent and it is deadlock-free. Therefore, the consistency of an SDF graph can be statically verified by computing the repetition counts of all nodes (sample rate consistency) and by constructing a static schedule.

2.2 Software Synthesis from SDF Graph

An SDF graph can be used as a graphical representation of a DSP algorithm, from which target codes are automatically generated. Software synthesis from an SDF graph includes determination of an appropriate schedule and a coding style for each dataflow node, both of which affect the memory requirements of the generated software. One of the main scheduling objectives for software synthesis is to minimize the total (sum of code and data) memory requirements.

For software synthesis, the kernel code of each node (function block) is assumed already optimized and provided from a predefined block library. Then the target software is synthesized by putting the function blocks into the scheduled position once a schedule is determined. There are two coding styles, *inline* and *function*, depending on how to put a function block into the target code. The former is to generate an inline code for each node at the scheduled position, and the latter is to define a separate function that contains the kernel of each node. Figure 5 shows three programs based on the same schedule $\Sigma 2$ of Fig. 2b. The first two use the inline coding style, and the third the function coding style.

If we use function calls, we have to pay, at run-time, the function-call overhead which can be significant if there are many function blocks of small granularity. If inlining is used, however, there is a danger of large code size if a node is instantiated multiple times. For the example of Fig. 2, schedule $\Sigma 1$ is not adequate for inlining unlike SA-schedules, $\Sigma 2$ and $\Sigma 3$. Figure 5b shows an alternative code that uses

inlining without a proportional increase of code size to the number of instantiations of nodes. The basic idea is to make a simple run-time system that executes the nodes according to the schedule sequence. It pays the run-time overhead of switch-statements and code overhead for schedule sequence management. Hence an appropriate coding style should be selected considering the node granularity and the schedule.

For each schedule, the buffer size requirement can be computed. If we assume that a separate buffer is allocated on each arc, as is usually the case, the minimum buffer requirement of an arc becomes the maximum number of samples accumulated on the arc during an iteration of the schedule. For the example of Fig. 2, we can compare the buffer size requirements of three schedules as shown in Table 1.

From Table 1, we can observe that the SA schedules usually require larger buffers while they guarantee the minimum code size for inline code generation. In multimedia applications, frame-based algorithms are common where the size of a unit sample may be as large as a video frame or an audio frame. In these applications minimizing the buffer size is as important as minimizing the code size. In general, both code size and buffer size should be considered when we construct a memory-optimal schedule.

Buffering requirements can be reduced if we use buffer sharing. Arc buffers can be shared if their life-times are not overlapped with each other during an iteration of the schedule. The life-time of an arc buffer is defined by a set of durations from the

a

```
main() {
for () {
 for (3) {
   code block of A;
   code block of D;
 }
 for (2) {
  code block of B;
  for (2) {
    code block of C;
  }
 }
} /* end of for */
} /* end of main */
```

b

```
main() {
 for () {
   switch (i) {
   case 1-3:
      code block of A;
      code block of D;
      break;
    case 4-5:
      code block of B;
      for (2) {code block of C;}
      break;
    default: i = 1; break;
   }
   i++;
 } /* end of for */
} /* end of main */
```

c

```
A() { code block of A; }
B() { code block of B; }
C() { code block of C; }
D() { code block of D; }

main() {
for() {
 for (3) { A() ; D(); }
 for (2) {
   B();
   for (2) { C(); }
 }
} /* end of for */
} /* end of main */
```

Fig. 5 Three programs based on the same schedule $\Sigma 2$ of Fig. 2b

Table 1 Buffer requirements for three schedules of Fig. 2b

Schedule	Arc AB	Arc AD	Arc BC	Total
$\Sigma 1$: ADADBCCADBCC	4	2	2	8
$\Sigma 2$: 3(AD)2(B2(C))	6	2	2	10
$\Sigma 3$: 3A3D2B4C	6	4	4	14

source node invocation that starts producing a sample to the buffer to the completion of the destination node that empties the buffer. Consider schedule $\Sigma 1$ of Fig. 2. The buffer life-time of arc BC consists of two durations, {BCC, BCC}, in the schedule. Since the buffer of arc AD is never empty, the buffer life-time of arc AD is the entire duration of the schedule. If we remove the initial sample on arc AD, the buffer life-time of arc AD consists of three durations, {AD, AD, AD}. Then we can share the two arc buffers of arc AD and arc BC since their life-times are not overlapped. A more aggressive buffer sharing technique has been developed by separating global sample buffers and local pointer buffers in case the sample size is large in frame-based applications [20]. The key idea is to allocate a global buffer whose size is large enough to store the maximum amount of live samples during an iteration of the schedule. Each arc is assigned a pointer buffer that stores pointers to the global buffer.

Code size can also be reduced by sharing the kernel of a function block when there are multiple instances of the same block [30] in a dataflow graph. Multiple instances of the same block are regarded as different blocks, and the same kernel, possibly with different local states, may appear several times in the generated code. A technique has been proposed to share the same kernel by defining a shared function. Separate state variables and buffers should be maintained for each instance, which define the *context* of each instance. The shared function is called with the context of an instance as an argument at the scheduled position of the instance. To decide whether sharing a code block is beneficial or not, the overhead and the gain of sharing should be compared. If Δ is an overhead that is incurred by function sharing, R is a code block size, and n is the number of instances of a block, the decision function for code sharing is summarized as the following inequality: $\frac{\Delta}{(n-1)R} < 1$.

For more detailed information on the code generation procedure and other issues related with software synthesis from SDF graphs, refer to [3].

2.3 Static Scheduling Techniques

Static scheduling of an SDF graph is the key technique of static analysis that checks the consistency of the graph and determines the memory requirement of the generated code. Since an SDF graph imposes only partial ordering constraints between the nodes, there exist many valid schedules and finding an optimal schedule has been actively researched.

2.3.1 Scheduling Techniques for Single Processor Implementations

Since the memory requirement of the automatically synthesized code depends on the schedule of a given SDF graph, finding an optimal schedule for single processor implementation has been actively researched. Since the problem of

finding a schedule with minimum buffer requirement for an acyclic graph is NP-complete, various heuristic approaches have been proposed. Since a single appearance schedule guarantees the minimum code size for inline code generation, a group of researchers have focused on finding a single appearance schedule that minimizes the buffer size. Bhattacharyya et al. developed two heuristics: APGAN and RPMC, to find an SA-schedule that minimizes the buffer requirements [4]. Ritz et al. used an ILP formulation to find a flat single appearance schedule that minimizes the buffer size [26] considering buffer sharing. Since a flat SA-schedule usually requires more data buffer than a nested SA-schedule, it is not evident which approach is better between these two approaches.

Another group of researches tries to minimize only the buffer size. Ade et al. presented an algorithm to determine the smallest possible buffer size for arbitrary SDF applications [1]. Though their work is mainly targeted for mapping an SDF application onto a Field Programmable Gate Array (FPGA) in the GRAPE environment, the computed lower bound on the buffer requirement is applicable to software synthesis. Govindarajan et al. [9] developed a rate optimal compile time schedule, which minimizes the buffer requirement by using linear programming formulation. Since the resultant schedule will not be an SA-schedule in general, a function coding style should be used to minimize the code size in the generated code.

No previous work exists that considers all design factors such as coding styles, buffer sharing, and code sharing. In spite of extensive prior research efforts, finding an optimal schedule that minimizes the total memory requirement still remains an open problem, even for single processor implementation.

2.4 Parallel Scheduling of SDF Graphs

Since an SDF graph imposes only partial ordering constraints between the nodes, it exposes the functional parallelism of an application explicitly, which is a very desirable feature for multiprocessor implementation. Unlike single processor implementation where the execution length is independent of scheduling, the execution length of an application heavily depends on how to parallelize the application. Thus, the main scheduling objective for multiprocessor implementation is to reduce the execution length or the throughput of a given SDF graph.

While there are numerous techniques developed for parallel scheduling, they usually assume a single rate dataflow graph where each node is executed only once in a single iteration. And they primarily focus on exploiting the functional parallelism of an application to minimize the length of the schedule, called *makespan*. In stream-based applications, however, maximizing the throughput is more important than minimizing the schedule length. Pipelining is a popular way of improving the throughput of a dataflow graph. For example Hoang et al. have proposed a pipelined mapping/scheduling technique based on a list scheduling

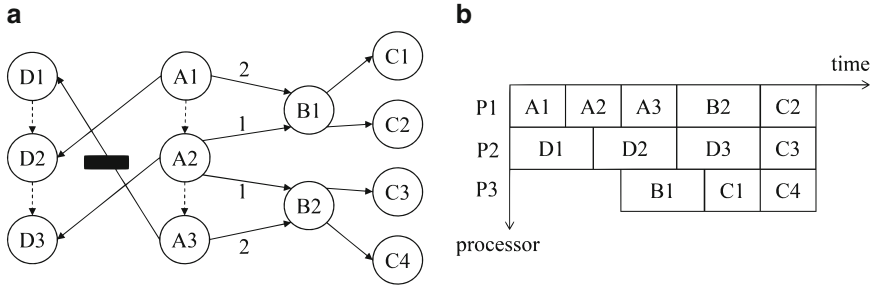


Fig. 6 (**a**) An APEG (acyclic precedence expanded graph) of the SDF graph in Fig. 2a, (**b**) a parallel scheduling result displayed with a Gantt chart

heuristic [12]. They maximize the throughput of a single rate dataflow graph on a homogeneous multi-processor architecture.

To apply these techniques to an SDF graph directly, we need to translate an SDF graph to a single rate task graph, called an APEG (Acyclic Precedence Expanded Graph) or simply EG (Expanded Graph) [24]. A node of an SDF graph is expanded to as many nodes in the EG as the repetition counts of the node. As an example, the corresponding EG of the graph of Fig. 2a is shown in Fig. 6a where nodes A and D are expanded to three invocations, node B to 2, and node C to four invocations, respectively. The number of samples communicated through each arc is unity unless specified otherwise; if an arc is annotated with a non-unity sample rate, such as an arc between nodes A1 and B2, the arc can be split into as many uni-rate arcs as the number to make a single-rate dataflow graph. If a node has any internal state, dependency arcs should be added between node invocations: In the figure, we assume that nodes A and D have internal states and the dependency between invocations is represented by dashed arcs. Note that the initial sample on arc AD is placed on the arc between A3 and D1. Since the initial sample breaks the execution dependency between A3 and D1 in the same iteration, D1 can be executed before A3.

After we translate an SDF graph to an APEG, we can apply an existent parallel scheduling algorithm to schedule an SDF graph on a multiprocessor architecture. Figure 6b shows a parallel scheduling result with a Gantt chart where the vertical axis represents the processing elements of the target system and the horizontal axis represents the elapsed time. There are some issues worth noting in this approach.

First, loop-level data parallelism in an SDF graph is translated into functional parallelism in the EG. Nodes B and C in Fig. 2a express data-level parallelism since multiple invocations can be executed in parallel. But all invocations are translated into separate nodes that can be scheduled independently, ignoring the loop structure, in the EG. As a result, the same block can appear several times in the schedule, which may incur significant code size overhead if inline coding style is used. While it is a reasonable way to exploit the loop-level data parallelism, it may result in a very expensive solution. Second, the total number of nodes in the EG is the sum

of all repetition counts of the nodes. A simple SDF graph with non-trivial sample rate changes may result in a huge EG. Therefore the algorithm complexity of a parallel scheduling technique should be low enough to scale well as the graph size increases. Third, multiple invocations of the same node are likely to be mapped to different processors. If a node has internal states (for instance node D in Fig. 6b), the internal states should be transferred between invocations, which incurs significant run-time overhead. And additional code should be inserted to manage the internal states in the generated code.

Therefore several parallel scheduling techniques have been proposed that work with the SDF graph directly without APEG translation. A node with internal states is constrained to be mapped to a single processor. The most popular approach is to use an evolutionary algorithm such as genetic algorithm and simulated annealing where node mapping to processors is improved iteratively until no further improvement can be found, A recent approach considers functional parallelism, loop-level parallelism, and pipelining simultaneously to minimize the throughput of the graph [32]. In this work, a node can be mapped to multiple processors if the kernel code of a coarse grain node has a parallel loop inside.

2.4.1 Scheduling Objectives

Unlike single processor implementation, there is a trade-off between resource requirements and the throughput performance in multi-processor implementation of SDF graphs. As more processing elements are used, higher throughput performance can be achieved if the application has sufficient degree of parallelism to utilize the processing elements effectively. Buffer sizes also affect the schedule of an SDF graph.

Figure 7 illustrates how the throughput is affected by the buffer size with the same node mapping to processors. Figure 2a shows a simple SDF graph that consists of three nodes whose mapping and execution times are given. In case the buffer size of an arc is fixed, we can add a feedback arc with the same number of initial tokens as the buffer size to the SDF graph to explicitly express the buffer size information. Figure 7b displays the schedule when the buffer size of arc AB is 4. The third invocation of node A (A3) can be executed after the first invocation of node B (B1) finishes and the next iteration starts at 5 time unit. If we increase the buffer size of arc AB from 4 to 6, a better throughput performance is achieved as shown in Fig. 7c.

Stuijk et al. have explored the trade-off and obtained the Pareto-optimal solutions in terms of the buffer size and the throughput, assuming that there is no constraint on the number of processors [29]. As long as the throughput constraint is satisfied, it is better to minimize the resource requirements. This work is extended to consider other resource constraints in [28] considering the case when multiple SDF graphs run on a heterogeneous multi-processor system. There is a limitation on the number of available processors and resource sharing between applications needs to be considered. Other extensions can be found in a web-site (http://www.es.ele.tue.nl/sdf3/) that they manage to make the proposed technique open to public under the name of SDF3 (SDF for Free) [29]. For more details on the throughput analysis of dataflow graphs, refer to [11].

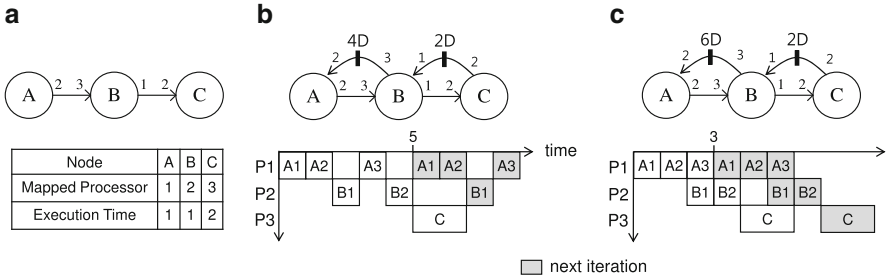


Fig. 7 (**a**) A simple SDF graph and its mapping information, (**b**) and (**c**) the same SDF graph with given channel buffer sizes and its parallel scheduling result displayed with a Gantt chart: buffer size of arc AB is 4 in (**b**) and 6 in (**c**)

There is also a trade-off between latency and throughput performance in an SDF graph. The latency of an SDF graph is defined as the time difference from the latest finish time to the start time of an iteration. For instance, the latency performance of schedule in Fig. 7b is 7 time units. Note that the same latency performance can be obtained by using a single processor. To increase the throughput performance, three processors are used. In case there are heterogeneous processors, we may want to minimize the cost or energy consumption. In summary, there are multiple objectives to consider in parallel scheduling of an SDF graph.

Another important factor to consider in multiprocessor implementation is the communication delay between two nodes mapped onto different processors. Even though the schedule diagram shown in Fig. 7 ignores the communication delay, the destination node cannot be executed immediately after the source node finishes its execution in reality. Thus it is necessary to model the communication network when finding a parallel schedule. Since the communication delay depends on the volume of the data transferred, we need to consider the data size when making a mapping decision.

2.4.2 Execution Strategies

When constructing a parallel schedule of an SDF graph, we assume that the execution time of a node on each processing element is known and communication delay between two nodes is fixed. In practice, however, the execution time of a node may change and the communication delay usually vary due to resource contention. How can we handle the dynamic behavior of the application at run-time? There are four different execution strategies to execute an SDF application at run-time [18]: fully static, self-timed, static-assignment, and fully dynamic.

In the fully static scheduling, we keep not only the mapping and scheduling decision made at compile-time but also the timing information. To this end, the execution time of each node and worst-case communication delay should be conservatively estimated and the scheduling is constructed based on those

conservatively estimated information. If a node finishes earlier than the assumed completion time in the schedule, the run-time scheduler delays the completion of the node. The rationale of using this strategy is to guarantee the real-time performance since it will produce the same scheduling result at run time as expected at compile-time. It is very desirable for hard real-time systems that are willing to pay the price to guarantee the timing correctness. Since the resource utilization may be severely degraded, however, it is seldom adopted in casual signal processing systems.

The self-timed scheduling strategy keeps the mapping and execution order of nodes on each processor, but ignore the timing information while the static assignment scheduling strategy keeps the mapping information only. Suppose a static scheduling list $O = \{o_1, o_2, \ldots, o_p\}$ is defined for each processor where p denotes the number of node instances running on the processor in one iteration and o_i indicates a node instance: lower index i of o_i means higher priority. In the self-timed scheduling strategy, the run-time scheduler examines the node instance of the list sequentially and executes the node if it is fireable. Otherwise, it waits until all input samples are available even though there are other fireable node instances in the list, which may decrease the processor utilization. Since it needs to check the firing condition of only a single node instance, run-time scheduling overhead is little.

On the other hand, we may change the execution order of nodes in the static assignment strategy. At run-time, node instances can be split into three sets at time t: F for finished instances, B for blocked instances that have any input arc e such that $cons(e) > b(e)$ where $b(e)$ means the number of samples on the input arc, and R for ready node instances where for all input arcs e, $cons(e) \leq b(e)$. The run-time scheduler executes a node instance o_i such that $o_i \in R$ and o_i has the highest priority of R. The scheduling information can be used to give priorities to the mapped nodes in each processor in this strategy.

The fully dynamic scheduling strategy ignores all mapping and scheduling decisions. The static scheduling result can be used to define the priority of nodes. A central run-time system maintains the set of unfinished node instances in each iteration. It finds a fireable node instance with the highest priority in the set and maps it to an available processor. Even though it may maximize the resource utilization, the real-time performance can hardly be guaranteed and run-time overhead may overweigh the benefit of resource utilization.

While most work on multiprocessor implementation of SDF graphs assumes self-timed scheduling, some recent researches considers static assignment scheduling as a viable implementation technique of SDF task. A main benefit of static assignment scheduling over self-timed scheduling is that it may tolerate large variation of node execution times or communication delays. Since we assume a fixed node execution time and a fixed communication delay when we make a static schedule, the run-time behavior may deviate largely from the static schedule. A processor may be idle while the current scheduled node is waiting for the arrival of input data from the other processors even though there are other executable nodes. Thus static scheduling may result in waste of resources while dynamic scheduling changes the execution order of nodes to increase resource utilization. A key issue for dynamic scheduling is how to assign priorities to the mapped nodes on each processor. There is a recent

work to find an optimal static mapping and priority assignment to minimize the resource requirement under a throughput constraint [16].

Note that self-timed and static-assignment scheduling strategies do not guarantee the satisfaction of timing constraints even if the static schedule is constructed based on the worst case execution time (WCET) of nodes. If the execution time of a node becomes smaller than its WCET, the order of node firings may vary at run-time, which may lengthen the total execution time of the application if there is a shared resource. The node may delay the execution of a node in the critical path by occupying the shared resource unexpectedly. This behavior is known as "scheduling anomaly" of multiprocessor scheduling [10]. If other tasks share the processor with an SDF application, the problem becomes severe. Therefore it is necessary to devise a technique to conservatively estimate the worst-case run-time performance of an SDF application when self-timed and static-assignment scheduling strategies are used.

2.4.3 Scheduling of Multiple SDF Graphs

Most work on parallel scheduling of SDF graphs assumes that an application uses the system exclusively. Since it becomes more popular to run multiple applications on a multiprocessor system, we need to consider multi-tasking in parallel scheduling of SDF graphs. One solution is to statically schedule the multiple tasks up to their hyper-period that is defined as a least common multiple of all task periods [15]. Since the hyper-period can be huge if the task periods are relatively prime, this approach is not practical in general. Also this approach requires that the starting offset of all application are known and fixed.

To avoid these limitations and allow processor sharing between SDF graphs and conventional real-time tasks, there have been proposed several techniques that transform dataflow graphs into a set of independent real-time tasks so as to take advantage of existing real-time scheduling techniques [2, 6, 27]. The basic philosophy they have in common is that each node of dataflow graphs is transformed to a periodic (or sporadic) real-time task. A starting offset is introduced to each transformed task in order to emulate the behavior of data-dependencies in the original graph. This approach, however, does not utilize the static analyzability of the SDF model so that it may produce poor resource utilization and excessive buffer requirement.

Another approach has been recently proposed that preserves the advantages of static scheduling while allowing arbitrary starting offsets and processor sharing among multiple SDF graphs [14]. It consists of two phases. In the first phase, each application is scheduled separately as if it monopolizes the entire system. The second phase uses a meta-heuristic to find the combination of per-graph schedules to minimize the resource requirement by processor sharing. It is claimed that this technique exhibits better resource and buffer efficiency than the transformation technique.

2.5 *Hardware Synthesis from SDF Graph*

While the target architecture is given as a constraint for software synthesis, the target hardware structure can be synthesized in hardware synthesis from an SDF graph. Therefore, we can achieve the *iteration bound* of an SDF graph in theory (see [22] to find the definition of the iteration bound of a graph) if there is no limitation on the hardware size. Since there is a trade-off between hardware cost and the throughput performance, however, architecture design and node scheduling should be considered simultaneously under given design constraints.

A key issue in hardware synthesis is to preserve the SDF semantics to maintain the correctness of the graph. In the SDF model, two samples that have the same value should be distinguished as separate samples while the same value is not identified as a new event in a hardware logic. So the arrival of an input sample should be notified somehow. And if a node has more than one input port, the node should wait until all input ports receive data samples before the node starts execution. It means that we need some control logic to perform scheduling of the nodes. There are two types of controllers: distributed controller and centralized controller. In the centralized control scheme, the execution timing of each node is controlled by a central scheduler. The execution timing can be determined at compile-time by static scheduling of the graph. In a distributed scheme, a node is associated with a control logic that monitors the input queues and triggering the node execution when all input queues have input samples to fire the node.

For hardware synthesis, a node should be specified by a hardware description language that will be synthesized by a CAD tool, or by a function block that is mapped to a pre-defined hardware IP. If an hardware IP is used, interface between the IP and the rest of the system should be designed carefully. Since the interface design is a laborious and error-prone task, extensive researches are being performed on the automatic interface synthesis.

In summary, hardware synthesis from a SDF graph involves the following problems: architecture and datapath synthesis, controller synthesis, and interface synthesis. The node granularity in a SDF graph also affects the hardware synthesis procedure. Various issues in hardware synthesis for a coarse grained graph is discussed in [13]. For FPGA synthesis from a fine-grained graph, see [31] for more detailed information.

3 Cyclo-Static Dataflow (CSDF)

The strict restriction of the SDF model, that all sample rates are constant, limits the expression capability of the model. Figure 8a shows a simple SDF graph that models a stereo audio processing application where the samples for the left and right channels are interleaved at the source. The interleaved samples are distributed by node D to the left (node L) and to the right (node R) channel. In this example, the

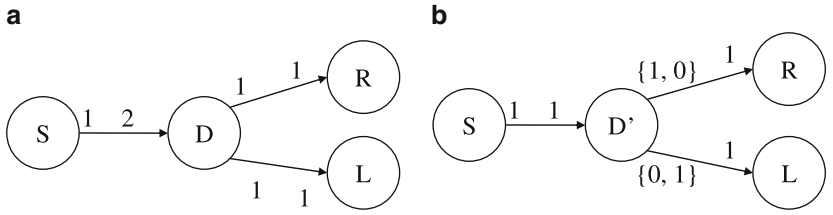


Fig. 8 (**a**) An SDF graph where node D is a distributor block and (**b**) a CSDF graph that shows the same behavior with a different version of a distributor block

distributor node (node D) waits until two samples are accumulated on its input arc to produce one sample at each output arc. A more natural implementation would be to make the distribution node route an input sample to two output ports alternatively at each arrival. A useful generalization of the SDF model, called the cyclo-static dataflow (CSDF), makes it possible [5].

In a cyclo-static dataflow graph, the sample rates of an input or output port may vary in a periodic fashion. Figure 8b shows how the CSDF model can specify the same application as Fig. 8a. To specify the periodic change of the sample rates, a tuple rather than an integer is annotated at the output ports of node D'. For instance, "{1,0}" on arc D'R denotes that the rate change pattern is repeated every other execution, where the rate is 1 at the first execution, and 0 at the second execution. Similarly, the periodic sample rate "{0,1}" means that the rate is 0 at every $(2n+1)$-th iteration and 1 at the other iterations.

Note that Fig. 8a, b represent the same application in functionality. One firing of node D in the SDF graph is broken down into two firings of node D' in the CSDF graph. Thus we have to split the behavior of node D' into phases. The number of phases is determined by the periods of the sample rate patterns of all input and output ports. In general, we can convert a CSDF graph to an *equivalent* SDF graph by merging as many firings of a CSDF node as the number of phases into a single firing of an equivalent SDF node. For instance, node D' in the CSDF graph repeats its behavior every two firings, and the number of phases becomes 2. So an equivalent SDF node can be constructed by merging two firings of node D' into one, which is node D in the SDF model. The sample rates of input and output ports are adjusted accordingly by summing up the number of samples consumed and produced during one cycle of periodic behavior.

The CSDF model has a big advantage over the SDF model in that it can reduce the buffer requirement on the arcs. In the example shown in Fig. 8, the minimum size of input buffer for node D should be 2 in the SDF model while it is 1 in the CSDF model.

3.1 Static Analysis

Since we can construct an equivalent SDF graph, static analysis and scheduling algorithms developed for SDF are also applicable to CSDF. For formal treatment, we use vectors to represent the periodic sample rates in CSDF: For an arc e, the output sample rate vector of $src(e)$ and the input sample rate vector of $snk(e)$ are denoted by **prod(e)** and **cons(e)** in CSDF. Figure 9a shows another CSDF graph that has non-trivial periodic patterns of sample rates. The sample rate vectors for the graph become the following:

$$\mathbf{prod(AC)} = (1, 0, 0), \mathbf{cons(AB)} = (0, 2), \mathbf{prod(BC)} = (0, 1, 0),$$
$$\mathbf{prod(AB)} = \mathbf{cons(BC)} = \mathbf{cons(AC)} = (1).$$

To make an equivalent SDF node for each CSDF node, we have to compute the repetition period for the phased operation of the CSDF node. First we obtain the period of the sample rate variation for each port, which is the size of the sample rate vector. Let dim(**v**) be the dimension of the sample rate vector **v**. Then the repetition period of a CSDF node A, denoted by $p(A)$ becomes the least common multiple (lcm) value of all dim(**v**) values for the input and output ports of the node. For the example of Fig. 9a, the repetition periods become the following:

$$p(A) = lcm(dim(\mathbf{prod(AB)}), dim(\mathbf{prod(AC)})) = lcm(3, 1) = 3.$$
$$p(B) = lcm(dim(\mathbf{cons(AB)}), dim(\mathbf{prod(BC)})) = lcm(2, 3) = 6.$$
$$p(C) = lcm(dim(\mathbf{cons(AC)}), dim(\mathbf{prod(BC)})) = lcm(1, 1) = 1.$$

If $p(A)$ firings of CSDF node A are merged into a single firing, an equivalent SDF actor A' is obtained. Hence the equivalent SDF graph is obtained as shown in Fig. 9b where node B' is obtained by merging 6 firings of node B in the CSDF graph. We denote this equivalence relation as $B' \approx 6B$. For an equivalent SDF node, the scalar sample rate of a port should be determined. Let $\sigma(\mathbf{v})$ be the sum of elements in vector **v**. The total number of samples produced or consumed on arc e of CSDF node A per the corresponding SDF node execution is given by $p(A)\frac{\sigma(\mathbf{prod(e)})}{dim(\mathbf{prod(e)})}$ or $p(A)\frac{\sigma(\mathbf{cons(e)})}{dim(\mathbf{cons(e)})}$. So, we can construct a topology matrix for the equivalent SDF graph as follows:

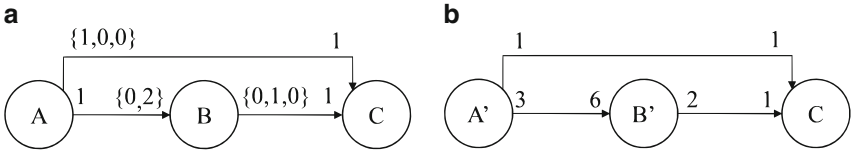


Fig. 9 (**a**) A cyclo-static dataflow graph and (**b**) its corresponding SDF graph

$$\Gamma(e, A) = \begin{cases} p(A)\frac{\sigma(\mathbf{prod(e)})}{dim(\mathbf{prod(e)})}, & \text{if } A = src(e) \\ -p(A)\frac{\sigma(\mathbf{cons(e)})}{dim(\mathbf{cons(e)})}, & \text{if } A = snk(e) \\ 0, & \text{otherwise} \end{cases} \quad (5)$$

We can check the sample rate consistency with this topology matrix. For the graph in Fig. 9b, the topology matrix and the repetition vector become:

$$\Gamma = \begin{pmatrix} 3 & -6 & 0 \\ 1 & 0 & -1 \\ 0 & 2 & -1 \end{pmatrix}$$

$$\mathbf{q_G} = (2, 1, 2)$$

Since rank of Γ is 2, the CSDF graph is sample rate consistent. Moreover, a valid schedule includes two invocations of nodes A' and C, and one invocation of node B'. This means that a valid CSDF schedule contains 6A, 6B and 2C since $A' \approx 3A$ and $B' \approx 6B$. The deadlock detection algorithm for an SDF graph in Sect. 2.1 is applicable for a CSDF graph, which is to construct a static schedule by simulating the graph.

3.2 Static Scheduling and Buffer Size Reduction

One strategy of scheduling a CSDF graph is to schedule the equivalent SDF graph and replace the execution of the equivalent node with the multiple invocations of the original CSDF node. We can obtain the following schedule for the graph in Fig. 9: $\Sigma 1 = 2A'B'2C = 6A6B2C$. Then the minimum buffer requirement on arc AB becomes 6. We can construct better schedules in terms of buffer requirements by utilizing the phased operation of a CSDF node. For the case of CSDF graph of Fig. 9a, we can construct a better schedule as follows.

1. Initially nodes A and B are fireable, so schedule nodes A and B: $\Sigma 2$ = "AB".
2. Since node A is the only fireable node, we schedule node A again: $\Sigma 2$ = "ABA"
3. Now two samples are accumulated on arc AB and the second phased of node B can start. So schedule node B: $\Sigma 2$ = "ABAB".
4. Node C becomes fireable. Schedule node C for the first time: $\Sigma 2$ = "ABABC".
5. We can schedule nodes A and B twice: $\Sigma 2$ = "ABABCABAB"
6. At this moment, only one sample is stored on arc AC and we can fire nodes A and B. We choose to schedule the fifth invocation of node B to produce one sample on arc BC. $\Sigma 2$ = "ABABCABABB"
7. Then, node C becomes fireable. Schedule node C: $\Sigma 2$ = "ABABCABABBC"
8. Finally we schedule node A twice and node B once to complete one iteration of the schedule: $\Sigma 2$ = "ABABCABABBCAAB"

9. Since schedule $\Sigma 2$ contains 6A, 6B and 2C, scheduling is finished and no deadlock is detected.

Schedule $\Sigma 2$ requires two buffers on arc AB, which is three times better than schedule $\Sigma 1$. Generally, as sample rates vary more, the buffer size reduction becomes more significant. This gain is obtained by splitting the CSDF node into multiple phases. But we have to pay the overhead of code size since a single appearance schedule is given up. In general, there are more valid schedules for a CSDF graph than for the equivalent SDF model. Therefore, discussion on the SDF scheduling can be applied to the CSDF model, but with increased complexity of scheduling problems.

3.3 Hierarchical Composition

Another advantage of CSDF is that it offers more opportunities of clustering when constructing a hierarchical graph. It also allows a seemingly delay-free cycle of nodes, while no delay-free cycle is allowed in SDF. Figure 10a shows an SDF graph with four nodes A,B,C and D. All sample rates are unity since no sample rate is annotated on any arc. The graph can be scheduled without deadlock since there is an initial delay sample between nodes A and B. One unique valid schedule of this graph is "BCDA". Suppose we cluster nodes A and B into an hierarchical node W in CSDF and W' in SDF as illustrated in Fig. 10a, b respectively. Since CSDF node W fires node B at every $(2n+1)$-th cycle and node A at every $2n$-th cycle, the input and the output sample rate vectors of node W become "{0,1}" and "{1,0}" respectively.

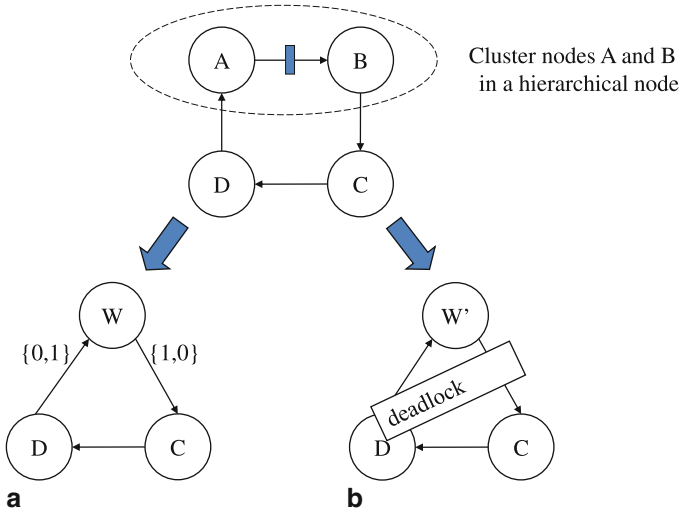


Fig. 10 Clustering of nodes A and B into an hierarchical node in (**a**) CSDF and (**b**) in SDF

Therefore, the CSDF graph can be scheduled without deadlock and a valid static schedule is "WCDW" where node B is fired at the first invocation of node W and node A is fired at the second invocation. On the other hand, SDF node W' should execute both nodes A and B when it is fired. Therefore, the SDF graph as shown in Fig. 10b is deadlocked since nodes W', D, and C wait for each other.

Clustering of nodes may cause deadlock in SDF even though the original SDF graph is consistent. On the other hand, a CSDF graph that includes a cyclic loop without an initial delay can be scheduled without deadlock if the periodic rates are carefully determined. Therefore, the CSDF model allows more freedom of hierarchical composition of the graph.

4 Other Decidable Dataflow Models

4.1 FRDF (Fractional Rate Dataflow)

The SDF model does not make any assumption on the data types of samples as long as the same data types are used between two communicating nodes. To specify multimedia applications or frame-based signal processing applications, it is natural to use composite data types such as video frames or network packets. If a composite data type is assumed, the buffer requirement for a single data sample can be huge, which amounts to 176×144 pixels for a QCIF video frame for instance. Then reducing the buffer requirement becomes more important than reducing the code size when we make a schedule.

Figure 11a shows an SDF subgraph of an H.263 encoder algorithm for QCIF video frames. A QCIF video frame consists of 11×9 macroblocks whose size is 16×16 pixels. Node ME that performs motion estimation consumes the current and the previous frames as inputs. Internally, the ME block divides the current frame into 99 macroblocks and computes the motion vectors and the pixel differences from the previous frame. And it produces 99 output samples at once where each output sample is a macroblock-size data that represents a 16×16 array of pixel differences. Node EN performs macroblock encoding by consuming one macroblock at a time and produces one encoded macroblock as its output sample.

This SDF representation is not efficient in terms of buffer requirement and performance. Since node ME produces 99 macroblock-size samples at once after consuming a single frame size sample at each invocation, we need a 99-macroblock-

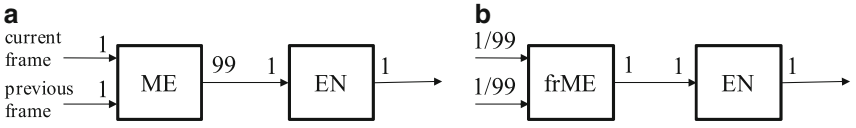


Fig. 11 A subgraph of an H.263 encoder graph (**a**) in SDF and (**b**) in FRDF

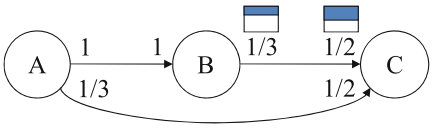


Fig. 12 An FRDF graph in which sample types on arc BC and AC are a composite and a primitive type, respectively

size buffer or a frame-size buffer ($99 \times 16 \times 16 = 176 \times 144$) to store the samples on the arc between nodes ME and EN. Moreover node EN cannot start execution before node ME finishes motion estimation for the whole input frame. As this example demonstrates, the SDF model has inherent difficulty of efficiently expressing the mixture of a composite data type and its constituents: a video frame and macroblocks in this example. A video frame is regarded as a unit of data sample in integer rate dataflow graphs, and should be broken down into multiple macroblocks explicitly by consuming extra memory space.

To overcome this difficulty, the fractional rate dataflow (FRDF) model in which a fractional number of samples can be produced or consumed has been proposed [21]. In FRDF, a fraction number can be used as a sample rate as shown in Fig. 11b where the input sample rates of node frME is set to $\frac{1}{99}$. The fractional number means that the input data type of node frME is a macroblock and it corresponds to $\frac{1}{99}$ of a frame data.

Figure 12 shows an FRDF graph where the data type of arc BC is a composite type as illustrated in the figure and the data type of arc AC is a primitive type such as integer or float. A fractional sample rate has different meaning for a composite data type from a primitive type. For a composite data type, the fractional sample rate really indicates the partial production or consumption of the sample. In the example graph, one firing of node B fills $\frac{1}{3}$ of a sample on arc BC and node C reads the first half of the sample at every $(2n + 1)$-th firing and the second half at every $2n$-th firing. Hence, if we consider the execution order of nodes B and C, a schedule "BBCBC" is valid since $\frac{2}{3}$ of a sample is available after node B is fired twice and node C becomes fireable.

For primitive types, partial production or consumption is not feasible. Then statistical interpretation is applied for a fractional rate. In the example graph, the output sample rate of node A is $\frac{1}{3}$ on arc AC. This means that node A produces a single sample every three executions. Similarly node C consumes one sample every two executions. Note that a fractional rate does not imply at which firings samples are produced. So node C becomes fireable only after node A is executed three times. If we are concerned about the execution order of nodes A and C only, schedule "3A2C" is valid while "2ACAC" is not. Consequently, a valid schedule for the FRDF graph of Fig. 12 is "3(AB)2C".

Regardless of the data type, a fractional sample rate $\frac{p}{q}$ guarantees that p samples are produced or consumed after q firings of the node. Similar to the CSDF graph, we can construct an equivalent SDF graph by merging q firings of an FRDF node into

an equivalent SDF node that produces or consumes p samples per firing. Therefore, static analysis techniques for the SDF model can be applied to the FRDF model. For the analysis of sample rate consistency, however, we can use fractional sample rates directly in the topology matrix. For the FRDF graph of Fig. 12, the topology matrix and repetition vector are:

$$\Gamma = \begin{pmatrix} 1 & -1 & 0 \\ 0 & \frac{1}{3} & -\frac{1}{2} \\ \frac{1}{3} & 0 & -\frac{1}{2} \end{pmatrix}$$

$$\mathbf{q_G} = (3, 3, 2)$$

Since the rank of Γ is 2, the graph is sample rate consistent. Deadlock can be detected by constructing a static schedule similarly to the SDF case. If there exists a valid static schedule, the graph is free from deadlock. A static schedule is simply constructed by inserting a fireable node into the schedule list and simulating its firing. An FRDF node has different firing conditions depending on the data types of input ports. An FRDF node is fireable, or executable, if all input arcs satisfy the following condition depending on the data type:

1. If the data type is primitive, there must be at least as many stored samples as the numerator value of the fractional sample rate. An integer sample rate is a special fractional rate whose denominator is 1.
2. If the data type is composite, there must be at least as large a fraction of samples stored as the fractional input sample rate.

Special care should be taken for a composite type data. If the consumer and the producer have a different interpretation on the fraction, then a composite type should be regarded as atomic like a primitive type when the firing condition is examined. Suppose that for a composite data type of a two-dimensional array, the producer regards it as an array of row vectors while the consumer regards it as an array of column vectors as shown in Fig. 13. In this case, the two-dimensional array may not be regarded as a composite type data. Therefore, schedule "DDUDU" is not valid while "3D 2U" is.

In general, the FRDF model results in an efficient implementation of a multimedia application in terms of buffer requirement and performance. Consider the example in Fig. 11b again. Since node frME uses a macroblock-size sample, the output arc requires only a single macroblock-size buffer. For each arrival of an input video frame, node frME is fired 99 times and consumes a macro-block size portion

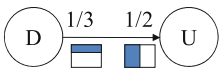


Fig. 13 If the consumer and the producer have the different access patterns for a composite type data then the type should be treated as atomic

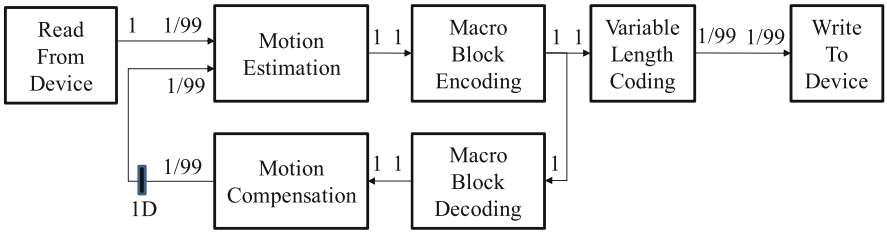
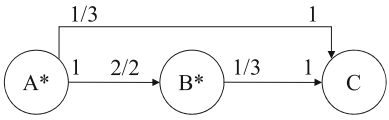


Fig. 14 H.263 encoder in FRDF

Fig. 15 An FRDF graph corresponding to Fig. 9



of the input frame per firing. Since node EN can be fired after each firing of node frME, shorter latency is experienced when compared with the SDF implementation. Figure 14 shows a whole H.263 encoding algorithm in FRDF where the sample types for "current frame" and "previous frame" are video frames. It is worth noting that the entire previous frame is needed to do motion estimation for each macroblock of the current frame while the sample rate of the bottom input port of node "Motion Estimation" is $\frac{1}{99}$. Hence, even though the previous frame is a composite data type, it should be regarded as atomic. Then node "Motion Estimation" is fireable only after the entire previous frame is available.

Figure 15 shows an FRDF graph that corresponds with Fig. 9a. Both have the same equivalent SDF graphs. Similar to the CSDF model, the FRDF model can reduce the buffer size when compared with the corresponding SDF model. Since node A* produces a single sample and B* consumes two samples every other execution, a valid schedule for the graph is "2A* 2B* 2A* B* C B* 2A* 2B* C". And the required buffer sizes on arcs A*B* and B*C* are equal to the sizes for the CSDF graph. The buffer size for arc A*C is, however, larger than the CSDF graph since the FRDF model does not know when samples are produced and consumed, and the schedule for the FRDF model should consider the worst case behavior. For an output port, the worst case is when output samples are all produced at the last phase while it is when input samples are all consumed at the first phase for an input port. Therefore, in the FRDF model, rate $\frac{p}{q}$ for a primitive data type corresponds to "$\{(q-1)\times 0$,p$\}$" for an output sample rate and "$\{$p,$(q-1)\times 0\}$" for an input sample rate in the CSDF model, where "$(q-1)\times 0$" denotes "$\underbrace{0, 0, \cdots, 0}_{q-1}$". Hence, the CSDF model may generate better schedules than the associated FRDF model since we can split the node execution into finer granularity of phases at compile-time scheduling; This is not possible in the FRDF if the date type is primitive. The expression capability of two models is, however, different. The CSDF model allows only a periodic pattern to express sample rate variations while the FRDF model

has no such restriction as long as the average value is constant during a period. So the FRDF model has more expression power than the CSDF model since it allows dynamic behavior of an FRDF node within a period and the periodic pattern can be regarded as a special case.

4.2 SPDF (Synchronous Piggybacked Dataflow)

The SDF model does not allow communication through a shared global variable since the access order to the global variable can vary depending on the execution order of nodes. Suppose a source block produces the frame header in a frame-based signal processing application that is to be used by several downstream blocks. A natural way of coding in C is to define a shared data structure that the downstream blocks access by pointers. But in a dataflow model, such sharing is not allowed. As a result, redundant copies of data samples are usually introduced in the automatically generated code from the dataflow model. Such overhead is usually not tolerable for embedded systems with tight resource and/or timing constraints. To overcome this limitation, an extended SDF model, called SPDF(Synchronous Piggybacked Dataflow) is proposed [23], by introducing the notion of "controlled global states" and by coupling a data sample with a pointer to the controlled global state.

The Synchronous Piggybacked Dataflow (SPDF) model was first proposed to support frame-based processing, or block-based processing, that frequently occurs in multimedia applications. In frame-based processing, the system receives input streams of frames that consist of a frame header and data samples. The frame header contains information on how to process data samples. So an intuitive implementation is to store the information in a global data structure, called *global states*, and the data processing blocks refer to the global states before processing the data samples.

Figure 16 shows a simple SPDF graph where node A reads a frame from a file and produces the header information and the data samples through different output ports. Suppose that a frame consists of 100 data samples in this example.

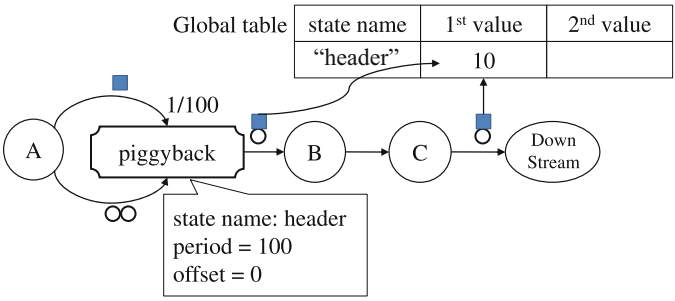


Fig. 16 An example SPDF graph that shows a typical frame-based processing: The Piggyback block writes the header information to the global state and the downstream blocks refer to the global state before processing the data samples

The header information and the data samples are both connected to a special FRDF(Fractional Rate Dataflow) block, called *"Piggyback"* block, that has three parameters: *"statename"*, *"period"*, and *"offset"*. The Piggyback block updates the global state of *"statename"* with the received header information periodically with the specified *"period"* parameter. It piggybacks a pointer to the global state on each input data sample before sending it through the output port. Since it needs to receive the header information in order to update the global state only once per 100 executions, the sample rate of the input port associated with the header information is set to the fractional value $\frac{1}{100}$, which means that it consumes one sample per 100 executions in the FRDF model. The input port of this fractional rate is called the *"state port"* of the Piggyback block. The sample rate of the data port, on the other hand, is unity.

The *"offset"* parameter indicates when to update the global state. The Piggyback block receives as many data samples as the *"offset"* value before updating the global state. In this example, the *"offset"* value is set to its default value, zero, which makes the Piggyback block consume the header information and update the global state before it piggybacks the data samples with a pointer to the global state.

Note that the Piggyback block with a fractional rate input port is the only extension to the SDF model. Since the sample rates of the SPDF graph are all static, the static analyzability of the SDF model is preserved even after addition of the Piggyback block. Also, piggybacking of data samples with pointers can be performed without any run-time overhead by utilizing the static schedule information of the graph. Suppose that the SDF graph in Fig. 16 has the following static schedule: "A 100(Piggyback, B, C, DownStream)". Then the pseudo code of the automatically generated code associated with the schedule is as follows:

```
code block of A
for (int i = 0; i < 100, i++) {
  if (i == offset_Piggyback)
    update the global state header
  code block of B
  code block of C
  if (i == offset_Piggyback)
    update the local state of DownSteam block
      from the global state information
  code block of DownStream
}
```

Figure 17 shows another example that produces a sinusoidal waveform with varying amplitude at run-time. The "Singen" block generates a sinusoidal waveform (N samples per period) of unit amplitude and the "Gain" block amplifies the input samples by the "gain" parameter of the block. To control the amplitude, the graph uses a Piggyback block after the "Singen" block. Another source block, "Ramp", is connected to the state port of the Piggyback block. The "statename" of the Piggyback is named "global_gain" and the "gain" parameter of the "Gain" block is also set to "global_gain". Then, the "gain" parameter of the "Gain" block is updated with a global state named by "global_gain" whose value is determined by

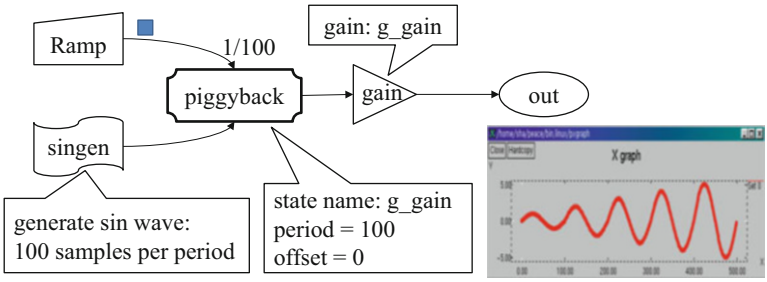


Fig. 17 An SPDF graph that produces a sinusoidal waveform with varying amplitude at run-time: the "gain" state of the "Gain" block is updated by the "Ramp" block through a global state, "global_gain"

the "Ramp" block. In this example, the period of the Piggyback block is set to N so that the amplitude of the sinusoidal waveform is incremented by one every period as shown in Fig. 17. If we insert two initial samples on the input arc of the "Gain" block, the "offset" parameter of the Piggyback block should be 2.

Thus the SPDF model provides a safe mechanism to deliver state values through shared memory instead of message passing. Communication through shared memory is prohibited in conventional dataflow models since the access order to the shared memory may vary depending on the schedule. But the SPDF model gives another solution by observing that the side effect is caused by an implicit assumption that the global state is assigned a memory location before scheduling is performed. The SPDF model changes the order: allocate the memory for the global state after the schedule is made. Since the scheduling decision is made at compile-time, we know the access order to the variable and the life time of each global state variable. Suppose that the schedule of Fig. 17 becomes "2(100(Singen) Ramp Piggyback) 200(Gain Display)". From the static schedule, we know that we need to maintain two memory locations for the global state, "global_gain" since the Piggyback block writes the second value to the global state before the "Gain" reads the first global state.

Allowing shared memory communication without side effects gives a couple of significant benefits over conventional dataflow models. First, it can remove the unnecessary overhead of data copying of message passing communication since the global state can be shared by multiple blocks. Second, it greatly enhances the expression capability of the SDF model without breaking the static analyzability. It provides an explicit mechanism of affecting the internal behavior of a block from the outside through global states.

4.3 SSDF (Scalable SDF)

DSP architectures have stringent on-chip memory limits and off-chip memory access is costly. They also allow vector processing of instructions and arithmetic pipelining like MAC in order to attain peak performance when the pipelining is fully utilized. Therefore, when an SDF block operates on primitive-type data and the granularity is small, it behaves inefficiently. For example, an adder block performs a single accumulation operation by reading two samples from memory and writing a sample into memory. It requires large run time overhead of off-chip memory access for two read and one write operations. In order to achieve efficient implementation, the scalable synchronous dataflow (SSDF) model is proposed [25]. The SSDF model has the same semantics as the SDF model except that a node may consume or produce any integer multiple of the fixed rates per invocation. The actual multiple, called *blocking factor*, is determined by a scheduler or an optimizer.

Figure 18a shows the code of an "Add" block in SDF. In SSDF, the code includes blocking factor N_b that is the number of iterations as shown in Fig. 18b. Since the function call overhead of "add()" is larger than the accumulation operation, the SSDF model amortizes the function call overhead by performing N_b accumulations per function call. When blocking factor N_b is 1, the SSDF graph degenerates to an SDF graph. Therefore, the SSDF model has the same sample rates as the SDF model and sample rate inconsistency can be checked using the topology matrix for the SDF model. Moreover, deadlock is detected by constructing a schedule by setting block factor $N_b = 1$. From the static analysis of the degenerated SDF graph, repetition vector q_G can be obtained, assuming N_b is 1 for all blocks. When $N_b > 1$, the repetition vector for SSDF becomes $N_b q_G$.

A straightforward scheduling technique for an SSDF graph is to increase the minimal scheduling period by an integer factor N_g where N_g is a global blocking factor. Each node A of the graph will be invoked $N_g x(A)$ times within one scheduling period, where $x(A)$ is the repetition count of node A. Increasing N_g reduces the function call overhead but requires larger buffer memory for graph execution. For instance, the "Add" node in Fig. 18 consumes N_b samples from each input port and produces N_b output samples, then all three buffers have size N_b while they have size 1 when the blocking factor is unity. Moreover, the increment of N_g delays the response time although it does not decrease the throughput.

a

```
void add()
{
   *addOut = *addIn1 + *addIn2;
}
```

b

```
void add()
{
   for(int i=0; i<Nb; i++)
     *addOut++ = *addIn1++ + *addIn2++;
}
```

Fig. 18 Code of an "Add" actor (**a**) in SDF and (**b**) in SSDF where N_b is the blocking factor

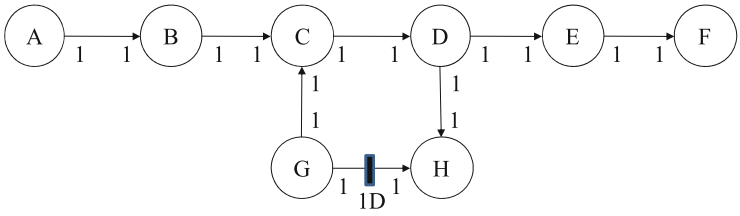


Fig. 19 A graph with feedback loop

Another major obstacle to increase the blocking factor is related with feedback loops. Vector processing is restricted to the number of initial delays on the feedback loop. If the number is smaller than N_g, the vector processing capability cannot be fully utilized. For example, the scheduling result for a graph shown in Fig. 19 is "A B G C D H E F" when the blocking factor is 1. If blocking factor N_g becomes 5 then the scheduling becomes "5A 5B 5(GCDH) 5E 5F" in which nodes G,C,D and H are repeated five times sequentially. Therefore, a scheduling algorithm for SSDF should consider the graph topology to minimize the program code size.

In case feedback loops exist, strongly-connected components are first clustered into a strong component. A *strong component* of graph G is defined as a subgraph $F \subset G$ if for all pairs of nodes $u, v \in F$ there exist paths p_{uv}(from u to v) and p_{vu}(from v to u). This clustering is performed in a hierarchical fashion until the top graph does not have any feedback loop. Then, a valid schedule for an SSDF graph can be constructed using the SDF scheduling algorithms. Each node is scheduled by applying the global blocking factor N_g. For the SSDF graph in Fig. 19, the top graph consists of five nodes "A B (CDGH) E F" where nodes C, D, G and H are merged into a clustered-node. When blocking factor N_g is set to 5, a schedule for the top graph becomes "5A5B5(clustered-node)5E5F".

Next, the strong components are scheduled. The blocking factor depends on the number of initial delay samples on a feedback loop. Let $N_l(L)$ denote the maximum bound of the blocking factor on feedback loop L. Since feedback loops can be nested, a feedback loop with the largest maximum bound $N_l(L)$ should be selected first. Subsequently, feedback loops are selected in a descending order of $N_l(L)$. Scheduling of the clustered subgraph starts with a node that has many initial delay samples on its input ports and allows a large blocking factor. When a strong component "(CDHG)" is scheduled in the SSDF graph, actor G should be fired since it has an initial delay sample.

For a selected strong component, we schedule the internal nodes as follows, depending on the number of delays on the feedback loop.

Case 1: N_g **is an integer multiple of** $N_l(L)$. The scheduling order is repeated $N_g/N_l(L)$ times using $N_b = N_l(L)$ for the internal nodes. In the example of Fig. 19, since $N_l(L) = 1$, $N_g = 5$, and $N_g/N_l(L)$ is an integer, schedule of "GCDH" is repeated five times. Moreover, the blocking factor for each node N_b is 1. Hence, the final schedule is "5A 5B 5(GCDH) 5E 5F".

Case 2: $N_g \leq N_l(L)$. Blocking factor $N_b = N_g$ is applied for all actors in the strong component. For example, if the number of delay samples increases to 5 in Fig. 19, then blocking factor $N_l(L)$ is 5 which is equal to N_g, and the schedule becomes "5A 5B (5G 5C 5D 5H) 5E 5F". Therefore, the blocking factor can be fully utilized.

Case 3: If $N_g > N_l(L)$ but not an integer multiple. One of two scheduling strategies can be applied:

1. The schedule for the strong component is repeated N_g times using $N_b = 1$ internally, which produces the smallest code at the cost of throughput.
2. The schedule is repeated with blocking factor $N_b = N_l(L)$, and then once more for the remainder to N_g. This improves throughput but also enlarges the code size.

When $N_l(L) = 2$ by increasing the number of delay samples to 2, a valid schedule is "5(GCDH)" if the first strategy is followed or "2(2G 2C 2D 2H) GCDH" if the second strategy is followed. Consequently, the final schedule is either "5A5B 5(GCDH) 5E 5F" or "5A 5B 2(2G 2C 2D 2H) GCDH 5E 5F".

Although the SSDF model is proposed to allow large blocking factors to utilize vector processing of simple operations in an node, the scheduling algorithm for SSDF is also applicable to an SDF graph in which every node has an inline style code specification. Without the modification of the SDF actor, the blocking factor can be applied to the SDF graph and the SDF schedule. For instance, when block factor $N_g = 3$ is applied to Fig. 2, a valid schedule is "9A 9D 6B 12C". For the given schedule with the blocking factor, programs can be synthesized as shown in Fig. 5 where each loop value in the codes will be multiplied by blocking factor N_g (=3).

References

1. Ade, M., Lauwereins, R., Peperstraete, J.A.: Implementing dsp applications on heterogeneous targets using minimal size data buffers. In: Proceedings of RSP'96, pp. 166–172 (1996)
2. Bamakhrama, M., Stefanov, T.: Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In: Proceedings of the Ninth ACM International Conference on Embedded Software, EMSOFT '11, pp. 195–204. ACM, New York, NY, USA (2011). http://doi.acm.org/10.1145/2038642.2038672
3. Bhattacharyya, S.S., Murthy, P.K., Lee, E.A.: Software Synthesis from Dataflow Graphs. Kluwer Academic Publisher, Norwell MA (1996)
4. Bhattachayya, S.S., Murthy, P.K., Lee, E.A.: Apgan and rpmc: Complementary heuristics for translating dsp block diagrams into efficient software implementations. In: Journal of Design Automation for Embedded Systems, vol. 2, pp. 33–60 (1997)
5. Bilsen, G., Engles, M., Lauwereins, R., Peperstraete, J.A.: Cyclo-static dataflow. In: IEEE Trans. Signal Processing, vol. 44, pp. 397–408 (1996)

6. Bodin, B., Kordon, A.M., de Dinechin, B.D.: Periodic schedules for cyclo-static dataflow. In: The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia, Montreal, QC, Canada, October 3–4, 2013, pp. 105–114 (2013). http://dx.doi.org/10.1109/ESTIMedia.2013.6704509
7. Buck, J.T., Ha, S., Lee, E.A., Messerschimitt, D.G.: Ptolemy: A framework for simulating and prototyping heterogeneous systems. In: Int. Journal of Computer Simulation, special issue on Simulation Software Development, vol. 4, pp. 155–182 (1994)
8. Dennis, J.B.: Dataflow supercomputers. In: IEEE Computer Magazine, vol. 13 (1980)
9. Govindarajan, R., Gao, G., Desai, P.: Minimizing memory requirements in rate-optimal schedules. In: Proceedings of the International Conference on Application Specific Array Processors, pp. 75–86 (1993)
10. Graham, R.L.: Bounds on multiprocessing timing anomalies. In: SIAM Journal on Applied Mathematics, vol. 17, pp. 416–429 (1969)
11. de Groote, R.: Throughput analysis of dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
12. Hoang, P.D., Rabaey, J.M.: Scheduling of dsp programs onto multiprocessors for maximum throughput. In: IEEE Transactions on Signal Processing, pp. 2225–2235 (1993)
13. Jung, H., Yang, H., Ha, S.: Optimized rtl code generation from coarse-grain dataflow specification for fast hw/sw cosynthesis. In: Journal of Signal Processing Systems, vol. 52, pp. 13–34 (2008)
14. Kang, S.h., Kang, D., Yang, H., Ha, S.: Real-time co-scheduling of multiple dataflow graphs on multi-processor systems. In: Proceedings of the 53rd Annual Design Automation Conference, DAC '16, pp. 159:1–159:6. ACM, New York, NY, USA (2016). http://doi.acm.org/10.1145/2897937.2898077
15. Kermia, O., Sorel, Y.: A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor. In: Proceedings of the ISCA 20th International Conference on Parallel and Distributed Computing Systems, September 24–26, 2007, Las Vegas, Nevada, USA, pp. 1–6 (2007)
16. Kim, J., Shin, T., Ha, S., Oh, H.: Resource minimized static mapping and dynamic scheduling of sdf graphs. In: ESTIMedia (2011)
17. Lauwereins, R., Engels, M., Peperstraete, J.A., Steegmans, E., Ginderdeuren, J.V.: Grape: A case tool for digital signal parallel processing. In: IEEE ASSP Magazine, vol. 7, pp. 32–43 (1990)
18. Lee, E.A., Ha, S.: Scheduling strategies for multiprocessor real-time DSP. In: GLOBECOM '89: IEEE Global Telecommunications Conference and Exhibition. Communications Technology for the 1990s and Beyond, vol. 2, pp. 1279–1283. IEEE, Los Alamitos, CA, USA (1989). http://dx.doi.org/10.1109/GLOCOM.1989.64160
19. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous dataflow programs for digital signal processing. In: IEEE Transaction on Computer, vol. C-36, pp. 24–35 (1987)
20. Oh, H., Ha, S.: Memory-optimized software synthesis from dataflow program graphs with large size data samples. In: EURASIP Journal on Applied Signal Processing, vol. 2003, pp. 514–529 (2003)
21. Oh, H., Ha, S.: Fractional rate dataflow model for memory efficient synthesis. In: Journal of VLSI Signal Processing, vol. 37, pp. 41–51 (2004)
22. Parhi, K.K., Chen, Y.: Signal flow graphs and data flow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2012)
23. Park, C., Chung, J., Ha, S.: Extended synchronous dataflow for efficient dsp system prototyping. In: Design Automation for Embedded Systems, vol. 3, pp. 295–322. Kluwer Academic Publishers (2002)
24. Pino, J., Ha, S., Lee, E.A., Buck, J.T.: Software synthesis for dsp using ptolemy. In: Journal of VLSI Signal Processing, vol. 9, pp. 7–21 (1995)
25. Ritz, S., Pankert, M., Meyr, H.: High level software synthesis for signal processing systems. In: Proceedings of the International Conference on Application Specific Array Processors (1992)

26. Ritz, S., Willems, M., Meyr, H.: Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In: Proceedings of the ICASSP 95 (1995)
27. Spasic, J., Liu, D., Cannella, E., Stefanov, T.: Improved hard real-time scheduling of csdf-modeled streaming applications. In: Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis, CODES '15, pp. 65–74. IEEE Press, Piscataway, NJ, USA (2015). http://dl.acm.org/citation.cfm?id=2830840.2830848
28. Stuijk, S., Basten, T., Geilen, M.C.W., Coporaal, H.: Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In: DAC, pp. 777–782 (2007)
29. Stuijk, S., Geilen, M.C.W., Basten, T.: Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In: DAC, pp. 899–904 (2006)
30. Sung, W., Ha, S.: Memory efficient software synthesis using mixed coding style from dataflow graph. In: IEEE Transaction on VLSI Systems, vol. 8, pp. 522–526 (2000)
31. Woods, R.: Mapping decidable signal processing graphs into FPGA implementations. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2012)
32. Yang, H., Ha, S.: Pipelined data parallel task mapping/scheduling technique for mpsoc. In: DATE (Design Automation and Test in Europe) (2009)

Systolic Arrays

Yu Hen Hu and Sun-Yuan Kung

Abstract This chapter reviews the basic ideas of systolic array, its design methodologies, and historical development of various hardware implementations. Two modern applications, namely, motion estimation of video coding and wireless communication baseband processing are reviewed. The application to accelerating deep neural networks is also discussed.

1 Introduction

Systolic array [2, 13, 15] is an on-chip multi-processor architecture proposed by Kung in late 1970s. It is proposed as an architectural solution to the anticipated on-chip communication bottleneck of modern very large scale integration (VLSI) technology. A systolic array features a mesh-connected array of identical, simple processing elements (PE). According to Kung [13], "*In a systolic system, data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory, much as blood circulates to and from the heart.*" As depicted in Fig. 1, a systolic array is often configured into a linear array, a two-dimensional rectangular mesh array, or sometimes, a two dimensional hexagonal mesh array. In a systolic array, every PE is connected only to its nearest neighboring PEs through dedicated, buffered local bus. This localized interconnects, and regular array configuration allow a systolic array to grow in size without incurring excessive on-chip global interconnect delays due to long wires.

Several key architectural concerns impacted on the development of systolic architecture [13]:

Y. H. Hu (✉)
University of Wisconsin - Madison, Department of Electrical and Computer Engineering, Madison, WI, USA
e-mail: yuhen.hu@wisc.edu

S.-Y. Kung
Princeton University, Department of Electrical Engineering, Princeton, NJ, USA
e-mail: kung@princeton.edu

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_26

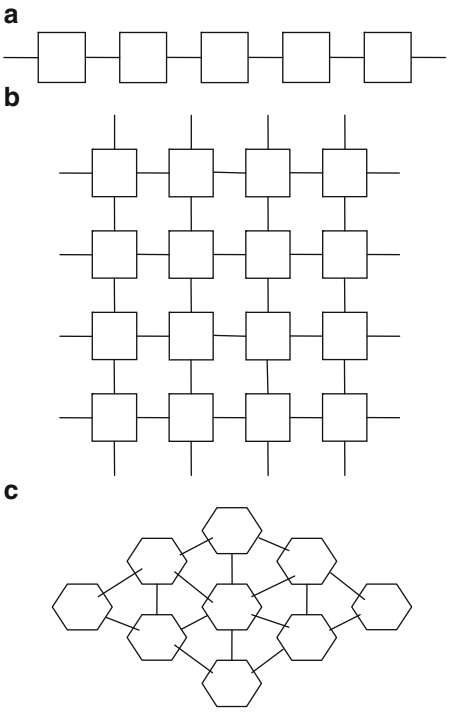


Fig. 1 Common configurations of systolic architecture: (**a**) linear array, (**b**) rectangular array, (**c**) hexagonal array

1. *Simple and regular design*—In order to reduce design complexity, design cost, and to improve testability, fault-tolerance, it is argued that VLSI architecture should consist of simple modules (cores, PEs, etc) organized in regular arrays.
2. *Concurrency and communication*—Concurrent computing is essential to achieve high performance while conserving power. On-chip communication must be constrained to be local and regular to minimize excessive overhead due to long wire, long delay and high power consumption.
3. *Balanced on-chip computation rate and on/off chip data input/output rate*—moving data on/off chip remains to be a communication bottleneck of modern VLSI chips. A sensible architecture must balance the demand of on/off chip data I/O to maximize the utilization of the available computing resources.

Systolic array is proposed to implement *application specific* computing systems. Toward this goal, one must *map* the computing algorithm to a systolic array. This requirement stimulated two complementary research directions that have seen numerous significant and fruitful research results. The first research direction is to reformulate existing computing algorithms, or develop novel computing algorithms that can be mapped onto a systolic architecture to enjoy the benefit of systolic computing. The second research direction is to develop a systematic design methodology that would automate the process of algorithm mapping. In Sect. 2 of this chapter, we will provide a brief overview of these *systolic algorithms* that have been proposed. In Sect. 3, the formal design methodologies developed for automated systolic array mappings will be reviewed.

Systolic array computing was developed based on a globally synchronized, fine-grained, pipelined timing model. It requires a global clock distribution network free of clock skew to distribute the clock signal over the entire systolic array. Recognizing the technical challenge of developing large scale clock distribution network, Kung et al. [14–16] proposed a self-timed, data flow based *wavefront array processor architecture* that promises to alleviate the stringent timing constraint imposed by the global clock synchronization requirement. In Sect. 4, the wavefront array architecture and its related design methodology will be discussed.

These architectural features of systolic array have motivated numerous developments of research and commercial computing architectures. Notable examples include the WARP and iWARP project at CMU [1, 3, 7, 10]; Transputer™of INMOS [8, 20, 26, 30]; and TMS 32040 DSP processor of Texas Instruments [27]. In Sect. 5 of this chapter, brief reviews of these systolic-array motivated computing architectures will be surveyed.

While the notion of systolic array was first proposed three decades ago, its impacts can be felt vividly today. Modern applications of the concept of systolic array can be found in field programmable gate array (FPGA) chip architectures, network-on-chip (NoC) mesh array multi-core architecture. Computation intensive special purpose architecture such as discrete cosine transform and block motion estimation algorithms in video coding standards, as well as the QR factorization for least square filtering in wireless communication standards have been incorporated in embedded chip designs. These latest real world applications of systolic array architecture will be discussed in Sect. 6.

2 Systolic Array Computing Algorithms

A systolic array exhibits characteristics of parallelism (in the form of fine-grained pipelining), regularity, and local communication. A large number of signal processing algorithms, and numerical linear algebra algorithms can be implemented using systolic arrays.

2.1 Convolution Systolic Array

For example, consider a convolution of two sequences $\{x[n]\}$ and $\{h[n]\}$:

$$y[n] = \sum_{k=0}^{K-1} h[k]x[n-k], \quad 0 \le n \le N-1. \tag{1}$$

A systolic array realization of this algorithm can be shown in Fig. 2 ($K = 4$). In Fig. 2a, the block diagram of the systolic array and the pattern of data movement are

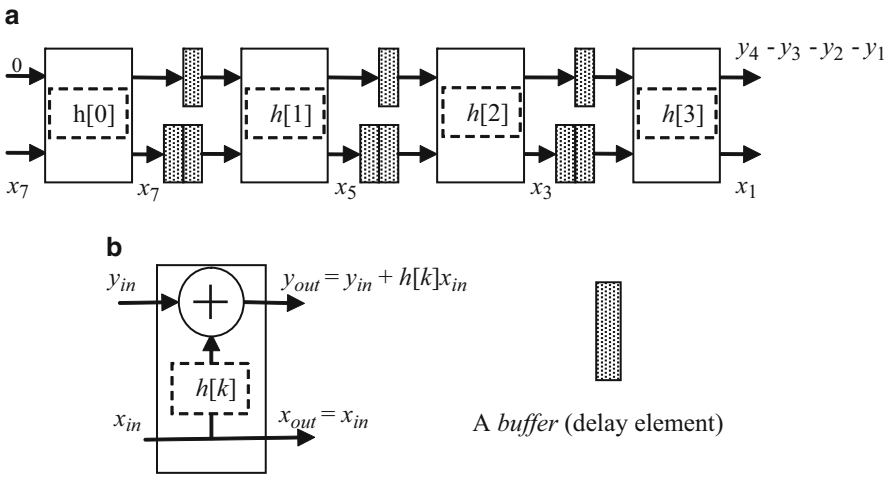


Fig. 2 (**a**) Fully pipelined systolic array for convolution, (**b**) internal architecture of a single processing element (PE)

depicted. The block diagram of an individual processing element (PE) is illustrated in Fig. 2b where a shaded rectangle represents a buffer (delay element) that can be implemented with a register. The output $y[n]$ begins its evaluation at the upper left input with initial value 0. When it enters into each PE, the multiply-and-accumulate (MAC) operation

$$y_{out} = y_{in} + h[k]x_{in} \tag{2}$$

will be performed. The systolic array is of the same length as the sequence $\{h[k]; 0 \leq k \leq K-1\}$ with each $h[k]$ resides in a register in each PE. The final result $\{y[n]\}$ appears at the upper right output port. Every other clock cycle, one output will be evaluated. The input $\{x[n]\}$ will be provided from the lower left input port. It will propagate toward the lower right output port without being modified ($x_{out} = x_{in}$). Along the way, it will be properly buffered to keep pace of the evaluation of the convolution.

2.2 Linear System Solver Systolic Array

Similar to above example, a systolic algorithm is often presented in the form of a high level block diagram of the systolic array configuration (e.g. Fig. 1) complemented with labels indicating data movement within the processor array, and a detailed block diagram explaining the operations performed within an individual PE.

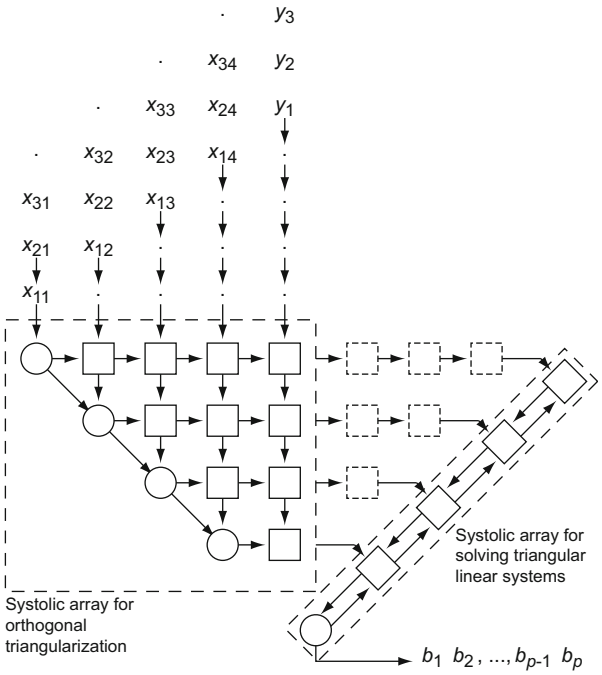


Fig. 3 Systolic array for solving linear systems [13]

Another example given in [13] is shown in Fig. 3. It consists of two systolic arrays for solving linear systems of equations. One triangular-configured systolic array is responsible for orthogonal triangulation of a matrix using QR factorization, and the other linear systolic array is responsible for solving a triangular linear system using back-substitution. A linear system of equations is represented as

$$\mathbf{Xb} = \mathbf{y}.$$

Using a Jacobi's rotation method, the first column of the X matrix will enter the upper-left circular PE where an angle θ_k is evaluated such that

$$\begin{bmatrix} \cos\theta_k & -\sin\theta_k \\ \sin\theta_k & \cos\theta_k \end{bmatrix} \begin{bmatrix} x_{11}^{(k-1)} \\ x_{k,1} \end{bmatrix} = \begin{bmatrix} x_{11}^{(k)} \\ 0 \end{bmatrix}, \quad k = 2, 3, \ldots \tag{3}$$

Clearly, in this circular PE, the operation to be performed will be

$$\theta_k = -\tan^{-1}\left(x_{k,1}/x_{11}^{(k-1)}\right). \tag{4}$$

This θ_k then will be propagated to the square PEs to the right of the upper left circular PE to perform rotation operations

$$\begin{bmatrix} \cos\theta_k & -\sin\theta_k \\ \sin\theta_k & \cos\theta_k \end{bmatrix} \begin{bmatrix} x_{12}^{(k-1)} & \dots & x_{1N}^{(k-1)} & y_1^{(k-1)} \\ x_{k2}^{(k-1)} & \dots & x_{kN}^{(k-1)} & y_k^{(k-1)} \end{bmatrix} = \begin{bmatrix} x_{12}^{(k)} & \dots & x_{1N}^{(k)} & y_1^{(k)} \\ x_{k2}^{(k)} & \dots & x_{kN}^{(k)} & y_k^{(k)} \end{bmatrix}. \tag{5}$$

The second row of the results of above equation will propagated downward to the next row in the triangular systolic array repeating what has been performed on the first row of that array. After $N-1$ iterations, the results will be ready within the triangular array. Note that during this rotation process, the right hand size of the linear systems of equations $\mathbf{y}$ is also subject to the same rotation operation. Equivalently, these operations taken places at the triangular systolic array amount to pre-multiply the $\mathbf{X}$ matrix with a unitary matrix $\mathbf{Q}$ such that $\mathbf{QX} = \mathbf{U}$ is an upper triangular matrix, and $\mathbf{z} = \mathbf{Qy}$. This yields an upper triangular system $\mathbf{Ub} = \mathbf{z}$.

To solve this triangular system of equations, a *back-propagation* solver algorithm is used. Specifically, given

$$\mathbf{Ub} = \begin{bmatrix} u_{11} & u_{11} & \dots & u_{1N} \\ 0 & u_{22} & & u_{2N} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & u_{NN} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_N \end{bmatrix} = \mathbf{Z}. \tag{6}$$

The algorithm begins by solving $u_{NN}b_N = z_N$ for $b_N = z_N/u_{NN}$. In the systolic array, u_{NN} are fed from the last (lower right corner) circular PE to the circular PE of the linear array to the right. The computed b_N then will be forwarded to the next square processor in the upper right direction to be substituted back into the next equation of

$$u_{N-1,N-1}b_{N-1} + u_{N-1,N}b_N = z_{N-1}. \tag{7}$$

In the rectangular PE, the operation performed will be $z_{N-1} - u_{N-1,N}b_N$. The result then will be fed back to the circular PE to compute $b_{N-1} = (z_{N-1} - u_{N-1,N}b_N)/u_{N-1,N-1}$.

2.3 Sorting Systolic Arrays

Given a sequence $\{x[n];\ 0 \le n \le N-1\}$, the sorting algorithm will output a sequence $\{m[n];\ 0 \le n \le N-1\}$ that is a permutation of the ordering of $\{x[n]\}$ such that $m[n] \ge m[n+1]$. There are many sorting algorithms available. A systolic array that implements the bubble sort algorithm is presented in Fig. 4.

Each PE in this systolic array will receive data a, b from both left and right sides. These two inputs will be compared and the maximum of the two will be output to

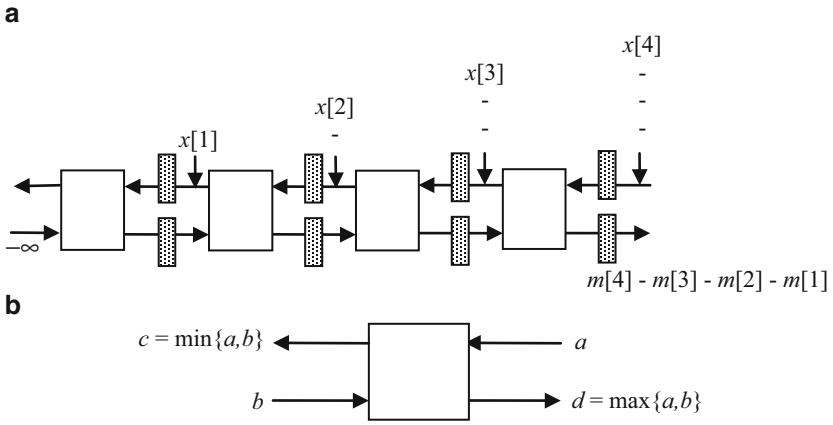


Fig. 4 (**a**) A bubble sort systolic array, (**b**) operation performed within a PE

the right side buffer while the minimum of the two output to the left side buffer. The input will be loaded into the upper buffer according to specific schedule. The left most input will be fixed at $-\infty$. It has been shown that systolic arrays of insertion sort and selection sort can also be derived using similar approach [15].

3 Formal Systolic Array Design Methodology

Due to the regular structure, localized interconnect, and pipelined operations, a formal systolic array design methodology has been proposed that greatly simplified the systolic array design complexity and opened new avenue to seek optimized systolic architecture. In order to introduce the formal systolic array design methodology in this section, a few important representations will be briefly surveyed.

3.1 Loop Representation, Regular Iterative Algorithm (RIA), and Index Space

Algorithms that are suitable for systolic array implementation must exhibit high degree of regularity, and require intensive computation. Such an algorithm often can be represented by a set of nested Do loops of the following general format:

$$
\begin{array}{ll}
L_1: & \text{DO } i_1 = p_1, q_1 \\
L_2: & \quad \text{DO } i_2 = p_2, q_2 \\
\vdots & \quad\quad \vdots \\
L_m: & \quad\quad \text{DO } i_m = p_m, q_m
\end{array}
$$

```
        H(i1, i2, ..., im)
      End  do
      ⋮
    End do
  Enddo
```

where $\{L_m\}$ specify the *level* of the loop nest, $\{i_m\}$ are *loop indices*, and $\mathbf{i} = [i_1, i_2, \cdots, i_m]^T$ is a $m \times 1$ *index vector*, representing an index point in a m-dimensional lattice. $\{p_m, q_m\}$ are *loop bounds* of the mth loop nest. $\mathrm{H}(i_1, i_2, \ldots, i_m)$ is the *loop body* and may have different *granularity*. That is, the loop body could represent bit-level operations, word-level program statements, or sub-routine level procedures. Whatever the granularity is, it is assumed that the loop body is to be executed in a single PE. For convenience, the execution time of a loop body in a PE will be assumed to be one clock cycle in this chapter. In other words, it is assumed that the execution of a loop body within a PE cannot be interrupted. All data needed to execute the loop body must be available before the execution of loop body can start; and none of the output will be available until the execution of the entire loop body is completed.

If the loop bounds are all constant, the set of indices corresponding to all iterations form a rectangular parallelepiped. In general, the loop bounds are linear (affine) function with integer coefficients of outer loop indices and can be represented with two inequalities:

$$\mathbf{p}_0 \leq \mathbf{Pi} \quad \text{and} \quad \mathbf{Pi} \leq \mathbf{q_0}, \tag{8}$$

where $\mathbf{p}_0$ and $\mathbf{q}_0$ are constant integer-valued vectors, and $\mathbf{P}$, $\mathbf{Q}$ respectively, are integer-valued upper triangular coefficient matrices. If $\mathbf{P} = \mathbf{Q}$, then the corresponding loop nest can be *transformed* in the index space such that the transformed algorithm has constant iteration bounds. Such a nested loop is called a *regular* nested loop. If an algorithm is formulated to contain only regular nested loops, it is called a *regular iterative algorithm (RIA)*.

Consider the convolution algorithm described in Eq. (1) of this chapter. The mathematical formula can be conveniently expressed with a 2-level loop nest as shown in Fig. 5.

In this formulation, n and k are *loop indices* having *loop bounds* $(0, N-1)$ and $(0, K-1)$ respectively. The *loop body* $\mathrm{H}(\mathbf{i})$ consists of a single statement

$$y[n] = y[n] + h[k]x[n-k].$$

Fig. 5 Convolution

```
For n = 0 to N − 1,
    y[n] = 0;
    For k = 0 to K − 1,
        y[n] = y[n] + h[k] ∗ x[n − k];
    end
end
```

Note that

$$\mathbf{i} = \begin{bmatrix} n \\ k \end{bmatrix} \; ; \quad \mathbf{p_0} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \leq \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} n \\ k \end{bmatrix} = \mathbf{Pi} = \mathbf{Qi} \leq \begin{bmatrix} N-1 \\ K-1 \end{bmatrix} = \mathbf{q_0}. \tag{9}$$

Hence, this is a RIA.

3.2 *Localized and Single Assignment Algorithm Formulation*

As demonstrated in Sect. 2, a systolic array is a parallel, distributed computing platform where each PE will execute identical operations. By connecting PEs with specific configurations, and providing input data at right timing, a systolic array will be able to perform data intensive computations in a rhythmic, synchronous fashion. Therefore, to implement a given algorithm on a systolic array, its formulation may need to be adjusted. Specifically, since computation takes place at physically separated PEs, data movement in a systolic algorithm must be explicitly specified. Moreover, unnecessary algorithm formulation restrictions that may impede the exploitation of inherent parallelism must be removed. A closer examination of Fig. 5 reveals two potential problems according to above arguments: (1) The variables $y[n]$, $h[k]$, $x[n-k]$ are one-dimensional arrays while each index vector $\mathbf{i} = (n, k)$ resides in a two-dimensional space. (2) The memory address locations $y[n]$ will be repeatedly assigned with new values during each k-loop K times before the final result is evaluated.

Having one-dimensional variable arrays in a two dimensional index space implies that the same input data will be needed when executing the loop body H($\mathbf{i}$) at different iterations (index points). In a systolic array, it is likely that H($\mathbf{i}$) and H($\mathbf{j}$), $\mathbf{i} \neq \mathbf{j}$ may be executed at different PEs. As such, how these variables may be distributed to different index points where they are needed should be explicitly specified in the algorithm. Furthermore, a design philosophy that dominates the development of systolic array is to discourage on-chip global interconnect due to many potential drawbacks. Hence, the data movement would be restricted to *local* communication. Namely passing the data from one PE to one or more of its nearest neighboring PEs in a systolic array. This restriction may be imposed by limiting the propagation of such a global variable from one index point to its nearest neighboring index points. For this purpose, we make the following modification of algorithm in Fig. 5:

$$h[k] \rightarrow h1[n,k] \text{ such that } h1\,[0,k] = h[k],\; h1\,[n,k] = h1[n-1,k]$$
$$x[n] \rightarrow x1[n,k] \text{ such that } x1\,[n,0] = x[n],\; x1\,[n,k] = x1[n-1,k-1].$$

Note that the equations for $h1$ and $x1$ are chosen based on the fact that $h[k]$ will be made available for the entire ranges of index n, and $x[n-k]$ will be made available to all (n', k') such that $n' - k' = n - k$. An algorithm with all its variables passing from one iteration (index point) to its neighboring index point is called a (variable) localized algorithm.

Fig. 6 Convolution (localized, single assignment version)

```
h1[0,k] = h[k], k = 0,...,K-1
x1[n,0] = x[n], n = 0,...,N-1
y1[n,-1] = 0, n = 0,1,...,N-1
n = 0,1,2,...,N-1 and k = 0,...,K-1
    y1[n,k] = y1[n,k-1] + h1[n,k] * x1[n,k]
    h1[n,k] = h1[n-1,k]
    x1[n,k] = x1[n-1,k-1]
y[n] = y1[n,K], n = 0,1,2,...,N-1
```

The repeated assignment of different intermediate results of $y[n]$ into the same memory location will cause an unwanted *output dependence* relation in the algorithm formulation. Output dependence is a type of false data dependence that would impede potential parallel execution of a given algorithm. The output dependence can be removed if the algorithm is formulated to obey a *single assignment* rule. That is, every memory address (variable name) will be assigned to a new value only once during the execution of an algorithm. To remedy, one would create new memory locations to be assigned to these intermediate results by expanding the one dimensional array $\{y[n]\}$ into a two-dimensional array $\{y1[n,k]\}$:

$$y[n] \rightarrow y1[n,k] \text{ such that} y1[n,-1] = 0,$$
$$y1[n,k] = y1[n,k-1] + h1[n,k]x1[n,k],$$

where the previously localized variables $h1$ and $x1$ are used. With above modifications, algorithm in Fig. 5 is reformulated as shown in Fig. 6.

3.3 Data Dependence and Dependence Graph

An iteration H(**j**) is *dependent* on iteration H(**i**) if H(**j**) will read from a memory location whose value is last written during execution of iteration H(**i**). The corresponding *dependence vector* **d** is defined as:

$$\mathbf{d} = \mathbf{j} - \mathbf{i}.$$

A matrix **D** consisting of all dependence vectors of an algorithm is called a *dependence matrix*. This *inter-iteration dependence relation* imposes a partial ordering on the execution of the iterative loop nest. From algorithm in Fig. 6, three dependence vectors can be derived:

$$\mathbf{d}_1 = \begin{bmatrix} n \\ k \end{bmatrix} - \begin{bmatrix} n \\ k-1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix};$$

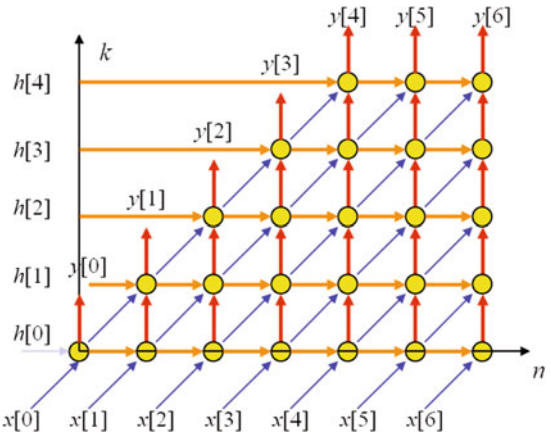


Fig. 7 Localized dependence graph of convolution algorithm

$$\mathbf{d}_2 = \begin{bmatrix} n \\ k \end{bmatrix} - \begin{bmatrix} n-1 \\ k \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}; \tag{10}$$

$$\mathbf{d}_3 = \begin{bmatrix} n \\ k \end{bmatrix} - \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

In the index space, a lattice point whose coordinates fall within the range of the loop bounds represents the execution of the loop body of the particular loop index values. The dependence vectors may be represented by directed arcs starting from the iteration that produces the data to the iteration where the data is needed. Together with these index points and directed arcs, one has a *dependence graph* (DG) representing the computation tasks required of a localized RIA. The corresponding DG of the convolution algorithm is depicted in Fig. 7 for $K = 5$ and $N = 7$.

The dependence graph in Fig. 7 is *shift-invariant* in that the dependence vector structure is identical of each (circled) lattice point of the iteration space. This regularity and modularity is the key feature of a systolic computing algorithm that lends itself for efficient systolic array implementation.

Due to the shift invariant nature, a DG of a localized RIA algorithm can be conveniently represented by the set of indices $\{\mathbf{i};\ \mathbf{p}_0 \leq \mathbf{Pi} \leq \mathbf{q}_0\}$ and the dependence vectors $\mathbf{D}$ at each index point.

3.4 Mapping an Algorithm to a Systolic Array

A *schedule* $\mathrm{S} : \mathbf{i} \rightarrow \mathrm{t}(\mathbf{i}) \in \mathbf{Z}^+$ is a mapping from each index point $\mathbf{i}$ in the index space R to a positive integer $\mathrm{t}(\mathbf{i})$ which dictates *when* this iteration is to be executed. An *assignment* $\mathrm{A} : \mathbf{i} \rightarrow \mathrm{p}(\mathbf{i})$ is a mapping from each index point $\mathbf{i}$ onto a PE index $\mathrm{p}(\mathbf{i})$ *where* the corresponding iteration will be executed. Given the dependence graph of a given algorithm, the development of a systolic array implementation amounts to find a mapping of each index point i in the DG onto $(\mathrm{p}(\mathbf{i}), \mathrm{t}(\mathbf{i}))$.

Toward this goal, two fundamental constraints will be discussed. First, it is assumed that each PE can only execute one task (loop body) at a time. As such, a *resource constraint* must be observed:

Resource Constraints

$$\text{If } \mathrm{t}(\mathbf{i}) = \mathrm{t}(\mathbf{j}),\ \mathbf{i} \neq \mathbf{j}, \text{ then } \mathrm{p}(\mathbf{i}) \neq \mathrm{p}(\mathbf{j});\ \text{ and if } \mathrm{p}(\mathbf{i}) = \mathrm{p}(\mathbf{j}), \mathbf{i} \neq \mathbf{j}, \text{ then } \mathrm{t}(\mathbf{i}) \neq \mathrm{t}(\mathbf{j}). \tag{11}$$

In addition, the data dependence also imposes a partial ordering of schedule. This *data dependence constraint* can be summarized as follows:

Data Dependence Constraint If index $\mathbf{j}$ can be reached from index $\mathbf{i}$ by following the path consisting of one or more dependence vectors, then $\mathrm{H}(\mathbf{j})$ should be scheduled after $\mathrm{H}(\mathbf{i})$. That is, if there exists a vector $\mathbf{m}$ consisting of non-negative integers such that

$$\text{if } \mathbf{j} = \mathbf{i} + \mathbf{D}\mathbf{m}, \text{ then } \mathrm{s}(\mathbf{j}) > \mathrm{s}(\mathbf{i}), \tag{12}$$

where $\mathbf{D}$ is the dependence matrix.

Since a systolic array often assumes a one or two dimensional regular configuration (cf. Fig. 1), the PE index $\mathrm{p}(\mathbf{i})$ can be associated with the lattice point in a *PE index space* just as each loop body in a loop nest is associated with an index point in the DG. To ensure the resulting systolic array features local inter-processor communication, the localized dependence vectors in the DG should not require global communication after the PE assignment $\mathbf{i} \rightarrow \mathrm{p}(\mathbf{i})$. A somewhat restrictive constraint to enforce this requirement would be

Local Mapping Constraint If $\mathbf{j} - \mathbf{i} = \mathbf{d}_k$ (a dependence vector), then

$$\|\mathrm{p}(\mathbf{j}) - \mathrm{p}(\mathbf{i})\|_1 \leq \|\mathbf{d}_k\|_1. \tag{13}$$

A number of performance metrics may be defined to compare the merits of different systolic array implementations. These include

Total Computing Time

$$T_C = \max_{i,j \in DG} (\mathrm{t}(\mathbf{i}) - \mathrm{t}(\mathbf{j})). \tag{14}$$

PE Utilization

$$U_{PE} = N_{DG} / (T_C N_{PE}), \tag{15}$$

where N_DG is the number of index points in the DG, and N_{PE} is the number of PEs in the systolic array.

Now we are ready to formally state the systolic array mapping and scheduling problem:

Systolic Array Mapping and Scheduling Problem Given a localized, shift invariant DG, and a systolic array configuration, find a PE assignment mapping p(**i**), and a schedule t(**i**) such that the performance is optimized, namely, the total computing time T_C is minimized, and the PE utilization U_{PE} is maximized; subject to (1) the resource constraint, (2) the data dependence constraint, and (3) the local mapping constraints.

Thus the systolic array implementation is formulated as a discrete constrained optimization problem. By fully exploiting of the regular (shift invariant) structure of both the DG and the systolic array, this problem can be further simplified.

3.5 Linear Schedule and Assignment

A *linear schedule* is an integer-valued scheduling vector **s** in the index space such that

$$\mathrm{t}(\mathbf{i}) = \mathbf{s}^T\mathbf{i} + t_0 \in \mathbf{Z}^+, \tag{16}$$

where t_0 is a constant integer. The data dependence constraint stipulates that

$$\mathbf{s}^T\mathbf{d} > 0 \text{ for any dependence vector } \mathbf{d}. \tag{17}$$

Clearly, all iterations that reside on a hyper-plane perpendicular to **s**, called *equi-temporal hyperplane* must be executed in parallel at different PEs. The equi-temporal hyperplane is defined as $Q = \left\{\mathbf{i} \mid \mathbf{s}^T\mathbf{i} = \mathrm{t}(\mathbf{i}) - t_0, \mathbf{i} \in DG\right\}$. According to the resource constraint, the maximum number of index points in Q determines the minimum size (number of PEs) of the systolic array.

Assume that the PE index space is a $m-1$ dimensional subspace in the iteration index space. Then the assignment of individual iterations i to a PE index p(**i**) can be realized by projecting **i** onto the PE subspace along an integer-valued assignment vector **a**. Define a $m \times (m-1)$ integer-valued *PE basis* matrix **P** such that $\mathbf{P}^T\mathbf{a} = \mathbf{0}$, then a *linear PE assignment* can be obtained via an affine transformation

$$\mathrm{p}(\mathbf{i}) = \mathbf{P}^T\mathbf{i} + \mathbf{p}_0. \tag{18}$$

Combining Eqs. (16) and (18), one has a node mapping procedure:

$$\textit{Node mapping} \quad \begin{bmatrix} \mathbf{s}^T \\ \mathbf{P}^T \end{bmatrix} \mathbf{i} = \begin{bmatrix} \mathrm{t}(\mathbf{i}) \\ \mathrm{p}(\mathbf{i}) \end{bmatrix}. \tag{19}$$

The node mapping procedure can also be extended to a subset of nodes where external data input and output take places. The same node mapping procedure will indicate where and when these external data I/O will take place in the systolic array. This special mapping procedure is also known as *I/O mapping*.

Different PEs in the systolic array are interconnected by local buses. These buses are implemented based on the need of passing data from an index point (iteration) to another as specified by the dependence vectors. Hence, the orientation of these buses as well as buffers on them can be determined also using **P** and **s**:

$$Arc\ mapping \quad \begin{bmatrix} \mathbf{s}^T \\ \mathbf{P}^T \end{bmatrix} \mathbf{D} = \begin{bmatrix} \tau \\ \mathbf{e} \end{bmatrix}, \tag{20}$$

where τ is the number of first-in-first-out buffers required on each local bus, and **e** is the orientation of the local bus within the PE index space.

Consider two iterations $\mathbf{i}, \mathbf{j} \in DG$, $\mathbf{i} \neq \mathbf{j}$. If p(**i**) = p(**j**), it implies that

$$\mathbf{0} = \mathrm{p}(\mathbf{i}) - \mathrm{p}(\mathbf{j}) = \mathbf{P}T(\mathbf{i} - \mathbf{j}) \Rightarrow \mathbf{i} - \mathbf{j} = k\mathbf{a}. \tag{21}$$

The resource constraint (cf. Eq. (11)) stipulates that if p(**i**) = p(**j**) $\mathbf{i} \neq \mathbf{j}$, then t(**i**) $\neq$ t(**j**). Hence,

$$\mathrm{t}(\mathbf{i}) - \mathrm{t}(\mathbf{j}) = \mathbf{s}^T(\mathbf{i} - \mathbf{j}) = k\mathbf{s}^T\mathbf{a} \neq 0. \tag{22}$$

Example 1 Let us now use the convolution algorithm in Fig. 6 and its corresponding DG in Fig. 7 as an example and set $\mathbf{a}^T = [1\ 0]$, and $\mathbf{s}^T = [1\ 1]$. It is easy to derive the PE basis matrix $\mathbf{P}^T = [0\ 1]$. Hence, the node mapping becomes

$$\begin{bmatrix} \mathbf{s}^T \\ \mathbf{P}^T \end{bmatrix} \begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n+k \\ k \end{bmatrix} = \begin{bmatrix} \mathrm{t}(\mathbf{i}) \\ \mathrm{p}(\mathbf{i}) \end{bmatrix}, \ 0 \leq n \leq 6, \ 0 \leq k \leq \min(4, n). \tag{23}$$

This implies every (n, k) iterations will be executed at PE #k of the systolic array and the scheduled execution time slot is $n+k$. Next, the arc mapping can be found as:

$$\begin{bmatrix} \mathbf{s}^T \\ \mathbf{P}^T \end{bmatrix} \mathbf{D} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 1 & 1 \end{bmatrix}. \tag{24}$$

The second row of the right-hand-side (RHS) of Eq. (24) indicates that there are three local buses. The first one has an entry "0" implies that this is a bus that starts and ends at the same PE. The other two have an entry "1", indicating that they are local buses in the increasing k direction. The first row of the RHS gives the number of registers required on each local bus to ensure the proper execution ordering is obeyed. Thus, the first two buses have a single buffer, while the third bus has two buffers. Note that the external data input $\{x[n]\}$ are fed into the DG at $\{(n, 0); 0 \leq n \leq 6\}$, and the final output $\{y[n]\}$ will be available at $\{(n, K); 0 \leq n \leq 6\}$ where $K = 4$. Thus, through I/O mapping, one has

$$\begin{bmatrix} \mathbf{s}^T \\ \mathbf{P}^T \end{bmatrix} \begin{bmatrix} n & n \\ 0 & K \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} n & n \\ 0 & K \end{bmatrix} = \begin{bmatrix} n & n+K \\ 0 & K \end{bmatrix}. \tag{25}$$

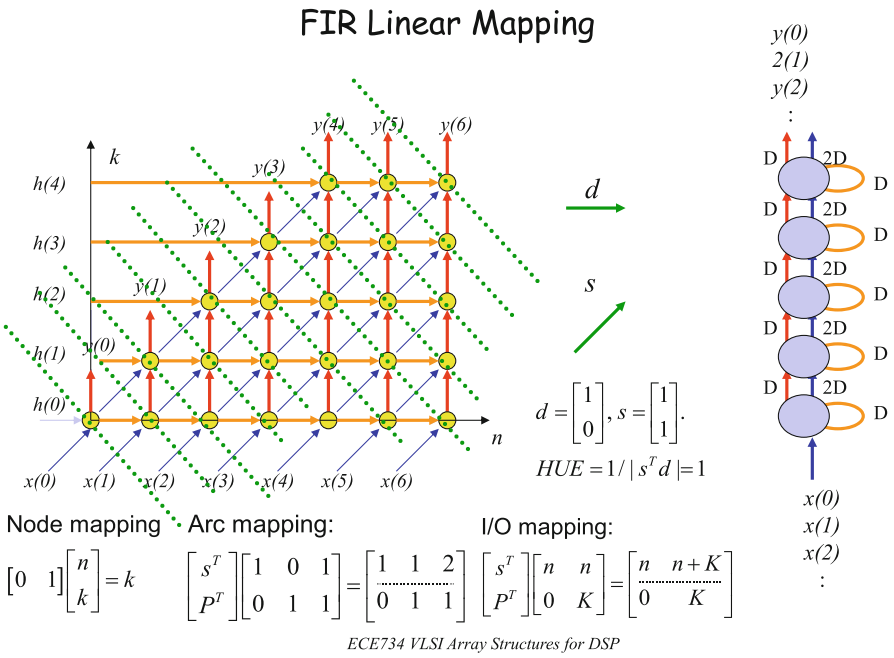


Fig. 8 Linear assignment and schedule of convolution algorithm

This implies that the input $x[n]$ will be fed into the #0 PE of the systolic array at the nth clock cycle; and the output $y[n]$ will be available at the $\#K$ PE at the $(n + K)$th clock cycle. The node mapping, arc mapping and I/O mapping are summarized in Fig. 8.

At the left of Fig. 8, the original DG is overlaid with the equi-temporal hyperplane which is depicted by parallel, dotted lines. To the right of Fig. 8 is the systolic array, its local buses, and the number of buffers (Delays) on each bus. This array is a more abstract version of what is presented in Fig. 2.

4 Wavefront Array Processors

4.1 Synchronous Versus Asynchronous Global On-Chip Communication

The original systolic array architecture adopted a globally synchronous communication model. It is assumed that a global clock signal is available to synchronize the state transition of every storage elements on chip. However, as predicted by the Moore's law, in modern integrated circuits, transistor sizes continue to shrink,

and the number of transistors on a chip continues to increase. These trends make it more and more difficult to implement globally synchronized clocking scheme on chip. On the one hand, the wiring propagation delay does not scale down as transistor feature sizes reduce. As such, the signal propagation delay becomes very prominent compared to logic gate propagation delay. As on-chip clock frequency exceeds giga-hertz threshold, adverse impacts of clock skew become more difficult to compensate. On the other hand, as the number of on-chip transistors increases, so does the complexity and size of on-chip clock distribution network. The power consumption required to distribute giga-hertz clock signal synchronously over entire chip becomes too large to be practical.

In view of the potential difficulties in realizing a globally synchronous clocking scheme as required by the original systolic array design, a asynchronous array processor, known as *wavefront array processor* has been proposed.

4.2 Wavefront Array Processor Architecture

According to [14, 16], a wavefront array is a computing network with the following features:

- *Self-timed, data-driven computation*: No global clock is needed, as the computation is self-timed.
- *Regularity, modularity and local interconnection*: The array should consist of modular processing units with regular and (spatially) local interconnections.
- *Programmability in wavefront language or data flow graph (DFG)*: Computing algorithms implemented on a wavefront array processor may be represented with a data flow graph. Computation activities will propagate through the processor array as if a series of wavefronts propagating through the surface of water.
- *Pipelinability with linear-rate speed-up*: A wavefront array should exhibit a linear-rate speed-up. With M PEs, a wavefront array promises to achieve an $O(M)$ speed-up in terms of processing rates.

The major distinction between the wavefront array the systolic array is that there is no global timing reference in the wavefront array. In the wavefront architecture, the information transfer is by mutual agreements between a PE and its immediate neighbors using, say, an asynchronous hand-shaking protocol [14, 16].

4.3 Mapping Algorithms to Wavefront Arrays

In general, there are three formal methodologies for the derivation of wavefront arrays [15]:

1. Map a localized dependence graph directly to a data flow graph (DFG). Here a DFG is adopted as a formal abstract model for wavefront arrays. A systematical procedure can be used to map a dependence graph (DG) to a DFG.
2. Convert an signal flow graph into a DFG (and hence a wavefront array), by properly imposing several key data flow hardware elements.
3. Trace the computational wavefronts and pipeline the fronts through the processor array. This will be elaborated below.

The notion of computational wavefronts offers a very simple way to design wavefront computing, which consists of three steps:

1. Decompose an algorithm into an orderly sequence of recursions;
2. Map the recursions onto corresponding computational wavefronts in the array;
3. Pipeline the wavefronts successively through the processor array.

4.4 Example: Wavefront Processing for Matrix Multiplication

The notion of computational wavefronts may be better illustrated by an example of the matrix multiplication algorithm where **A**, **B**, and **C**, are assumed to be $N \times N$ matrices:

$$\mathbf{C} = \mathbf{A} \times \mathbf{B}. \tag{26}$$

The topology of the matrix multiplication algorithm can be mapped naturally onto the square, orthogonal $N \times N$ matrix array as depicted in Fig. 9. The computing network serves as a (data) wave-propagating medium. To be precise, let us examine the computational wavefront for the first recursion in matrix multiplication. Suppose that the registers of all the PEs are initially set to zero, that is, $C_{ij}(0) = 0$. The elements of **A** are stored in the memory modules to the left (in columns) and those of **B** in the memory modules on the top (in rows). The process starts with PE (1, 1) which computes:

$$C_{11}(1) = C_{11}(0) + a_{11}b_{11}.$$

The computational activity then propagates to the neighboring PEs (1, 2) and (2, I), which execute:

$$C_{12}(1) = C_{12}(0) + a_{11}b_{12} \text{ and } C_{21}(1) = C_{21}(0) + a_{21}b_{11}.$$

The next front of activity will be at PEs (3,1), (2,2), and (1,3), thus creating a computation wavefront traveling down the processor array. This computational wavefront is similar to optical wavefronts (they both obey Huygens' principle), since each processor acts as a secondary source and is responsible for the propagation of the wavefront. It may be noted that wave propagation implies localized data flow.

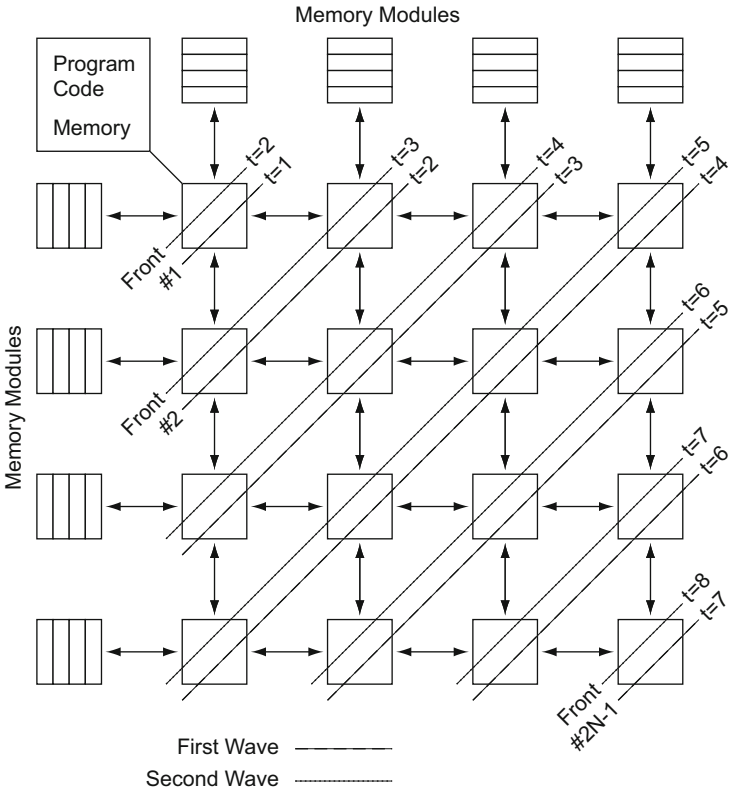


Fig. 9 Wavefront processing for matrix multiplication [15]

Once the wavefront sweeps through all the cells, the first recursion is over. As the first wave propagates, we can execute an identical second recursion in parallel by pipelining a second wavefront immediately after the first one. For example, the (1, 1) processor executes

$$C_{11}(2) = C_{11}(1) + a_{12}b_{21} = a_{11}b_{11} + a_{12}b_{21}.$$

Likewise each processor (i, j) will execute (from $k = 1$ to N)

$$C_{ij}(k) = C_{ij}(k+1) + a_{ik}b_{kj} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{ik}b_{kj}$$

and so on.

In the wavefront processing, the pipelining technique is feasible because the wavefronts of two successive recursions would never intersect. The processors executing the recursions at any given instant are different, thus any contention problems are avoided.

Note that the successive pipelining of the wavefronts furnishes additional dimension of concurrency. The separated roles of pipeline and parallel processing also become evident when we carefully inspect how parallel processing computational wavefronts are pipelined successively through the processor arrays. Generally speaking, parallel processing activities always occur at the PEs on the same front, whereas pipelining activities are perpendicular to the fronts. With reference to the wavefront processing example in Fig. 9, PEs on the anti-diagonals of the wavefront array execute in parallel, since each of the PEs process information independently. On the other hand, pipeline processing takes place along the diagonal direction, in which the computational wavefronts are piped.

In this example, the wavefront array consists of $N \times N$ processing elements with regular and local interconnections. Figure 9 shows the first 4×4 processing elements of the array. The computing network serves as a (data) wave propagating medium. Hence the hardware has to support pipelining the computational wavefronts as fast as resource and data availability allow. The (average) time interval T between two separate wavefronts is determined by the availability of the operands and operators.

4.5 Comparison of Wavefront Arrays Against Systolic Arrays

The main difference between a wavefront array processor and a systolic array lies in hardware design, e.g., on clock and buffer arrangements, architectural expandability, pipeline efficiency, programmability in a high-level language, and capability to cope with time uncertainties in fault-tolerant designs.

As to the synchronization aspect, the clocking scheme is a critical factor for large-scale array systems, and global synchronization often incurs severe hardware design burdens in terms of clock skew. The synchronization time delay in systolic arrays is primarily due to the clock skew which can vary drastically depending on the size of the array. On the other hand, in the data-driven wavefront array, a global timing reference is not required, and thus local synchronization suffices. The asynchronous data-driven model, however, incurs fixed time delay and hardware overhead due to hand-shaking.

From the perspective of pipelining rate, the data-driven computing in the wavefront array may improve the pipelinability. This becomes especially helpful in the case where variable processing times are used in individual PEs. A simulation study on a recursive least squares minimization computation also reports a speedup by a factor of almost two, in favor of the wavefront array over a globally clocked systolic array [4].

In general, a systolic array is useful when the PEs are simple primitive modules, since the handshaking hardware in a wavefront array would represent a non-negligible overhead for such applications. On the other hand, a wavefront array is more applicable when the modules of the PEs are more complex (such as floating-point multiply-and-add), when synchronization of a large array becomes impractical or when a reliable computing environment (such as fault tolerance) is essential.

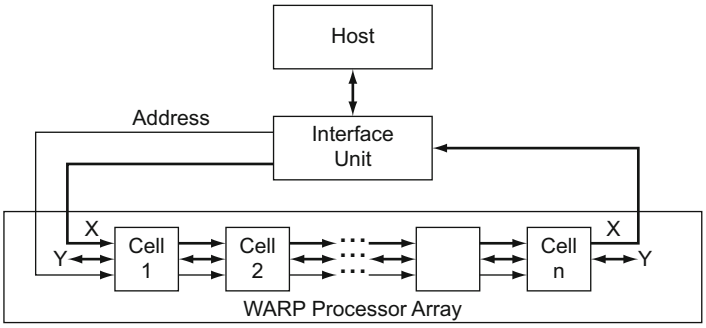


Fig. 10 Warp system overview [1]

5 Hardware Implementations of Systolic Array

5.1 Warp and iWARP

Warp [1] is a prototype linear systolic array processor developed at CMU in mid-1980s. As illustrated in Fig. 10, the Warp array contains 10 identical *Warp cells* interconnected as a linear array. It is designed as an attach processor to a host processor through an interface unit. Each Warp cell has three inter-cell communication links: one address link and two data links (X and Y). They are connected to nearest neighboring cells or the interface unit. Each cell contains two floating point units (one for multiply and one for addition) with corresponding register files, two local memory banks (2K words each with 32 bits/word) for resident and temporary data, each communication link also has a 512 words buffer queue. All these function units are interconnected via a cross-bar switch for intra-cell communication.

The Warp cell is micro-programmed with horizontal micro-code. Although all cells will execute the same cell program, broadcasting micro-code to all cells is not practical and would violate the basic principle of localized communication.

A noticeable feature of the WARP processor array is that its inter-cell communication is *asynchronous*. It is argued [1] that the synchronous, fine-grained inter-PE communication schemes of the original systolic array is too restrictive and is not suitable for practical implementations. Instead, a hard-ware assisted run-time flow control scheme together with a relatively large queue size would allow more efficient inter-cell communication without incurring excessive overheads.

The Warp array uses a specially designed programming language called “W2”. It explicitly supports communication primitives such as “receive” and “send” to transfer data between adjacent cells. The program execution at a cell will stall if either the send or receive statement cannot be realized due to an empty receiving queue (nothing to receive from) or a full sent queue (nowhere to send to). Thus, the programmer bears the responsibility of writing a deadlock free parallel program to run on the Warp processor array.

The performance of the Warp processor array is reported as hundreds of times faster than running the same type of algorithm in a VAX 11/780, a popular mini-computer at the time of Warp development. The development of the Warp processor array is significant in that it is the first hardware systolic array implementation. Lessons learned from this project also motivated the development of iWarp. The iWarp project [3, 7] was a follow-up project of WARP and started in 1988. The purpose of this project is to investigate issues involved in building and using high performance computer systems with powerful communication support. The project led to the construction of the iWarp machines, jointly developed by Carnegie Mellon University and Intel Corporation.

As shown in Fig. 11, the basic building block of the iWarp system is a full custom VLSI component integrating a LIW (long instruction word) microprocessor, a network interface and a switching node into one single chip of 1.2 cm × 1.2 cm silicon. The iWarp cell consists of a computation agent, a communication agent, and a local memory. The computation agent includes a 32-bit micro-processor with 96-bit wide instruction words, an integer/logic unit, a floating point multiplier, and a floating point adder. It runs at a clock speed of 20 MHz. The communication agent has 4 separate full duplex physical data links capable of transferring data at 40 MB/s. These data links can be configured into 20 virtual channels. The clock speed of the communication agent is 40 MHz. Each cell is attached to a local memory sub-system including up to 4 MB static RAM (random access memory) or/and 16 MB DRAM. The iWarp system is designed to be configured as a $n \times m$ torus array. A typical system would have 64 cells configured as a 8×8 torus array and yields 1.2 GFlop/s peak performance.

The communication agent supports word-level flow control between connecting cells and transfers messages word by word to implement wormhole routing [19]. Exposing this mechanism to the computation agents allows programs to communicate systolically. Moreover, a communication agent can automatically route messages to the appropriate destination without the intervention of the computation agent.

5.2 SAXPY Matrix-1

Claimed to be "the first commercial, general-purpose, systolic computer", Matrix-1 [6] is a vector array processor developed by the SAXPY computer co. in 1987 for scientific, signal processing applications. It promises 1 GFLOP throughput by means of 32-fold parallelism, fast (64 ns) pipelined floating-point units, and fast and flexible local memories.

At system level, a Matrix-1 system (cf. Fig. 12) consists of a system controller, system memory, and mass storage in addition to the matrix processor. These system components are interconnected with a high-speed (320 MB/s) bus (S-bus). The system memory has a maximum capacity of 128 MB. It uses only physical addresses and hence allows faster access.

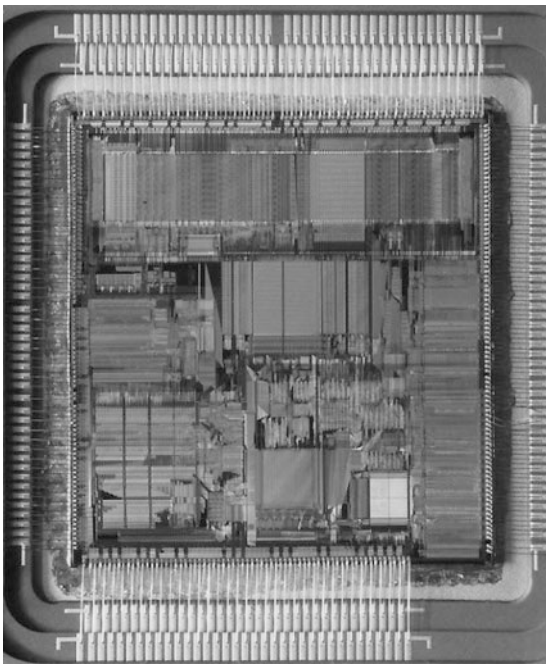


Fig. 11 Photograph of a iWARP chip [3]

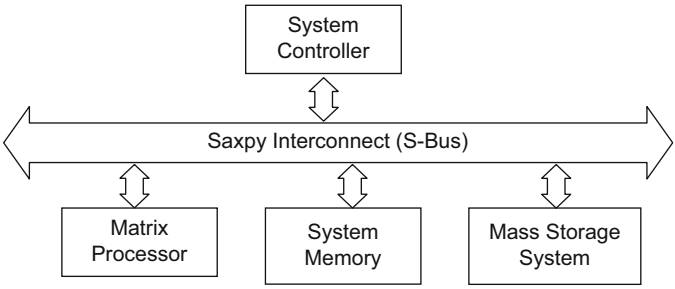


Fig. 12 Block diagram of a Matrix-1 system

The Matrix Processor (Fig. 13) is a ring-connected linear array of 8, 16, 24, or 32 vector processors. Each processor is called a computational zone. All zones receive the same control and address instructions at each clock cycle. The Matrix Processor can function in a systolic mode (in which data are transferred from one zone to the next in a pipelined fashion) or in a block mode (in which all zones operate simultaneous to execute vector operations). Each zone has a pipelined, 32-bit floating-point multiplier; a pipelined, 32-bit floating-point adder with logic capabilities, and a 4K-word local memory implemented as a two-way interleaved zone buffer. These components operate at a clock frequency of 16 MHz. With 32 zones, the maximum computing power would approach 960 MFLOP.

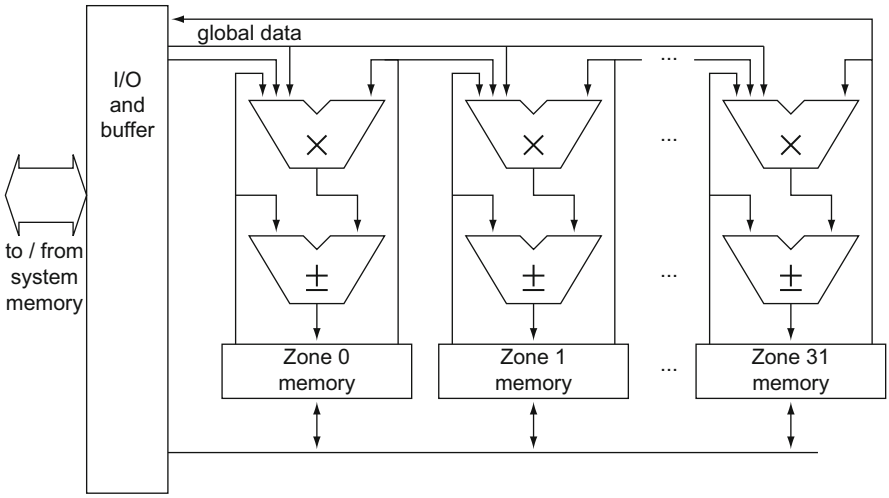


Fig. 13 The Matrix Processor Zone architecture of SAXPY Matrix-1 computer

The Matrix-1 employs an application programming interface (API) approach to interface with the host processor. The user program will be written in C or Fortran and makes calls to the matrix processor subroutines. Experienced programmers may also write their own matrix processor subroutines or directly engage assembly level programming of the matrix processors.

5.3 Transputer

The Transputer (transistor computer) [8, 20, 26, 30] is a microprocessors developed by Inmos Ltd. in mid-1980s to support parallel processing. The name was selected to indicate the role the individual Transputers would play: numbers of them would be used as basic building blocks, just as transistors in integrated circuits.

A most distinct feature of a Transputer chip is that there are four serial links to communicate with up to four other Transputers simultaneously each at 5, 10, or 20 Mbit/s. The circuitry to drive the links is all on the Transputer chip and only two wires are needed to connect two Transputers together. The communication links between processors operate concurrently with the processing unit and can transfer data simultaneously on all links without the intervention of the CPU. Supporting the links was additional circuitry that handled scheduling of the traffic over them. Processes waiting on communications would automatically pause while the networking circuitry finished its reads or writes. Other processes running on the transputer would then be given that processing time. These unique properties allow multiple Transputer chips to be configured easily into various topologies such as linear or mesh array, or trees to support parallel processing.

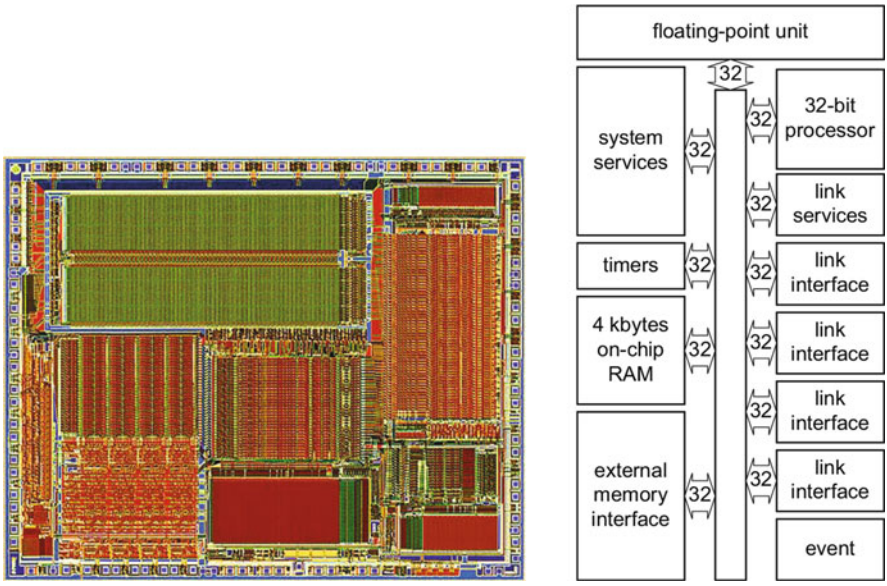


Fig. 14 INMOS T805 floating-point processor (http://www.classiccmp.org/transputer/)

Depicted in Fig. 14 is a chip layout picture and a floor plan of Transputer T805. It has a 32-bit architecture running at 25 MHz clock frequency. It has an IEEE 754 64-bit on-chip floating point unit, 4 KB on-chip static RAM, and may connect to 4 GB directly addressable external memory (no virtual memory) at 33 MB/s sustained data rate. It uses a 5 MHz clock input and runs on a single 5 V power supply.

Transputers were intended to be programmed using the OCCAM programming language, based on the CSP process calculus. Occam supported concurrency and channel-based inter-process or inter-processor communication as a fundamental part of the language. With the parallelism and communications built into the chip and the language interacting with it directly, writing code for things like device controllers became a triviality. Implementations of more mainstream programming languages, such as C, FORTRAN, Ada and Pascal were also later released by both INMOS and third-party vendors.

5.4 TMS 32040

TMS 32040 [27] is Texas Instruments' floating point digital signal processor developed in early 1990. The '320C40 has six on-chip communication ports for processor-to-processor communication with no external-glue logic. The communication ports remove input/output bottlenecks, and the independent smart DMA coprocessor is able to relieve the CPU input/output burden.

Each of the six serial communication ports is equipped with a 20M-bytes/s bidirectional interface, and separate input and output 8-word-deep FIFO buffers. Direct processor-to-processor connection is supported by automatic arbitration and handshaking. The DMA coprocessor allows concurrent I/O and CPU processing for sustained CPU performance.

The processor features single-cycle 40-bit floating-point and 32-bit Integer multipliers, 512-byte instruction cache, and 8K Bytes of single-cycle dual-access program or data RAM. It also contains separate internal program, data, and DMA coprocessor buses for support of massive concurrent input/output (I/O) program and data throughput.

The TMS 32040 is designed to support general purpose parallel computation with different configurations. With six bidirectional serial link ports, it would directly support a hypercube configuration containing up to $2^6 = 64$ processing elements. It, of course, also can be easily configured to form a linear or two-dimensional mesh-connected processor array to support systolic computing.

6 Recent Developments and Real World Applications

6.1 Block Motion Estimation

Block motion estimation is a critical computation step in every international video coding standard, including MPEG-I, MPEG-II, MPEG-IV, H.261, H.263, and H.264. This algorithm consists of a very simple loop body (sum of absolute difference) embedded in a six-level nested loop. For real time, high definition video encoding applications, the motion estimation operation must rely on special purpose on-chip processor array structures that are heavily influenced by the systolic array concept.

The notion of block motion estimation is demonstrated in Fig. 15. To the left of this figure is the *current frame* which is to be encoded and transmitted from the encoding end. To the right is the *reference frame* which has already been transmitted and reconstructed at the receiver end. The encoder will compute a copy of this reconstructed reference frame for the purpose of motion estimation. Both the current frame and the reference frame are divided into *macro-blocks* as shown with dotted lines. Now focus on the *current block* which is the shaded macro-block at the second row and the fourth column of the current frame. The goal of motion estimation is to find a matching macro-block in the reference frame, in the vicinity of the location of the current block such that it resembles the current block in the current frame. Usually, the current frame and the reference frame are separated by a couple of frames temporally, and are likely to contain very similar scene. Hence, there exists high degree of *temporal correlation* among them. As such, there is high probability that the current block can find a very similar matching block in the reference frame. The displacement between the location of the current block and that of the matching

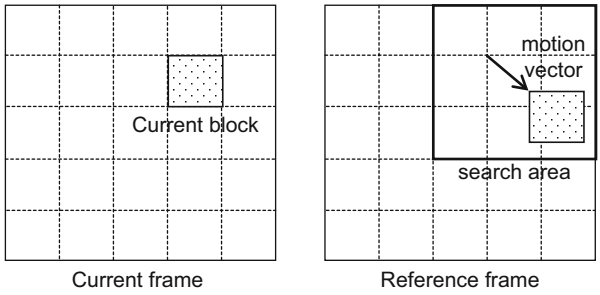


Fig. 15 Block motion estimation

macro-block is the *motion vector* of the current block. This is shown to the right hand side of Fig. 15. By transmitting the motion vector alone to the receiver, a predicted copy of the current block can be obtained by copying the matching macro-block from the reference frame. That process is known as *motion compensation*.

The similarity between the current block in the current frame and corresponding matching block in the reference frame is measured using a *mean of absolute difference (MAD) criterion*:

$$MAD(m,n) = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |x(i,j) - y(i+m, j+n)|. \tag{27}$$

where the size of the macro-block is N pixels by N pixels. $x(i,j)$ is the value of the (i,j)th pixel of the current frame and $y(i+m, j+n)$ is the value of the $(i+m, j+n)$th pixel of the reference frame. $MAD(m,n)$ is the mean absolute difference value between the current block and the candidate matching block with a displacement of (m,n), $-p \le m,n \le p$, where p is a bounded, pre-set *search range* which is usually twice or thrice the size of a macro-block. The motion vector (MV) of the current block is found as

$$MV = \arg\left\{ \min_{-p \le m,n \le p} MAD(m,n) \right\}. \tag{28}$$

We assume each video frame is partitioned into $N_h \times N_v$ macro-blocks. With Eqs. (27) and (28), one may express the whole frame full-search block matching motion estimation algorithm as a six-level nested loop as shown in Fig. 16.

The performance requirement for such a motion estimation operation is rather stringent. Take MPEG-II for example, a typical video frame of 1080p format contains 1920×1080 pixels. With a macro-block size $N = 16$, one has $N_h = 1920/16 = 120$, $Nv = \lceil 1080/16 \rceil = 68$. Usually, $N = 16$, and $p = N/2$. Since there are 30 frames per second, the number of the sum of absolute difference operations that need to be performed would be around $30 \times N_h \times N_v \times (2p+1)^2 \times N^2 \approx 1.8 \times 10^{10}$ operations/second.

```
Do h = 0 to N_h − 1
  Do v = 0 to N_v − 1
    MV(h, v) = (0, 0)
    Dmin(h, v) = ∞
    Do m = −p to p
      Do n = −p to p
        MAD(m, n) = 0
        Do i = h ∗ N to (h + 1) ∗ N − 1
          Do j = v ∗ N to (v + 1) ∗ N − 1
            MAD(m, n) = MAD(m, n) + |x(i, j) − y(i + m, j + n)|
          End do j
        End doi
        If Dmin(h, v) > MAD(m, n)
          Dmin(h, v) = MAD(m, n)
          MV(h, v) = (m, n)
        End if
      End do n
    End do m
  End do v
End do h
```

Fig. 16 Full search block matching motion estimation

Since motion estimation is only part of video encoding operations, an application specific hardware module would be a desirable implementation option. In view of the regularity of the loop-nest formulation, and the simplicity of the loop-body operations (addition/subtraction), a systolic array solution is a natural choice. Toward this direction, numerous motion estimation processor array structures have been proposed, including 2D mesh array, 1D linear array, tree-structured array, and hybrid structures. Some of these realizations focused on the inner 4-level nested loop formulation of algorithm in Fig. 16 [12, 22], and some took the entire 6-level loop nest into accounts [5, 11, 31]. An example is shown in Fig. 17. In this configuration, the search area pixel y is broadcast to each processing elements in the same column; and current frame pixel x is propagated along the spiral interconnection links. The constraint of $N = 2p$ is imposed to achieve low input/output pin count. A simple PE is composed of only two 8-bit adders and a comparator as shown in Fig. 18.

A number of video encoders micro-chips including motion estimation have been reported over the years. Earlier motion estimation architectures often use some variants of a pixel-based systolic array to evaluate the MAD operations. Often a fast search algorithm is used in lieu of the full search algorithm due to speed and power consumption concerns. One example is a MPEG-IV standard profile encoder chip reported in [18]. Some chip characteristics are given in Table 1.

As shown in Fig. 19, the motion estimation is carried out with 16 adder tree (processing units, PU) for sum of absolute difference calculation and the motion vectors are selected based on these results. A chip micro-graph is depicted in Fig. 20.

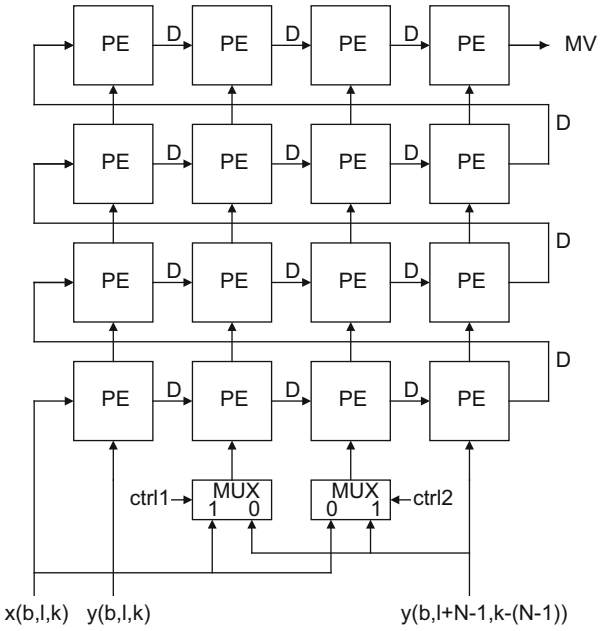


Fig. 17 2-D array with spiral interconnection ($N = 4$ and $p = 2$) [31]

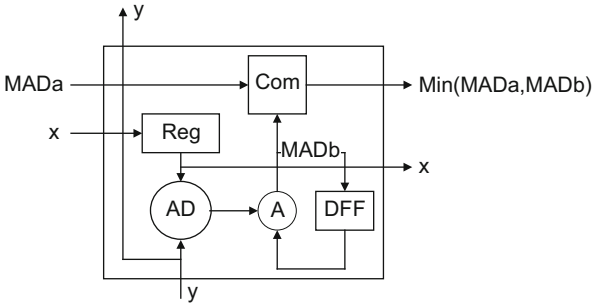


Fig. 18 Block diagram of an individual processing element [31]

6.2 Wireless Communication

Systolic array has also found interesting applications in wireless communication baseband signal processing applications. A typical block diagram of wireless transceiver baseband processing algorithms is depicted in Fig. 21. It includes fast Fourier transform (FFT) and inverse fast Fourier transform (IFFT), channel estimator/equalizer, data interleaver, and channel encoder/decoder, etc.

In [25], a reconfigurable systolic array of CORDIC (Coordinate Rotation Digital Computer) processing nodes (PN) is proposed to realize the computation intensive

Table 1 MPEG-IV motion estimation chip features [18]

Technology	TSMC 0.18 μm, 1P6M CMOS
Supply voltage	1.8 V (Core)/3.3 V (I/O)
Core area	$1.78 \times 1.77\ mm^2$
Logic gates	201 K (2-input NAND gate)
SRAMs	4.56 kB
Encoding feature	MPEG-4 SP
Search range	H[−16, +15.5] V[−16, +15.5]
Operating frequency	9.5 MHz CIF, 28.5 MHz VGA
Power consumption	5 mW (CIF, 9.5 MHz, 1.3 V)
	18 mW (VGA, 28.5 MHz, 1.4 V)

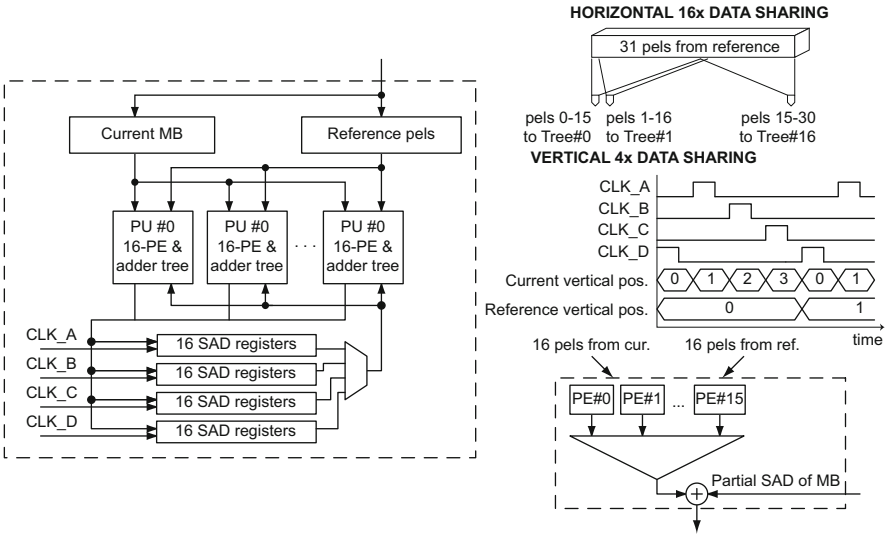


Fig. 19 Motion estimation architecture [18]

portion of the wireless baseband operations. CORDIC [28, 29] is an arithmetic computing algorithm that has found many interesting signal processing applications [9]. Specifically, it is an efficient architecture to realize unitary rotation operations such as Jacobi rotation described in Eqs. (3)–(5) in this chapter. With CORDIC, the rotation angle θ is represented with a weighted sum of a sequence of elementary angels $\{a(i); 0 \le i \le n-1\}$ where $a(i) = \tan^{-1} 2^{-i}$. That is,

$$\theta = \sum_{i=0}^{n-1} \mu_i a(i) = \sum_{i=0}^{n-1} \tan^{-1} 2^i\ , \quad \mu_i \in \{-1, +1\}. \tag{29}$$

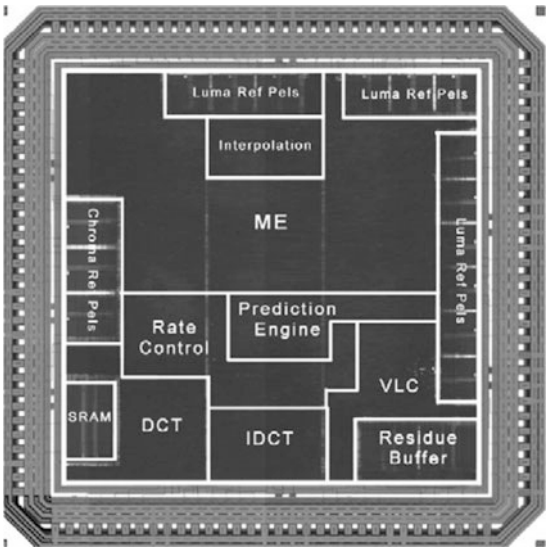


Fig. 20 MPEG-IV encoder chip die micro-graph [18]

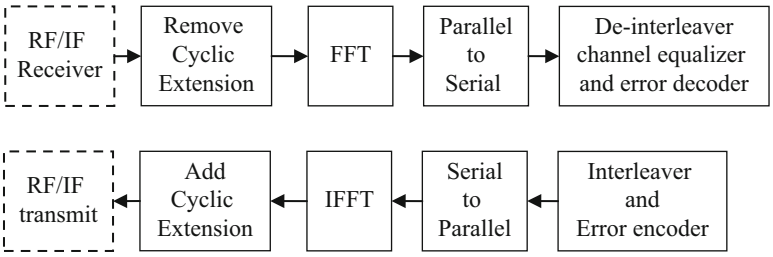


Fig. 21 A typical block diagram of wireless transceiver baseband processing

As such, the rotation operation through each elementary angle may be easily realized with simple shift-and-add operations

$$\begin{bmatrix} x(i+1) \\ y(i+1) \end{bmatrix} = \begin{bmatrix} \cos a(i) & -\mu_i \sin a(i) \\ \mu_i \sin a(i) & \cos a(i) \end{bmatrix} \begin{bmatrix} x(i) \\ y(i) \end{bmatrix} \tag{30}$$

$$= k(i) \begin{bmatrix} 1 & -m\mu_i 2^{-i} \\ \mu_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x(i) \\ y(i) \end{bmatrix}$$

where $k(i) = 1/\sqrt{1+2^{-2i}}$.

A block diagram of a 4×4 CORDIC reconfigurable systolic array is shown in Fig. 22. The control unit is a general purpose RISC (reduced instruction set computer) micro-processor. The PN array employs a data driven (data flow) paradigm so that globally synchronous clocking is not required. During execution phase, the address generator provides an address stream to the data memory bank.

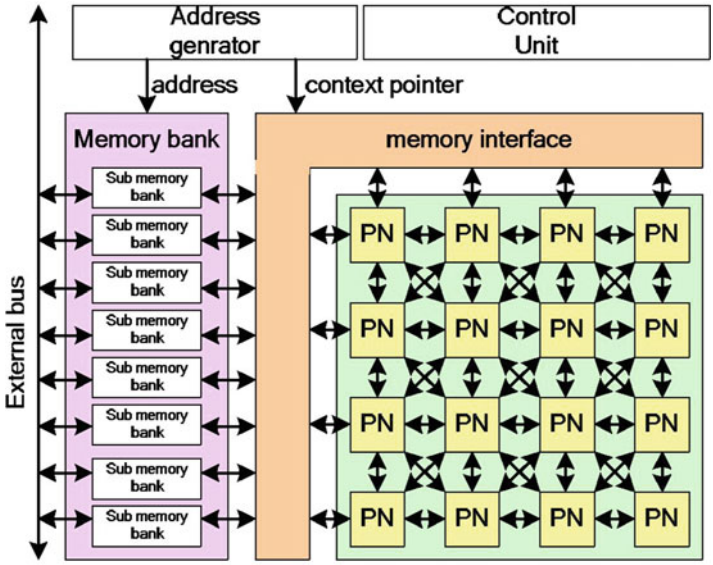


Fig. 22 CORDIC systolic array [25]

Accessed data is fed from the data memory bank to the PN array and back, via the memory interface, which adds a context pointer to the data. With the context pointer, dynamic reconfiguration of the PN array within a single clock cycle becomes possible.

The PN architecture is depicted in Fig. 23 where two CORDIC processing elements, two delay processing elements are interconnected via the communication agent, which also handles external communications with other PNs.

Using this CORDIC reconfiguration systolic array, a minimum mean square error detector is implemented for an OFDM (orthogonal frequency division modulation) MIMO (multiple input, multiple output) wireless transceiver. A QR decomposition recursive least square (QRD-RLS) triangular systolic array is implemented on a FPGA prototype system and is shown in Fig. 24.

6.3 Deep Neural Network

Since mid-1980s, artificial neural network (ANN), especially multilayer perceptron (MLP) has attracted many attention for its promise of solving challenging pattern recognition problems such as speech recognition, image object recognition. However, ANN MLP often requires tremendous amount of computation power that cannot be offered with the information technology at that time. Even so, the regular computation requirement of MLP has attracted researchers' attention to

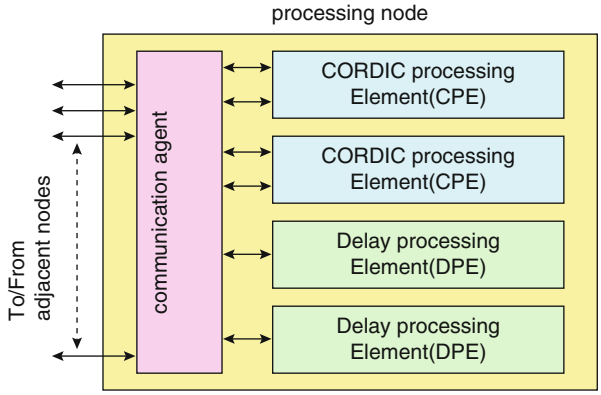


Fig. 23 Processing node architecture [25]

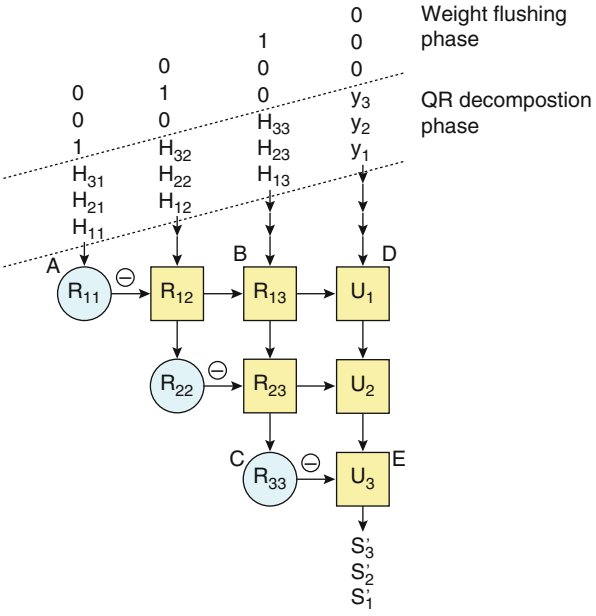


Fig. 24 QRD-RSL triangular systolic array [25]

develop special purpose hardware platform, leveraging systolic array technology to accelerate computation. A chapter in this handbook [24] provides an overview of the algorithmic aspects of DNN. In this section, we focus on applications of systolic array for a couple of DNN implementations.

The basic computing unit in a DNN is called a McCulloch-Pitts model of neuron. Referring to Fig. 25a, the ith neuron consists of N inputs forming a $N \times 1$ input vector $\mathbf{x}$ and a single output, called the *activation* a_i. Inside the neuron, a net

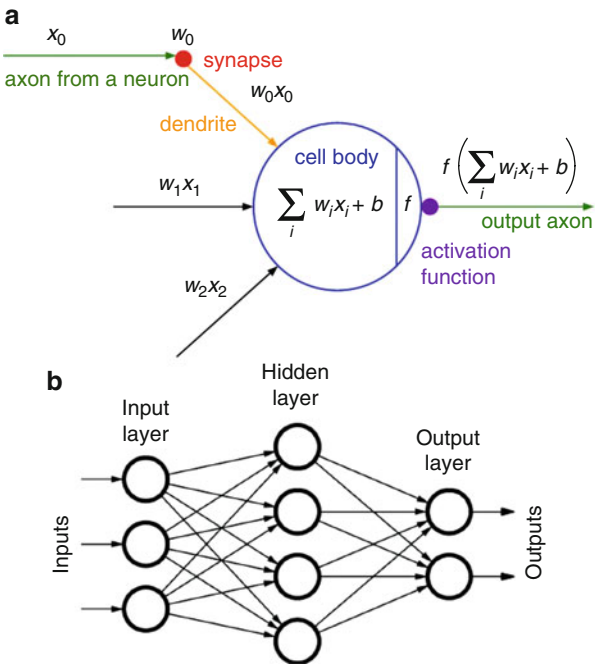


Fig. 25 (**a**) A McCulloch-Pitts neuron model; (**b**) organization of a feed-forward multi-layer perceptron network

function u_i is evaluated as

$$u_i = \mathbf{w}_i^T \mathbf{x} + \theta_i, \tag{31}$$

where $\mathbf{w}_i$ is a $N \times 1$ weight vector and θ_i is a scalar bias term. Once u_i is evaluated, it will pass through a nonlinear transformation to form the activation:

$$a_i = f(u_i). \tag{32}$$

A popular choice of the nonlinear transformation is called a sigmoidal function that has the form

$$f(u) = \frac{1}{1 + exp(-\alpha u)}.$$

Other popular nonlinear transformation functions include the hyperbolic tangent function, rectified linear unit (ReLU), as well as Max-pooling. When the output nonlinear function is a threshold function with binary output values of 0 or 1, such a neuron model is also known as a *perceptron*.

A neuron can be abstracted as a node in a graph with multiple incoming edges from external inputs or activations of other neurons, and a single out-going edge (the activation). By connecting neurons together, a directed network (graph) may be configured to form a *neural network*. If the corresponding directed graph model of a neural network consists of one or more cycles, such a neural network is called a *recurrent neural network*. Otherwise, a neural network corresponding to an acyclic graph is known as a feed-forward network. As illustrated in Fig. 25b, in a feed-forward network, neurons may be organized into *layers* based on their graphic distance from the input (or from the output). A most popular feed-forward neural network is known as *multi-layer perceptron* (MLP), despite the fact that the sigmoidal nonlinearity is used in lieu of the threshold function. A deep neural network is usually a MLP with large number of layers.

The MLP structure allows a vectorized representation of the computation performed in such network. Specifically, assume that there are m neurons forming the ℓth layer. Their activation values form an $m \times 1$ vector $\mathbf{y}$. $\mathbf{y}$ is evaluated by

$$\mathbf{y} = f(\mathbf{u}) = f(\mathbf{W}\mathbf{x}), \tag{33}$$

where $\mathbf{W}$ is a $m \times N$ *weight matrix*, and $\mathbf{x}$ represents all inputs to the neurons in the ℓth layer. For convenience, the bias term may be absorbed as a separate column of the $\mathbf{W}$ matrix and a constant input of value 1 in the $\mathbf{x}$ vector. The nonlinearity is applied element by element to the net function vector $\mathbf{u}$.

A neural network is operated in two different modes: learning (training) and inferencing (testing). During learning mode, annotated input-output pairs (training data) are provided to train the weights (including bias) of a neural network so that it behaves as close to that pre-scribed in the training data as possible. Once a network is successfully trained, it may be deployed to faciliate inferencing where the trained weight matrices will be used so that the network can provide outputs to inputs that are not part of the training data. The operation in Eq. (33) is performed for each layer of a MLP from input toward the output. This forward pass is performed during the training phase as well as during the inference phase. In the training phase, a *back-propagation* procedure will be performed to update the weights after the forward pass. In the inference phase, the output will be provided to the user immediately without further processing. The training phase is often conducted off-line with large computation resources and long training time (weeks, months or longer). However, for inference applications such as real time language translation, speech conversation, short latency becomes a requirement. Thus, most existing systolic realization of neural networks have been focused on accelerating the inference process given a trained network (given weights).

In [23], a specific VLSI Neural Signal Processor called the MA-16 is proposed. Each MA-16 is a custom systolic multiply-accumulate model that performs sixteen 16-bit multiplies concurrently. It uses custom hardware units to realize the activation function.

Recently, Microsoft reported a *Catapult* FPGA accelerator card, as shown in Fig. 26, that leverage a systolic array of processing elements to accelerate evaluation of deep convolutional neural network (CNN) [21]. Each Catapult card consists of

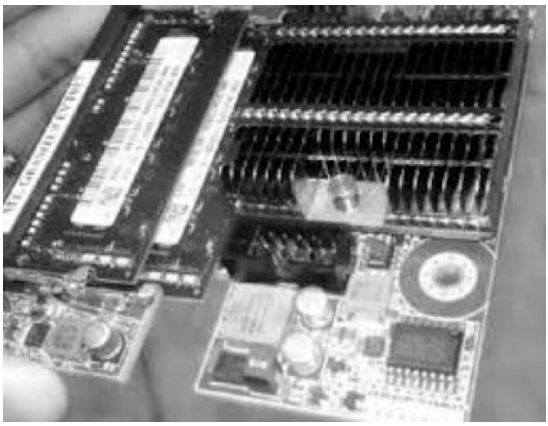


Fig. 26 Catapult FPGA accelerator card [21]

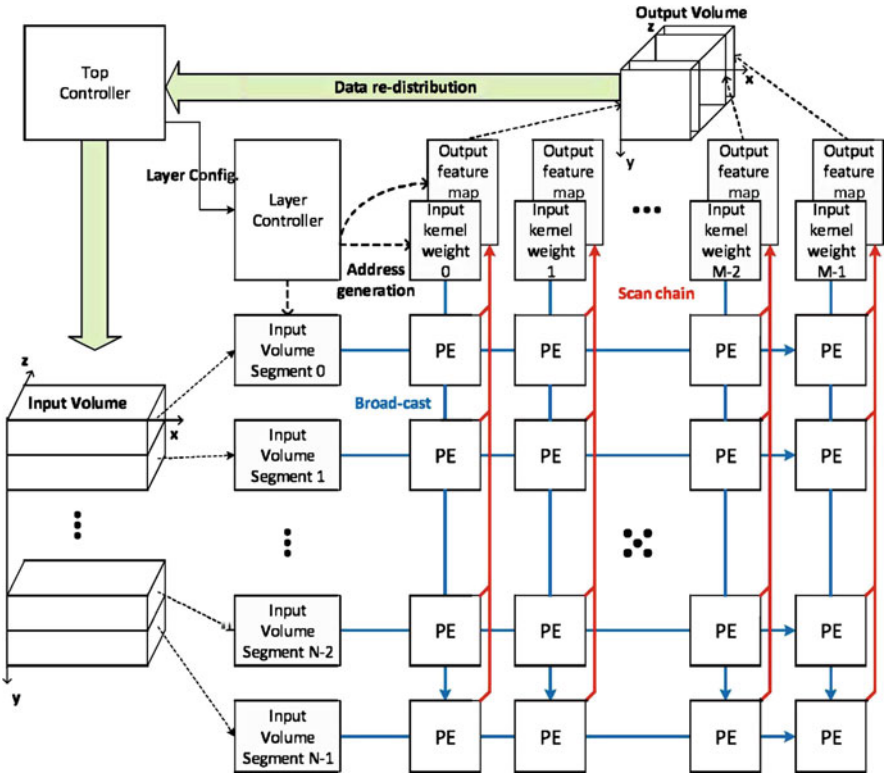


Fig. 27 Systolic array microarchitecture of Catapult [21]

an Altera(R) Stratix V D5 FPGA chip, 8 GB DDR3 DRAM module, and a PCIe Gen 3 $\times$ 8 bus interface. A systolic array micro-architecture is implemented on the FPGA. As shown in Fig. 27, the systolic array consists of a $m \times n$ rectangular

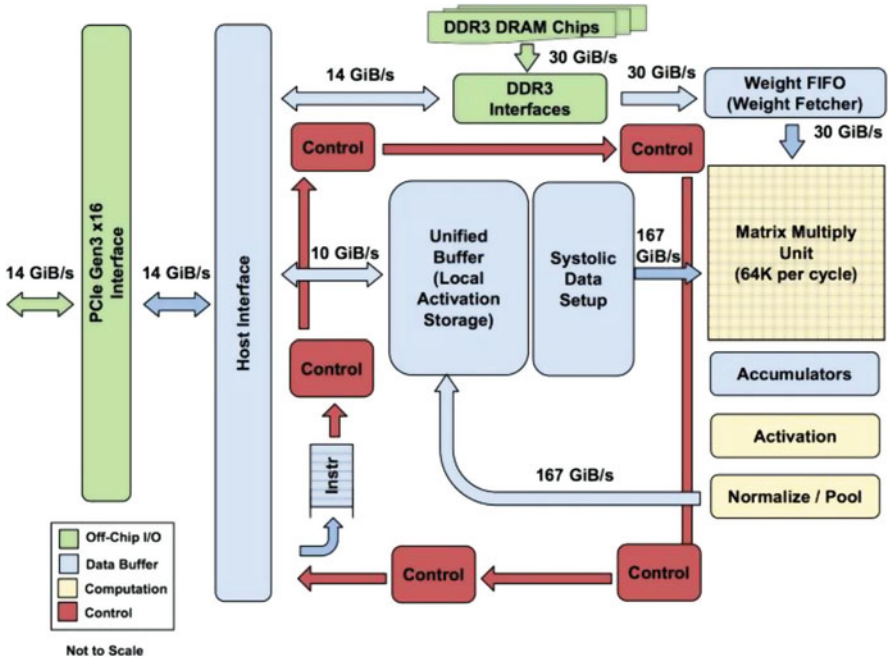


Fig. 28 Tensor Processing Unit system block diagram [17]

array of function units (FU) that implement the multiply-and-accumulate (MAC) operation and a simple data forwarding control. The output will be sent to an array of output buffers (OB), adding to bias values, and then passing through hardware-implemented non-linear activation functions as desired, and finally passing through the max-pooling elements (MPE).

The company Google reported [17] a *Tensor Processing Unit* (TPU) as a customed systolic array chip for inference processing of a variety of DNNs, including MLP, Short-Long Term Memory (SLTM), and CNN. A block diagram of the TPU unit is shown in Fig. 28. The floor plan of the TPU chip is depicted in Fig. 29. The matrix multiplication unit is implemented by a systolic array. However, most of the chip area is dedicated to on-chip storage of data and weights. In [17], it is observed that memory bandwidth is the limiting factor of the overall performance. The hardware-software design objective is to keep the matrix multiplication array busy as much as possible. The TPU's systolic array micro-architecture is depicted in Fig. 30. It contains 256×256 multiply-and-accumulate units that can perform 8-bit multiply-and-adds on signed or unsigned integers. The matrix unit produces one 256-element partial sum per clock cycle. The 16-bit products are collected in the 256 32-bit Accumulators below the matrix unit.

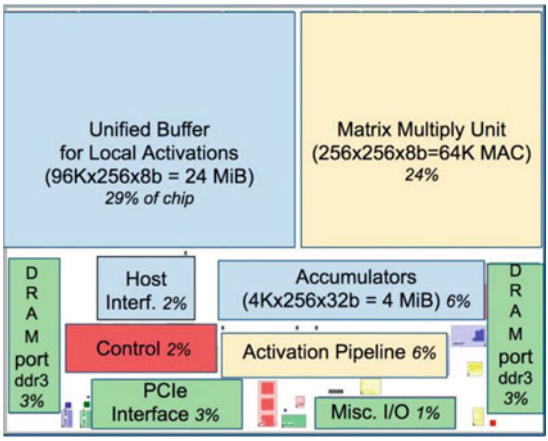


Fig. 29 Tensor Processing Unit floor plan [17]

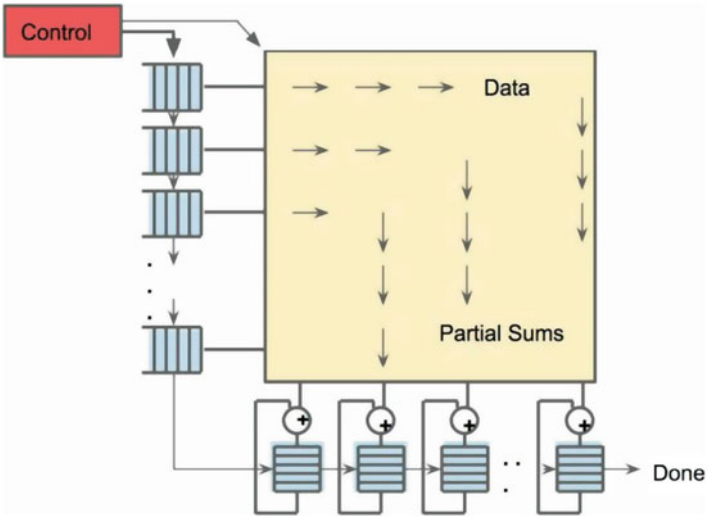


Fig. 30 Systolic array data flow of matrix multiplication unit of the TPU [17]

7 Summary

In this chapter, the historically important systolic array architecture is discussed. The basic systolic design methodology is reviewed, and the wavefront array processor architecture has been surveyed. Several existing implementations of systolic array like parallel computing platforms, including WARP, SAXPY Matrix-1, Transputer, and TMS320C40 have been briefly reviewed. Real world applications of systolic arrays to video coding motion estimation and wireless baseband processing have also been discussed.

References

1. Annaratone, M., Arnould, E., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., and Webb, J.A.: The WARP computer: Architecture, implementation, and performance. IEEE Trans. Computers **36**, 1523–1538 (1987)
2. Arnould, E., Kung, H., et al.: A systolic array computer. In: Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 10, pp. 232–235 (1985)
3. Borkar, S., Cohn, R., Cox, G., Gross, T., Kung, H.T., Lam, M., Levine, M., Moore, B., Moore, W., Peterson, C., Susman, J., Sutton, J., Urbanski, J., Webb, J.: Supporting systolic and memory communication in iwarp. In: Proc. 17th Intl. Symposium on Computer Architecture, pp. 71–80 (1990)
4. Broomhead, D., Harp, J., McWhirter, J., Palmer, K., Roberts, J.: A practical comparison of the systolic and wavefront array processing architectures. In: Proc. Intl. Conf. Acoustics, Speech, and Signal Processing, vol. 10, pp. 296–299 (1985)
5. Chen, Y.K., Kung, S.Y.: A systolic methodology with applications to full-search block matching architectures. J. of VLSI Signal Processing **19**(1), 51–77 (1998)
6. Foulser, D.E.: The Saxpy Matrix-1: A general-purpose systolic computer. IEEE Computer **20**, 35–43 (1987)
7. Gross, T., O'Hallaron, D.R.: iWarp: Anatomy of a Parallel Computing System. MIT Press, Boston, MA (1998)
8. Homewood, M., May, D., Shepherd, D., Shepherd, R.: The IMS T800 Transputer. IEEE Micro **7**(5), 10–26 (1987)
9. Hu, Y.H.: CORDIC-based VLSI architectures for digital signal processing. IEEE Signal Processing Magazine **9**, 16–35 (1992)
10. iWarp project. URL http://www.cs.cmu.edu/afs/cs/project/iwarp/archive/WWW-pages/iwarp.html
11. Kittitornkun, S., Hu, Y.: Systolic full-search block matching motion estimation array structure. IEEE Trans. Circuits Syst. Video Technology **11**, 248–251 (2001)
12. Komarek, T., Pirsch, P.: Array architectures for block matching algorithms. IEEE Trans. Circuits Syst. **26**(10), 1301–1308 (1989)
13. Kung, H.T.: Why systolic array. IEEE Computers **15**, 37–46 (1982)
14. Kung, S.Y.: On supercomputing with systolic/wavefront array processors. Proc. IEEE **72**, 1054–1066 (1984)
15. Kung, S.Y.: VLSI Array Processors. Prentice Hall, Englewood Cliffs, NJ (1988)
16. Kung, S.Y., Arun, K.S., Gal-Ezer, R.J., Bhaskar Rao, D.V.: Wavefront array processor: Language, architecture, and applications. IEEE Trans. Computer **31**(11), 1054–1066 (1982)
17. Jouppi, N. P., et al: In-Datacenter Performance Analysis of a Tensor Processing Unit. IEEE 44th International Symposium on Computer Architecture (ISCA), pp. 1–12, Toronto, Canada, (2017)
18. Lin, C.P., Tseng, P.C., Chiu, Y.T., Lin, S.S., Cheng, C.C., Fang, H.C., Chao, W.M., Chen, L.G.: A 5mW MPEG4 SP encoder with 2D bandwidth-sharing motion estimation for mobile applications. In: Proc. International Solid-State Circuits Conference, pp. 1626–1635. San Francisco, CA (2006)
19. Ni, L.M., McKinley, P.: A survey of wormhole routing techniques in direct networks. IEEE Computer **26**, 62–76 (1993)
20. Nicoud, J.D., Tyrrell, A.M.: The transputer T414 instruction set. IEEE Micro **9**(3), 60–75 (1989)
21. Ovtcharov, K., Ruwase, O., Kim, J.Y., Fowers, J., Strauss, K. and Chung, E.S.: Toward accelerating deep learning at scale using specialized hardware in the datacenter. IEEE Hot Chips 27 Symposium, 1–38 (2015)
22. Pan, S.B., Chae, S., Park, R.: VLSI architectures for block matching algorithm. IEEE Trans. Circuits Syst. Video Technol. **6**(1), 67–73 (1996)

23. Ramacher, U., Beichter, J., Raab, W., Anlauf, J., Bruels, N., Hachmann, U. and Wesseling, M.: Design of a 1st Generation Neurocomputer. VLSI Design of Neural Networks, Springer US. (1991)
24. Huttunen, H.: Deep neural networks: A signal processing perspective. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
25. Seki, K., Kobori, T., Okello, J., Ikekawa, M.: A cordic-based reconfigrable systolic array processor for MIMO-OFDM wireless communications. In: Proc. IEEE Workshop on Signal Processing Systems, pp. 639–644. Shanghai, China (2007)
26. Taylor, R.: Signal processing with occam and the transputer. IEE Proceedings F: Communications, Radar and Signal Processing **131**(6), 610–614 (1984)
27. Texas Instruments: TMS320C40 Digital Signal Processors (1996). URL http://focus.ti.com/docs/prod/folders/print/tms320c40.html
28. Volder, J.E.: The CORDIC trigonometric computing technique. IRE Trans. on Electronic Computers **EC-8**(3), 330–334 (1959)
29. Walther, J.S.: A unified algorithm for elementary functions. In: Spring Joint Computer Conf. (1971)
30. Whitby-Strevens, C.: Transputers-past, present and future. IEEE Micro **10**(6), 16–19, 76–82 (1990)
31. Yeo, H., Hu, Y.: A novel modular systolic array architecture for full-search block matching motion estimation. IEEE Trans. Circuits Syst. Video Technol. **5**(5), 407–416 (1995)

Compiling for VLIW DSPs

Christoph W. Kessler

Abstract This chapter describes fundamental compiler techniques for VLIW DSP processors. We begin with a review of VLIW DSP architecture concepts, as far as relevant for the compiler writer. As a case study, we consider the TI TMS320C6x™ clustered VLIW DSP processor family. We survey the main tasks of VLIW DSP code generation, discuss instruction selection, cluster assignment, instruction scheduling and register allocation in some greater detail, and present selected techniques for these, both heuristic and optimal ones. Some emphasis is put on phase ordering problems and on phase coupled and integrated code generation techniques.

1 VLIW DSP Architecture Concepts and Resource Modeling

In order to satisfy high performance demands, modern processor architectures exploit various kinds of parallelism in programs: *thread-level parallelism* (i.e., running multiple program threads in parallel on multi-core and/or hardware-multithreaded processors), *data-level parallelism* (i.e., executing the same instruction or operation on several parts of a long data word or on a vector of multiple data words together), *memory-level parallelism* (i.e., overlapping memory access latency with other, independent computation on the processor), and *instruction-level parallelism* (i.e., overlapping the execution of several instructions in time, using different resources of the processor in parallel at a time).

By pipelined execution of subsequent instructions, a certain amount of instruction level parallelism (ILP) can already be exploited in ordinary sequential RISC processors that issue a single instruction at a time. More ILP can often be leveraged by *multiple-issue* architectures, where execution of several independent instructions can be started in parallel, resulting in a higher throughput (instructions per clock cycle, IPC). The maximum number of instructions that can be issued simultaneously

C. W. Kessler (✉)
Department of Computer Science (IDA), Linköping University, Linköping, Sweden
e-mail: christoph.kessler@liu.se

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_27

is called the *issue width*, denoted by ω. In this chapter, we focus on multiple-issue instruction-level parallel DSP architectures, i.e., $\omega > 1$.

ILP in programs can either be given explicitly or implicitly. With *implicit ILP*, dependences between instructions are implicitly given in the form of register and memory addresses read and written by instructions in a sequential instruction stream. It is the task of a run-time (usually hardware) scheduler to identify instructions that are independent and do not compete for the same resource. Such instructions could then be issued in parallel to different available functional units of the processor. Superscalar processors use a hardware scheduler to analyze data dependences and resource conflicts on-the-fly within a given fixed-size window over the next instructions in the instruction stream. While convenient for the programmer, superscalar processors require high energy and silicon overhead for analyzing dependences and dispatching instructions.

With *explicit ILP*, the assembler-level programmer or compiler is responsible to identify independent instructions that should execute in parallel, and group them together into *issue packets* (also known as *instruction groups* e.g. in the Intel Itanium IA-64 processor, see e.g. [104]). The elementary instructions in an issue packet will be dispatched simultaneously to different functional units for parallel execution. In the following, we will consider explicit ILP architectures.

The issue packets do not necessarily correspond one-to-one to the units of instruction fetch. The processor's instruction fetch unit usually reads properly aligned, fixed-sized blocks of bytes from program memory, which contain a fixed number of elementary instructions, and decodes them together. We refer to these blocks as *fetch packets* (also known as *instruction bundles* in the Itanium IA-64 literature). For instance, a fetch packet for the Itanium IA-64 processor family contains three instructions, and fetch packets for the TI 'C62x contain eight instructions.

In the traditional VLIW architectures (see Fig. 1), the issue packets coincide with the fetch packets; they have a fixed length of L bytes that are L-byte aligned in instruction (cache) memory, and are called *Very Long Instruction Words (VLIWs)*. A VLIW contains $\omega > 1$ predefined slots for elementary instructions. Each instruction slot may be dedicated to a certain kind of instructions or to controlling a

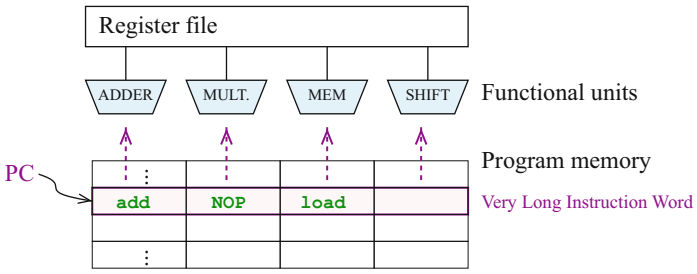


Fig. 1 A traditional VLIW processor with very long instruction words consisting of four issue slots, each one controlling one functional unit

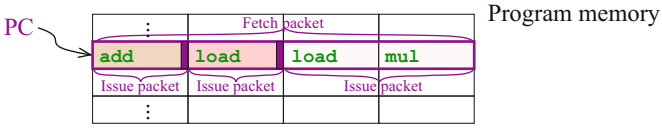


Fig. 2 Several issue packets may be accommodated within a single fetch packet. Here, the framed fetch packet contains three issue packets: the first two contain just one elementary instruction each, while the third one contains two parallel instructions

specific functional unit of the processor. Not all instruction slots have to be used; unused slots are marked by `NOP` (no operation) instructions. While decoding is straightforward, code density can be low if there is not enough ILP to fill most of the slots; this wastes program memory space and instruction fetch bandwidth.

Instead, most explicit ILP architectures nowadays allow to pack and encode instructions more flexibly in program memory. An instruction of specific kind may be placed in several or all possible instruction slots of a fetch packet. Also, a fetch packet may accommodate several issue packets, as illustrated in Fig. 2; the boundaries between these may, for instance, be marked by special delimiter bits. The different issue packets in a fetch packet will be issued subsequently for execution. The hardware is responsible for extracting the issue packets from a stream of fetch packets.[1] In the DSP domain, the Texas Instruments TI TMS320C6x processor family [97] uses such a flexible encoding schema, which we will present in Sect. 2.

The existence of multiple individual RISC-like elementary instructions as separate slots within an issue packet to express parallel issue is a key feature of VLIW and EPIC architectures. In contrast, consider dual-MAC (multiply-accumulate) instructions that are provided in some DSP processors, but encoded as a single instruction (albeit a very powerful one) in a linear instruction stream. Such instructions are, by themselves, not based on VLIW but should rather be considered as a special case of SIMD (single instruction multiple data) instructions. Indeed, SIMD instructions can occur as elementary instructions in VLIW instruction sets. Generally, a *SIMD instruction* applies the same arithmetic or logical operation to multiple operand data items in parallel. These operand items usually need to reside in adjacent registers or memory locations to be treated and addressed as single long data words. Hence, SIMD instructions have only one opcode, while issue packets in VLIW/EPIC architectures have one opcode per elementary instruction.

The appropriate issue width and the number of parallel functional units for a VLIW processor design depends, beyond architectural constraints, on the characteristics of the intended application domain. While the average ILP degree achievable in general-purpose programs is usually low, it can be significantly higher in the computational kernels of typical DSP applications. For instance, Gangwar et al. [43] report for DSPstone and Mediabench benchmark kernels an achievable ILP degree

[1] Processors that decouple issue packets from fetch packets are commonly also referred to as *Explicitly Parallel Instruction set Computing* (EPIC) architectures.

of 20 on average for a (clustered) VLIW architecture with 16 ALUs and 8 load-store units. Moreover, program transformations can be applied to increase exploitable ILP; we will discuss some of these later in this chapter.

1.1 Resource Modeling

We model instruction issue and resource usage explicitly. An instruction i issued at time t occupies an *issue slot* (e.g., a slot in a VLIW) at time t and possibly[2] several resources (such as functional units or buses) at time t or later.

For each instruction type, its required resource reservations relative to the issue time t are specified in a *reservation table* [26], a boolean matrix where the entry in row j and column u indicates if the instruction uses resource u in clock cycle $t + j$.

If an instruction is issued at a time t, its reservations of resources are committed to a *global resource usage map* or *table*. Two instructions are in conflict with each other if their resource reservations overlap in the global resource usage map; this is also known as a *structural hazard*. See Fig. 3 for an example. In such a case, one of the two instructions has to be issued at a later time to avoid duplicate reservations of the same resource.

Non-pipelined resources that have to be reserved for more than 1 clock cycle in sequence can thus lead to delayed issuing of subsequent instructions that should use the same resource. The *occupation time* $o(i, j)$ denotes the minimum distance in issue time between two (data-independent) instructions i and j that are to be issued subsequently on the same issue unit and that subscribe to a common resource. Hence, the occupation time only depends on the instruction types. For instance, in Fig. 3, $o(\texttt{add}, \texttt{add}) = 1$. In fact, for most processors, occupation times are generally 1.

Sets of time slots on one or several physical resources (such as pipeline stages in functional units or buses) can often be modeled together as a single virtual resource. This can be done if an analysis of the instruction set shows that, once an instruction is assigned the earliest one of the resource slots in this subset, no other instruction could possibly interfere with it in later slots or with other resources in the same subset.

A processor is called *fully pipelined* if it can be modeled with virtual resources such that all occupation times are 1, there are no exposed structural hazards, and the reservation table for an instruction thus degenerates to a vector over the virtual resources. On regular VLIW architectures, these virtual resources often correspond one-to-one to functional units.

[2] `NOP` (no operation) instructions only occupy an issue slot but no further resources.

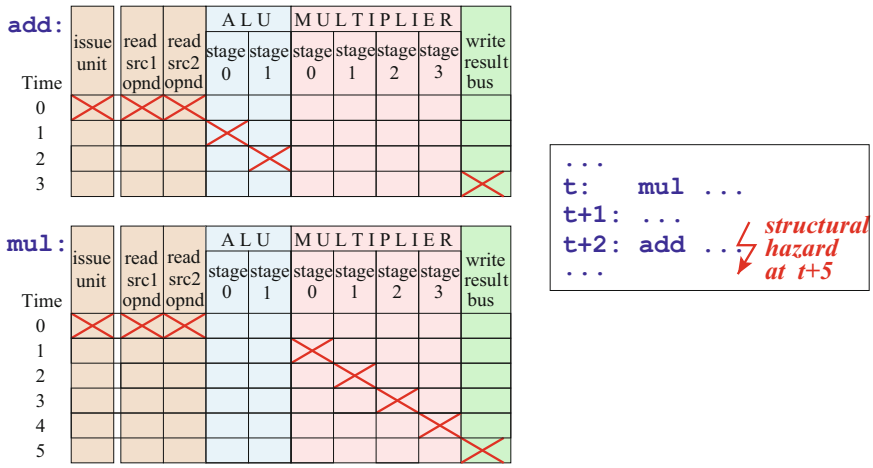


Fig. 3 Left: Example reservation tables for addition and multiplication on a pipelined processor with an ALU and multiplier unit. Resources such as register file access ports and pipeline stages on the functional units span the horizontal axis of the reservation tables while time flows downwards. Time slot 0 represents the issue time. Right: Scheduling an `add` instruction 2 clock cycles after a `mul` instruction would lead to conflicting subscriptions of the result resource (write back to register file). Here, the issue of `add` would have to be delayed to, say, time $t + 3$. If exposed to the programmer/compiler, a `nop` instruction could be added before the `add` to fill the issue slot at time $t + 2$. Otherwise, the processor will handle the delay automatically by stalling the pipeline for one cycle

1.2 Latency and Register Write Models

Consider two instructions i_1 and i_2 where i_1 issued at time t_1 produces (writes) a value so that it is available at the beginning of time slot $t_1 + \delta_{w1}$ in some register r, which is to be consumed (read) by i_2 at time $t_2 + \delta_{r2}$. The time of writing the result relative to the issue time, δ_{w1} is called the *write latency*[3] of i_1, and δ_{r2} the *read latency* of i_2. For the earliest possible issue time t_2 of i_2 we have to preserve the constraint

$$t_2 \geq t_1 + \delta_{w1} - \delta_{r2}$$

to make sure that the operand value for i_2 is available in the register.

We refer to the minimum difference in issue times induced by data dependence, i.e.,

[3]For simplicity of presentation, we assume here that write latency and read latency are constants for each instruction. In general, they may in some cases depend on run-time conditions exposed by the hardware and vary in an interval between earliest and latest write resp. read latency. See also our remarks on the LE model further below. For a more detailed latency model, we refer to Rau et al. [93].

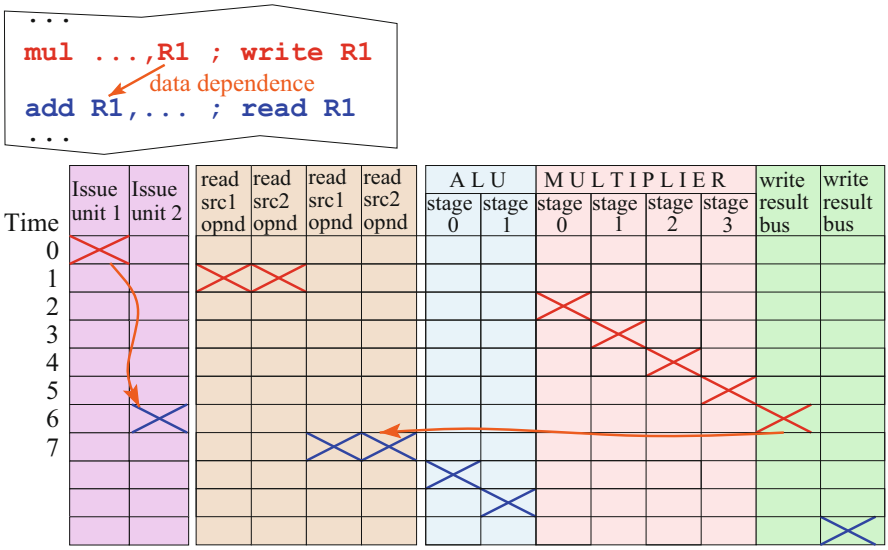


Fig. 4 A read-after-write (flow) data dependence forces the instruction scheduler to await the latency of 6 clock cycles between the producing and consuming instruction to make sure that the value written to register `R1` is read

$$\ell(i_1, i_2) = \delta_{w1} - \delta_{r2}$$

as *latency* between i_1 and i_2. See Fig. 4 for an illustration. For memory data dependences between store and load instructions, latency is defined accordingly. The difference $\ell(i_1, i_2) - o(i_1, i_2)$ is usually referred to as the *delay*[4] of instruction i_1.

Latencies are normally positive, because operations usually read operands early in their execution and write results just before terminating. For the same reason, the occupation time usually does not exceed the latency. Only for uncommon combinations of an early-writing i_1 with a late-reading i_2, or in the case of write-after-read dependences, negative latencies could occur, which means that a successor instruction actually could be issued before its predecessor instruction in the data dependence graph and still preserve the data dependence. However, this only applies to the EQ model, which we now explain.

There exist two different latency models with respect to the result register write time: EQ (for "equal") and LE (for "less than or equal"). Both models are being used in VLIW DSP processors. The *EQ model* specifies that the result register of an instruction i_1 issued at time t_1 will be written exactly at the end of time slot $t_1 + \delta_{w1} - 1$, not earlier and not later. Hence, the destination register r only needs to be reserved from time $t_1 + \delta_{w1}$ on.

[4]Note that in some papers and texts, the meanings of the terms *delay* and *latency* are reversed.

In the *LE model*, $t_1 + \delta_{w1}$ is an upper bound of the write time, but the write could happen at any time between issue time t_1 and $t_1 + \delta_{w1}$, depending on hardware-related issues. In the LE model, the destination register r must hence be reserved already from the issue time on. The EQ model allows to better utilize the registers, but the possibility of having several in-flight result values to be written to the same destination register makes it more difficult to handle interrupts properly.

In some architectures, latency only depends on the instruction type of the source instruction. If the latency $\ell(i, j)$ is the same for all possible instructions j that may directly depend on i (e.g., that use the result value written by i) we set $\ell(i) = \ell(i, j)$. Otherwise, on LE architectures, we could instead set $\ell(i) = \max_j \ell(i, j)$, i.e., the maximum latency to any possible direct successor instruction consuming the output value of i. The assumption that latency only depends on the source instruction is then a conservative simplification and may lead in some cases to somewhat longer register live ranges than necessary.

1.3 Clustered VLIW: Partitioned Register Sets

In VLIW architectures, possibly many instructions may execute in parallel, each accessing several operand registers and/or producing a result value to be written to some register. If each instruction should be able to access each register in a homogenous register set, the demands on the number of parallel read and write ports to the register set, i.e., on the access bandwidth to the register set, become extraordinarily high. Register files with many ports have very high silicon area and energy costs, and even access latency grows.

A solution is to constrain general accessibility and partition the set of functional units and likewise the register set to form *clusters*. A cluster consists of a set of functional units and a local register set, see Fig. 5. Within a cluster, each functional unit can access each local register. However, the number of accesses to a remote register is strictly limited, usually to one per clock cycle and cluster. A task for

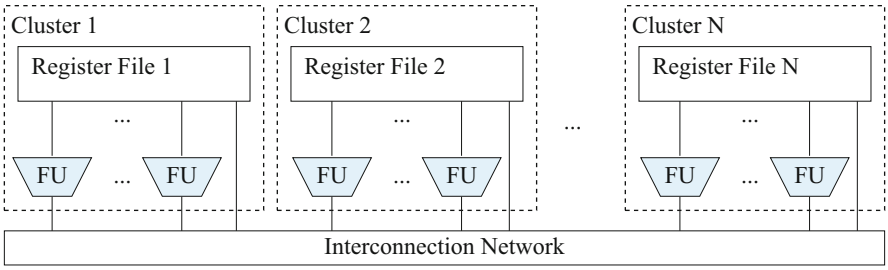


Fig. 5 Clustered VLIW processor with partitioned register set

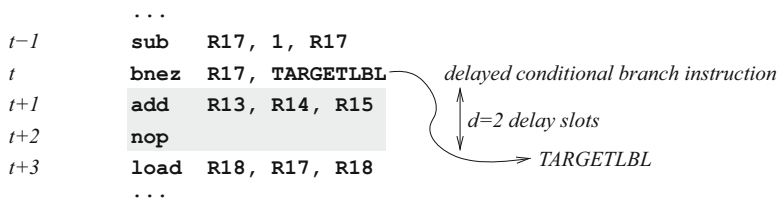


Fig. 6 A delayed branch with d delay slots takes effect only d clock cycles after the branch was executed. In this example, we have simple RISC code with a delayed conditional branch at position t with $d = 2$ delay slots. The first delay slot at position $t + 1$ could here be filled with an `add` instruction that the branch condition (`R17`$\neq 0$) does not depend on. For the second delay slot at position $t + 2$, a `nop` instruction has been inserted to fill the slot. The subsequent `load` instruction at position $t + 3$ will only execute if the branch was not taken

the programmer (or compiler) is thus to plan in which cluster data should reside at runtime and on which cluster each operation is to be performed to minimize the loss of ILP due to the clustering constraints.

1.4 Control Hazards

The pipelined execution of instructions on modern processors, including VLIW processors, achieves maximum throughput only in the absence of data hazards, structural hazards, and control hazards. In VLIW processors, these hazards are exposed to the assembler-level programmer or compiler. Data hazards and structural hazards have been discussed above.

Control hazards denote the fact that branch instructions may disrupt the linear fetch-decode-execute pipeline flow. Branch instructions are detected only in the decoding phase and the branch target may, in the case of conditional branches, be known even later during execution. If subsequent instructions have been fetched, decoded and partly executed on the "wrong" control flow branch when the branch is detected or the branch target is known, the effect of these instructions must be rolled back and the pipeline must restart from the branch target. This implies a non-zero delay in execution that may differ depending on the type of branch instruction (nonconditional branch, conditional branch taken as expected, or conditional branch not taken as expected). There are basically two possibilities how processors manage branch delays:

(1) *Delayed branch*: The branch instruction semantics is re-defined to take its effect on the program counter only after a certain number $d > 0$ of delay time slots, see also Fig. 6 for an example. It is a task for global instruction scheduling (see Sect. 7) to try filling these d branch delay slots with useful instructions that need to be executed anyway but do not influence the branch condition. If no other instructions can be moved to a branch delay slot, it has to be filled with a `NOP` instruction as placeholder.

```
          ...                                  ...
          sub   R17, 1, R17                    sub   R17, 1, R17
          bnez  R17, ELSELBL                   cmpne  R17, 0, P1
          store  R13, R17, R15       [P1]      store   R13, R15
          jump  NEXTLBL              [!P1]     load  R18, R17, R18
ELSELBL:load  R18, R17, R18                    ...
NEXTLBL:...
```

Fig. 7 Predication example. Left hand side: a simple RISC code implementing an if-then-else like computation, using one conditional branch and one unconditional branch instruction (which are not delayed here, for simplicity). Right hand side: An equivalent predicated code. By the compare instruction (`cmpne`), the branch condition (a boolean value) is evaluated and written into a predicate register, here `P1`. The subsequent two instructions (a `load` and a `store`) are both issued and executed, but take effect only if their guarding predicate (`[P1]` and `[!P1]` respectively) evaluates to true

(2) *Pipeline stall*: The entire processor pipeline is frozen until the first instruction word has been loaded from the branch target. The delay is not explicit in the program code and may vary depending on the branch instruction type.

In particular, conditional branches have a detrimental effect on processor throughput. For this reason, hardware features and code generation techniques that allow to reduce the need for (conditional) branches are important. The most prominent one is *predicated execution*: Each instruction takes an additional operand, a boolean predicate, which may be a constant or a variable in a predicate register. If the predicate evaluates to true, the instruction executes as usual. If it evaluates to false, the effect of that instruction is rolled back such that it behaves like a `NOP` instruction. Figure 7 gives a simple example for predicated execution.

1.5 Hardware Loops

Many innermost loops in digital signal processing applications have a fixed number of iterations and a fixed-length loop body consisting of straight-line code. Some DSP processors therefore support a hardware loop construct. A special hardware loop setup instruction at the loop entry initializes an iteration count register and also specifies the number of subsequent instructions that are supposed to form the loop body. The iteration count register is advanced automatically after every execution of the loop body; no separate add instruction is necessary for that purpose. A backward branch instruction from the end to the beginning of the loop body is now no longer necessary either, as the processor automatically resets its program counter to the first loop instruction, unless the iteration count has reached its final value, see Fig. 8 for an example. Hardware loops have thus no overhead for loop control per loop iteration and only a marginal constant loop setup cost. Also, they do not suffer from control hazards, as the processor hardware knows well ahead of time where and whether to execute the next backward branch.

```
        ...                                  ...
        add  8192, R17 ; trip count in R17   repeat  2, 8192    ; loop count in LR
LOOPLBL:sub  R17, 1, R17                     load  R15, LR, R18
        load  R15, R17, R18                  store  R18, R16, LR
        store  R18, R16, R17                 ...
        bnez  R17, LOOPLBL
NEXTLBL:...
```

2 instructions

Fig. 8 Hardware loop example. Left hand side: A simple RISC code for an ordinary copying loop, using a conditional branch instruction (`bnez`) to reiterate if the loop count stored in register R17 has not reached value 0 yet. Right hand side: The loop has been rewritten using a hardware loop construct. The `repeat` instruction sets up a hardware loop consisting of the 2 subsequent instructions (`load`, `store`) and implicitly initializes a special loop count register LR to the loop trip count (8192). Decrementing LR and branching are implicit by `repeat`

1.6 Examples of VLIW DSP Processors

In the next section, we will consider the TI 'C6x DSP processor family as a case study. Other VLIW/EPIC DSP processors include, e.g., the HP Lx/STMicroelectronics ST200, Analog Devices TigerSHARC ADSP-TS20xS [5], NXP (formerly Philips Semiconductors) TriMedia [86], Qualcomm Hexagon [91] and Recore Xentium [94].

Due to their relatively low power and silicon area usage, VLIW DSP cores are also often used in low-power multi- and manycore architectures. For instance, multiple TI 'C66 DSP cores (and ARM Cortex A15 cores) are aggregated in the TI KeyStone II multicore system-on-chip architecture. Another example is Kalray MPPA-256 clustered manycore architecture: it is organized as a distributed memory architecture with 16 compute clusters connected by a network-on-chip; each compute cluster contains 16 VLIW (5-issue) DSP compute cores (plus one system core) sharing 2 MB cluster-local memory, where each compute core has a peak performance of 2.4GFlops (single precision) at only 600 MHz, amounting to an accumulated peak performance of 634GFlops at 25 W [28].

2 Case Study: TI 'C6x DSP Processor Family

As a case study, we consider TI 'C6201, a fixed-point digital signal processor (DSP) of the Texas Instrument's 'C62x™ / 'C64x™/ 'C66x™/ 'C67x™ family of clustered VLIW DSPs with the VelociTI™ instruction set. We also shortly mention SIMD support in 'C64x and floatingpoint support in 'C66x/'C67x; a detailed treatment of floatingpoint issues is however beyond the scope of this section. Finally, we also briefly describe TI's programming models for TI 'C6x DSPs.

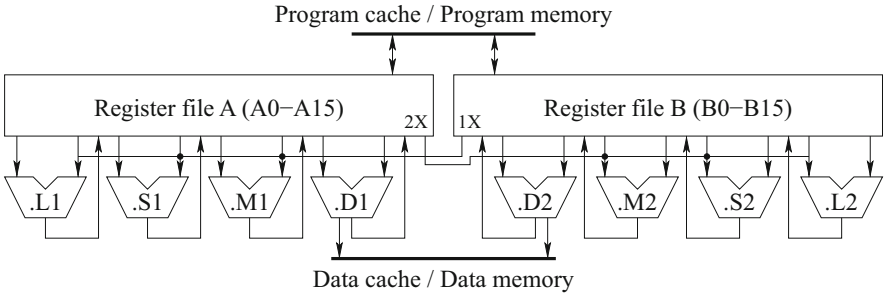


Fig. 9 The TI 'C6201 clustered VLIW DSP processor

2.1 TI 'C6201 DSP Processor Architecture

The Texas Instruments TI TMS320C6201™[97] (shorthand: 'C6201) is a high-performance fixed-point digital signal processor (DSP) clocked at 200 MHz. It is a clustered VLIW architecture with issue width $\omega = 8$. A block diagram is given in Fig. 9. The 'C6201 has 128 KB on-chip static RAM, 64 KB for data and 64 KB for instructions.

The 'C6201 has eight functional units, including two 16-bit multipliers for 32-bit results (the .M-units) and six 32/40-bit arithmetic-logical units (ALUs), of which two (the .D-units) are connected to on-chip data cache memory. The 'C62x CPUs are load-store architectures, i.e., all operands of arithmetic and logical operations must be constants or reside in registers, but not in memory. The data addressing (.D) units are used to load data from (data) memory to registers and store register contents to (data) memory. The load and store instructions exist in variants for 32-bit, 16-bit and 8-bit data. The two .L units (logical units) mainly provide 32-bit and 40-bit arithmetic and compare operations and 32-bit logical operations like and, or, xor. The two .S units (shift units) mainly provide 32-bit arithmetic and logical operations, 32-bit bit-level operations, 32-bit and 40-bit shifts, and branching. Some instructions are available on several units. For instance, additions can be done on the .L units, .S units and .D units.

The 'C62x architecture is fully pipelined. The reservation table of each instruction[5] is a 10×1 matrix, consisting of eight slots for the eight functional units and the two cross paths `1X` and `2X` (which will be described later) at issue time. In particular, each instruction execution occupies exactly one of the functional units at

[5]Exception: For load and store instructions, two more resources are used to model load destination register resp. store source register access to the two register files, as only one loaded or stored register can be accessed per register file and clock cycle. Furthermore, load instructions can cause additional implicit delays (pipeline stalls) by unbalanced access to the internal memory banks (see later). This effect could likewise be modeled with additional resources representing the different memory banks. However, this will only be useful for predicting stalls where the alignment of the accessed memory addresses is statically known.

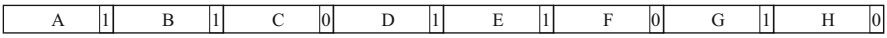


Fig. 10 A fetch packet for the 'C62x can contain up to eight issue packets, as marked by the chaining bits. In this example, there are three issue packets: instructions A, B, C issued together, followed by D, E, F issued together and finally G and H issued together

issue time. Separate slots for modeling instruction issue are thus not required, they coincide with the slots for the corresponding functional units. The occupation time is 1 for all instructions.

The 'C62x architecture offers the EQ latency model (there it is called "multiple assignment") for non-interruptable code, while the LE model (called "single assignment") should be used for interruptable code. Global enabling and disabling of interrupts is done by changing a flag in the processor's control status register.

The latency for load instructions is 5 clock cycles, for most arithmetic instructions it is 1, and for multiply 2 clock cycles. Load and store instructions may optionally have an address autoincrement or -decrement side effect, which has latency one.

Each of the two clusters *A* and *B* has sixteen 32-bit general purpose registers, which are connected to four units (including one multiplier and one load-store unit). The units of Cluster *A* are called .D1, .M1, .S1 and .L1, those of Cluster *B* are called .D2, .M2, .S2 and .L2. All units are fully pipelined (with occupation time 1), i.e., in principle, a new instruction could be issued to each unit in each clock cycle.

An instruction fetch packet for the 'C62x family is 256 bit wide and is partitioned into 8 instruction slots of 32 bit each. The least significant bit position of a 32-bit instruction slot is used as a *chaining bit* to indicate the limits of issue packets: If the chaining bit of slot i is 0, the instruction in the following slot $(i + 1)$ belongs to the next issue packet (see Fig. 10). Technically, issue packets cannot span several fetch packets.[6] Hence, the maximum issue packet size of $\omega = 8$ occurs when all chaining bits (except perhaps the last one) are set in a fetch packet. As the other extreme, up to eight issue packets could occur in a fetch packet (if all chaining bits are cleared). The next fetch packet is not fetched before all issue packets of the previous one have been dispatched.

As each functional unit can do simple integer operations like addition, the 'C6201 can thus run up to eight integer operations per cycle, which amounts to 1600 MIPS (million instructions per second).

The ADD2 instruction, which executes on .S units, allows to perform a pair of 16-bit additions in a single clock cycle on the same functional unit, if the 16-bit operands (and results) each are packed into a common 32-bit register, see Fig. 11. One of these two 16-bit additions accesses the lower 16 bit (bits 0..15) of the

[6]Even though 'C62x assembly language allows an issue packet to start in a fetch packet and continue into the next one, the assembler will automatically create and insert a fresh fetch packet after the first one, move the pending issue packet there, and fill up the remainder of the first issue packet with `NOP` instructions.

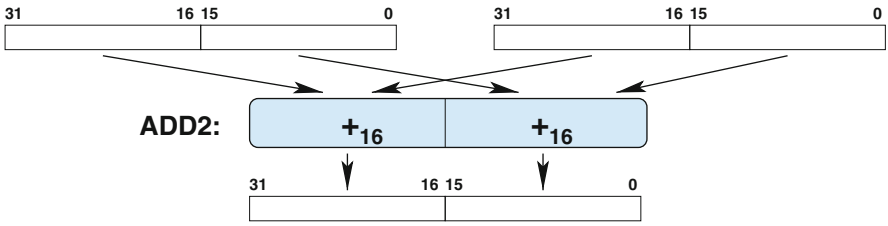


Fig. 11 The SIMD instruction ADD2 performs two 16-bit integer additions on the same functional unit in 1 clock cycle. The 32-bit registers shared by the operands and results are shown as rectangles

registers, the other the higher 16 bit (bits 16..31). No carry is propagated from the lower to the higher 16-bit addition, which differs from the behavior of the 32-bit ADD instruction and therefore requires the separate opcode ADD2. The SUB2 instruction, also available on the .S units, works similarly for two 16-bit subtractions. Other instructions like bitwise AND, bitwise OR, etc. work for 16-bit operand pairs in the same way as for 32-bit operands and thus do not need a separate opcode.

Within each cluster, each functional unit can access any register. At most one instruction per cluster and clock cycle can take one operand from the other cluster's register file, for which it needs to reserve the corresponding *cross path* (`1X` for accessing B registers from cluster A, and `2X` for the other way), which is also modeled as a resource for this purpose. Assembler mnemonics encode the used resources as a suffix to the instruction's opcode: For instance, `ADD.L1` is an ordinary addition on the .L1 unit using operands from cluster A only, while `ADD.S2X` denotes an addition on the .S2 unit that accesses one A register via the cross path `2X`. In total, there are twelve different instructions for addition (not counting the ADD2 option for 16-bit additions).

It becomes apparent that the problems of resource allocation (including cluster assignment) and instruction scheduling are not independent but should preferably be handled together to improve code quality. An example (adapted from Leupers [70]) is shown in Table 1: A basic block consisting of eight independent load (`LDW`) instructions is to be scheduled. The address operands are initially available (i.e., live on entry) in registers A0,...,A7 in register file A, the results are expected to be written to registers B0,...,B7 (i.e., live on exit) in register file B. Load instructions execute on the cluster containing the address operand register. The result can be written to either register file. However, only one load or store instruction can access a register file per clock cycle to write its destination register resp. read its source register; otherwise, the processor stalls for 1 clock cycle to serialize the competing accesses. Copying registers between clusters (which occupies the corresponding cross path) can be done by Move (`MV`), which is a shorthand for `ADD` with one zero operand, and has latency 1. As the processor has 2 load/store units and load has latency 5, a lower bound for the makespan (the time until all results are available) is 8 clock cycles; it can be sharpened to 9 clock cycles if we

Table 1 (a) schedule generated by an early version of the TI-C compiler (12 cycles) [70]; (b) optimal schedule generated by OPTIMIST with dynamic programming (9 cycles) [62]

(a)	(b)
LDW.D1 *A4,B4	LDW.D1 *A0,A8 || MV.L2X A1,B8
LDW.D1 *A1,A8	LDW.D2 *B8,B1 || LDW.D1 *A2,A9 || MV.L2X A3,B10
LDW.D1 *A3,A9	LDW.D2 *B10,B3 || LDW.D1 *A4,A10 || MV.L2X A5,B12
LDW.D1 *A0,B0	LDW.D2 *B12,B5 || LDW.D1 *A6,A11 || MV.L2X A7,B14
LDW.D1 *A2,B2	LDW.D2 *B14,B7 || MV.L2X A8,B0
LDW.D1 *A5,B5	MV.L2X A9,B2
LDW.D1 *A7,A4	MV.L2X A10,B4
LDW.D1 *A6,B6	MV.L2X A11,B6
NOP 1	NOP 1; (last delay slot of LDW to B7)
MV.L2X A8,B1	
MV.L2X A9,B3	
MV.L2X A4,B7	

consider that at least one of the addresses has to be moved early to register file B to enable parallel computing, which takes one more clock cycle. A naive solution (a) sequentializes the computation by executing all load instructions on cluster A only. A more advanced schedule (b) utilizes both load/store units in parallel by transporting four of the addresses to cluster B as soon as possible, so the loads can run in parallel. Note also that no implicit pipeline stalls occur as the parallel load instructions always target different destination register files in their write-back phase, 5 clock cycles after issue time. Indeed, (b) is an optimal schedule; it was computed by the dynamic programming algorithm in OPTIMIST [62]. Generally, there can exist several optimal schedules. For instance, another one for this example is reported by Leupers [70], which was computed by a simulated annealing based heuristic.

Branch instructions on the 'C62x, which execute on the .S units, are delayed branches with a latency of 6 clock cycles, thus 5 delay slots are exposed. If two branches execute in the same issue packet (on .S1 and .S2 in parallel), control branches to the target for which the branch condition evaluates to true. This can be used to realize three-way branches. If both branch conditions evaluate to true, the behavior is undefined.

All 'C62x instructions can be predicated. The four most significant bits in the opcode form a *condition field*, where the first three bits specify the condition register tested, and the fourth bit specifies whether to test for equality or non-equality of that register with zero. Registers A1, A2, B0, B1 and B2 can serve as condition registers. The condition field code 0000 denotes unconditional execution.

Usually, branch targets will be at the beginning of an issue packet. However, branch targets can be any word address in instruction memory and thereby any instruction, which may also be in the middle of an issue packet. In that case, the instructions in that issue packet that appear in the program text before the branch target address will not take effect (are treated as `NOP`s).

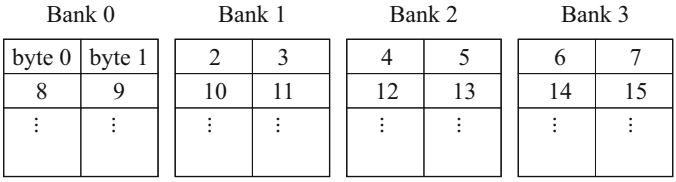


Fig. 12 Interleaved internal data memory with four memory banks, each 16 bit (2 bytes) wide

Most 'C62x processor types use interleaved memory banks for the internal (on-chip) data memory. In most cases, data memory is organized in four 16-bit wide memory banks, and byte addresses are mapped cyclically across these (see Fig. 12). Each bank is single-ported memory, thus only one access is possible per clock cycle. If two load or store instructions try to access addresses in the same bank in the same clock cycle, the processor stalls for one cycle to serialize the accesses. For avoiding such delays, it is useful to know statically the alignment of addresses to be accessed in parallel, and make sure that these end up in different memory banks. Note also that load-word (`LDW`) and store-word (`STW`) instructions, which access 32-bit data, access two neighbored banks simultaneously. Word addresses must be aligned on word boundaries, i.e., the two least significant address bits are zero. Halfword addresses must be aligned on halfword boundaries.

2.2 SIMD and Floatingpoint Support

All DSPs in the 'C6x family are based on the 'C6x instruction set and have a two-clustered VLIW architecture with 2×4 functional units. TI 'C62x and 'C64x processors are fixed point DSP processors, where the 'C64x processors have instruction set extensions that include, for instance, further support for SIMD processing (beyond ADD2, such as four-way 8 bit SIMD addition etc., four-way 16×16 bit multiply and eight-way 8×8 bit multiply), further instructions such as 32×32 bit multiply and complex multiply, compact (16-bit) instructions that can be mixed with 32-bit instructions [52], hardware support for software pipelining of loops, and more (2×32) registers.

The TI 'C66x and 'C67x DSP processor families also support floatingpoint computations,[7] by providing additional floatingpoint and complex data types, floatingpoint arithmetic instructions with same occupation time and latency as their fixed point counterparts, as well as instructions for fast conversion between fixed point and floatingpoint values. These extensions give more flexibility to the

[7] 'C66x and 'C67x support, for the basic arithmetic instructions, both single-precision and double-precision floatingpoint variants as defined by the IEEE 754 standard [98]. 'C66x combines the floatingpoint features of 'C67x with the advanced fixed point features of 'C64x.

programmer. Floatingpoint support in a DSP processor is very convenient if an early prototype code in C or similar high-level language using floatingpoint arithmetics is already given for a DSP problem at hand: the code can be compiled and executed as is, and it can be used as a base-line for further code modifications that can leverage fixed-point/floatingpoint performance trade-offs. Most DSP computations are, as long as the precision is sufficient, more efficient when implemented using fixed point computation, while there exist some operations such as calculating $1/x$ or $1/\sqrt{x}$ that execute faster on a floatingpoint representation. Hence, code switching considerately between fixed point and floatingpoint computing can lead to considerable speedups. For instance, TI [99] reports a 6.8x speedup by using mixed fixed point/single-precision floatingpoint code on 'C66x compared to fixed-point computation only on 'C64x for a loop calculating normalized values of the elements in an array of complex numbers, which involves calculating $1/\sqrt{x}$. 'C66x can calculate floatingpoint $1/\sqrt{x}$ by a single instruction, while 'C64x needs to invoke a library function with a software implementation for fixed point $1/\sqrt{x}$, which takes multiple clock cycles. For mixed code, fast conversions are essential. For example, 'C66x provides 2-way SIMD conversion instructions, for converting two single-precision (32-bit) floatingpoint values stored in registers into two 16-bit fixed point values stored in registers, or vice versa.

2.3 Programming Models

Beyond the 'C6x assembly language, TI provides three further *programming models* for 'C6x processors: (1) ANSI C, (2) C with calls to intrinsic functions that map one-to-one to 'C6x-specific instructions, such as `_add2()`, and (3) linear assembly code, which is RISC-like serial unscheduled code that uses 'C6x instructions, but assumes no resource conflicts and only unit latencies. In general, the more processor-specific programming models allow to generate more efficient code. For instance, for an IIR filter example, TI reports that the software pipeline (see Sect. 7.2) generated from plain C code has a kernel length of 5 clock cycles, from C with intrinsics only 4, while the linear assembly optimizer achieves 3 clock cycles and thus the best throughput [95].

3 VLIW DSP Code Generation Overview

In this section, we give a short overview of the main tasks in code generation that produce target-specific assembler code from a (mostly) target-independent intermediate representation of the program. We will consider these tasks and the main techniques used for them in some more detail in the following sections.

Most modern compilers provide not just one but several *intermediate representations* (IR) of the program module being translated. These representations

differ in their level of abstraction and degree of language independence and target independence. High-level representations such as abstract syntax trees follow the syntactic structure of the programs and represent e.g. loops and array accesses explicitly, while these constructs are, in low-level representations, lowered to branches and pointer arithmetics, respectively; such low-level IRs include control flow graphs, three-address code or quadruple sequences. A compiler supporting several different representations allows the different program analyses, optimizations and transformations to be implemented each on the level that is most appropriate for it. For instance, *common subexpression elimination* is best performed on a lower-level representation because more common subexpressions can be found after array accesses and other constructs have been lowered.

Code generation usually starts from a low-level intermediate representation (LIR) of the program. This LIR may be, to some degree, target dependent. For instance, IR operations for which no equivalent instruction exists on the target (e.g., there is no division instruction on the 'C62x) are lowered to equivalent sequences of LIR operations or to calls to corresponding routines in the compiler's run-time system.

For simple target architectures, the main tasks in code generation include instruction selection, instruction scheduling and register allocation:

- *Instruction selection* maps the abstract operations of the IR to equivalent instructions of the target processor. If we associate fixed resources (functional units, buses etc.) to be used with each instruction, this also includes the *resource allocation* problem. Details will be given in Sect. 4.
- *Instruction scheduling* arranges instructions in time, usually in order to minimize the overall execution time, subject to data dependence, control dependence and resource constraints. In particular, this includes the subproblems of *instruction sequencing*, i.e., determining a linear (usually, topological) order of instructions for scheduling, and *code compaction*, i.e. determining which independent instructions to execute in parallel and mapping these to slots in instruction issue packets and fetch packets. Local, loop-level and global instruction scheduling methods will be discussed in Sect. 7.
- *Register allocation* selects which values should, at what time during execution, reside in *some* target processor register. If there may not be enough registers available at a time, some values must be temporarily *spilled* to memory, which requires the generation and scheduling of additional spill code in the form of load and store instructions. *Register assignment* maps the run-time values that were allocated a register to a concrete register number, which is a simpler problem than register allocation. Details will be given in Sect. 6.

Advanced architectures such as clustered ones may require additional tasks in code generation, in particular *cluster assignment* and data transfer generation (see Sect. 5), which may be considered separately or be combined with some of the above tasks. For instance, cluster assignment for instructions could be considered part of instruction selection, and cluster assignment for data may be modeled as an extension of register allocation.

Another task that is typical for DSP processors is that of *address code generation* for address generation units (AGUs). AGUs provide auto-increment and auto-

decrement functionality as a parallel side effect to ordinary instructions that use special address registers for accessing data in memory. The AGUs may provide fixed offset values or offset registers to be used for in-/decrementing. A compiler could thus assign address registers and select offsets in an attempt to minimize the amount of residual addressing code that would still be computed with ordinary processor instructions on the main functional unit. See Section 3.1 in the chapter on *C Compilers and Code Optimizations for DSPs* [40] in the previous (second) edition of this book for further details.

Further code generation problems frequently occurring with VLIW DSPs include exploiting available SIMD instructions, which can be regarded a subproblem of instruction selection, and optimizing memory data layout to avoid stalls caused by memory bank access conflicts. Also here we refer to the above-mentioned chapter [40], Sects. 3.3 and 3.4, for a discussion of SIMD code generation and optimization of memory bank assignment, respectively.

4 Instruction Selection and Resource Allocation

The instruction selection problem is often modeled as a pattern matching problem. For each available instruction of the target processor, the compiler writer describes its semantics as a pattern consisting of IR operations that is considered equivalent. Then, the IR is to be covered completely with such patterns, where each IR operation has to be covered by exactly one pattern node. Some examples are shown in Fig. 13. As intermediate results corresponding to inner edges of a multi-node pattern are no longer accessible in registers if that instruction is selected, such a pattern is only applicable as a cover if no other operations (such as `SUB` in Fig. 13b) access such an intermediate result. In other words, all outgoing edges from IR nodes covered by non-root pattern nodes must be covered by pattern edges.

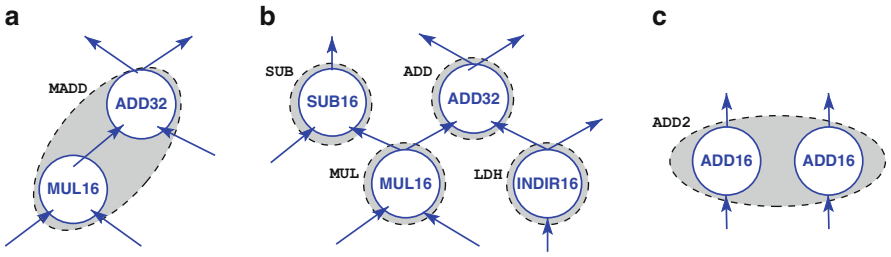


Fig. 13 Examples for covering IR nodes (solid circles) and edges (arrows) with patterns (dashed ovals) corresponding to instructions. (**a**) The pattern of a multiply-add (`MADD`) instruction covers two IR nodes, a 32-bit addition operation being the only consumer of the result of a 16-bit multiplication operation. (**b**) Here, covering by the `MADD` pattern is not applicable as the intermediate product value is also used by the 16-bit subtraction operation. (**c**) The pattern of a 2-way 32-bit SIMD-add instruction (`ADD2`) may cover two independent 16-bit addition operations

Each pattern is associated with a *cost*, which is typically its occupation time or its latency as an a-priori estimation of the instruction's impact on overall execution time in the final code (the exact impact will only be known after the remaining tasks of code generation, in particular instruction scheduling, have been done). But also other cost metrics, such as register space requirements, are possible. The optimization goal is then to minimize the accumulated total cost of all covering patterns, subject to the condition that each IR operation is covered by exactly one pattern node.

The optimizing pattern matching problem can be solved in various ways. A common technique, *tree parsing*, is applicable if the patterns are tree-shaped and the data flow graph of the current basic block (for instruction selection, we usually consider one basic block of the input program at a time) is a *tree*. The patterns are modeled as tree rewrite rules in a tree grammar describing admissible derivations of coverings of the input tree. A heuristic solution could be determined by a LR-parser that selects, in each step of constructing bottom-up a derivation the input tree, in a greedy way whenever there are several applicable rules (patterns) to choose from [45]. An optimal solution (i.e., a minimum-cost covering of the tree with respect to the given costs) can be computed in polynomial time by a dynamic programming algorithm that keeps track of all possibly optimal coverings in a subtree [1, 41].

Computing a minimum cost covering for a directed acyclic graph (DAG) is assumed to be NP-complete, but by splitting the DAG into trees processed separately and forcing the shared nodes' results to be stored in registers, dynamic-programming based bottom-up tree pattern matching techniques can still be used as heuristic methods. Ertl [37] gives an algorithm to check if a given processor instruction set (i.e., tree grammar) belongs to a special class of architectures (containing e.g. MIPS and SPARC processors), where the constrained tree pattern matching always produces optimal coverings for DAGs.

Another way to compute a minimum cost covering, albeit a more expensive one, is to apply advanced optimization methods such as integer linear programming [11, 35, 72, 103], partitioned boolean quadratic programming [30] or constraint programming [56]. This may be an applicable choice if a similar technique is also used for solving other subtasks, such as register allocation or instruction scheduling, the basic block and the number of patterns are not too large, and a close integration between these tasks is desired to produce high-quality code. Furthermore, solving the problem by such general optimization techniques is by no means restricted to tree-shaped patterns or tree-shaped IR graphs. In particular, they work well with complex patterns, such as forest patterns and directed acyclic graph (DAG) patterns. Forest patterns are non-connected trees of IR operators and can be used, for instance, to model SIMD instructions, as in Fig. 13c. DAG patterns can model powerful instructions that imply an internal reuse of common IR subexpressions or operands, such as autoincrement load and store instructions or memory read-modify-write instructions. The advanced optimization methods can even handle cyclic IR structures, such as static single assignment (SSA) representation. Because covering several IR nodes with a complex pattern corresponds to merging these nodes, special care has to be taken with forest and DAG patterns to avoid the

creation of artificial dependence cycles by the matching, which could lead to non-schedulable code [29]. For a comprehensive survey and classification of instruction selection problems and techniques we refer to the recent book by Hjort-Blindell [55].

Instruction selection can be combined with *resource allocation*, i.e., the binding of instructions to executing resources. For some instructions, there may be no choice: For instance, on 'C62x, a multiply instruction can only be executed on a .M unit. In case that the same instruction can execute on different functional units, each with its own cost, latency and resource reservations, one could model these variants as different instructions that just share the pattern defining the semantics. Like instruction selection, resource allocation is often done before instruction scheduling, but a tighter integration with scheduling would be helpful because resource allocation clearly constrains the scheduler.

A natural extension of this approach is to also model cluster assignment for instructions as a resource allocation problem, and thus as extended instruction selection problem. However, cluster allocation has traditionally been treated as a separate problem; we will discuss it in Sect. 5.

Further target-level optimizations could be modeled as part of instruction selection. For instance, for special cases of IR patterns there could exist alternative code sequences that may be faster, shorter, or more appropriate for later phases in code generation. As an example, an ordinary integer multiplication by 2 can be implemented in at least three different ways (*mutations*): by a `MUL` instruction, maybe running on a separate multiplier unit, by an `ADD` instruction, and by a left-shift (`SHL`) by one, each having different latency and resource requirements. The ability to consider such mutations during instruction scheduling increases flexibility and can thus improve code quality [85].

Another extension of instruction selection concerns the identification of several independent IR operations with same operator and short operand data types that could be merged to utilize a SIMD instruction instead.

Also, the selection of short instruction formats to reduce code size can be considered a subproblem of instruction selection. While beneficial for instruction fetch bandwidth, short instruction formats constrain the number of operands, the size of immediate fields or the set of accessible registers, which can have negative effects on register allocation and instruction scheduling. For the 16-bit compact instructions of 'C64x, Hahn et al. [52] explore the trade-off between performance and code size.

5 Cluster Assignment for Clustered VLIW Architectures

Cluster assignment for clustered VLIW architectures can be done at IR level or at target level. It maps each IR operator or instruction, respectively, to one of the clusters. Also, variables and temporaries to be held in registers must be mapped to a register file. Indeed, a value could reside in several register files if appropriate data transfer instructions are added; this is also an issue of register allocation and instruction scheduling and typically solved later than instruction cluster assignment

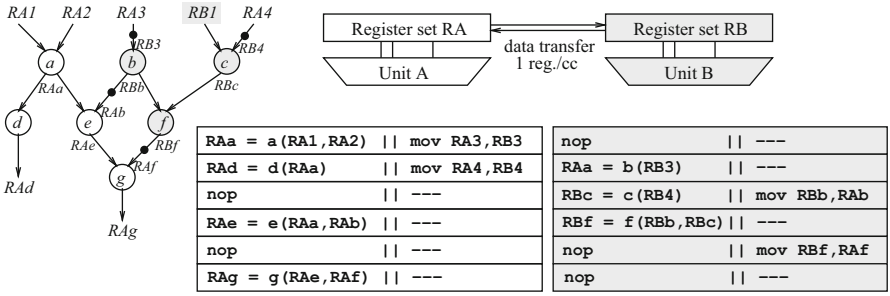


Fig. 14 Cluster assignment example. We consider a simple clustered architecture with two clusters, each with a register set and (for simplicity) one general-purpose functional unit that can only access local registers as operands. Given is the intermediate representation of a basic block in the form of a data flow graph. Some cluster assignment has been applied, which maps ingoing values $RA1,\ldots,RA4$ to register set RA, value $RB1$ (shaded) to register set RB, operations a, d, e and g to unit A, operations b, c and f (shaded) to unit B, and all outgoing values to register set RA. This cluster assignment implies data transfers between RA and RB along some data flow edges, which are marked by black dots. We assume that a data transfer takes 1 time step, and that at most one register value can be transfered per time step and direction, by using a `mov` instruction in parallel to a local operation. A possible schedule based on this cluster assignment is also shown. It has a makespan of 6 time steps; some slots are unused (`nop`) because no operation is data-ready at that time. Keeping all data and operations in a single cluster would require at least 7 steps. Note that the given cluster assignment is not optimal here; for instance, if b were computed on cluster A instead, the resulting code could be scheduled in 5 time steps

in most compilers, although there exist obvious interdependences. Figure 14 shows an example of a cluster assignment for a very simple two-cluster architecture, together with a possible schedule based on this clustering that exhibits the resulting data transfers and their impact on execution time.

There exist various techniques for cluster assignment for basic blocks. The goal is to minimize the number of transfer instructions, especially on the critical path(s). Usually, heuristic solutions are applied.

Ellis [33] gives a heuristic algorithm for cluster assignment called bottom-up greedy (BUG) for basic blocks and traces (see later) that is applied before instruction scheduling. Desoli [27] identifies sub-DAGs of the target-level dataflow graph that are mapped to a cluster as a whole. Gangwar et al. [43] first decompose the target-level dataflow graph into disjoint chains of nodes connected by dataflow edges. The nodes of a chain will always be mapped to the same cluster. Chains are grouped together by a greedy heuristic until there are as many chain groups left as clusters. Finally, chain groups are heuristically assigned to clusters so that the residual cross-chaingroup dataflow edges coincide with direct inter-cluster communication links wherever possible. For many-cluster architectures where no fully connected inter-cluster communication network is available, the algorithm tries to minimize the communication distance accordingly, such that communicating chain groups are preferably mapped to clusters that are close to each other.

Hierarchical partitioning heuristics are used e.g. by Aleta et al. [3] and Chu et al. [24]. Aleta et al. also consider replication of individual instructions in order to reduce the amount of communication.

Beg and van Beek [12] use constraint programming to solve the cluster assignment problem optimally for an idealized multi-cluster architecture with unlimited inter-cluster communication bandwidth.

Usually, cluster assignment precedes instruction scheduling in phase-decoupled compilers for clustered VLIW DSPs, because the resource allocation for instructions must be known for scheduling. On the other hand, cluster allocation could benefit from information about free communication slots in the schedule. The quality of the resulting code suffers from the separate handling of cluster assignment, instruction scheduling and register allocation. We will discuss phase-coupled and integrated code generation approaches for clustered architectures in Sect. 8.

6 Register Allocation and Generalized Spilling

In the low-level IR, all program variables that could be held in a register and all temporary variables are modeled as *symbolic registers*, which the register allocator then maps to the hardware registers, of which only a limited number is available.

A symbolic register s is *live* at a program point p if s is defined (written) on a control path from the procedure's entry point to p, and there exists a program point q where s is used (read) and s may not be (re-)written on the control path $p \ldots q$. Hence, s is live at p if it is used in a control flow successor q of p. The *live range* of s is the set of program points where s is live. The number of all symbolic registers live at a program point p is called the *register pressure* at p.

Live ranges could be defined on the low-level IR (if register allocation is to be done before instruction selection), but usually, they are defined at target instruction level, because instruction selection may introduce additional temporary variables to be kept in registers. If the schedule is given, the live ranges are fixed, which constrains the register allocator, and generated spill code has to be inserted into the schedule. If register allocation comes first, some pre-scheduling (sequencing) at LIR or target level is required to bring the operations/instructions of a basic block in a linear order that defines the live range boundaries. Early register allocation constrains the subsequent scheduler, but generated spill code will then be scheduled and compacted together with the remaining instructions.

Two live ranges *interfere* if they may overlap. Interfering live ranges cannot share the same register. The *live range interference graph* is an undirected graph whose nodes represent the live ranges of a procedure and edges represent interferences. Register assignment now means coloring the live range interference graph by assigning a color (i.e., a specific physical register) to each node such that interfering nodes have different colors (see Fig. 15 for a simple example). Moreover, the coloring must not use more colors than the number K of machine registers available. Determining if a general graph is colorable with K colors is NP-complete for $K \geq 3$. If a coloring cannot be found, the register allocator must restructure the program to make the interference graph colorable.

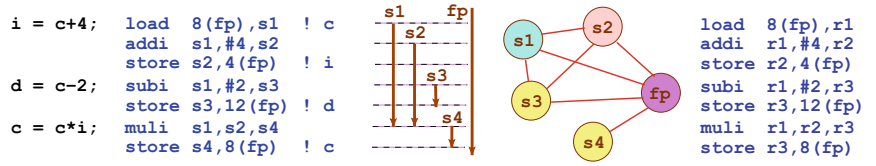


Fig. 15 Graph coloring example. For the C example code on the left hand side, equivalent RISC assembler pseudocode, using symbolic registers `s1`, `s2` etc. and the frame pointer register `fp`, is shown next to it. (`fp` is used in address calculations for stack-allocated local variables, here `i`, `d` and `c`.) The arrows in the center show how the live ranges for the symbolic registers overlap in time. To the right, we see the live range interference graph, including a vertex representing `fp`. The vertices are colored so that interfering live ranges do not get the same color (physical register). Here, `s3` and `s4` are assigned the same color, i.e., they will share a register (`r3`). The graph contains a 4-clique, involving live ranges `s1`, `s2`, `s3` and `fp`, hence at least 4 physical registers will be required in spill-free code. Code after register assignment is shown on the right hand side

Fig. 16 Coalescing example. In the pseudocode on the left hand side, we assume that the live ranges (symbolic registers) `s1` and `s2` do not interfere, i.e., `s2` is not accessed before the copy operation `s2 = s1` and `s1` not afterwards, such that the live ranges just touch each other at the copy operation. Right hand side: Coalescing `s1` and `s2` virtually merges both live ranges into one by forcing them to use the same physical register `r1`. The copy operation is eliminated

Chaitin [20] proposed a heuristic for coloring the interference graph with K colors, where K denotes the number of physical registers available. The algorithm works iteratively. In each step, it tries to find a node with degree $< K$, because then there must be some color available for that node, and removes the node from the interference graph. If the algorithm cannot find such a node, the program must be transformed to change the interference graph into a form that allows the algorithm to continue. Such transformations include coalescing, live range splitting, and spilling with rematerialization of live ranges. After the algorithm has removed all nodes from the interference graph, the nodes are colored in reverse order of removal. The optimistic coloring algorithm by Briggs [16] improves Chaitin's algorithm by delaying spilling transformations.

Coalescing is a transformation applicable to copy instructions $s_2 \leftarrow s_1$, where the two live ranges s_1, s_2 do not overlap except at that copy instruction, which marks the end (last use) of s_1 and beginning (write) of s_2. Coalescing merges s_1 and s_2 together to a single live range by renaming all occurrences of s_1 to s_2, which forces the register allocator to store them in the same physical register, and the copy instruction can now be removed. See Fig. 16 for an example.

Long live ranges tend to interfere with many others and thus may make the interference graph harder to color. As coalescing yields longer live ranges, it should

be applied with care. *Conservative coalescing* [17] merges live ranges only if the degree of the merged live range in the interference graph would still be smaller than the number of physical registers, K.

The reverse of coalescing is *live range splitting* i.e., insertion of register-to-register copy operations and renaming of accesses to split a long live range into several shorter ones. Splitting can make an interference graph colorable without having to apply spilling; this is often more favorable, as register-to-register copying is faster and less energy consuming than memory accesses. Live range splitting can be done considerately with a small number of sub-live-ranges, or aggressively, with one sub-live-range per access.

Spilling removes a live range as symbolic register, by storing its value in main memory (or other non-register location). For each writing access, a store instruction is inserted that saves the value to a memory location (e.g., in the variable's "home" location or in a temporary location on the stack), and for each reading access, a load instruction to a temporary register is inserted. This spill code leads to increased execution time, energy consumption and code size. In some cases, it could be more efficient to realize the *rematerialization* [17] of a spilled value not by an expensive load from memory, but by recomputing it instead. The choice between several spill candidates could be made greedily by considering the *spill cost* for a live range s, which contains the number of required store and load (or other rematerialization) instructions (to model the code size penalty), often also weighted by predicted execution frequencies (to model the performance and energy penalty).

A live range may not have to be spilled everywhere in the program. For instance, even if a symbolic register has a long live range, it may not be accessed during major periods in its live range where register pressure is high, for instance during an inner loop. Such periods are good candidates for *partial spilling*.

Register allocation can be implemented as a two-step approach [6, 29], where a global *pre-spilling* phase is run first to limit the remaining register pressure at any program point to the available number of physical registers, which makes the subsequent coloring phase easier.

Coloring-based heuristics are used in many standard compilers. While just-in-time compilers and dynamic optimizations require fast register allocators such as linear scan allocators [89, 100], the VLIW DSP domain rather calls for static compilation with high code quality, which justifies the use of more advanced register allocation algorithms. The first register allocator based on integer linear programming was presented by Goodwin and Wilken [47].

Optimal spilling selects just those live ranges for spilling whose accumulated spill cost is minimal, while making the remaining interference graph colorable. Optimal selection of spill candidates (pre-spilling) and optimal a-posteriori insertion of spill code for a given fixed instruction sequence and a given number of available registers are NP-complete even for basic blocks and have been solved by dynamic programming or integer linear programming for various special cases of processor architecture and dependency structure [6, 57, 58, 80]. In most compilers, heuristics are used that try to estimate the performance penalty of inserted load and store instructions [13]. More recently, several practical methods based on integer linear

programming for general optimal pre-spilling and for optimal coalescing have been developed, e.g., by Appel and George [6].

Another more recent trend is towards performing register allocation on the SSA form: For SSA programs, the interference graph belongs to the class of chordal graphs, which can be K-colored in quadratic time [14, 18, 51]. The generation of optimal spill code and minimization of copy instructions by coalescing remain NP-complete problems also for SSA programs. For the problem of optimal coalescing in spill-free SSA programs, a good heuristic method was proposed by Brisk et al. [18], and an optimal method based on integer linear programming was given by Grund and Hack [50]. *Ultimate coalescing* [19] considers all copy-related live ranges for coalescing that do not interfere as they hold the same value, as is the case in SSA-based IRs. For optimal pre-spilling in SSA programs, Ebner [29] models the problem as a constrained min-cut problem and applies a transformation that yields a polynomial-time near-optimal algorithm that does not rely on an integer linear programming solver.

7 Instruction Scheduling

In this section, we review fundamental instruction scheduling methods for VLIW processors at basic block level, loop level, and global level. We also discuss the automatic generation of the most time consuming part of instruction schedulers from a formal description of the processor.

7.1 Local Instruction Scheduling

The *control flow graph* at the IR level or target level representation of a program is a graph whose nodes are the IR operations or target instructions, respectively, and its edges denote possible control flow transitions between nodes.

A *basic block* is any (maximum-length) sequence of textually consecutive operations (at IR level) or instructions (at target level) in the program that can be entered by control flow only via the first and left only via the last operation or instruction, respectively. Hence, branch targets are always at the entry of a basic block, and a basic block contains no branches except maybe its last operation or instruction.

Control flow executes all operations of a basic block from its entry to its exit. Hence, the data dependences in a basic block form a directed acyclic graph, the *data flow graph* of the basic block. This data flow graph defines the partial execution order that constrains instruction scheduling: The instruction/operation at the target of a data dependence must not be issued before the latency of the instruction/operation at the source has elapsed. Leaf nodes in the data flow graph do not depend on any other node and have therefore no predecessor (within the basic block), root nodes have no successor node (within the basic block).

A path from a leaf node with maximum accumulated latency over its edges towards a root node is called a *critical path* of the basic block; its length is a lower bound for the makespan of any schedule for the basic block.

Methods for instruction scheduling for basic blocks (i.e., local scheduling) are simpler than global scheduling methods, because control flow in basic blocks is straightforward and can be ignored. Only data dependences and resource conflicts need to be taken into account. Interestingly, most basic blocks in real-world programs are quite small and consist of only a few instructions. However, program transformations, such as function inlining, loop unrolling or predication, can yield considerably larger basic blocks.

Traditionally, heuristic methods have been considered for local instruction scheduling, mostly because of fast optimization times. A simple and well-known heuristic technique is *list scheduling*.

List scheduling [49] is based on topological sorting of the operations or instructions in the basic block's data flow graph, taking the precedence constraints by data dependences into account and using a heuristic ordering to decide priorities in the case of multiple possible choices. The algorithm schedules nodes iteratively and maintains a list of data-ready nodes, the *ready list*. Initially, it consists of the leaves of the data flow graph, i.e., those nodes that do not depend on any other and could be scheduled immediately. The nodes in the ready list are assigned priorities that could, for instance, be the estimated maximum accumulated latency on any path from that node to a root of the data flow graph. In each step, list scheduling picks, in a greedy way, as many highest-priority nodes as possible from the ready list that fit into the next issue packet and for which resource requirements can be satisfied. The resource reservations of these issued nodes are then committed to the global resource usage map, and the issued nodes are removed from the data flow graph. Some further nodes may now become data ready in the next steps after the latency after all their predecessors has elapsed. The ready list is accordingly updated, and the process repeats until all nodes have been scheduled. The above description is for *forward scheduling*. *Backward scheduling* starts with the roots of the data flow graph and works in an analogous way in reversed topological order.

Another technique is *critical path scheduling*. First, a critical path in the basic block is detected; the nodes of that path are removed from the data flow graph and scheduled in topological order, each in its own issue packet. For the residual data flow graph, a critical path is determined, and so on, and this process is repeated until all nodes in the data flow graph have been scheduled. If there is no appropriate free slot in an issue packet to accommodate a node to be scheduled, a new issue packet is inserted.

Time-optimal instruction scheduling for basic blocks is NP-complete for almost any nontrivial target architecture, including most VLIW architectures. For special combinations of simple target architectures and restricted data flow graph topologies such as trees, polynomial-time optimal scheduling algorithms are known.

In the last decade, more expensive optimal methods for local instruction scheduling have become more and more popular, driven by (1) the need to generate high-quality code for embedded applications, (2) the fact that modern computers offer

the compiler many more CPU cycles that can be spent on advanced optimizations, and (3) advances in general optimization problem solver technology, especially for integer linear programming. For local instruction scheduling on general acyclic data flow graphs, optimal algorithms based on integer linear programming [11, 36, 61, 73, 102], branch-and-bound [23, 53], constraint logic programming [10] and dynamic programming [63, 65] have been developed. Also, more expensive heuristic optimization techniques, such as genetic programming [36, 77, 108] have been used successfully.

In practice, the scope limitation of instruction scheduling to a single basic block is too restrictive. Local instruction scheduling techniques are nevertheless significant, because they are also used in several global scheduling algorithms for larger acyclic code regions and even in certain cyclic scheduling algorithms, which we will discuss in Sect. 7.2.

7.2 Modulo Scheduling for Loops

Most DSP programs spend most of their execution time in some (inner) loops. Efficient loop transformation and loop scheduling techniques are therefore key to high code quality.

Loop unrolling is a simple transformation that can increase the scope of a local scheduler (and also other code optimizations) beyond the iteration boundaries, such that independent instructions from different iterations could be scheduled in parallel. However, loop unrolling increases code size considerably, which is often undesirable in embedded applications.

Software pipelining is a technique to overlap the execution of subsequent loop iterations such that independent instructions from different iterations can be scheduled in parallel on an instruction-level parallel architecture, without having to replicate the loop body code as in unrolling. As most scheduling problems with resource and dependence constraints, (rate-)optimal software pipelining is NP-complete.

Software pipelining has been researched intensively, both as a high-level loop transformation (performed in the middle end of a compiler or even as source-to-source program transformation) and as low-level optimization late in the code generation process (performed in the back end of a compiler), after instruction selection with resource allocation has been performed. The former approaches are independent of particular instructions and functional units to be selected for all operations in the loop, and thus have to rely on inaccurate cost estimations for execution time, energy, or register pressure when comparing various alternatives, while the actual cost will also depend on decisions made late in the code generation process. The latter approaches are bound to fixed instructions and functional units, and hence the flexibility of implementing the same abstract operation by a variety of different target machine instructions, with different resource requirements and latency behavior, is lost. In either case, optimization opportunities are missed

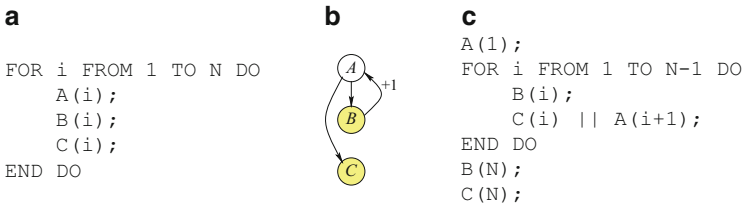


Fig. 17 Simple example: (**a**) Original loop, where $A(i)$, $B(i)$, $C(i)$ denote operations in the loop body that may compete for common resources, in this example $B(i)$ and $C(i)$, and that may involve both loop-independent data dependences, here $A(i) \rightarrow B(i)$ and $A(i) \rightarrow C(i)$, and loop-carried data dependences, here $B(i) \rightarrow A(i+1)$, see the dependence graph in (**b**). (**c**): After software pipelining

because interdependent problems are solved separately in different compiler phases. Approaches to integrate software pipelining with other code generation tasks will be discussed in Sect. 8.

Software pipelining, also called *cyclic scheduling*, transforms a loop into an equivalent loop whose body contains operations from different iterations of the original loop, which may result in faster code on an instruction-level parallel architecture. For example, the loop in Fig. 17a with data dependence graph in Fig. 17b can be transformed in the equivalent loop in Fig. 17c, where instructions $C(i)$ and $A(i+1)$ could now be executed in parallel ($||$) because they are statically known to be independent of each other and not to subscribe to the same hardware resources. This parallel execution was not possible in the original version of the loop because the code generator usually treats the loop body (a basic block) as a unit for scheduling and resource allocation, and furthermore separates the code for $C(i)$ and $A(i+1)$ by a backward branch to the loop entry. The body of the transformed loop is called the *kernel*, the operations before the kernel that "fill" the pipeline (here $A(1)$) are called the *prologue*, and the operations after the kernel that "drain" the pipeline (here $B(N)$ and $C(N)$), are called the *epilogue* of the software-pipelined loop. Software pipelining thus overlaps in the new kernel the execution of operations originating from different iterations of the original loop, as far as permitted by given dependence and resource constraints, in order to solicit more opportunities for parallel execution on instruction-level parallel architectures, such as superscalar, VLIW or EPIC processors. Software pipelining can be combined with loop unrolling.

In their survey of software pipelining methods, Allan et al. [4] divide existing approaches into two general classes. Based on a lower bound determined by analyzing dependence distances, latencies, and resource requirements, the *modulo scheduling* methods, as introduced by Rau and Glaeser [92] and refined in several approaches [68, 76], first guess the kernel size (in terms of clock cycles), called the *initiation interval (II)*, and then fill the instructions of the original loop body into a modulo reservation table of size II, which produces the new kernel. If no such modulo schedule could be found for the assumed II, the kernel is enlarged

by incrementing II, and the procedure is repeated. The *kernel-detection methods*, such as those by Aiken and Nicolau [2] (no resource constraints) and Vegdahl [101], continuously peel off iterations from the loop and schedule their operations until a pattern for a steady state emerges, from which the kernel is constructed.

Modulo scheduling starts with an initial initiation interval given by the lower bound $MinII$ (minimum initiation interval), which is the maximum of the recurrence-based minimum initiation interval (*RecMinII*) and the resource-based minimum initiation interval (*ResMinII*). *RecMinII* is the maximum accumulated sum of latencies along any dependence cycle in the dependence graph, divided by the number of iterations spanned by the dependence cycle. If there is no such cycle, *RecMinII* is 0. *ResMinII* is the maximum accumulated number of reserved slots on any resource in the loop body.

Modulo scheduling attempts to find a valid modulo schedule by filling all instructions in the modulo reservation table for the current II value. Priority is usually given to dependence cycles in decreasing order of accumulated latency per accumulated distance. If the first attempt fails, most heuristic methods allow backtracking for a limited number of further attempts. An exhaustive search is usually not feasible, because of the high problem complexity. Instead, if no attempt was successful, the II is incremented and the procedure is repeated with a one larger modulo reservation table. As there exists a trivial upper bound for the II (namely, the accumulated sum of all latencies in the loop body), this iterative method will eventually find a modulo schedule.

The main goal of software pipelining is to maximize the throughput by minimizing II, i.e., *rate-optimal* software pipelining. Moreover, minimizing the makespan (the elapsed time between the first instruction issue and last instruction termination) of a single loop iteration in the modulo scheduled loop is often a secondary optimization goal, because it directly implies the length of prologue and epilogue and thereby has an impact on code size (unless special hardware support for rotating predicate registers allows to represent prologue and epilogue code implicitly with the predicated kernel code).

Register allocation for software pipelined loops is another challenge. Software pipelining tends to increase register pressure. If a live range is longer than II cycles, it will interfere with itself (e.g., with its instance starting in the next iteration of the kernel) and thus a single register will not be sufficient; special care has to be taken to access the "right" one at any time. There are two kinds of techniques for such self-overlapping live ranges: hardware based techniques, such as rotating register sets and register queues, and software techniques such as modulo variable expansion [68] and live range splitting [96]. With *modulo variable expansion*, the kernel is unrolled and symbolic registers renamed until no live range self-overlaps any more: If μ denotes the maximum length of a self-overlapping live range, the required unroll factor is $\rho = \lceil \mu / II \rceil$, and the new initiation interval of the expanded kernel is $II' = \rho\ II$. The drawback of modulo variable expansion is increased code size and increased register need. An alternative approach is to avoid self-overlapping live ranges *a priori* by splitting long live ranges on dependence cycles into shorter ones, by inserting copy instructions.

Optimal methods for software pipelining based on integer linear programming have been proposed e.g. by Badia et al. [8], Govindarajan et al. [48] and Yang et al. [106]. Combinations of modulo scheduling with other code generation tasks will be discussed in Sect. 8.

Software pipelining is often combined with loop unrolling. Especially if the lower bound *MinII* is a non-integer value, loop unrolling before software pipelining can improve throughput. Moreover, loop unrolling reduces loop overhead (at least on processors that do not have hardware support for zero-overhead loops). The downside is larger code size.

7.3 Global Instruction Scheduling

Basic blocks are the units of (procedure-)global control flow analysis. The *basic block graph* of a program is a directed graph, where the nodes correspond to the basic blocks and edges show control flow transitions between basic blocks, such as branches or fall-through transitions to branch targets.

Global instruction scheduling methods consider several basic blocks at a time and allow to move instructions between basic blocks. The (current) scope of a global scheduling method is referred to as a *region*. Regions used for global scheduling include traces, superblocks, hyperblocks and treegions. Local scheduling methods are extended to address entire regions. Because the scope is larger, global scheduling has more flexibility and may generate better code than local scheduling. Program transformations such as function inlining, loop unrolling or predication can be applied to additionally increase the size of basic blocks and regions.

The idea of *trace scheduling* [38] is to make the most frequently executed control flow paths fast while accepting possible performance degradations along less frequently used paths. Execution frequencies are assigned to the outgoing edges at branch instructions based on static predictions or on profile data. A *trace* is a linear path (i.e., free of backwards edges and thereby of loops) through the basic block graph, where, at each basic block B_i in the trace except for the last one, its successor B_j in the trace is the target of the more frequently executed control flow edge leaving B_i. Traces may have side entrances and side exits of control flow from outside the trace. Forward edges are possible, and likewise backwards edges to the first block in the trace. See Fig. 18 for an example.

Trace scheduling repeatedly identifies a maximum-length trace in the basic block graph, removes its basic blocks from the graph and schedules the instructions of the trace with a local scheduling method as if it were a single basic block. As instructions are moved across a control flow transition, either upwards or downwards, correctness of the program must be re-established by inserting compensation code into the other predecessor or successor block of the original basic block of that instruction, respectively. Two of the possible cases are shown in Fig. 19. The insertion of compensation code may lead to considerable code size expansion on the less frequently executed branches. After the trace has been scheduled, its basic

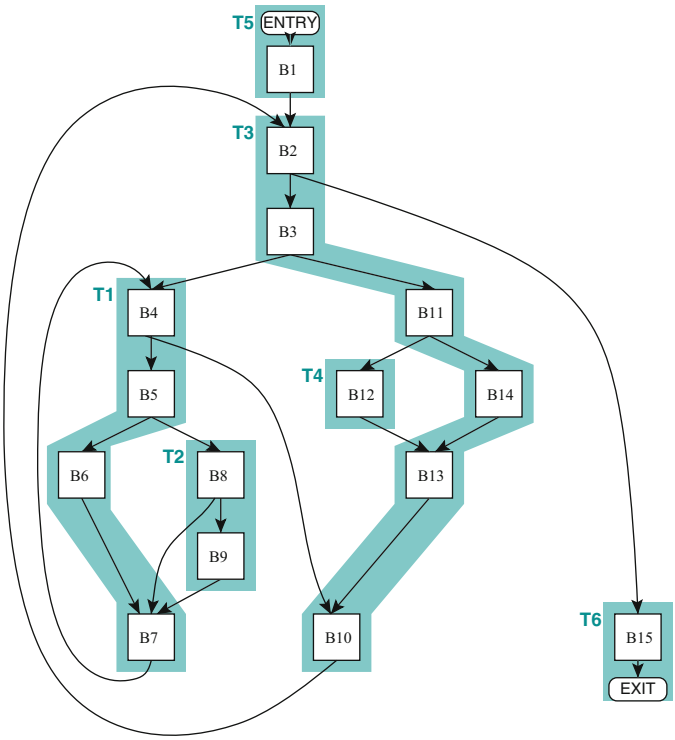


Fig. 18 Traces (shaded) in a basic block graph, constructed and numbered T1, T2, ... in order of decreasing predicted execution frequency. A trace ends at a backwards branch or at a join point with another trace of higher execution frequency (which thus was constructed earlier). Trace T1 represents the more frequent control path in the inner loop starting at basic block B4

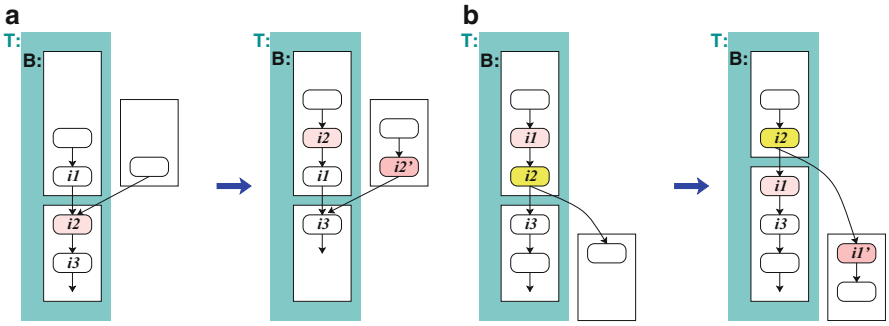


Fig. 19 Two of the main cases in trace scheduling where compensation code must be inserted. The trace T is being compacted. Case (**a**): Hoisting instruction $i2$ into the predecessor basic block B requires inserting a copy $i2'$ of $i2$ into the other predecessor block(s). Case (**b**): Moving instruction $i1$ forward across the branch instruction $i2$ requires inserting a copy $i1'$ of $i1$ into the other branch target basic block

blocks are removed from the basic block graph, the next trace is determined, and this process is repeated until all basic blocks of the program have been scheduled.

Superblocks [59] are a restricted form of traces that do not contain any branches into it (except possibly for backwards edges to the first block). This restriction simplifies the generation of compensation code in trace scheduling. A trace can be converted into a superblock by replicating its tail parts that are targets of branches from outside the trace. Tail duplication is a form of generating compensation code ahead of scheduling, and can likewise increase code size considerably.

While traces and superblocks are linear chains of basic blocks, *hyperblocks* [78] are regions in the basic block graph with a common entry block and possibly several exit blocks, with acyclic internal control flow. Using predication, the different control flow paths in a hyperblock could be merged to a single superblock.

A *treegion* [54], also known as *extended basic block* [81], is an out-tree region in the basic block graph. There are no side entrances of control flow into a treegion, except to its root basic block.

Recently, optimal methods for global instruction scheduling on instruction-level parallel processors have become popular. Winkel used integer linear programming for optimal global scheduling for Intel IA-64 (Itanium) processors [104] and showed that it can be used in a production compiler [105]. Malik et al. [79] proposed a constraint programming approach for optimal scheduling of superblocks.

7.4 Generated Instruction Schedulers

Whenever a forward scheduling algorithm, such as list scheduling, inserts another data-ready instruction at the end of an already computed partial schedule, it needs to fit the required resource reservations of the new instruction against the already committed resource reservations, and likewise obey pending latencies of predecessor instructions where necessary, in order to derive the earliest possible issue time for the new instruction relative to the issue time of the last instruction in the partial schedule. While the impact of dependence predecessors can be checked quickly, determining that issue time offset is more involved with respect to the resource reservations. The latter calculation could be done, for instance, by searching through the partial schedule's resource usage map, resource by resource. For advanced scheduling methods that try lots of alternatives, faster methods for detecting resource conflicts, respectively for computing the issue time offset, are desirable.

Note that the new instruction's issue time relative to the currently last instruction of the partial schedule only depends on the most recent resource reservations, not the entire partial schedule. Each possible contents of this still relevant, recent part of the pipeline can be interpreted as a pipeline *state*, and appending an instruction will result in a new state and an issue time offset for the new instruction, such that scheduling can be described as a finite state automaton. The initial state is an empty pipeline. The set of possible states and the set of possible transitions depend only on the processor, not on the input program. Hence, once all possible states have been

determined and encoded and all possible transitions with their effects on successor state and issue time have been precomputed once and for all, scheduling can be done very fast by looking up the issue time offset and the new state in the precomputed transition table for the current state and inserted instruction.

This automaton-based approach was introduced by Müller [82] and was improved and extended in several works [9, 31, 90]. The automaton can be generated automatically from a formal description of the processor's set of instructions with their reservation tables. A drawback is that the number of possible states and transitions can be tremendous, but there exist techniques to reduce the size of the scheduling automaton, such as standard finite state machine minimization, automata factoring, and replacement of several physical resources with equivalent contention behavior by a single virtual resource.

8 Integrated Code Generation for VLIW and Clustered VLIW

In most compilers, the subproblems of code generation are treated separately in subsequent *phases* of the compiler back-end. This is easier from a software engineering point of view, but often leads to suboptimal results because decisions made in earlier phases constrain the later phases.

For instance, early instruction scheduling determines the live ranges for a subsequent register allocator; when the number of physical registers is not sufficient, spill code must be inserted a-posteriori into the existing schedule, which may compromise the schedule's quality. Conversely, early register allocation introduces additional ("false") data dependences, which are an artifact caused by the reuse of registers but constrain the subsequent instruction scheduling phase.

Interdependences exist also between instruction scheduling and instruction selection. In order to formulate instruction selection as a separate minimum-cost covering problem, phase-decoupled code generation assigns a fixed, context-independent cost to each instruction, such as its expected or worst-case execution time. However, the actual cost also depends on interference with resource occupation and latency constraints of other instructions, which depends on the schedule. For instance, a potentially concurrent execution of two independent IR operations will be prohibited if instructions are selected that require the same resource.

For loops, instruction selection can likewise depend on modulo scheduling and vice versa; for instance, in the example loop of Fig. 17, there might exist an efficient instruction covering the chain $B(i) \longrightarrow A(i+1)$ that only is exposed to (local) instruction selection after software-pipelining the loop as in Fig. 17c.

Even the subdivision of instruction scheduling into separate phases for sequencing and compaction can have negative effects on schedule quality if instructions with non-block reservation tables occur [64].

Furthermore, on clustered VLIW processors, concurrent execution may be possible only if the operands reside in the right register sets at the right time, as

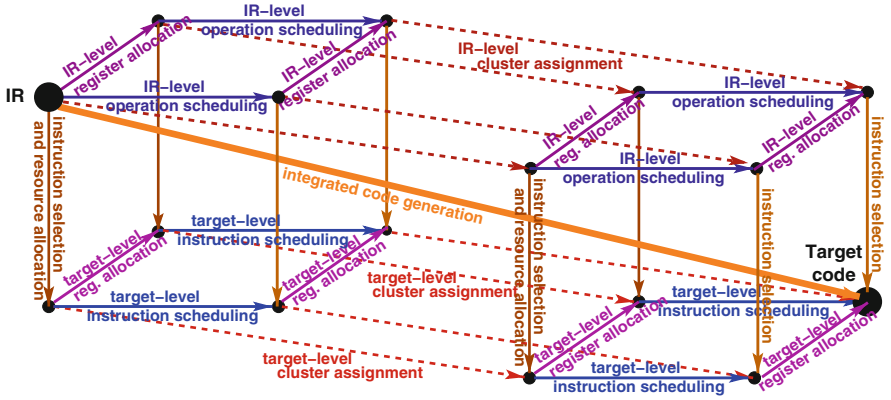


Fig. 20 Phase-decoupled vs. fully integrated code generation for clustered VLIW processors. Only the four main tasks of code generation are shown: Cluster assignment (red dashed arrows), instruction selection (brown vertical arrows), instruction scheduling (blue horizontal arrows), and register allocation (purple arrows in z-direction). While often performed in just this order, many phase orderings are possible in phase-decoupled code generation, visualized by the paths along the edges of the four-dimensional hypercube from the processor-independent low-level IR to final target code. Fully integrated code generation solves all tasks simultaneously as a monolithic combined optimization problem, thus following the main diagonal (orange thick arrow)

discussed earlier. While instruction scheduling and register allocation need to know about the cluster assignment of instructions and data, cluster assignment could profit from knowing about free slots where transfer instructions could be placed, or free registers where transfered copies could be held. Any phase decoupled approach may result in bad code quality because the later phases are constrained by decisions made in the early ones.

Hence, the integration of these subproblems to solve them as a single optimization problem, as visualized in Fig. 20, is highly desirable, but unfortunately this increases the overall complexity of code generation considerably. Despite the recent improvements in general optimization problem solver technology, this ambitious approach is limited in scope to basic blocks and loops. Other methods take a more conservative approach based on a phase-decoupled code generator and make, heuristically, an early phase aware of possibly different goals of later phases. For instance, register pressure aware scheduling methods trade less instruction level parallelism for shorter live ranges in program regions where register pressure is predicted to be high, which can lead to better register allocation with less spill code later.

8.1 Integrated Code Generation at Basic Block Level

There exist several heuristic approaches that aim at a better integration of instruction scheduling and register allocation [15, 42, 46, 67]. For the case of clustered VLIW processors, the heuristic algorithm proposed by Kailas et al. [60] integrates cluster assignment, register allocation, and instruction scheduling.

Heuristic methods that couple or integrate instruction scheduling and cluster assignment were proposed by Özer et al. [88], Leupers [71], Chu et al. [24], and by Nagpal and Srikant [84]. For example, Leupers [71] uses a simulated-annealing based approach where cluster allocation and instruction scheduling are applied alternatingly in an iterative optimization loop.

For the computationally intensive kernels of DSP application programs to be used in an embedded product throughout its lifetime, the manufacturer is often willing to afford spending a significant amount of time in optimizing the code during the final compilation. However, there are only a few approaches that have the potential—given sufficient time and space resources—to compute an optimal solution to an integrated problem formulation, mostly combining local scheduling and register allocation [10, 61, 69].

Some of these approaches are also able to partially integrate instruction selection problems, even though for rather restricted machine models. For instance, Wilson et al. [103] consider architectures with a single, non-pipelined ALU, two non-pipelined parallel load/store/move units, and a homogeneous set of general-purpose registers. Araujo and Malik [7] consider integrated code generation for expression trees with a machine model where the capacity of each sort of memory resource (register classes or memory blocks) is either one or infinity, a class that includes, for instance, the TI C25.

The integrated method adopted in the retargetable framework AVIV [53] for clustered VLIW architectures builds an extended data flow graph representation of the basic block that explicitly represents all alternatives for implementation; then, a branch-and-bound heuristic selects an alternative among all representations that is optimized for code size.

Chang et al. [21] use integer linear programming for combined instruction scheduling and register allocation with spill code generation for non-pipelined, non-clustered multi-issue architectures.

Kessler and Bednarski [63] propose a dynamic programming algorithm for fully integrated code generation for clustered and non-clustered VLIW architectures at the basic block level, which was implemented in the retargetable integrated code generator OPTIMIST. Bednarski and Kessler [11] and Eriksson et al. [34, 36] solve the problem with integer linear programming; the latter work also gives a heuristic approach based on a genetic algorithm.

Castañeda-Lozano et al. [19] present a method that works on the (linear) SSA form and applies constraint programming to integrate register allocation including ultimate coalescing and spill code optimization with instruction scheduling for non-clustered VLIW architectures.

8.2 Loop-Level Integrated Code Generation

There exist several heuristic algorithms for modulo scheduling that attempt to reduce register pressure, such as Hypernode Resource Modulo Scheduling [76] and Swing Modulo Scheduling [75]. Nyström and Eichenberger [87] couple cluster assignment and modulo scheduling for clustered VLIW architectures. Codina et al. [25] give a heuristic method for modulo scheduling integrated with register allocation and spill code generation for clustered VLIW processors. Zalamea et al. [107] consider the integration of register pressure aware modulo scheduling with register allocation, cluster assignment and spilling for clustered VLIW processors and present an iterative heuristic algorithm with backtracking. Aleta et al. [3] use a phase-coupled heuristic approach to cluster-assignment and modulo scheduling that also considers replication of instructions for reduced inter-cluster communication. Kim and Krall [66] present an iterative heuristic approach that couples modulo scheduling and cluster assignment heuristics for the 'C64x architecture, implemented in the LLVM compiler framework. Stotzer and Leiss [96] propose a preprocessing transformation for modulo scheduling for the 'C6x clustered VLIW DSP architecture that attempts to reduce self-overlapping cyclic live ranges in a preprocessing phase and thereby eliminate the need for modulo variable expansion or rotating register files.

Eisenbeis and Sawaya [32] propose an integer linear programming method for modulo scheduling integrated with register allocation, which gives optimal results if the number of schedule slots is fixed. Nagarakatte and Govindarajan [83] provide an optimal method for integrating register allocation and spill code generation with modulo scheduling for non-clustered architectures. Eriksson and Kessler [34, 35] give an integer linear programming method for optimal, fully integrated code generation for loops, combining modulo scheduling with instruction selection, cluster assignment, register allocation and spill code generation, for clustered VLIW architectures.

9 Concluding Remarks

Compilers for VLIW DSP processors need to apply a considerable amount of advanced optimizations to achieve code quality comparable to hand-written code. Current advances in general optimization problem solver technology are encouraging, and heuristic techniques developed for standard compilers are being complemented by more aggressive optimizations. For small and medium sized program parts, even optimal solutions are within reach. Also, most problems in code generation are strongly interdependent and should be considered together in an integrated or at least phase-coupled way to avoid poor code quality due to phase ordering effects. We expect further improvements in optimized and integrated code generation techniques for VLIW DSPs in the near future.

Acknowledgements The author thanks Mattias Eriksson and Dake Liu for discussions and commenting on a draft of this chapter. The author also thanks Eric Stotzer from Texas Instruments for interesting discussions about code generation for the TI 'C6x DSP processor family.

This work was funded by Vetenskapsrådet (project *Integrated Software Pipelining*), SSF (project *DSP platform for emerging telecommunication and multimedia*) and by SeRC, *Parallel Software and Data Engineering* (www.e-science.se).

References

1. Alfred V. Aho, Mahadevan Ganapathi, and Steven W.K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.
2. Alexander Aiken and Alexandru Nicolau. Optimal loop parallelization. *SIGPLAN Notices*, 23(7):308–317, July 1988.
3. Alex Aleta, Josep M. Codina, Jesus Sanchez, Antonio Gonzalez, and David Kaeli. AGAMOS: A graph-based approach to modulo scheduling for clustered microarchitectures. *IEEE Transactions on Computers*, 58(6):770–783, June 2009.
4. Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3), September 1995.
5. Analog Devices. TigerSHARC embedded processor ADSP-TS201S. Data sheet, www.analog.com/en/embedded-processing-dsp/tigersharc, 2006.
6. Andrew W. Appel and Lal George. Optimal Spilling for CISC Machines with Few Registers. In *Proc. ACM conf. on Programming language design and implementation*, pages 243–253. ACM Press, 2001.
7. Guido Araujo and Sharad Malik. Optimal code generation for embedded memory non-homogeneous register architectures. In *Proc. 7th Int. Symposium on System Synthesis*, pages 36–41, September 1995.
8. Rosa M. Badia, Fermin Sanchez, and Jordi Cortadella. OSP: Optimal Software Pipelining with Minimum Register Pressure. Technical Report UPC-DAC-1996-25, DAC Dept. d'arquitectura de Computadors, Univ. Polytecnica de Catalunya, Barcelona, Campus Nord. Modul D6, E-08071 Barcelona, Spain, June 1996.
9. Vasanth Bala and Norman Rubin. Efficient instruction scheduling using finite state automata. In *Proc. 28th int. symp. on miocroarchitecture (MICRO-28)*, pages 46–56. IEEE, 1995.
10. Steven Bashford and Rainer Leupers. Phase-coupled mapping of data flow graphs to irregular data paths. *Design Automation for Embedded Systems (DAES)*, 4(2/3):119–165, 1999.
11. Andrzej Bednarski and Christoph Kessler. Optimal integrated VLIW code generation with integer linear programming. In *Proc. Int. Euro-Par 2006 Conference*. Springer LNCS, August 2006.
12. Mirza Beg and Peter van Beek. A constraint programming approach for instruction assignment. In *Proc. Int. Workshop on Interaction between Compilers and Computer Architectures (INTERACT-15)*, pp. 25–34, February 2011.
13. D. Bernstein, M.C. Golumbic, Y. Mansour, R.Y. Pinter, D.Q. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compilers. In *Proc. Int. Conf. on Progr. Lang. Design and Implem.*, pages 258–263, 1989.

14. F. Bouchez, A. Darte, C. Guillon, and F. Rastello. Register allocation: what does the NP-completeness proof of Chaitin et al. really prove? [...]. In *Proc. 19th int. workshop on languages and compilers for parallel computing, New Orleans*, November 2006.
15. Thomas S. Brasier, Philip H. Sweany, Steven J. Beaty, and Steve Carr. Craig: A practical framework for combining instruction scheduling and register assignment. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'95)*, 1995.
16. Preston Briggs, Keith Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *Proc. Int. Conf. on Progr. Lang. Design and Implem.*, pages 275–284, 1989.
17. Preston Briggs, Keith Cooper, and Linda Torczon. Rematerialization. In *Proc. Int. Conf. on Progr. Lang. Design and Implem.*, pages 311–321, 1992.
18. Philip Brisk, Ajay K. Verma, and Paolo Ienne. Optimistic chordal coloring: a coalescing heuristic for SSA form programs. *Des. Autom. Embed. Syst.*, 13:115–137, 2009.
19. Roberto Castañeda-Lozano, Mats Carlsson, Gabriel Hjort-Blindell and Christian Schulte. Combinatorial spill code optimization and ultimate coalescing. In *Proc. LCTES'14*, pp. 23–32, June 2014.
20. G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.
21. Chia-Ming Chang, Chien-Ming Chen, and Chung-Ta King. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers Mathematics and Applications*, 34(9):1–14, 1997.
22. Chung-Kai Chen, Ling-Hua Tseng, Shih-Chang Chen, Young-Jia Lin, Yi-Ping You, Chia-Han Lu, and Jenq-Kuen Lee. Enabling compiler flow for embedded VLIW DSP processors with distributed register files. In *Proc. LCTES'07*, pages 146–148. ACM, 2007.
23. Hong-Chich Chou and Chung-Ping Chung. An Optimal Instruction Scheduler for Superscalar Processors. *IEEE Trans. on Parallel and Distr. Syst.*, 6(3):303–313, 1995.
24. Michael Chu, Kevin Fan, and Scott Mahlke. Region-based hierarchical operation partitioning for multicluster processors. In *Proc. Int. Conf. on Progr. Lang. Design and Implem. (PLDI'03)*, pp. 300–311, ACM, June 2003.
25. Josep M. Codina, Jesus Sánchez, and Antonio González. A unified modulo scheduling and register allocation technique for clustered processors. In *Proc. PACT-2001*, September 2001.
26. Edward S. Davidson, Leonard E. Shar, A. Thampy Thomas, and Janak H. Patel. Effective control for pipelined computers. In *Proc. Spring COMPCON75 Digest of Papers*, pages 181–184. IEEE Computer Society, February 1975.
27. Giuseppe Desoli. Instruction assignment for clustered VLIW DSP compilers: a new approach. Technical Report HPL-98-13, HP Laboratories Cambridge, February 1998.
28. Benoit Dupont de Dinechin. Kalray MPPA® Massively Parallel Processor Array. Slide set, Hot Chips 27 Symposium, IEEE, August 2015.
29. Dietmar Ebner. *SSA-based code generation techniques for embedded architectures*. PhD thesis, Technische Universität Wien, Vienna, Austria, June 2009.
30. Erik Eckstein, Oliver König, and Bernhard Scholz. Code instruction selection based on SSA-graphs. In A. Krall, editor, *Proc. SCOPES-2003, Springer LNCS 2826*, pages 49–65, 2003.
31. Alexandre E. Eichenberger and Edward S. Davidson. A reduced multipipeline machine description that preserves scheduling constraints. In *Proc. Int. Conf. on Progr. Lang. Design and Implem. (PLDI'96)*, pages 12–22, New York, NY, USA, 1996. ACM Press.
32. Christine Eisenbeis and Antoine Sawaya. Optimal loop parallelization under register constraints. In *Proc. 6th Workshop on Compilers for Parallel Computers (CPC'96)*, pages 245–259, December 1996.
33. John Ellis. Bulldog: A Compiler for VLIW Architectures. MIT Press, Cambridge, MA, 1986.
34. Mattias Eriksson and Christoph Kessler. Integrated Code Generation for Loops. *ACM Transactions on Embedded Computing Systems* 11S(1), Article 19, 24 pages, ACM, June 2012.
35. Mattias Eriksson and Christoph Kessler. Integrated modulo scheduling for clustered VLIW architectures. In *Proc. HiPEAC-2009 High-Performance and Embedded Architecture and Compilers, Paphos, Cyprus*, pages 65–79. Springer LNCS 5409, January 2009.

36. Mattias Eriksson, Oskar Skoog, and Christoph Kessler. Optimal vs. heuristic integrated code generation for clustered VLIW architectures. In *Proc. 11th int. workshop on software and compilers for embedded systems (SCOPES'08)*. ACM, 2008.
37. M. Anton Ertl. Optimal Code Selection in DAGs. In *Proc. Int. Symposium on Principles of Programming Languages (POPL'99)*. ACM, 1999.
38. Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, C–30(7):478–490, July 1981.
39. Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier / Morgan Kaufmann, 2005.
40. Björn Franke. C Compilers and Code Optimization for DSPs. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, eds., *Handbook of Signal Processing Systems*, Second Edition, Springer 2012.
41. Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a Simple, Efficient Code Generator Generator. *Letters of Programming Languages and Systems*, 1(3):213–226, September 1992.
42. Stefan M. Freudenberger and John C. Ruttenberg. Phase ordering of register allocation and instruction scheduling. In *Code Generation: Concepts, Tools, Techniques [44]*, pages 146–170, 1992.
43. Anup Gangwar, M. Balakrishnan, and Anshul Kumar. Impact of intercluster communication mechanisms on ILP in clustered VLIW architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 12(1):1, 2007.
44. Robert Giegerich and Susan L. Graham, editors. *Code Generation - Concepts, Tools, Techniques*. Springer Workshops in Computing, 1992.
45. R.S. Glanville and S.L. Graham. A New Method for Compiler Code Generation. In *Proc. Int. Symposium on Principles of Programming Languages*, pages 231–240, January 1978.
46. James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *Proc. ACM Int. Conf. on Supercomputing*, pages 442–452. ACM press, July 1988.
47. David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0–1 integer programming. *Softw. Pract. Exper.*, 26(8):929–965, 1996.
48. R. Govindarajan, Erik Altman, and Guang Gao. A framework for resource-constrained rate-optimal software pipelining. *IEEE Trans. Parallel and Distr. Syst.*, 7(11):1133–1149, November 1996.
49. R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, November 1966.
50. Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In *Proc. 16th int. conf. on compiler construction*, pages 111–125, March 2007.
51. Sebastian Hack and Gerhard Goos. Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters*, 98:150–155, 2006.
52. Todd Hahn, Eric Stotzer, Dineel Sule, and Mike Asal. Compilation strategies for reducing code size on a VLIW processor with variable length instructions. In *Proc. HiPEAC'08 conference*, pages 147–160. Springer LNCS 4917, 2008.
53. Silvina Hanono and Srinivas Devadas. Instruction scheduling, resource allocation, and scheduling in the AVIV retargetable code generator. In *Proc. Design Automation Conf.* ACM, 1998.
54. W. A. Havanki. Treegion scheduling for VLIW processors. M.S. thesis, Dept. Electrical and Computer Engineering, North Carolina State Univ., Raleigh, NC, USA, 1997.
55. Gabriel Hjort-Blindell. *Instruction Selection – Principles, Methods, and Applications.* Springer, 2016.
56. Gabriel Hjort-Blindell, Mats Carlsson, Roberto Castaneda-Lozano, and Christian Schulte. Complete and practical univeral instruction selection. *ACM Trans. on Embedded Computing Systems (TECS)*, 16(5s), Art. 119, Sep. 2017
57. L.P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index register allocation. *Journal of the ACM*, 13(1):43–61, January 1966.

58. Wei-Chung Hsu, Charles N. Fischer, and James R. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Trans. Softw. Eng.*, 15(10):1252–1262, October 1989.
59. Wen-Mei Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *J. Supercomput.*, 7(1-2):229–248, 1993.
60. Krishnan Kailas, Kemal Ebcioglu, and Ashok Agrawala. CARS: A new code generation framework for clustered ILP processors. In *Proc. 7th Int. Symp. on High-Performance Computer Architecture (HPCA'01)*, pages 133–143. IEEE Computer Society, June 2001.
61. Daniel Kästner. *Retargetable Postpass Optimisations by Integer Linear Programming*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2000.
62. Christoph Kessler and Andrzej Bednarski. Optimal integrated code generation for clustered VLIW architectures. In *Proc. ACM SIGPLAN Conf. on Languages, Compilers and Tools for Embedded Systems / Software and Compilers for Embedded Systems, LCTES-SCOPES'2002*. ACM, June 2002.
63. Christoph Kessler and Andrzej Bednarski. Optimal integrated code generation for VLIW architectures. *Concurrency and Computation: Practice and Experience*, 18:1353–1390, 2006.
64. Christoph Kessler, Andrzej Bednarski, and Mattias Eriksson. Classification and generation of schedules for VLIW processors. *Concurrency and Computation: Practice and Experience*, 19:2369–2389, 2007.
65. Christoph W. Keßler. Scheduling Expression DAGs for Minimal Register Need. *Computer Languages*, 24(1):33–53, September 1998.
66. Nikolai Kim and Andreas Krall. Integrated modulo scheduling and cluster assignment for TI TMS320C64x+ architecture. In Proc. 11th Worksh. on Optim. for DSP and Embedded Syst. (ODES'14), pp. 25–32, ACM, 2014.
67. Tokuzo Kiyohara and John C. Gyllenhaal. Code scheduling for VLIW/superscalar processors with limited register files. In *Proc. 25th int. symp. on miocroarchitecture (MICRO-25)*. IEEE CS Press, 1992.
68. Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. CC'88*, pages 318–328, July 1988.
69. Rainer Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer, 1997.
70. Rainer Leupers. *Code Optimization Techniques for Embedded Processors*. Kluwer, 2000.
71. Rainer Leupers. Instruction scheduling for clustered VLIW DSPs. In *Proc. PACT'00 int. conference on parallel architectures and compilation*. IEEE Computer Society, 2000.
72. Rainer Leupers and Steven Bashford. Graph-based code selection techniques for embedded processors. *ACM TODAES*, 5(4):794–814, October 2000.
73. Rainer Leupers and Peter Marwedel. Time-constrained code compaction for DSPs. *IEEE Transactions on VLSI Systems*, 5(1):112–122, 1997.
74. Dake Liu. *Embedded DSP processor design*. Morgan Kaufmann, 2008.
75. Josep Llosa, Antonio Gonzalez, Mateo Valero, and Eduard Ayguade. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. In *Proc. PACT'96 conference*, pages 80–86. IEEE, 1996.
76. Josep Llosa, Mateo Valero, Eduard Ayguade, and Antonio Gonzalez. Hypernode reduction modulo scheduling. In *Proc. 28th int. symp. on miocroarchitecture (MICRO-28)*, 1995.
77. M. Lorenz and P. Marwedel. Phase coupled code generation for DSPs using a genetic algorithm. In *Proc. conf. on design automation and test in Europe (DATE'04)*, pages 1270–1275, 2004.
78. Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proc. 25th int. symp. on microarchitecture (MICRO-25)*, pages 45–54, December 1992.
79. Abid M. Malik, Michael Chase, Tyrel Russell, and Peter van Beek. An application of constraint programming to superblock instruction scheduling. In *Proc. 14th Int. Conf. on Principles and Practice of Constraint Programming*, pages 97–111, September 2008.

80. Waleed M. Meleis and Edward D. Davidson. Dual-issue scheduling with spills for binary trees. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 678 – 686. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.
81. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
82. Thomas Müller. Employing finite automata for resource scheduling. In *Proc. 26th int. symp. on microarchitecture (MICRO-26)*, pages 12–20. IEEE, December 1993.
83. S. G. Nagarakatte and R. Govindarajan. Register allocation and optimal spill code scheduling in software pipelined loops using 0-1 integer linear programming formulation. In *Proc. int. conf. on compiler construction (CC-2007)*, pages 126–140. Springer LNCS 4420, 2007.
84. Rahul Nagpal and Y. N. Srikant. Integrated temporal and spatial scheduling for extended operand clustered VLIW processors. In *Proc. 1st conf. on Computing Frontiers*, pages 457–470. ACM Press, 2004.
85. Steven Novack and Alexandru Nicolau. Mutation scheduling: A unified approach to compiling for fine-grained parallelism. In *Proc. Workshop on compilers and languages for parallel computers (LCPC'94)*, pages 16–30. Springer LNCS 892, 1994.
86. NXP. Trimedia TM-1000. Data sheet, www.nxp.com, 1998.
87. Erik Nyström and Alexandre E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proc. 31st annual ACM/IEEE Int. symposium on microarchitecture (MICRO-31)*, IEEE CS Press, 1998.
88. Emre Özer, Sanjeev Banerjia, and Thomas M. Conte. Unified assign and schedule: a new approach to scheduling for clustered register file microarchitectures. In *Proc. 31st annual ACM/IEEE Int. Symposium on Microarchitecture*, pages 308–315. IEEE CS Press, 1998.
89. Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5), September 1999.
90. Todd A. Proebsting and Christopher W. Fraser. Detecting pipeline structural hazards quickly. In *Proc. 21st symp. on principles of programming languages (POPL'94)*, pages 280–286. ACM Press, 1994.
91. Qualcomm Technologies, Inc. Hexagon DSP Processor. Qualcomm Developer Network, https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor, last accessed March 2017
92. B. Rau and C. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. 14th Annual Workshop on Microprogramming*, pages 183–198, 1981.
93. B. Ramakrishna Rau, Vinod Kathail, and Shail Aditya. Machine-description driven compilers for EPIC and VLIW processors. *Design Automation for Embedded Systems*, 4:71–118, 1999. Appeared also as technical report HPL-98-40 of HP labs, Sep. 1998.
94. Recore Systems. Xentium VLIW DSP IP core. Product brief, http://www.recoresystems.com/fileadmin/downloads/Product_briefs/2016-1.0_Xentium_Product_Brief.pdf, 2016.
95. Richard Scales. Software development techniques for the TMS320C6201 DSP. Texas Instruments Application Report SPRA481, www.ti.com, December 1998.
96. Eric J. Stotzer and Ernst L. Leiss. Modulo scheduling without overlapped lifetimes. In *Proc. LCTES-2009*, pages 1–10. ACM, June 2009.
97. Texas Instruments, Inc. TMS320C62x DSP CPU and instruction set reference guide. Document SPRU731A, www.ti.com, 2010.
98. Texas Instruments, Inc. TMS320C66x DSP CPU and instruction set reference guide. Document SPRUGH7, www.ti.com, Nov. 2010.
99. Texas Instruments, Inc. Optimizing loops on the C66x DSP. Application report SPRABG7, www.ti.com, Nov. 2010.
100. Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and Speed in Linear-scan Register Allocation. In *Proc. ACM SIGPLAN Conf. on Progr. Lang. Design and Implem. (PLDI'98)*, pages 142–151, 1998.
101. Steven R. Vegdahl. A Dynamic-Programming Technique for Compacting Loops. In *Proc. 25th annual ACM/IEEE Int. symposium on microarchitecture (MICRO-25)*, pages 180–188. IEEE CS Press, 1992.

102. Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *Proc. Int. Conf. on Progr. Lang. Design and Implem. (PLDI'00)*, pages 121–133, 2000.
103. Tom Wilson, Gary Grewal, Ben Halley, and Dilip Banerji. An integrated approach to retargetable code generation. In *Proc. Int. Symposium on High-Level Synthesis*, pages 70–75, May 1994.
104. Sebastian Winkel. *Optimal global instruction scheduling for the Itanium processor architecture*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, September 2004.
105. Sebastian Winkel. Optimal versus heuristic global code scheduling. In *Proc. 40th annual ACM/IEEE Int. symposium on microarchitecture (MICRO-40)*, pages 43–55, 2007.
106. Hongbo Yang, Ramaswamy Govindarajan, Guang R. Gao, George Cai, and Ziang Hu. Exploiting schedule slacks for rate-optimal power-minimum software pipelining. In *Proc. Workshop on Compilers and Operating Systems for Low Power (COLP-2002)*, September 2002.
107. Javier Zalamea, Josep Llosa, Eduard Ayguade, and Mateo Valero. Modulo scheduling with integrated register spilling for clustered VLIW architectures. In *Proc. ACM/IEEE Int. symp. on microarchitecture (MICRO-34)*, pages 160–169, 2001.
108. Thomas Zeitlhofer and Bernhard Wess. Operation scheduling for parallel functional units using genetic algorithms. In *Proc. Int. Conf. on ICASSP '99: Proceedings of the Acoustics, Speech, and Signal Processing (ICASSP'99)*, pages 1997–2000. IEEE Computer Society, 1999.

Software Compilation Techniques for Heterogeneous Embedded Multi-Core Systems

Rainer Leupers, Miguel Angel Aguilar, Jeronimo Castrillon, and Weihua Sheng

Abstract The increasing demands of modern embedded systems, such as high-performance and energy-efficiency, have motivated the use of heterogeneous multi-core platforms enabled by Multiprocessor System-on-Chips (MPSoCs). To fully exploit the power of these platforms, new tools are needed to address the increasing software complexity to achieve a high productivity. An *MPSoC compiler* is a tool-chain to tackle the problems of application modeling, platform description, software parallelization, software distribution and code generation for an efficient usage of the target platform. This chapter discusses various aspects of compilers for heterogeneous embedded multi-core systems, using the well-established single-core C compiler technology as a baseline for comparison. After a brief introduction to the MPSoC compiler technology, the important ingredients of the compilation process are explained in detail. Finally, a number of case studies from academia and industry are presented to illustrate the concepts discussed in this chapter.

1 Introduction

1.1 MPSoCs and MPSoC Compilers

The current design trend in embedded systems show that heterogeneous Multiprocessor System-on-Chip (MPSoC) is the most promising way to keep on exploiting

R. Leupers (✉) · M. A. Aguilar
Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Aachen, Germany
e-mail: leupers@ice.rwth-aachen.de; aguilar@ice.rwth-aachen.de

J. Castrillon
Center for Advancing Electronics Dresden, TU Dresden, Dresden, Germany
e-mail: jeronimo.castrillon@tu-dresden.de

W. Sheng
Silexica GmbH, Köln, Germany
e-mail: sheng@silexica.com

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_28

the high level of integration provided by the semiconductor technology and, at the same time, matching the constraints imposed by the embedded systems market in terms of performance and power consumption. Looking at today's smartphones, it is clear to see that they are integrated with a great number of functions, such as camera, personal digital assistant applications, voice/data communications and multi-band wireless standards. Moreover, like many other consumer electronic products, many non-functional parameters are evenly critical for their successes in the market, e.g., energy consumption and form factor. All these requirements need the emergence of heterogeneous MPSoC architectures. They usually consist of programmable cores of various types, special hardware accelerators and efficient Networks-on-Chips (NoCs), to execute a large amount of complex software, in order to catch up with the next wave of integration.

Compared to high-performance computing systems in supercomputers and computer clusters, embedded computing systems require a different set of constraints that need to be taken into consideration during the design process:

- *Real-time constraints:* Real-time performance is key to the embedded devices, especially in the signal processing domain, such as wireless and multimedia. Meeting real-time constraints requires not only the hardware being capable of satisfying the demands of high-performance computations, but also the predictable behavior of the running applications.
- *Energy-efficiency:* Most mobile devices are battery powered, therefore, energy-efficiency is one of the most important factors during the system design.
- *Area-efficiency:* How to efficiently use the limited chip area becomes critical, especially for consumer electronics, where portability is a must-to-have.
- *Application Domain:* Unlike in general-purpose computing, embedded products usually target at specific market segments, which in turn ask for the specialization of the system design tailored for specific applications.

With these design criteria, heterogeneous MPSoC architectures are called to outperform the previous single-core or homogeneous solutions. For a detailed discussion on the architectures, the readers are referred to Chapter [15]. MPSoC design methodologies, also referred as *Electronic System-Level* (ESL) tools, are growing in importance to tackle the challenge of exploring the exploding design space brought by the heterogeneity [53]. Many different tools are required for completing a successful MPSoC design, or a series of MPSoC product generations, such as the Texas Instruments Keystone family [73]. The *MPSoC compiler* (or *Multi-Core Compiler*) is one important tool among those, which is the main focus of this chapter.

First of all, what is an *MPSoC Compiler*? The large majority of the current compilers are targeted to single-core, and the design and implementation of special compilers optimized for various core types (RISC, DSP, VLIW, among others) has been well understood and practiced. Now, the trend moving to MPSoCs raises the level of complexity of the compilers targeting these platforms. The problems of application modeling, platform description, software parallelization, software distribution, and code generation for an efficient usage of these platforms,

still remain as open issues both in academia and industry [17]. In this chapter, *MPSoC Compiler* is defined as the tool-chain to tackle those problems for a given (pre-)verified MPSoC platform.

It is worth mentioning that this definition of MPSoC compiler is slightly different from the term *software synthesis* as it appears in the hardware-software co-design community [28]. In this context, *software synthesis* emphasizes that starting from a single high-level system specification, the tools perform hardware/software partitioning and automatically synthesize the software part so as to meet the system performance requirements of the specifications. The flow is also called an application-driven "top-down" design flow. In contrast, the MPSoC compiler is used mostly in *platform-based* design, where the semiconductor suppliers evolve the MPSoC designs in generations targeting a specific application domain. The function of an MPSoC compiler is very close to that of a single-core compiler, where the compiler translates the high-level programming language (e.g., C/C++) into the machine binary code. The difference is that an MPSoC compiler needs to perform additional (and more complex) jobs over the single-core one, such as software parallelization and distribution, as the underlying MPSoC platform is by orders of magnitude more complex. Although, software synthesis and MPSoC compilers share some similarities, the major difference is that they exist in the context of different methodologies, thus focusing on different objectives [18].

The rest of the chapter is organized as follows. Section 1.2 briefly introduces the challenges of building MPSoC compilers, using a comparison of an MPSoC compiler to a single-core compiler, followed by Sect. 2, where detailed discussions are carried out. Finally, Sect. 3 looks into how the challenges are tackled by presenting case studies of MPSoC compilers from the academia and the industry.

1.2 Challenges of Building MPSoC Compilers

Before the multi-core era, single-core systems have been very successful in creating a comfortable and convenient programming environment for software developers. The success is largely due to the fact that the sequential programming model is very close to the natural way humans think and that it has been taught for decades in basic engineering courses. Also, the compilers of high-level programming languages (e.g., C/C++) for single-core are well studied, which hide nearly all hardware details from the programmers as a holistic tool [34]. User-friendly graphical integrated development environments (IDEs) like Eclipse [1] and debugging tools like `gdb` [2] also contribute to the ecosystem of hardware and software in the single-core era.

The complexity of programming and compiling for MPSoC architectures has greatly increased compared to single-core. The reasons are manifold and the most important ones are as follows. On the one hand, MPSoCs inherently ask for applications being written in parallel programming models so as to efficiently utilize the hardware resources. Parallel programming (or thinking) has been proven to be difficult for programmers, despite years of efforts invested in high-performance

computing. On the other hand, the heterogeneity of MPSoC architectures requires the compilation process to be ad-hoc. The programming models for different Processing Elements (PEs) can be different. The granularity of the parallelism might also vary. The compiler tool-chains can originate from different vendors for PEs. All those make MPSoC compilation an extremely sophisticated process, which is most likely not anymore "the holistic compiler" for the end users. Neither the software tool-chains are fully prepared to handle MPSoCs, nor productive multi-core debugging solutions are available. The software tool-chains are not yet fully prepared to well handle MPSoC systems, plus the lack of productive multi-core debugging solutions.

An MPSoC compiler, as the key tool to enable the power of MPSoCs, is known to be difficult to build. A brief list of the fundamental challenges is provided below, with an in-depth discussion in the following Sect. 2.

1. *Programming Models*: Evidently the transition to parallel programming models impacts the MPSoC compiler fundamentally.
2. *Platform Description*: The traditional single-core compiler requires architecture information, such as the instruction set and latency table in the backend to perform code generation. In contrast, the MPSoC compiler needs another type of platform description including further details, such as information about the PEs and available communication resources. This information is used in multiple phases of the compilation process beyond the backend.
3. *Software Parallelization*: While Instruction-Level Parallelism (ILP) is exploited by single-core compilers, MPSoC compilers focus on a wider variety of forms of parallelism, which are more coarse-grained.
4. *Software Distribution*: An MPSoC compiler distributes coarse-grained tasks (or code blocks), while the single-core compiler performs this at instruction-level.
5. *Code generation*: It is yet another leaping complexity for the MPSoC compiler to be able to generate the final binaries for heterogeneous PEs and the NoC compared to generate the binary for a one-ISA architecture.

2 Foundation Elements of MPSoC Compilers

This section delves into the details of the problems mentioned in the introduction of this chapter. The discussion is based on the general structure of a single-core compiler, shown in Fig. 1. The issues that make the tasks of an MPSoC compiler particularly challenging are highlighted, taking the single-core compiler technology as a reference.

A single-core compiler is typically divided into three phases: the *front end*, the *middle end* and the *back end*. The front end checks for the lexical, syntactic and semantic correctness of the application. Its output is an abstract *Intermediate Representation* (IR) of the application, which is suitable for optimizations and for code generation in the following phases of the compiler. The middle end, sometimes

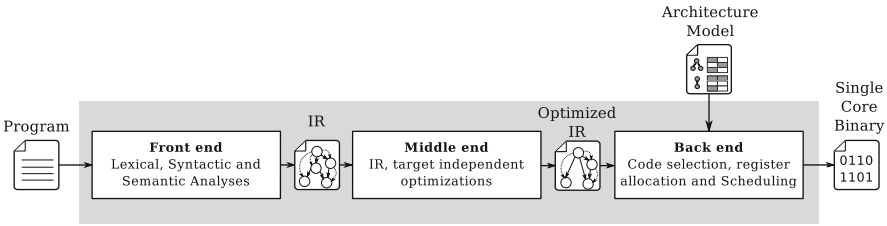


Fig. 1 Coarse view of a single-core compiler

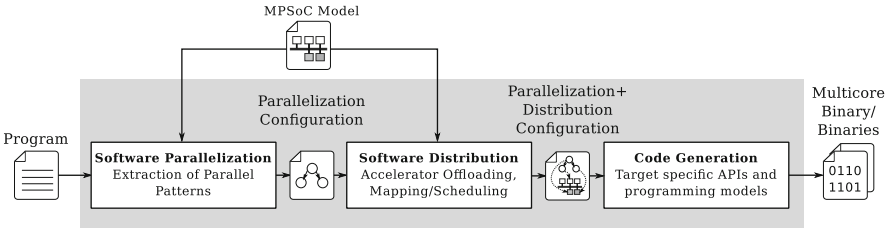


Fig. 2 Coarse view of a MPSoC compiler

conceptually included within the front end, performs different analyses on the IR. These analyses enable several target-independent optimizations that mainly aim at improving the performance of the posterior generated code. The backend is in charge of the actual code generation and is divided into phases as well. Typical backend steps include *code selection*, *register allocation* and *instruction scheduling*. These steps are machine dependent and therefore require a model of the target architecture.

MPSoC compilers are also divided into phases in order to manage complexity. The overall structure of the single-core compiler (in Fig. 1) will suffer some changes though, as Fig. 2 shows. In general, an MPSoC compiler is also divided into three phases: *software parallelization*, *software distribution* and *code generation*. Throughout this section more details about these phases will be provided, to help understanding the differences between single-core and MPSoC compilers.

2.1 Programming Models

The main entry for any compiler is a representation of an application using a given programming model, as shown in Fig. 1. A programming model is a bridge that provides humans access to the resources of the underlying hardware platform. Designing such a model is a delicate art, in which hardware details are hidden for the sake of productivity and usually at the cost of performance. In general, the more details remain hidden, the harder the job of the compiler is to close the performance gap. In this sense, a given programming model may reduce the work of the compiler

a

```
1: function S = fir(coeff, X)
2: S = coeff * X';
```

b

```
1: float coeff[N] = {0.7, ...}
2:
3: float fir (float *ptr) {
4:  int i;
5:  float sum = 0;
6:  for (i=0; i<N; i++)
7:   sum += coeff[i]*(*ptr++);
8:  return sum;
9: }
```

c

```
1: X fract coeff[N] = {0.7, ...}
2:
3: fract fir (Y fract * Y ptr) {
4:   int i;
5:   accum sum = 0;
6:   for (i=0; i<N; i++)
7:    sum += coeff[i] *
8:             (accum)(*ptr++);
9:   return sum;
10: }
```

Fig. 3 FIR implementation on different programming languages. (**a**) Matlab. (**b**) C. (**c**) DSP-C

but will never circumvent using one. Figure 3 shows an implementation of an FIR filter using different programming languages representing different programming models. This figure shows an example of the productivity-performance trade-off. On one extreme, the Matlab implementation (Fig. 3a) features high simplicity and no information of the underlying platform. The C implementation (Fig. 3b) provides more information, having types and the memory model visible to the programmer. On the other extreme, the DSP-C implementation (Fig. 3c) has explicit memory bank allocation (through the *memory qualifiers* `X` and `Y`) and dedicated data types (`accum`, `fract`). Programming at this level requires more knowledge and careful thinking, but will probably lead to better performance. Without this explicit information, a traditional C compiler would need to perform complex memory disambiguation analysis in order to place the arrays in separate memory banks.

In [11], the authors classify programming models as being either *hardware-centric*, *application-centric* or *formalism-centric*. Hardware-centric models strive for efficiency and usually require a very experienced programmer (e.g., Intel IXP-C [51]). Application-centric models strive for productivity allowing fast application development cycles (e.g., Matlab [65], LabView [57]), and formalism-centric models strive for safeness due to the fact of being verifiable (e.g., Actors [30]). Practical programming models for embedded MPSoCs cannot pay the performance overhead brought by a pure application-centric approach and will seldom restrict programmability for the sake of *verifiability*. As a consequence, programming models used in industry are typically hardware-centric and provide some means to ease programmability, as will be discussed later in this section.

Orthogonal to the previous classification, programming models can be broadly classified into *sequential* and *parallel ones*. The latter being of particular interest for MPSoC programming and this chapter's readers, though having its users outnumbered by the sequential programming community. As a matter of fact, C

and C++ are still the top languages in the embedded domain [23], which have underlying sequential semantics. Programmers have been educated for decades to program sequentially. They find it difficult to describe an application in a parallel manner, and when doing so, they introduce a myriad of (from their point of view) unexpected errors. Apart from that, there are millions of lines of sequential legacy code that will not be easily rewritten within a short period of time to make use of the new parallel architectures. Parallel programming models for heterogeneous architectures can be further classified as *host-centric* and *non-host centric*. In the host-centric approach the PEs in the platform have specific roles, either as hosts or accelerators. Here the execution is controlled by the hosts and eventually they offload computationally intensive code blocks to specialized accelerators to improve performance. In contrast, in the non-host centric approach code blocks are assigned to PEs without assuming any specific role for each of them and the control flow is distributed.

Compiling a sequential application, for example written in C, for a simple core is a very mature field. Few people would program an application in assembly language for a single issue embedded RISC processor, such as the ARM7 or the MIPS32. In general, compiler technology has advanced greatly in the single-core domain. Several optimizations have been proposed for superscalar processors [40], DSPs [49], VLIW processors [24] and for exploiting *Single Instruction Multiple Data* (SIMD) architectures [50]. Nonetheless, high performance routines for complex processor architectures with complex memory hierarchies are still hand-optimized and are usually provided by processor vendors as library functions. In the MPSoC era, the optimization space is too vast to allow hand-crafted solutions across different cores. The MPSoC compiler has to help the programmer to optimize the application, possibly taking into account optimized routines for some of the processing elements.

In spite of the efforts invested in classical compiler technology, plain C programming is not likely to be able to leverage the processing power of future MPSoCs. When coding a parallel application in C, the parallelism is hidden due to the inherent sequential semantics of the language and its centralized control flow. Retrieving this parallelism requires complex dataflow and dependence analyses which are usually NP-complete and sometimes even undecidable (see Sect. 2.3.2). For this reason MPSoC compilers need also to cope with parallel programming models, some of which will be introduced in the following.

2.1.1 Mainstream Parallel Programming Models

There are manifold parallel programming models. Modern parallel programming models are built on top of traditional sequential languages like C or C++ by means of compiler directives, libraries or language extensions. These models are usually classified by the underlying memory architecture that they support; either shared or distributed. They can be further classified by the parallel patterns that they allow to express (see Sect. 2.3.3). Today a great majority of the mainstream parallel programming models are industry standards, which have a solid tooling support and

are constantly evolving to satisfy the needs of developers and to exploit the new features of modern multi-core platforms. These programming models have their roots in the *High Performance Computing* (HPC) community, however, they have gained acceptance in the embedded domain [5, 41, 71, 74]. Prominent examples of these models are presented in the following:

- **POSIX Threads (Pthreads):** This is a library-based shared memory parallel programming model [69]. Pthreads is a low level approach, as the developer has to explicitly create and destroy threads, partition the workload, map the threads to cores and ensure a proper thread synchronization. The accesses to shared data (critical sections) have to be carefully designed to avoid *data races* and *deadlocks*. The protection to the critical sections can be achieved by means of mutual exclusion (mutex) or semaphores.
- **OpenMP:** This is an industry standard parallel programming model for shared memory systems based on compiler directives [3]. The use of compiler directives implies minimal source code modifications in contrast to Pthreads. Moreover, thread management in OpenMP is performed by a runtime system, which further simplifies the challenging task of multi-core programming. Initially, OpenMP focused on regular loop level parallelism for homogeneous multi-core platforms. However, it was later extended to support both irregular parallelism by means its *tasking model*, and heterogeneous platforms by means of its *accelerator model*. The accelerator model is particular important for the embedded domain, as it enables the designer to exploit all types of cores in heterogeneous MPSoC, including DSPs [71, 74]. Furthermore, recent research efforts have confirmed the applicability of OpenMP in the embedded domain, as it has been demonstrated that it is feasible to use it in real time systems [77].
- **OpenCL:** This is a parallel programming model for heterogeneous systems, which is also an industry standard [70]. OpenCL follows a host-centric approach in which a host device (e.g., CPU) offloads data and computation typically to accelerator devices (e.g., GPUs or DSPs). In this programming model, computations are described as kernels, which are the basic units of execution (e.g., one iteration of a parallel loop). Kernels are written in a language called *OpenCL C*, which is simultaneously a subset and a superset of the C99 standard. In addition, OpenCL offers an API that allows the host to manage data transfers and kernel execution on the target devices. In the embedded domain OpenCL has also gained acceptance, and it is already available for a wide variety of heterogeneous embedded platforms [41, 74].
- **MPI:** This is a parallel programming model for distributed systems based on a library. It relies on the message passing principle, where both point-to-point and collective form communications are supported. MPI can be used in combination with other parallel programming models for shared memory systems, such as OpenMP. While MPI allows to exploit parallelism across nodes in a distributed system, OpenMP allows to exploit parallelism within each node. This approach is usually referred as *hybrid programming* [64]. MPI is currently the *de facto* standard for distributed systems in HPC, and it has been also applied in the embedded domain [5, 74].

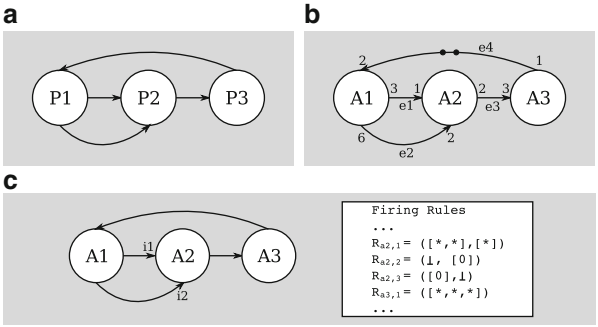


Fig. 4 Example of concurrent MoCs (P: Process, A: Actor). (**a**) KPN. (**b**) SDF. (**c**) DDF

2.1.2 Dataflow Programming Models

Dataflow or streaming *Models of Computation* (MoCs) appear to be one promising choice for describing signal processing applications. In dataflow programming models, an application is represented as a graph. The nodes of this graph (also called processes or actors) perform computation whereas the edges (also called channels) are used to transfer data among nodes. These MoCs originated from theoretical computer science for formally describing a computing system and were initially used to compute bounds on complexity. MoCs were thereafter used in the early 1980s to model VLSI circuits and only in the 1990s started to be utilized for modeling parallel applications. Dataflow programming models based on concurrent MoCs like *Synchronous Dataflow* (SDF) [46] and some extensions (like *Boolean Dataflow* (BDF) [47]) have been deeply studied in [68]. More general dataflow programming models based on *Dynamic Dataflow* (DDF) and *Kahn Process Networks* (KPN) [36] MoC have also been proposed [44, 58] (see also Chapters [12, 26]).

- **KPN Programming Model:** In this programming model, an application is represented as a graph $G = (V, E)$ like the one in Fig. 4a. In such a graph, a node $p \in V$ is called *process* and represent computation. The edges represent unbounded FIFO channels for processes communication by means of data items or *tokens*. Processes can only be in one of two states: ready or blocked. The blocked state can only be reached by reading from *only one* empty input channel — *blocking read* semantics. A KPN is said to be determinate: the history of tokens produced on the communication channels is independent of the scheduling.
- **DDF Programming Model:** In this programming model, an application is also represented as a graph $G = (V, E, R)$ with R a family of sets, one set for every node in V. Edges have the same semantics as in the KPN model. Nodes are called *actors* and do not feature the blocking read semantics of KPN. Instead, every actor $a \in V$ has a set of *firing rules* $R_a \in R$, $R_a = \{R_{a,1}, R_{a,2}, \ldots\}$.

A firing rule for an actor $a \in V$ with p inputs is a p-tuple $R_{a,i} = (c_1, \ldots, c_p)$ of conditions. A condition describes a sequence of tokens that has to be available at the given input FIFO. Parks introduced a notation for such conditions in [61]. The condition $[X_1, X_2, \ldots, X_n]$ requires n tokens with values $X_1, X_2, \ldots, X_n$ to be available at the top of the input FIFO. The conditions $[*]$, $[*, *]$, $[*_{(1)}, \ldots, *_{(m)}]$ require at least 1, 2 and m tokens respectively with arbitrary values to be available at the input. The symbol $\perp$ represents any input sequence, including an empty FIFO. For an actor a to be in the ready state at least one of its firing rules need to be satisfied. An example of a DDF graph is shown in Fig. 4c. In this example, the actor a_2 has three different firing rules. This actor is ready if there are at least two tokens in input `i1` and at least 1 token in input `i2`, or if the next token on input `i2` or `i1` has value 0. Notice that more than one firing rule can be activated, in this case the dataflow graph is said to be non-determinate.

- **SDF Programming Model:** An SDF can be seen as a simplification of DDF model,[1] in which an actor with p inputs has only one firing rule of the form $R_{a,1} = (n_1, \ldots, n_p)$ with $n \in \mathbb{N}$. Additionally, the amount of tokens produced by one execution of an actor on every output is also fixed. An SDF can be defined as a graph $G = (V, E, W)$ where $W = \{w_1, \ldots, w_{|E|}\} \subset \mathbb{N}^3$ associates three integer constants $w_e = (p_e, c_e, d_e)$ to every channel $e = (a_1, a_2) \in E$. p_e represents the number of tokens produced by every execution of actor a_1, c_e represents the number of tokens consumed in every execution of actor a_2 and d_e represents the number of tokens (called delays) initially present on edge e. An example of an SDF is shown in Fig. 4b with delays represented as dots on the edges. For the SDF in the example, $W = \{(3, 1, 0), (6, 2, 0), (2, 3, 0), (1, 2, 2)\}$.

Different dataflow models differ in their expressiveness, some being more general, some being more restrictive. By restricting the expressiveness, models possess stronger formal properties (e.g., determinism) which make them more amenable to analysis. For example, since the token consumption and production of an SDF actor are known beforehand, it is possible for a compiler to compute a plausible static schedule for an SDF. For a KPN instead, due to control dependent access to channels, it is impossible to compute a pure static schedule.

Apart from explicitly exposing parallelism, dataflow programming models became attractive mainly for two reasons. On the one hand, they are well-suited for graphical programming, similar to the block diagrams used to describe signal processing algorithms. On the other hand, some of the underlying MoC's properties facilitate the analysis performed by the tools. For example, channels explicitly expose data dependencies among computing processes/actors, and they have a distributed control flow which is easily mapped to different PEs.

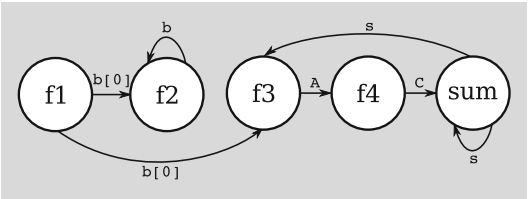
To understand how dataflow models can potentially reduce the compilation effort, an example of an application written in a sequential and in two parallel forms is shown in Fig. 5. Let us assume that the KPN parallel specification in Fig. 5a

[1] Being more closely related to the so-called *Computation Graphs* [38].

a

```
1:  int b[N];
2:  int A[4][N];
3:  int c[4];
4:
5:  void foo()
6:  {
7:      int i,j,s;
8:
9:      b[0] = f1();
10:     for (i = 0; i < N; i++)
11:       f2(&b[0], &b[i], &A[i%4][0]);
12:     s = b[0];
13:     b[0] = 0;
14:     for (i = 0; i < 4*16; i++) {
15:         j = i % 4;
16:       f3(s, &A[j][0]);
17:       c[j] = f4(&A[j][0]);
18:       if (j == 3)
19:         s += sum(&c[0]);
20:     }
21: }
```

b



c

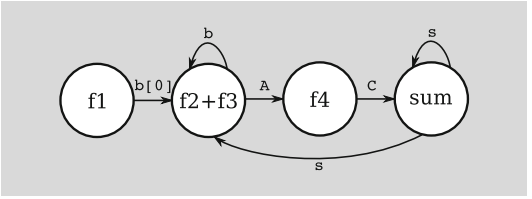


Fig. 5 KPN example. (**a**) C implementation. (**b**) A "Good" KPN representation. (**c**) A "Bad" KPN representation

represents the desired output of a parallelizing compiler. In order to derive this KPN from the sequential specification in Fig. 5b, complex analyses have to be performed. For example, the compiler needs to identify that there is no dependency on array `A` among lines `11` and `16` (i.e., between `f2` and `f3`), which is a typical example of dataflow analysis (see Sect. 2.3.4). Only for a restricted subset of C programs, namely *Static Affine Nested Loop Programs* (SANLP), similar transformations to that shown in Fig. 5 have been implemented in [78]. Therefore, starting from a specification already parallel greatly simplifies the work of the compiler.

However, even with a parallel specification at hand, an MPSoC compiler has to be able to look inside the nodes in order to attain higher performance. With the applications becoming more and more complex, a compiler cannot completely rely on the programmer's knowledge when decomposing the application into blocks. A block diagram can hide lots of parallelism in the interior of the blocks and thus, computing nodes cannot always be considered as *black boxes* but rather as *gray/white boxes* [52]. As an example of this, consider the KPN shown in Fig. 5a. Assume that this parallel specification was written by a programmer to represent the same application logic in Fig. 5a. This KPN might seem appropriate to a programmer, because the communication is reduced (five instead of six edges). However, if functions `f2` and `f3` are time consuming, running them in parallel could be advantageous. However, in this representation the parallelism remains hidden inside block `f2+f3`.

Summary
Currently, MPSoC compilers should support sequential programming models as input, both because of the great amount of existing sequential legacy code and because of the generations of programmers that were taught to program sequentially. At the same time, the MPSoC compilers need to be aware of the properties of the target parallel programming models, particularly the forms of parallelism that they allow to express, as it will be discussed in Sect. 2.3.3.

2.2 Platform Description for MPSoC Compilers

After performing optimizations in the middle end, a single-core compiler backend generates code for the target platform based on a model of it. Such a platform model is also required by an MPSoC compiler, but in contrast to a single-core compiler flow, the architecture model may also be used during multiple phases of the compiler and not just by the backend, as Fig. 2 shows. For example, if the programming model exposes some hardware details to the user, the front end needs to be able to cope with that and eventually perform consistency checks. Besides, some MPSoC optimizations in the middle end may need some information about the target platform as discussed in Sect. 2.3. Traditionally an architecture model describes:

- **Available operations**: In form of an abstract description of the *Instruction Set Architecture* (ISA). This information is mainly used by the code selection phase.
- **Available resources**: A list of hardware resources such as registers and functional units (in case of a superscalar or a VLIW). This information is used, for example, by the register allocator and the scheduler.
- **Communication links**: Describe how data can be moved among functional units and register files (e.g., cross paths in a cluster VLIW processor).
- **Timing behavior**: In form of *latency* and *reservation tables*. For each available operation, the latency table tells the compiler how long it takes to generate a result, whereas the reservation table tells the compiler which resources are blocked and for how long. This information is mainly used to compute the schedule.

In the case of an MPSoC, a platform description has to provide similar information but at a different level. Instead of a representation of an ISA, the available operations describe which kinds of processors and hardware accelerators are in the platform. Instead of a list of functional units, the model provides a list of PEs and a description of the memory subsystem. The communication links represent no longer interconnections among functional units and register files, but possibly a complex Network-On-Chip (NoC) that interconnects the PEs among them and with

the memory elements. Finally, the timing behavior has to be provided for individual operations (instructions).

Usually, the platform description is a graph representation provided in a given format (usually XML files, see Sect. 3 for practical examples). Recently, the *Multi-core Association* has introduced a standard to specify multi-core platforms called *Software-Hardware Interface for Multi-Many-Core* (SHIM) [56]. This standard allows the abstraction of hardware properties that are key to enable multi-core tools. The SHIM implementation is based on XML files that describe the core types and the platform itself.

One of the main uses of the platform description is to enable the performance estimation of applications. Getting the timing behavior of given code blocks running on a particular MPSoC platform, is a major research topic and a requisite for an MPSoC compiler. Several performance estimation techniques, are applied in order to get specific execution times: *Worst/Best/Average Case Execution Time* (W/B/ACET) [79]. These techniques can be roughly categorized as follows [16]:

- **Analytical:** Analytical or static performance estimation tries to find theoretical bounds to the WCET, without actually executing the code. Using compiler techniques, all possible control paths are analyzed and bounds are computed by using an abstract model of the architecture. This task is particularly difficult in the presence of caches and other non-deterministic architectural features. For such architectures, the WCET might be too pessimistic and thus induces bad decisions (e.g., wrong schedules). There are already some commercial tools available for such purposes, aiT [4] is a good example.
- **Emulation-based:** The simulation time of cycle accurate models can be prohibitively high. Typical simulation speeds range from 1 to 100 KIPS (Kilo Instructions per Second). Therefore, some techniques emulate the timing behavior of the target platform in the host machine without modeling every detail of the processor by means of instrumentation. Source level timing estimation has proven to be useful for simple architectures [33, 39], the accuracy for VLIW or DSP processors is however not satisfactory. The authors in [25] use so-called *virtual back ends* to perform timing estimation by emulating the effects of the compiler back end and thus improving the accuracy of source level timing estimation considerably. With these techniques, simulation speeds of up to 1 GIPS are achievable.
- **Simulation-based:** In this case the execution times are measured on a simulator. Usually cycle accurate virtual platforms are used for this purpose [72]. Virtual platforms allow full system simulation, including complex chip interconnects and memory subsystems. Simulation-based models suffer from the *context-subset* problem, i.e., the measurements depend on the selection of the inputs.
- **Table-based:** This is a performance estimation technique based on source code instrumentation and a table with the costs of elementary processor operations. The cost of executing every elementary operation is based on the cost provided by the architecture model and the execution counts provided by the profiling information resulting from the execution of the instrumented code. This approach

allows to identify application hot spots and provides an early idea of the application runtime. However, it is not very accurate, in particular for non-scalar architectures such as VLIW.

Summary
Platform models for MPSoC compilers describe similar features to those of traditional compilers but at a higher level. Processing elements and NoCs take the place of functional units, register files and their interconnections. On an MPSoC compiler, the platform model is no longer restricted to be used on the back end but a subset of it may be used by the front end and the middle end. Out of the information needed to describe the platform, the timing behavior is the most challenging. This timing information is needed for performing successfully software parallelization and distribution, as it will be described in the next sections.

2.3 Software Parallelization

The software parallelization phase of an MPSoC compiler aims at identifying profitable parallelization opportunities hidden in legacy sequential code. The following sections will give more insights on the main challenges for software parallelization, namely the selection of an intermediate representation, the granularity issue, prominent parallel patterns and the problem of dataflow analysis.

2.3.1 Intermediate Representation (IR)

In a classical compiler, the front end translates application code into an *Intermediate Representation* (IR). Complex constructs of the original high level programming languages are lowered into the IR while keeping machine independence. The IR serves as basis for performing analysis (e.g., control and data flow), upon which many compiler optimizations can be performed. Although there is no *de facto* standard for IRs, most compiler IRs use graph data structures to represent the application. The fundamental analysis units used in traditional compilers are the so-called *Basic Blocks* (BB), where a BB is defined as *a maximal sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end* [10]. A procedure or function is represented as a *Control Flow Graph* whose nodes are BBs and edges represent the control transitions in the program. Data flow is analyzed inside a BB and as a result a *Data Flow Graph* is produced, where nodes represent statements (or instructions) and edges represent data dependencies (or precedence constraints).

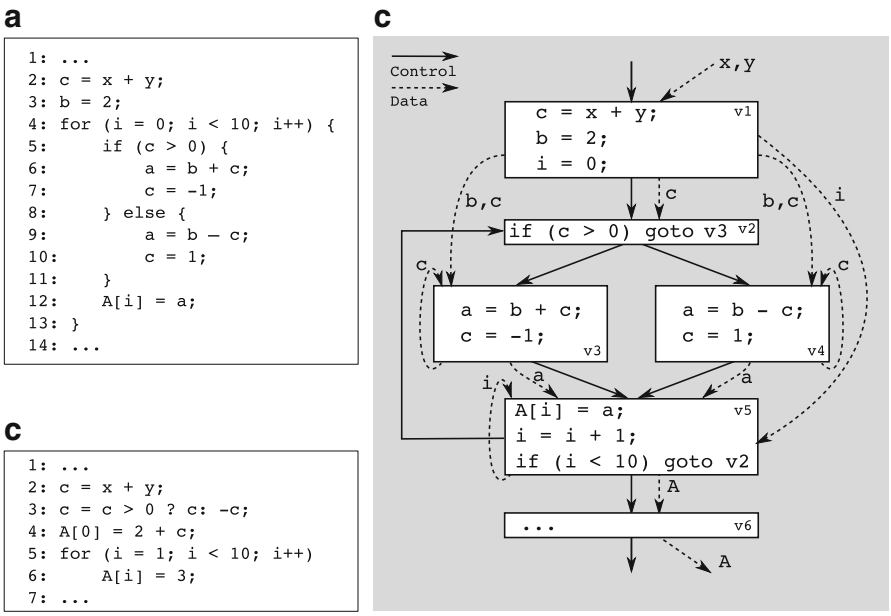


Fig. 6 Example of a CDFG. (**a**) Sample C code. (**b**) Optimized code. (**c**) CDFG for (**a**)

With intra-procedural analysis, data dependencies that span across BB borders can be identified. As a result both control and data flow information can be summarized in a *Control Data Flow Graph* (CDFG). A sample CDFG for the code in Fig. 6a is shown in Fig. 6c. BBs are identified with the literals `v1`,`v2`,...,`v6`. For this code it is easy to identify the data dependencies by simple inspection. Notice however, that the self-cycles because of variable `c` in `v3` and `v4` will never be executed, i.e., the definition of `c` in line `7` will never *reach* line `6`. Moreover, notice that the code in Fig. 6a is equivalent to that in Fig. 6b. Even for such a small program, a compiler needs to be equipped with powerful analysis to derive such an optimization.

For simple processors, the analysis at the BB *granularity* has been considered the state-of-the-art during the last decades. The instructions inside a BB will always be executed one after another in an in-order processor, and for that reason BBs are very well-suited for exploiting ILP. Already for more complex processors, like VLIW, BBs fall short to leverage the available ILP. *Predicated execution* and *software pipelining* [24] are just some examples of optimizations that cross the BB borders seeking for more parallelism. This quest for parallelism is even more challenging in the case of MPSoC compilers, as they must go beyond ILP. The question of granularity and its implication on parallelism becomes a major issue. The *ideal* granularity depends on the characteristics of the form of parallelism and of the target platform. Therefore, extensions to the CDFG have been proposed to address the granularity issue. One example of this is the *Statement Control Data Flow Graph* (SCDFG) [19] in which nodes are single statements instead of BBs to allow more flexibility. More insights on the granularity issue are provided in Sect. 2.3.2.

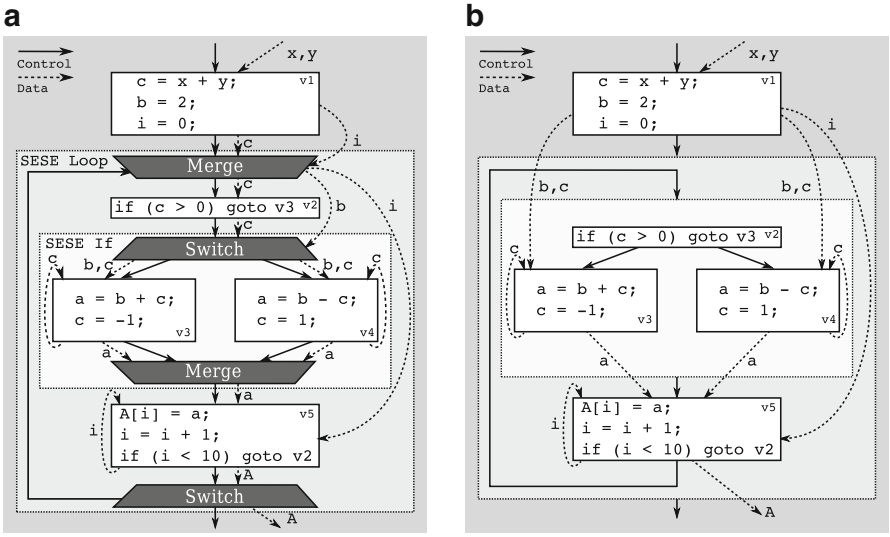


Fig. 7 Hierarchical IRs examples for the code in Fig. 6a. (**a**) DFG. (**b**) HTG

Another major issue for MPSoC compilers is the size of the solution space, which could be prohibitively large even for small applications. This issue has been addressed by introducing the notion of hierarchy in the IR, by also retaining high level information about program structure in the intermediate representation, such as loops and conditional blocks. This is a powerful property that enables a divide-and-conquer parallelization approach in which code regions can be analyzed in isolation based on their type. The *Dependence Flow Graph* (DFG) [35] and the *Hierarchical Task Graph* (HTG) [63] are examples of representations that incorporate the notion of hierarchy, which have been already used in existing MPSoC compilers [7, 8, 22]. Figure 7a shows an example of a DFG for the code presented in Fig. 6a. The DFG incorporates the notion of hierarchy by means of the so-called *Single-Entry Single-Exit* (SESE) regions. A SESE region is a sub-graph of the DFG, which has a unique incoming control edge leading to the region execution, and a unique outgoing control edge that exits the region. Regions can be nested or sequentially ordered and they can be statements, basic blocks, loops or conditional blocks (e.g. if or switch-case constructs). SESE regions related to loops and conditional blocks are enclosed by artificial nodes, namely `switch` and `merge`, as Fig. 7a illustrates. A key feature of the artificial nodes is that they allow to re-route data dependencies inside regions where they are relevant. For example, in Fig. 7a the data dependencies edges on `b` and `c` are re-routed inside the region `SESE If`, while the data dependency edge on `i` is bypassed, as it is not relevant for that particular region. This feature is useful not only for software parallelization analysis, but also for parallel code generation [9]. An example of a HTG for the code in Fig. 6a is presented in Fig. 7b. The aim of the HTG is to hide cyclic dependencies by leveraging the explicit

hierarchy in a program. In general, the HTG has two main types of nodes: *simple* and *compound*. Single nodes are used to encapsulate a single statement or basic block, while compound nodes introduce hierarchy, as they contain other single or compound nodes. Compound nodes are the counter part of SESE regions in a DFG, as they represent high level program constructs (e.g., loops or conditional blocks). However, the drawback of the HTG is that it has no artificial nodes that allow to re-route data dependencies in and out of the compound nodes, which makes data dependence analysis more challenging.

2.3.2 Granularity and Partitioning

Granularity is one major issue for software parallelization and has a direct impact on the form and degree of parallelism that can be achieved [6]. We define partitioning as the process of analyzing an application and fragmenting it into blocks with a given granularity suitable for parallelism extraction. In this sense, the process of constructing CFGs out of an application as discussed before can be seen as a partitioning. The following are the most intuitive granularities for MPSoC compilers:

- **Statement:** A statement is the smallest standalone entity of a programming language. An application can be broken to the level of statements and the relations among each of them. The statements could be simple expressions, such as arithmetic operations or function calls. This granularity provides the highest degree of freedom to the analysis but could prevent ILP from being exploited at the single-core level. Moreover, the parallelization overhead for such small granularity could be prohibitively large.
- **Basic Block:** As already discussed, traditional compilers work on the level of BBs, as they are well suited for ILP. However, in practice BBs could be either too big or too small for coarse-grained parallelism extraction. A BB composed of a sequence of function calls inside a loop would be seen as a single node, and potential parallelism will be therefore hidden. On the other extreme, a couple of small basic blocks divided by simple control constructs could be better handled by a single-core compiler with support for predicated execution.
- **Function:** A function is defined as a subroutine with its own stack. At this level, only function calls are analyzed and the rest of the code is considered as irrelevant. As with BBs, this granularity can be too coarse or too fine-grained depending on the application. It is possible to force a coding style, where parallelism is explicitly written in a way that the behavior is factorized into functions. However, an MPSoC compiler should not make any assumption on the coding style.

As an example, partitions at different granularity levels for the program introduced in Fig. 5a are shown in Fig. 8. The partition at statement level is shown in Fig. 8a. In this example the statements at lines `12`, `13` and `15` are too light weight. The partition of function `foo` at BB level is shown in Fig. 8b. The BB on line `9` is

a

```
1:   ...
9:   b[0] = f1();
10:  for (i = 0;...)
11:     f2(&b[0],...);
12:  s = b[0];
13:  b[0] = 0;
14:  for (i = 0;...) {
15:     j = i % 4;
16:     f3(s,&A[j][0]);
17:     c[j] = f4(...);
18:     if (j == 3)
19:        s += sum(&c[0]);
20: }
```

b

```
1:   ...
9:   b[0] = f1();
10:  for (i = 0;...)
11:     f2(&b[0],...);
12:  s = b[0];
13:  b[0] = 0;
14:  for (i = 0;...) {
15:     j = i % 4;
16:     f3(s,&A[j][0]);
17:     c[j] = f4(...);
18:     if (j == 3)
19:        s += sum(&c[0]);
20: }
```

c

```
1:   ...
9:   b[0] = f1();
10:  for (i = 0;...)
11:     f2(&b[0],...);
12:  s = b[0];
13:  b[0] = 0;
14:  for (i = 0;...) {
15:     j = i % 4;
16:     f3(s,&A[j][0]);
17:     c[j] = f4(...);
18:     if (j == 3)
19:        s += sum(&c[0]);
20: }
```

Fig. 8 Granularity examples. (**a**) Statement. (**b**) Basic block. (**c**) Function

too light weight in comparison to the other BBs, whereas the BB in lines `16-17` may be too coarse. Finally, the partition at function level is shown in Fig. 8c. This partition happens to match the KPN derivation introduced in Fig. 5b. Whether this granularity is appropriate or not, depends on the amount of data flowing between the functions and the timing behavior of each one of the functions.

As illustrated with the examples, it is not clear what will be the ideal granularity for an MPSoC compiler to work on. Existing research efforts have been directed towards the identification of a suitable granularity for particular parallelism patterns and platforms [7, 20, 22]. The approach is usually based on partitioning an application into *code blocks* of arbitrary granularity by means of heuristics or clustering algorithms, which use the previously described granularities as the starting point (i.e., a code block is built by clustering multiple statements). In the remaining of this chapter we refer to code blocks as statements, BBs, SESE regions, functions or the result of clustering algorithms.

2.3.3 Parallelism Patterns

While a traditional compiler tries to exploit fined-grained ILP, the goal of an MPSoC compiler is to extract coarser parallelism. The most prominent forms of coarse-grained parallelism are illustrated in Fig. 9 and described in the following.

- ***Task Level Parallelism* (TLP):** In TLP different tasks can compute in parallel on different data sets as shown in Fig. 9a. This form of parallelism is inherent to programming models based on concurrent MoCs (see Sect. 2.1). Tasks may have dependencies to each other, but once a task has its data ready, it can execute in parallel with the already running tasks in the system. Typically, TLP can be exploited by the parallel execution of independent function calls or loops.
- ***Data Level Parallelism* (DLP):** In DLP the same computational task is carried out on several disjoint *Data Sets*, as illustrated in Fig. 9b. This is one of the most scalable forms of parallelism. DLP is typically present in multimedia applications, where a decoding task performs the same operations on different portions of an image or video. Several programming models provide support for DLP, e.g. OpenMP by means of its `for` construct.

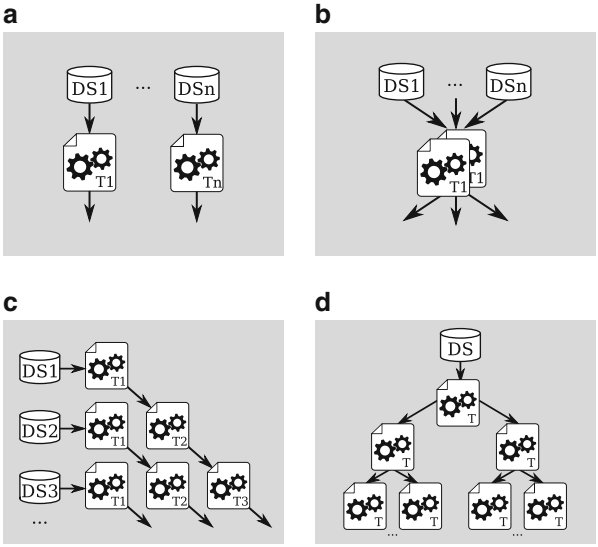


Fig. 9 Parallelism patterns. (**a**) TLP. (**b**) DLP. (**c**) PLP. (**d**) RLP

- ***Pipeline Level Parallelism* (PLP):** In PLP a computation within a loop is broken into a sequence of tasks called *stages*, as Fig. 9c shows. These tasks follow a producer-consumer relationship in which there is a flow of data from the first to the last stage. PLP is a well-suited form of parallelism for streaming applications in the embedded domain, in which there are serially dependent tasks that continuously operate on a flow of data (e.g., audio/video encoding-decoding).
- ***Recursion Level Parallelism* (RLP):** In RLP tasks are created from self-calls in functions that exhibit multiple recursion (i.e., recursive functions that contain two or more self-calls). Applications with multiple recursion typically implement divide-and-conquer algorithms, which recursively break problems into smaller sub-problems that are more simple to solve. A scalable form of nested parallelism can be exploited if the sub-problems are independent (i.e., the recursive call-sites are mutually independent). In RLP each task can further spawn parallel work as nested tasks in subsequent recursive calls, as illustrated in Fig. 9d.

Exploiting these kinds of parallelism is a must for an MPSoC compiler, which has to be therefore equipped with powerful flow and dependence analysis capabilities.

2.3.4 Flow and Dependence Analysis

Flow analysis includes both control and data flow. The result of these analyses can be summarized in a CDFG, a DFG or a HTG, as discussed at the beginning of this section. Data flow analysis serves to gather information at different program points,

e.g., about available defined variables (*reaching definitions*) or about variables that will be used later in the control flow (*liveness analysis*). As an example, consider the CDFG in Fig. 6c in which a reaching definitions analysis is carried out. The analysis tells, for example, that the value of variable `c` in line `5` can come from three different definitions in lines `2`, `7` and `10`.

Data flow analysis deals mostly with scalar variables, like in the previous example, but falls short when analyzing the flow of data when explicit memory accesses are included in the program. In practice, memory accesses are very common through the use of pointers, structures or arrays. Additionally, in the case of loops, data flow analysis only says if a definition reaches a point but does not specify exactly in which iteration the definition is made. The analyses that answer these questions are known as *array analysis*, *loop dependence analysis* or simply *dependence analysis*.

Given two statements $S1$ and $S2$, dependence analysis determines if $S2$ depends on $S1$, i.e., if $S2$ cannot execute before $S1$. If there is no dependency, $S1$ and $S2$ can execute in any order or in parallel. Dependencies are classified into control and data:

- Control Dependency: A statement $S2$ is control dependent on $S1$ ($S1\ \delta^c\ S2$) if whether or not $S2$ is executed depends on $S1$'s execution. In the following example, $S1\ \delta^c\ S2$:

```
S1: if (a > 0) goto L1;
S2: a = b + c;
S3: L1: ...
```

- Data Dependencies:
  - *Read After Write* (RAW, also *true/flow dependency*): There is a RAW dependency between statements $S1$ and $S2$ ($S1\ \delta^f\ S2$) if $S1$ modifies a resource that $S2$ reads thereafter. In the following example, $S1\ \delta^f\ S2$:

```
S1: a = b + c;          S2: d = a + 1;
```

  - *Write After Write* (WAW, also *output dependency*): There is a WAW dependency between statements $S1$ and $S2$ ($S1\ \delta^o\ S2$) if $S2$ modifies a resource that was previously modified by $S1$. In the following example, $S1\ \delta^o\ S2$:

```
S1: a = b + c;          S2: a = d + 1;
```

  - *Write After Read* (WAR, also *anti-dependency*): There is a WAR dependency between statements $S1$ and $S2$ ($S1\ \delta^a\ S2$) if $S2$ modifies a resource that was previously read by $S1$. In the following example, $S1\ \delta^a\ S2$:

```
S1: d = a + 1;          S2: a = b + c;
```

Obviously, two statements can exhibit different kinds of dependencies simultaneously. Computing these dependencies is one of the most complex tasks inside a compiler, both for single-core and for multi-core systems. For a language like C,

a

```
S1: for (i = N; i > X; i++) {
S2:     for (j = M; j > Y; j--) {
S3:         A[3*i + 2*j + 2][i + j + 1] = ...;
S4:         ... = A[4*i + j + 1][2*i + j + 1];
S5:     }
S6: }
```

b

```
S1: while (i) {
S2:     f1(&A[i]);
S3:     ... = A[f2(i)];
S4: }
```

c

```
S1: scanf("%d", &i);
S2: scanf("%d", &j);
S3: A[i] = ...;
S4: ... = A[j];
```

Fig. 10 Examples of dependence analysis. (**a**) NP complete. (**b**) Inter-procedural analysis. (**c**) Undecidable

the problem of finding all dependencies *statically* is NP complete and in some cases undecidable. The main reason for this being the use of pointers [31] and indexes to data structures that can only be resolved at runtime. Figure 10 shows three sample programs to illustrate the complexity of dependence analysis. In Fig. 10a, in order to determine if there is a RAW dependency between $S3$ and $S4$ ($S3\delta^f S4$) across iterations, one has to solve a constrained integer linear system of equations, which is NP complete. For the example, the system of equations is:

$$3x_1 + 2x_2 + 2 = 4y_1 + y_2 + 1$$
$$x_1 + x_2 + 1 = 2y_1 + y_2 + 1$$

subject to $X < x_1, y_1 < N$ and $Y < x_2, y_2 < M$. Notice for example that there is a RAW dependency between iterations $(1, 1)$ and $(2, -2)$ on `A[7][3]`. In order to analyze the sample code in Fig. 10a, b compiler has to perform inter-procedural analysis to identify if `f1` modifies the contents of `A[i]` and to sort out the potential return values of `f2(i)`. This problem could be potentially undecidable. Finally, the code in Fig. 10c is an extreme case of the previous one, in which it is impossible to know the values of the indexes at compile time. The complexity of dependence analysis motivated the introduction of memory disambiguation at the programming language level, such as the `restrict` keyword in C99 standard [80].

For an MPSoC compiler, the situation is not different. The same kind of analysis has to be performed at the granularity produced by the partitioning step. Array analysis could still be handled by a vectorizing compiler for one of the processors in the platform. The MPSoC compiler has to perform the analysis at a coarser granularity level in which function calls will not be an exception. This is for example the case for the code in Fig. 5a. In order to derive KPN representations, like those presented in Fig. 5b and c, the compiler needs to be aware of the side effects of all functions. For example, it has to make sure that function `f2` does not modify

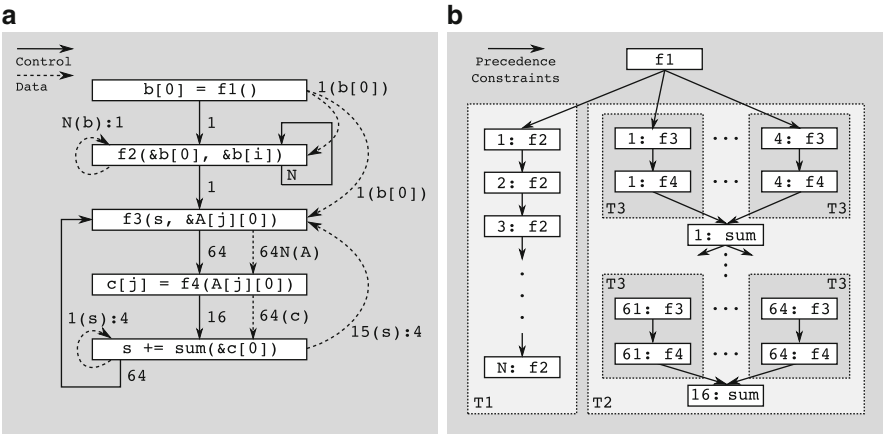


Fig. 11 Dependence analysis on example in Fig. 5a. (**a**) Summarized CDFG. (**b**) Unrolled dependencies

the array `A`, otherwise there would be a dependency (an additional channel in the KPN) between processes `f2` and `f3` in Fig. 5b. The dependence analysis should also provide additional information, for example, that the `sum` function is only executed every four iterations of the loop. This means that every four instances of `f3` followed by `f4` can be executed in parallel. This is illustrated in Fig. 11. A summarized version of the CDFG is shown in Fig. 11a. In this graph, data edges are annotated with the variable that generates the dependency and, in the case of *loop-carried* dependencies, with the *distance* of the dependency [54]. The distance of a dependency tells after how many iterations a defined value will be actually used. With the dependency information, it is possible to represent the precedence constraints along the execution of the whole program as shown in Fig. 11b. In the figure, `n: f` represents the *n*-th execution of function `f`. With this partitioning, it is possible to identify two different forms of parallelism: `T1` and `T2` represent TLP, whereas `T3` represents DLP. This is a good example where flow and dependence analysis help determining a partitioning that exposes coarse grained parallelism.

Due to the complexity of static analyses, multiple research groups started to rely on *Dynamic Data Flow Analysis* DDFA [7, 20, 76]. Unlike static analyses, where dependencies are determined at compile time, DDFA uses traces obtained from profiling runs. This analysis is of course not fully safe and the results need approval from the developer. In general, DDFA is used to obtain a coarse measure of the data flowing among different portions of the application in order to derive plausible partitions and in this way identify DLP, TLP, PLP and/or RLP. Being a profile-based technique, the quality of DDFA depends on a careful selection of the input stimuli. In interactive programming environments, DDFA can provide hints to the programmer about where to perform code modifications to expose more parallelism.

Summary
Traditional compilers work at the basic block granularity which is well suited for ILP. MPSoC compilers in turn need to be equipped with powerful flow analysis techniques, that allow to partition the application into a suitable granularity. This granularity may not match any mainstream granularity and may depend on the parallel pattern. The partitioning step must break the application into code blocks from which coarse level parallelism such as DLP, TLP, PLP or RLP can be extracted.

2.4 Software Distribution

The software distribution phase in an MPSoC compiler aims at deciding *where* and *when* to execute tasks of a parallel application on the target platform. In this chapter we discuss two forms of software distribution: (1) *accelerator offloading* in host-centric programming models and (2) *mapping and scheduling* of dataflow MoCs.

2.4.1 Accelerator Offloading

The use of specialized accelerators, such as DSPs and GPUs, has gained popularity due to their high peak performance/watt ratio in contrast to homogeneous multi-cores. However, the heterogeneity introduced by the accelerators makes the programmability of these platforms a complex task. Therefore, multiple host-centric parallel programming models have been proposed to address the challenge of accelerator computing (see Sect. 2.1.1). These models can be classified as *low-level*, such as OpenCL, or high-level directive-based, such as the OpenMP accelerator model. Despite these efforts to provide a convenient programming model, developers still have to manually specify the code regions to be offloaded and the data to be transferred, while at the same time taking into account that profitable accelerator computing is enabled by abundant DLP and low offloading overhead.

The accelerator offloading analysis in MPSoC compilers is enabled by hierarchical IRs in which applications are decomposed into structured code regions or blocks. An example of these IRs is the DFG introduced in Sect. 2.3.1, which has been successfully used for accelerator offloading analysis in [9]. The use of hierarchical IRs together with the architectural model of the target platform, enables a divide-and-conquer approach in which every region (typically loops with DLP) can be analyzed in isolation to reason about its potential performance improvement when it is offloaded to a particular accelerator. On the one hand, the region-based analysis allows to compare the performance of a particular region running on a host core with the performance running on an accelerator device. On the other

hand, this approach allows to estimate the offloading overhead by looking at the incoming and outgoing data dependencies of the region. Therefore, region-based analysis enables MPSoC compilers to decide whether or not to offload a given region to a particular accelerator, as it provides information about the key aspects for profitable accelerator computing, namely region execution performance and offloading overhead. Finally, the compiler has to be also aware of the desired target programming model to synthesize the appropriate code to offload code regions (see Sect. 2.5).

2.4.2 Mapping and Scheduling of Dataflow MoCs

Mapping and scheduling in a traditional compiler is done in the backend provided a description of the architecture. Mapping refers to the process of assigning operations to instructions and functional units (code selection) and variables to registers (register allocation). Scheduling refers to the process of organizing the instructions in a timed sequence. The schedule can be computed statically (for RISC, DSPs and VLIWs) or dynamically at runtime (for Superscalars), whereas the mapping of operations to instructions is always computed statically. The main purpose of mapping and scheduling in single-core compilers had been always to improve performance. Code size is also an important objective for embedded processors (specially VLIW). Only recently, power consumption became an issue. However, the reduction in power consumption with backend techniques does not have a big impact on the overall system power consumption.

In an MPSoC compiler similar operations have to be performed. Mapping, in this context, refers to the process of assigning code blocks to PEs and logical communication links to physical ones. In contrast to the single-core case, mapping can be also dynamic. A code block could be mapped at runtime to different PEs, depending on availability of resources. Scheduling for multi-cores has a similar meaning as for single-core, but instead of scheduling instructions, the compiler has to schedule code blocks. The presence of different application classes, e.g. real time, add complexity to the optimizations in the compiler. Particularly, there is much more room for improving power consumption in an MPSoC; after all, power consumption is one of the MPSoC drivers in the first place.

The result of scheduling and mapping is typically represented in form of a *Gantt Chart*, similar to the ones presented in Fig. 12. The PEs are represented in the vertical axis and the time in the horizontal axis. Code blocks are located in the plane, according to the mapping and the scheduling information. In Fig. 12a functions `f1` and `f2` are mapped to PE 1, the functions `f3` and `sum` are mapped to PE 2 and function `f4` to processor PE 3.

Given that code blocks have a higher time variability than instructions, scheduling can be rarely performed statically. Pure static scheduling requires full knowledge of the timing behavior and is only possible for very predictable architectures and regular computations, like in the case of *systolic arrays* [42]. If it is not possible to obtain a pure static schedule, some kind of synchronization is needed. Different

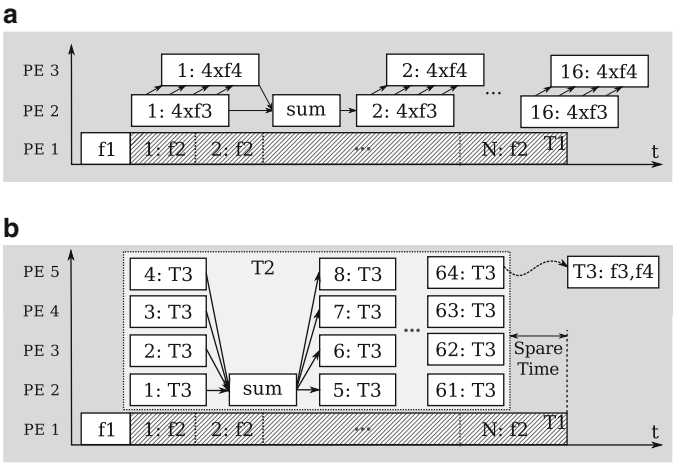


Fig. 12 Mapping and scheduling examples for code in Fig. 5a. (**a**) Partition with PLP, direct implementation of Fig. 5b. (**b**) Full parallelism exposed in Fig. 11b

scheduling approaches require different synchronization schemes with different associated performance overhead. In the example, the timing information of task `T3` is not known precisely. Therefore the exact starting time of function `sum` cannot be determined and a synchronization primitive has to be inserted to ensure correctness of the result. In this example, a simple *barrier* is enough in order to ensure that the execution of `T3` in PE 3, PE 4 and PE 5 has finished before executing function `sum`.

Scheduling Approaches

Which scheduling approach to utilize depends on the characteristics of the application and the properties of the underlying MoC used to describe it. Apart from pure static schedules, one can distinguish among the following scheduling approaches:

- **Self-timed Scheduling:** Typical for applications modeled with dataflow MoCs. A self-timed schedule is close to a static one. Once a static schedule is computed, the code blocks are ordered on the corresponding PEs, and synchronization primitives are inserted that ensure the presence of data for the computation. This kind of scheduling is used for SDF applications. For a more detailed discussion the reader is referred to [68].
- **Quasi-static Scheduling:** Used in the case where control paths introduce a predictable time variation. In this approach, unbalanced control paths are balanced and a self-timed schedule is computed. Quasi-static scheduling for dynamically parameterized SDF graphs is explored in [14] (see also Chapter [75]).
- **Dynamic Scheduling:** Used when the timing behavior of the application is difficult to predict and/or when the number of applications is not known in advance

(like in the case of general purpose computing). The scheduling overhead is usually higher, but so is also the average utilization of the processors in the platform. There are many dynamic scheduling policies. *Fair queue scheduling* is common in general purpose operating systems (OSs), whereas different flavors of priority based scheduling are typically used in embedded systems with real time constraints, e.g., *Rate Monotonic* (RM) and *Earliest Deadline First* (EDF).

- **Hybrid Scheduling:** Term used to refer to scheduling approaches in which several static or self-timed schedules are computed for a given application at compile time, and are switched dynamically at run-time depending on the *scenario* [27]. This approach is applied to streaming multimedia applications, and allows to adapt at runtime making it possible to save energy [52].

Virtually every MPSoC platform provides support for implementing mapping and scheduling. The support can be provided in software or in hardware and might restrict the available policies that can be implemented. This has to be taken into account by the compiler, which needs to generate/synthesize appropriate code (see Sect. 2.5).

Computing a Schedule

Independent of which scheduling approach and how this is supported, the MPSoC compiler has to compute a schedule (or several of them). Finding an optimal one in terms of performance is known to be NP-complete even for simple *Directed Acyclic Graphs* (DAGs). Single-core compilers therefore employ heuristics, most of them being derived from the classical *List Scheduling* algorithm [32]. Computing a schedule for multi-core platforms is by no means simpler. The requirements and characteristics of the schedule depend on the underlying MoC with which the application was modeled. In this chapter we distinguish between application modeled with centralized and distributed control flow.

Centralized Control Flow

Single-core compilers deal with centralized control flow, i.e., instructions are placed in memory and a central entity dictates which instructions to execute next, e.g., the *program counter* generator. The scheduler in a traditional single-core compiler leaves the control decisions out of the analysis and focus on scheduling instructions inside a BB. Since the control flow inside a BB is linear, there are no circular data dependencies and the data dependence graph is therefore acyclic. The resulting DAG is typically scheduled with a variant of the list scheduling algorithm.

In order to achieve a higher level of parallelism, single-core compilers apply different techniques that go beyond BBs. Typical examples of this techniques include *loop unrolling* and *software pipelining* [45]. An extreme example of loop unrolling was introduced in the previous section, where the dependence graph in

Fig. 11a was completely unrolled in Fig. 11b. Note that the graph in Fig. 11b is acyclic and could be scheduled with the list scheduling algorithm. The results of list scheduling with five resources would look similar to the scheduling traces in Fig. 12b.

In principle, the same scheduling approach can be used for multi-core. However, since every core in a MPSoC has its own control flow, a mechanism has to be implemented to transfer control. In the example in Fig. 12b, some core has the control code for the loop in line `14` of Fig. 5 and activates the four parallel tasks `T3`. There are several ways of handling this distribution of control. Parallel programming models like Pthreads and OpenMP offer source level primitives to implement *forks* and *joins*. Some academic research platforms offer dedicated instructions to send so-called *control tokens* among processors [81].

Distributed Control Flow

Parallel programming models based on concurrent MoCs like the ones discussed in Sect. 2.1.2 feature distributed control flow. For applications represented in this way, the issue of synchronization is greatly simplified and can be added to the logic of the channel implementation. Simple applications represented as acyclic task precedence graphs with predictable timing behavior can be scheduled with a list scheduling algorithm or with one of many other available algorithms for DAGs. For a survey on DAG scheduling algorithms the reader is referred to [43]. Applications, where precedence constraints are not explicit in the programming model and where communication can be control dependent, e.g., KPNs are usually scheduled dynamically. Finally, for applications represented as SDF, a self-timed schedule can be easily computed.

- **KPN scheduling:** KPNs are usually scheduled dynamically. There are two major ways of scheduling a KPN: *data* and *demand* driven. In data driven scheduling, every process in the KPN with available data at its input is in the ready state. A dynamic scheduler then decides which process gets executed on which processor at runtime. A demand driven scheduler first schedules processes with no output channels. These processes execute until a read blocks in one of the input channels. The scheduler triggers then only the processes from which data has been requested (*demanded*). This process continues recursively. For further details the reader is referred to [61].
- **SDF scheduling:** As mentioned before, SDFs are usually scheduled using a self-timed schedule, which requires a static schedule to be computed in the first place. There are two major types of schedules: *blocked* and *non-blocked schedules*. In the former, a schedule for one cycle is computed and is repeated without overlapping, whereas in the latter, the execution of different iterations of the graph are allowed to overlap. For computing a blocked schedule, a *complete cycle* in the SDF has to be determined. A complete cycle is a sequence of actor firings that brings the SDF to its initial state. Finding a complete cycles requires that

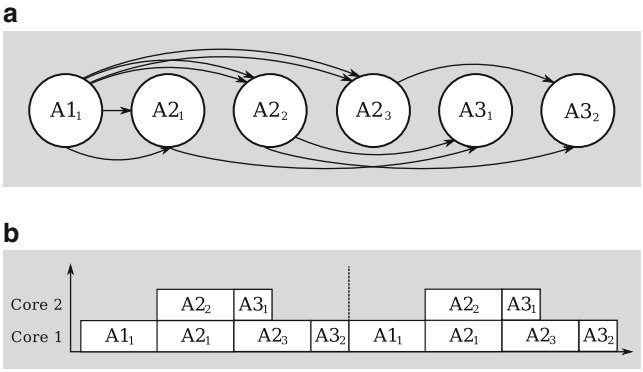


Fig. 13 Example of SDF scheduling, for SDF in Fig. 4a. (**a**) Derived DAG with $\mathbf{r} = 1\ 3\ 2^T$. (**b**) Possible schedule on two cores

(1) enough initial tokens are provided in the edges and (2) there is a non trivial solution for the system of equations $\Gamma \cdot \mathbf{r} = 0$, where $[\Gamma_{ij}] = p_{ij} - c_{ij}$, and p_{ij} c_{ij} are the number of tokens that actor i produces to and consumes from channel j respectively. In the literature, $\mathbf{r}$ is called *repetition vector* and Γ *topology matrix*. As an example, consider the SDF in Fig. 4b. This SDF has a topology matrix:

$$\Gamma = \begin{pmatrix} 3 & -1 & 0 \\ 6 & -2 & 0 \\ 0 & 2 & -3 \\ -2 & 0 & 1 \end{pmatrix}$$

and a repetition vector is $\mathbf{r} = [1\ 3\ 2]^T$. By *unfolding* the SDF according to its repetition vector and removing the feedback edges (those with delay tokens) one obtains the DAG shown in Fig. 13a with a possible schedule on two cores sketched in Fig. 13b. Using this procedure, the problem of scheduling an SDF is turned into DAG scheduling, and once again, one of the many heuristics for DAGs can be used. See Chapter [29] for further details.

For general application models and with the aim to obtain better results than with human-designed heuristics, several optimization methods are used. *Integer Linear Programming* is used in [59] and a combination of Integer Linear Programming and *Constraint Programming* (CP) is employed in [13]. *Genetic Algorithms* have also been used for this purpose, see Chapter [12]. Apart from scheduling and mapping code blocks and communication, a compiler also needs to map data. Data locality is already an issue for single-core systems with complex memory architectures: caches and *Scratch Pad Memories* (SPM). In multi-core systems, maximizing data locality and minimizing *false sharing* is an even bigger challenge [37].

Summary
Software distribution in the form of accelerator offloading and mapping and scheduling is one of the major challenges of MPSoC compilers. Different application constraints lead to new optimization objectives. Besides, different programming models with their underlying MoC allow different scheduling approaches. Most of these techniques work under the premise of accurate performance estimation (Sect. 2.2) which is by itself a hard problem. In addition, due to the high heterogeneity of signal processing multi-core systems, mapping of data represents a bigger challenge than in single-core systems.

2.5 Code Generation

The code generation phase of an MPSoC compiler is ad-hoc due to the heterogeneity of MPSoCs. To name a few examples: the cores are heterogeneous where the programming models may differ, the communications networks (and thus the APIs) are heterogeneous, and the OS-service libraries implementations can vary from one to another. After the software parallelization and the distribution phases, the code generation of an MPSoC compiler acts like a *meta-compiler* on top of multiple off-the-shelf compilers of the target MPSoC, to coordinate the compilation process. In this process, the code generator first performs a source-to-source transformation of the input application (which is either a sequential code or an abstract dataflow MoC), into a concrete parallel implementation, which is then further compiled with the tool-chain (including assemblers, compilers and linkers) of the target MPSoC. This tool-chain in turn can enable its own optimization features to further improve the code quality.

During the source-to-source transformation multiple steps take place, such as implementation of the parallel patterns according to the programming model, assignment of code blocks to cores, generation of the code for communication and scheduling, linking with the low-level libraries, among others. The complexity of the code generation process depends on the parallel programming model. For example, the code transformations for OpenMP are minimal, since it only implies inserting simple compiler directives. In contrast, other parallel programming models, such as Pthreads or OpenCL require heavy program transformations. For example, in OpenCL the kernels have to be extracted and the host code managing kernel execution and data transfers has to be added. Similarly, for abstract dataflow MoCs, the code generator has to make use of target specific OS APIs and libraries to create concrete implementations of actors/processes and FIFO channels.

In an MPSoC, PEs will communicate with each other using the NoC, which requires communication/synchronization primitives (e.g., semaphores, message passing) correctly set in place of the code blocks that the MPSoC compiler

distributes to the PEs. Again, due to the heterogeneous nature of the underlying architecture, the same communication link may look very different in the implementation, e.g., when the sending/receiving points are in different PEs. Embedded applications often need to be implemented in a portable fashion for the sake of software re-use. *Abstraction* of the communication functions to a higher level into the programming model is widely practiced, though it is still very ad-hoc and platform-specific. Recently, the Multicore Association has published the first draft of Multicore Communications API (MCAPI), which is a message-passing API to capture the basic elements of communication and synchronization that are required for closely distributed embedded systems [55]. This might have been a good first step in this area.

As discussed in Sect. 2.4.2, the scheduling decision is a key factor in the MPSoC compiler, especially in dataflow MoCs for embedded computing where real-time constraints have to be met. No matter which scheduling policy is determined for the final design, the functionality has to be implemented, in hardware, or software, or in a hybrid manner. A common approach is to use an off-the-shelf OS, often an RTOS, to enable the scheduling. There are many commercial solutions available such as QNX and WindRiver. The scheduler implementation in hardware is not uncommon for embedded devices, as software solutions may lead to larger overhead, which is not acceptable for RT-constrained embedded systems. Industry and academia have delivered promising results in this area, though more successful stories are still needed to justify this approach. A hybrid solution is a mixture, where some acceleration for the scheduler is implemented in hardware while flexibility is provided by software programmability, therefore customizing a trade-off between efficiency and flexibility. If the scheduling is not helped by e.g., an OS or a hardware scheduler, the code generation phase needs to generate or synthesize the scheduler e.g., [21] and [44].

Summary
Code generation is a complicated process, where many efforts are made to hide the compilation complexity via layered SW stacks and APIs. Heterogeneity will cause ad-hoc tool-chains to exist for a long time. The complexity of the code generation process depends of the parallel programming model.

3 Case Studies

As discussed in Sect. 2, the complexity of MPSoC compilers grows rapidly compared to single-core compilers. Nowadays, MPSoC compiler constructions for different industrial platforms and academic prototypes are still very much ad-hoc. This section surveys some prominent examples to show the readers how concrete implementations address the various challenges of MPSoC compilers.

3.1 Academic Research

In academia, vast research efforts have been recently directed towards MPSoC compiler technologies. Since the topic is very heterogeneous in nature, it has caught the attention of different research communities, such as real-time computing, compiler optimization, parallelization and fast simulation. A considerable amount of efforts have been invested in areas, such as MoCs, automatic parallelization and virtual simulation platforms. Compared to their counterparts in industry, the academic researchers focus mostly on the *upstream* of the MPSoC compilation flow, e.g., using MoCs to model applications, automatic task-to-processor mapping, early system performance estimation and holistic construction of MPSoC compilers.

3.1.1 Shapes

SHAPES [60] is a European Union FP6 Integrated Project whose objective is to develop a prototype of a *tiled* scalable hardware and software architecture for embedded applications featuring inherent parallelism. The major SHAPES building block, the RISC-DSP tile (RDT), is composed of an Atmel Magic VLIW floating-point DSP, an ARM9 RISC processor, on-chip memory, and a network interface for on- and off-chip communication. On the basis of RDTs and interconnect components, the architecture can be easily scaled to meet the computational requirements.

The SHAPES framework is shown in Fig. 14a. The starting point is the Model-driven compiler/Functional simulator, which takes an application specification in the form of process networks as input. High-level mapping exploration involves the trace information from the Virtual Shapes Platform (VSP) and the performance results from the Analytical Estimator, based on multi-objective optimization considering throughput, delay, predictability and efficiency. With the mapping information, the Hardware dependent Software (HdS) phase then generates the necessary dedicated communication and synchronization primitives, together with OS services.

The central part of the SHAPES software environment is the Distributed Operation Layer (DOL) framework [67]. The DOL structure and interactions with other tools and elements are shown in Fig. 14b. DOL mainly provides the MPSoC software developers two main services: system level performance analysis and process-to-processor mapping exploration.

- *DOL Programming Model:* DOL uses process networks as its programming model — the structure of the application is specified in an XML format consisting of processes, software channels and connections, while the application functionality is specified in C/C++ and process communications are performed by the DOL APIs, e.g., `DOL_read()` and `DOL_write()`. DOL uses a special *iterator* element to allow the user to instantiate several processes of the same type. For the process functionality in C/C++, a set of coding rules needs to be

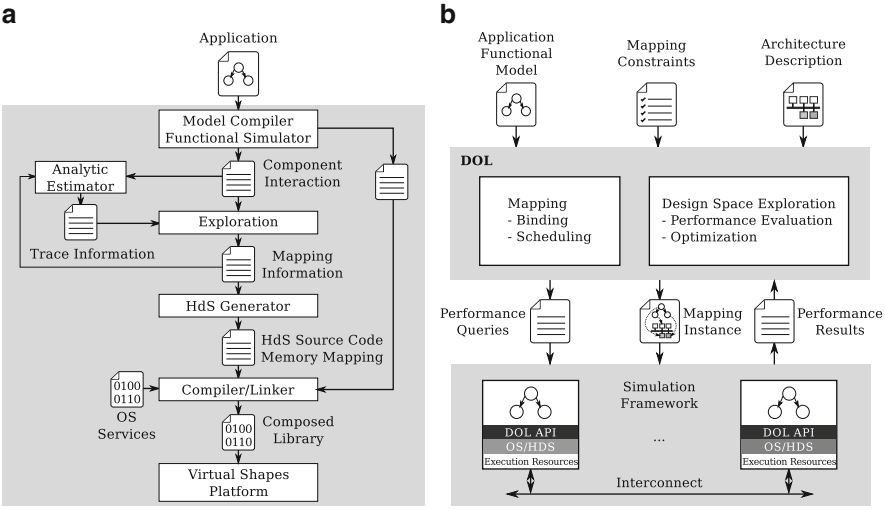


Fig. 14 SHAPES design flow. (**a**) Software development environment. (**b**) DOL framework

followed. In each process there must be an `init` and a `fire` procedure. The `init` procedure allocates and initializes data, which is called once during the application initialization. The `fire` procedure is called repeatedly afterwards.

- *Architecture Description:* DOL aims at mapping, therefore its architecture description abstracts away several details of the underlying platform. The XML format contains three types of information: *structural elements* such as processors/memories, *performance data* such as bus throughputs, and *parameters* such as memory sizes.
- *Mapping Exploration:* DOL mapping includes two phases: performance evaluation and optimization. Performance evaluation collects the data from both analytical performance evaluation and the simulation. The designer defines the optimization objectives and DOL uses evolutionary algorithms to generate the mapping.

With the mapping descriptor the HdS layer generates hardware dependent implementation codes and makefiles. Thereafter, the application can be compiled and linked against communication libraries and OS services. The final binary can be executed on the VSP or on the SHAPES hardware prototype.

3.1.2 Daedalus

Daedalus framework [58] is a tool-flow developed at Leiden University for automated design, programming and implementation of MPSoCs starting at a high level of abstraction. The Daedalus design-flow is shown in Fig. 15. It consists of three key tools, PNgen tool, Sesame (Simulation of Embedded System Architectures for

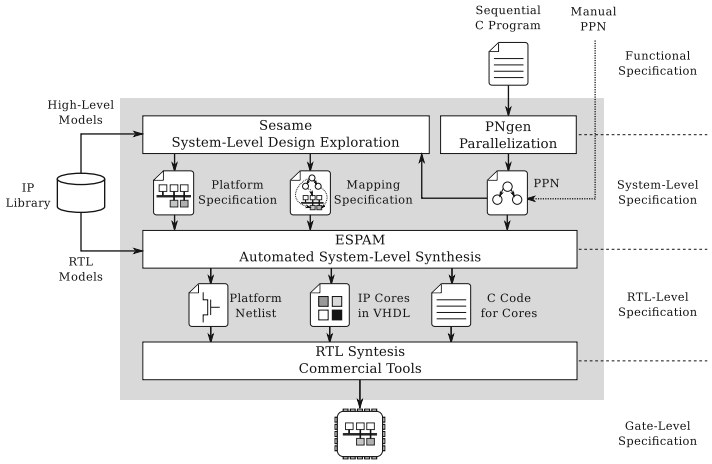


Fig. 15 Daedalus framework

Multilevel Exploration) and ESPAM (Embedded System-level Platform synthesis and Application Modeling), which work together to offer the designers a single environment for rapid system-level architectural exploration and automated programming and prototyping of multimedia MPSoC architectures. The PNgen tool automatically transforms the sequential application into a parallel specification in the form of *Polyhedral Process Networks* (PPNs), which are a subset of KPNs. The code that can be expressed in PPNs should be analyzable in the *polyhedral model* [48], which implies that the input sequential code is restricted to *Static Affine Nested Loop Programs* (SANLP). Then, the PPNs are used by Sesame modeling and simulation tool to perform a system-level *design space exploration* (DSE), where the performance of multiple mappings, HW/SW partitions and target platform architectures is quickly evaluated using high-level models from the IP library. Finally, the most promising mapping and platform specifications resulting from the DSE, together with the application specification (PPN) are the inputs to the ESPAM synthesis tool. The ESPAM tool uses these inputs along with the low-level RTL models from the IP library to automatically generate synthesizable VHDL code that implements the hardware architecture. It also generates, from the XML specification of the application, the C code for those processes that are mapped on to programmable cores, including the code for synchronization of the communication between the processors. Furthermore, commercial synthesis tools and the component compilers can be used to process the outputs for fast hardware/software prototyping.

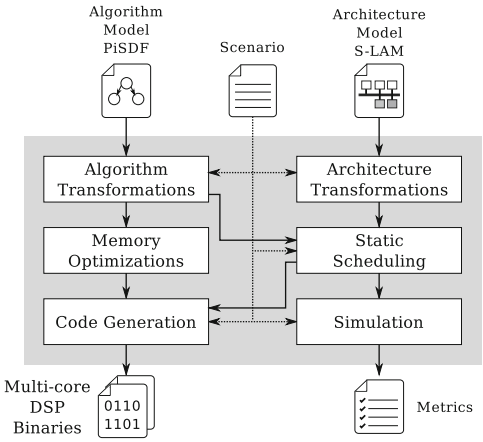


Fig. 16 PREESM framework

3.1.3 PREESM

The *Parallel and Real-time Embedded Executives Scheduling Method* (PREESM) is a framework for rapid prototyping and code generation, whose primary target is multi-core DSP platforms [62]. PREESM is developed at the Institute of Electronics and Telecommunications-Rennes (IETR) in collaboration with Texas Instruments. The PREESM framework is shown in Fig. 16. It takes as input an algorithm specification, an architectural model and a scenario that links the algorithm with the architecture. The *Parameterized and Interfaced Synchronous Dataflow* (PiSDF) is the MoC used here for the algorithm specification. PiSDF is an extension of SDF in which the production and consumption rates of the actors and the FIFO delays can be parameterized. The System-Level Architecture Model (S-LAM) describes the target platform as a graph in which the processing elements offer the processing capabilities for the actors and the communication elements offer the FIFO communication capabilities. The algorithm and architecture models are then transformed to enable scheduling and memory optimizations. On the one hand, the scheduling optimization aims at providing a static schedule that is deadlock-free. On the other hand, the memory optimization aims at reducing the memory requirements by allowing the re-utilization of memory for the FIFOs during code generation. Finally, the PREESM simulation facilities allow to assess the system performance by providing a gantt chart of the parallel execution of the algorithm, speedup estimates and memory requirements. Finally, the code generation stage emits the software for the selected multi-core DSP platform, which includes the necessary instructions for proper inter-core communication, cache management and synchronization. PREESM has been successfully evaluated in commercial multi-core DSP platforms, such as the ones from the Keystone family from Texas Instruments described in Sect. 3.2.1.

3.2 Industrial Case Studies

Several large semiconductor companies have already a few mature product lines aiming at different segments of the market due to the application-specific nature of the embedded devices. The stringent time-to-market window calls for the necessity to adopt platform-based MPSoC design methodology. That is, a new generation of an MPSoC architecture is based on a previous successful model with some evolutionary improvements. Compared to their counterparts in academia, the MPSoC software architects in industry focus more on the software tools reuse (considering the huge amount of certified code), providing abstractions and conveniences to the programmers for software development and efficient code-generation.

3.2.1 TI Keystone Multi-Core DSP Platform

The Keystone is a family of MPSoCs from Texas Instruments for high performance systems [73], which integrates RISC and DSP cores together with application specific co-processors and peripherals. The application domains of the Keystone platforms include high performance computing, wireless communications, networking, and audio/video processing. The Keystone architecture provides a high internal bandwidth by allowing non-blocking accesses to the processing cores, co-processors and peripherals. This is enabled by four main components: Multicore Navigator, TeraNet, Multicore Shared Memory Controller (MSMC) and HyperLink. The Multicore Navigator is a hardware controller for packet-based communication. Typical use cases are: message exchange or data transfer among cores, and data transfers between cores and co-processors or peripherals. The TeraNet is a low latency switch fabric that allows the movement of the Multicore Navigator packets among the main components within the Keystone platforms. The Multicore Shared Memory Controller allows to access the shared memory without using the TeraNet, which avoids interference with the packet movement. Finally, the HyperLink allows to interconnect multiple Keystone MPSoC.

Currently, there are two generations of the Keystone family. In the first generation, only DSPs were integrated as programmable cores. The architecture of the DSPs used in the Keystone platforms is called *C66x*. One interesting feature of the C66x cores is that they have both fixed-point and floating-point computation capabilities. In the second generation, the major enhancement is the integration of Cortex-A15 multi-core processors. In addition, the storage and bandwidth capacities of the main components were increased. Figure 17a shows the 66AK2H12 devices of the Keystone II family. This device offers a quad-core Cortex-A15 processor and eight C66x DSP cores, along with the main components of the Keystone family.

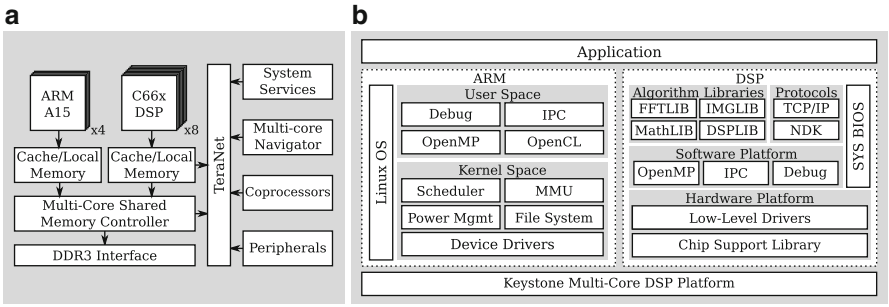


Fig. 17 TI keystone multi-core DSP platform. (**a**) 66AK2H12 keystone II device. (**b**) Keystone software stack

Figure 17b illustrates the software stack that TI provides for the Keystone platforms [74]. This software stack is divided into two coordinated sub-stacks, one for the ARM cores and another one for DSP cores. TI promotes the philosophy of *abstractions* among the software layers to hide just enough details for the developers at different roles/layers.

- OS Level: At the OS level the choice on the ARM side is Linux and on the DSP side is the TI-RTOS kernel (formerly known as SYS BIOS, which was the successor of DSP/BIOS) [74]. The TI-RTOS is optimized for real-time multi-tasking and scheduling. Along with the OS, low-level device drivers are provided to enable the use of hardware components in the Keystone platforms by higher software layers.
- Software Platform Level: The support for multi-core programming is at software platform level, including the TI IPC package [74] for inter-core communication and the support for industry standards, such as OpenMP and OpenCL. At this level there are also packages that enable tools for debugging, instrumentation and multi-core performance.
- Algorithm Level: Algorithms/codecs are usually allocated onto the DSP due to its computation power. At this level TI provides optimized libraries for multiple domains from general purpose math and signal processing libraries (e.g., DSPLIB and MATHLIB) to application specific libraries (e.g., IMGLIB and FFTLIB) [74].
- Application Level: The application developer uses the software layers introduced earlier to build the final system. Third-party tools that provide valuable add-ons such as GUI or streaming frameworks can be ported here.

The *abstractions* among the layers are realized by the standardized interfaces. Therefore, different teams can work in different domains at the same time thus boosting the productivity. Moreover, this also enables the possibility of third-parties participating in the TI software stack to provide valuable/commercial solutions, e.g., multi-core development tools and application-level GUI frameworks.

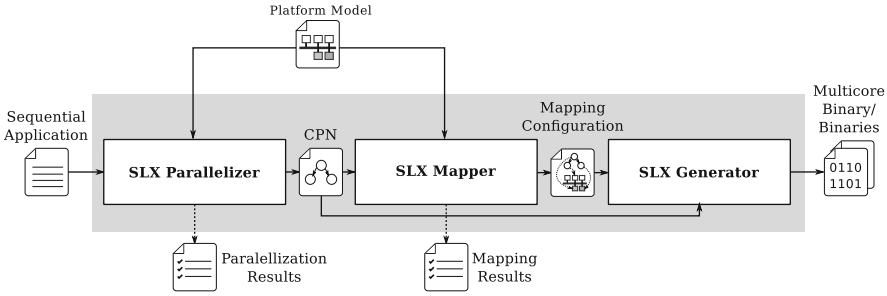


Fig. 18 SLX tool suite

3.2.2 Silexica: SLX Tool Suite

Silexica (SLX) [66] is a provider of software automation tools that addresses the increasingly complex task of multi-core programming in a variety of application domains, such as embedded vision, automotive and wireless telecommunications. Silexica is a spin-off of the Institute for Communication Technologies and Embedded Systems (ICE) at RWTH Aachen University. Its core technology is the *SLX Tool Suite* shown in Fig. 18. This tool suite has its roots in the academic project called *MPSoC Application Programming Studio* (MAPS), which started over a decade ago at ICE. The SLX Tool Suite is an excellent example of the adoption by the industry of the MPSoC compiler technologies described in this chapter, since it addresses the challenges of application modeling, platform description, software parallelization, software distribution, and code generation.

The SLX Tool Suite is composed of three main tools: *SLX Parallelizer*, *SLX Mapper* and *SLX Generator*. For an effective target-specific analysis, this tool suite uses fast and accurate software performance estimation technologies and an architectural model of the target platform. First, the SLX Parallelizer helps to migrate legacy C/C++ applications into the multi-core domain by identifying profitable parallelization opportunities. This parallelizer focuses on parallel patterns, such as DLP, PLP and TLP (see Sect. 2.3.3). As an output it provides source level information, which helps developers to understand the parallelization opportunities and its potential. In addition, the parallelized application can be exported using industry standards, such as OpenMP, or as the SLX specification called *C for Process Networks* (CPN). CPN is a language extension that allows to specify applications as dataflow MoCs (e.g. KPNs). The CPN specification can be either derived from the SLX Parallelizer analysis or manually by the developer. The SLX Mapper performs the task of software distribution by analyzing the computation and communication behavior of the CPN specification, to automatically distribute the processes on the platform cores and the FIFO channels on the platform interconnects. Finally, the SLX Generator is a source-to-source translation tool that takes both the CPN and the mapping specification generated by the SLX Mapper, to emit architecture-aware code, which is further compiled with the native tool-chain of the target platform.

4 Summary

In this chapter is presented an overview of the challenges for building MPSoC compilers and described some of the techniques, both established and emerging, that are being used to leverage the computing power of current and yet to come MPSoC platforms. The chapter concluded with selected academic and industrial examples that show how the concepts are applied to real systems.

It can be observed how new programming models are being proposed that change the requirements of the MPSoC compiler. It was discussed that, independent of the programming model, an MPSoC compiler has to find a suitable granularity to expose parallelism beyond the instruction level (ILP), demanding advanced analysis of the data and control flows. Software distribution is one of the most complex tasks of the MPSoC compiler and can only be achieved successfully with accurate performance estimation or simulation. Most of these analyses are target-specific, hence the MPSoC itself needs to be abstracted and fed to the compiler. With this information, the compiler can tune the different optimizations to the target MPSoC and finally generate executable code.

The whole flow shares similarities with that of a traditional single-core compiler, but is much more complex in the case of a multi-core embedded system. In this chapter it was presented some foundations and described approaches to deal with these problems. However, there is still a great amount of research to be done to make the leap from a high level specification to executable code as transparent as it is in the single-core case.

References

1. Eclipse. http://www.eclipse.org/. Visited on Jan. 2010
2. GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/. Visited on Jan. 2010
3. OpenMP Application Programming Interface. Version 4.5. http://www.openmp.org. Visited on Mar. 2017
4. AbsInt: aiT worst-case execution time analyzers. http://www.absint.com/ait/. Visited on Nov. 2009
5. Agbaria, A., Kang, D.I., Singh, K.: LMPI: MPI for heterogeneous embedded distributed systems. In: 12th International Conference on Parallel and Distributed Systems - (ICPADS'06), vol. 1, pp. 8 pp.– (2006)
6. Aguilar, M.A., Aggarwal, A., Shaheen, A., Leupers, R., Ascheid, G., Castrillon, J., Fitzpatrick, L.: Multi-grained Performance Estimation for MPSoC Compilers: Work-in-progress. In: Proceedings of the 2017 International Conference on Compilers, Architectures and Synthesis for Embedded Systems Companion, CASES '17, pp. 14:1–14:2. ACM, New York, NY, USA (2017)
7. Aguilar, M.A., Eusse, J.F., Ray, P., Leupers, R., Ascheid, G., Sheng, W., Sharma, P.: Towards parallelism extraction for heterogeneous multicore Android devices. International Journal of Parallel Programming pp. 1–33 (2016)
8. Aguilar, M.A., Leupers, R., Ascheid, G., Kavvadias, N.: A toolflow for parallelization of embedded software in multicore DSP platforms. In: Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, SCOPES '15, pp. 76–79. ACM, New York, NY, USA (2015)

9. Aguilar, M.A., Leupers, R., Ascheid, G., Murillo, L.G.: Automatic parallelization and accelerator offloading for embedded applications on heterogeneous MPSoCs. In: Proceedings of the 53rd Annual Design Automation Conference, DAC '16, pp. 49:1–49:6. ACM, New York, NY, USA (2016)
10. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)
11. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: A view from Berkeley. Tech. rep., EECS Department, University of California, Berkeley (2006)
12. Bacivarov, I., Haid, W., Huang, K., Thiele, L.: Methods and tools for mapping process networks onto multi-processor systems-on-chip. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
13. Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation and scheduling for MPSoCs via decomposition and no-good generation. Principles and Practices of Constrained Programming - CP 2005 (DEIS-LIA-05-001), 107–121 (2005)
14. Bhattacharya, B., Bhattacharyya, S.S.: Parameterized dataflow modeling for DSP systems. IEEE Transactions on Signal Processing **49**(10), 2408–2421 (2001)
15. Carro, L., Rutzig, M.B.: Multi-core systems on chip. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)
16. Castrillon, J., Leupers, R.: Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap. Springer Publishing Company, Incorporated (2013)
17. Castrillon, J., Sheng, W., Jessenberger, R., Thiele, L., Schorr, L., Juurlink, B., Alvarez-Mesa, M., Pohl, A., Reyes, V., Leupers, R.: Multi/many-core programming: Where are we standing? In: 2015 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1708–1717 (2015)
18. Castrillon, J., Sheng, W., Leupers, R.: Trends in embedded software synthesis. In: SAMOS, pp. 347–354 (2011)
19. Ceng, J.: A methodology for efficient multiprocessor system on chip software development. Ph.D. thesis, RWTH Aachen University (2011)
20. Ceng, J., Castrillon, J., Sheng, W., Scharwächter, H., Leupers, R., Ascheid, G., Meyr, H., Isshiki, T., Kunieda, H.: MAPS: an integrated framework for MPSoC application parallelization. In: DAC '08: Proceedings of the 45th annual conference on Design automation, pp. 754–759. ACM, New York, NY, USA (2008)
21. Cesario, W., Jerraya, A.: Multiprocessor Systems-on-Chips, chap. Chapter 9. Component-Based Design for Multiprocessor Systems-on-Chip, pp. 357–394. Morgan Kaufmann (2005)
22. Cordes, D.A.: Automatic parallelization for embedded multi-core systems using high-level cost models. Ph.D. thesis, TU Dortmund (2013)
23. Diakopoulos, N., Cass, S.: The top programming languages 2016. http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016. Visited on Feb. 2017
24. Fisher, J., P., F., Young, C.: Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools. Morgan-Kaufmann (Elsevier) (2005)
25. Gao, L., Huang, J., Ceng, J., Leupers, R., Ascheid, G., Meyr, H.: TotalProf: a fast and accurate retargetable source code profiler. In: CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis, pp. 305–314. ACM, New York, NY, USA (2009)
26. Geilen, M., Basten, T.: Kahn process networks and a reactive extension. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)
27. Gheorghita, S., T. Basten, H.C.: An overview of application scenario usage in streaming-oriented embedded system design. www.es.ele.tue.nl/esreports/esr-2006-03.pdf. Visited on Mar. 2017

28. Gupta, R., Micheli, G.D.: Hardware-software co-synthesis for digital systems. In: IEEE Design & Test of Computers, pp. 29–41 (1993)
29. Ha, S., Oh, H.: Decidable signal processing dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
30. Hewitt, C., Bishop, P., Greif, I., Smith, B., Matson, T., Steiger, R.: Actor induction and meta-evaluation. In: POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 153–168. ACM, New York, NY, USA (1973)
31. Hind, M.: Pointer analysis: Haven't we solved this problem yet? In: PASTE '01, pp. 54–61. ACM Press (2001)
32. Hu, T.C.: Parallel sequencing and assembly line problems. Oper. Res. **9**(6), 841–848 (1961)
33. Hwang, Y., Abdi, S., Gajski, D.: Cycle-approximate retargetable performance estimation at the transaction level. In: DATE '08: Proceedings of the conference on Design, automation and test in Europe, pp. 3–8. ACM, New York, NY, USA (2008)
34. Hwu, W.M., Ryoo, S., Ueng, S.Z., Kelm, J.H., Gelado, I., Stone, S.S., Kidd, R.E., Baghsorkhi, S.S., Mahesri, A.A., Tsao, S.C., Navarro, N., Lumetta, S.S., Frank, M.I., Patel, S.J.: Implicitly parallel programming models for thousand-core microprocessors. In: DAC '07: Proc. of the 44th Design Automation Conference, pp. 754–759. ACM, New York, NY, USA (2007)
35. Johnson, R.C.: Efficient program analysis using dependence flow graphs. Ph.D. thesis, Cornell University (1994)
36. Kahn, G.: The semantics of a simple language for parallel programming. In: J.L. Rosenfeld (ed.) Information Processing '74: Proceedings of the IFIP Congress, pp. 471–475. North-Holland, New York, NY (1974)
37. Kandemir, M., Dutt, N.: Multiprocessor Systems-on-Chips, chap. Chapter 9. Memory Systems and Compiler Support for MPSoC Architectures, pp. 251–281. Morgan Kaufmann (2005)
38. Karp, R.M., Miller, R.E.: Properties of a model for parallel computations: Determinacy, termination, queuing. SIAM Journal of Applied Math **14**(6) (1966)
39. Karuri, K., Al Faruque, M.A., Kraemer, S., Leupers, R., Ascheid, G., Meyr, H.: Fine-grained application source code profiling for ASIP design. In: DAC '05: Proceedings of the 42nd annual conference on Design automation, pp. 329–334. ACM, New York, NY, USA (2005)
40. Kennedy, K., Allen, J.R.: Optimizing compilers for modern architectures: A dependence-based approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2002)
41. Khronos Group: OpenCL embedded boards comparison 2015. https://www.khronos.org/news/events/opencl-embedded-boards-comparison-2015. Visited on Mar. 2017
42. Kung, H.T.: Why systolic architectures? Computer **15**(1), 37–46 (1982)
43. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv. **31**(4), 406–471 (1999)
44. Kwon, S., Kim, Y., Jeun, W.C., Ha, S., Paek, Y.: A retargetable parallel-programming framework for MPSoC. ACM Trans. Des. Autom. Electron. Syst. **13**(3), 1–18 (2008)
45. Lam, M.: Software pipelining: An effective scheduling technique for VLIW machines. SIGPLAN Not. **23**(7), 318–328 (1988)
46. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE **75**(9), 1235–1245 (1987)
47. Lee, E.A.: Consistency in dataflow graphs. IEEE Trans. Parallel Distrib. Syst. **2**(2), 223–235 (1991)
48. Lengauer, C.: Loop parallelization in the polytope model. In: Proceedings of the 4th International Conference on Concurrency Theory, CONCUR '93, pp. 398–416. Springer-Verlag, London, UK, UK (1993)
49. Leupers, R.: Retargetable Code Generation for Digital Signal Processors. Kluwer Academic Publishers, Norwell, MA, USA (1997)
50. Leupers, R.: Code selection for media processors with SIMD instructions. In: DATE '00, pp. 4–8. ACM (2000)
51. Li, L., Huang, B., Dai, J., Harrison, L.: Automatic multithreading and multiprocessing of C programs for IXP. In: PPoPP '05: Proc. of the 10th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 132–141. ACM, New York, NY, USA (2005)

52. Ma, Z., Marchal, P., Scarpazza, D.P., Yang, P., Wong, C., Gmez, J.I., Himpe, S., Ykman-Couvreur, C., Catthoor, F.: Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogenous Platforms. Springer (2007)
53. Martin, G.: ESL requirements for configurable processor-based embedded system design. http://www.us.design-reuse.com/articles/article12444.html. Visited on Mar. 2017
54. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1997)
55. Multicore Association: MCAPI - Multicore Communications API. http://www.multicore-association.org/workgroup/mcapi.php. Visited on Mar. 2017
56. Multicore Association: Software-hardware interface for multi-many-core (SHIM) specification v1.00. http://www.multicore-association.org. Visited on Mar. 2017
57. National Instruments: LabView. http://www.ni.com/labview/. Visited on Mar. 2017
58. Nikolov, H., Thompson, M., Stefanov, T., Pimentel, A., Polstra, S., Bose, R., Zissulescu, C., Deprettere, E.: Daedalus: Toward composable multimedia MP-SoC design. In: DAC '08: Proceedings of the 45th annual conference on Design automation, pp. 574–579. ACM, New York, NY, USA (2008)
59. Palsberg, J., Naik, M.: Multiprocessor Systems-on-Chips, chap. Chapter 12. ILP-based Resource-aware Compilation, pp. 337–354. Morgan Kaufmann (2005)
60. Paolucci, P.S., Jerraya, A.A., Leupers, R., Thiele, L., Vicini, P.: SHAPES:: a tiled scalable software hardware architecture platform for embedded systems. In: CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis, pp. 167–172. ACM, New York, NY, USA (2006)
61. Parks, T.M.: Bounded scheduling of process networks. Ph.D. thesis, Berkeley, CA, USA (1995)
62. Pelcat, M., Desnos, K., Heulot, J., Guy, C., Nezan, J.F., Aridhi, S.: Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In: 2014 6th European Embedded Design in Education and Research Conference (EDERC), pp. 36–40 (2014). https://doi.org/10.1109/EDERC.2014.6924354
63. Polychronopoulos, C.D.: The hierarchical task graph and its use in auto-scheduling. In: Proceedings of the 5th International Conference on Supercomputing, ICS '91, pp. 252–263. ACM, New York, NY, USA (1991)
64. Rabenseifner, R., Hager, G., Jost, G.: Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 427–436 (2009)
65. Sharma, G., Martin, J.: MATLAB (R): A language for parallel computing. International Journal of Parallel Programming **37**(1) (2009)
66. Silexica: SLX Tool Suite. http://www.silexica.com. Visited on Mar. 2017
67. Sporer, T., Franck, A., Bacivarov, I., Beckinger, M., Haid, W., Huang, K., Thiele, L., Paolucci, P., Bazzana, P., Vicini, P., Ceng, J., Kraemer, S., Leupers, R.: SHAPES - a scalable parallel HW/SW architecture applied to wave field synthesis. In: Proc. 32nd Intl Audio Engineering Society Conference, pp. 175–187. Audio Engineering Society, Hillerod, Denmark (2007)
68. Sriram, S., Bhattacharyya, S.S.: Embedded Multiprocessors: Scheduling and Synchronization. Marcel Dekker, Inc., New York, NY, USA (2000)
69. Standard for information technology - portable operating system interface (POSIX). Shell and utilities. IEEE Std 1003.1-2004, The Open Group Base Specifications Issue 6, section 2.9: IEEE and The Open Group
70. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. IEEE Des. Test **12**(3), 66–73 (2010)
71. Stotzer, E.: Towards using OpenMP in embedded systems. OpenMPCon: Developers Conference (2015)
72. Synopsys: Virtual Platforms. https://www.synopsys.com/verification/virtual-prototyping.html. Visited on Mar. 2017
73. Texas Instruments: Keystone Multicore Devices. http://processors.wiki.ti.com/index.php/Multicore. Visited on Mar. 2017

74. Texas Instruments: Software development kit for multicore DSP Keystone platform. http://www.ti.com/tool/bioslinuxmcsdk. Visited on Mar. 2017
75. Theelen, B.D., Deprettere, E.F., Bhattacharyya, S.S.: Dynamic dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
76. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.: Towards a holistic approach to auto-parallelization – integrating profile-driven parallelism detection and machine-learning based mapping. In: PLDI 0-9: Proceedings of the Programming Language Design and Implementation Conference. Dublin, Ireland (2009)
77. Vargas, R., Quinones, E., Marongiu, A.: OpenMP and timing predictability: A possible union? In: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE '15, pp. 617–620. EDA Consortium, San Jose, CA, USA (2015)
78. Verdoolaege, S., Nikolov, H., Stefanov, T.: pn: A tool for improved derivation of process networks. EURASIP J. Embedded Syst. **2007**(1), 19–19 (2007)
79. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3), 1–53 (2008)
80. Working Group ISO/IEC JTC1/SC22/WG14: C99, Programming Language C ISO/IEC 9899:1999
81. Zalfany Urfianto, M., Isshiki, T., Ullah Khan, A., Li, D., Kunieda, H.: Decomposition of task-level concurrency on C programs applied to the design of multiprocessor SoC. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **E91-A**(7), 1748–1756 (2008)

Analysis of Finite Word-Length Effects in Fixed-Point Systems

D. Menard, G. Caffarena, J. A. Lopez, D. Novo, and O. Sentieys

Abstract Systems based on fixed-point arithmetic, when carefully designed, seem to behave as their infinite precision analogues. Most often, however, this is only a macroscopic impression: finite word-lengths inevitably approximate the reference behavior introducing quantization errors, and confine the macroscopic correspondence to a restricted range of input values. Understanding these differences is crucial to design optimized fixed-point implementations that will behave "as expected" upon deployment. Thus, in this chapter, we survey the main approaches proposed in literature to model the impact of finite precision in fixed-point systems. In particular, we focus on the rounding errors introduced after reducing the number of least-significant bits in signals and coefficients during the so-called quantization process.

1 Introduction

The use of *fixed-point (FxP)* arithmetic is widespread in computing systems. Demanding applications often force computing systems to specialize their hardware and software architectures to reach the required levels of efficiency (in terms of

D. Menard (✉)
INSA Rennes, IETR, UBL, Rennes, France
e-mail: daniel.menard@insa-rennes.fr

G. Caffarena
CEU San Pablo University, Madrid, Spain
e-mail: gabriel.caffarena@ceu.es

J. A. Lopez
ETSIT, Universidad Politécnica de Madrid, Madrid, Spain
e-mail: juanant@die.upm.es

D. Novo
CNRS, LIRMM, Montpellier, France
e-mail: david.novo@lirmm.fr

O. Sentieys
INRIA, University of Rennes I, Rennes, France
e-mail: olivier.sentieys@inria.fr

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_29

energy consumption, execution speed, etc.). In such cases, the use of fixed-point arithmetic is usually not negotiable. Yet, the cost benefits of fixed-point arithmetic are not for free and can only be reached through an elaborated design methodology able to restrain finite word-length—or quantization—effects.

Digital systems are invariably subject to nonidealities derived from their finite precision arithmetic. A digital operator (e.g., an adder or a multiplier) imposes a limited number of bits (i.e., word-length) upon its inputs and outputs. As a result, the values produced by such an operator suffer from (small) deviations with respect to the values produced by its "equivalent" (infinite precision) mathematical operation (e.g., the addition or the multiplication). The more the bits allocated the smaller the deviation—or quantization error—but also the larger, the slower and the more energy hungry the operator. The so-called word-length optimization—or quantization—process determines the word-length of every signal (and corresponding operations) in a targeted algorithm. Accordingly, the best possible quantization process needs to select the set of word-lengths leading to the cheapest implementation while bounding the precision loss to a level that is tolerable by the application in hand. The latter can formally be defined as the following optimization problem:

$$\begin{aligned} \underset{\mathbf{w}}{\text{minimize}} \quad & C(\mathbf{w}) \\ \text{subject to} \quad & D(\mathbf{w}) \leq \Omega, \end{aligned} \tag{1}$$

where $\mathbf{w}$ is a vector containing the word-lengths of every signal, $C(\cdot)$ is a cost function that propagates variations in word-lengths to design objectives such as energy consumption, $D(\cdot)$ computes the degradation in precision caused by a particular $\mathbf{w}$ and Ω represents the maximum precision loss tolerable by the application.

From a methodological perspective, the word-length optimization process can be approached in two consecutive steps: (1) range selection and (2) precision optimization. The *range selection* step defines the left hand limit—or *Most-Significant Bit (MSB)*—and the subsequent *precision optimization* step fixes the right hand limit—or *Least-Significant Bit (LSB)*—of each word-length. Typically, the range selection step is designed to avoid overflow errors altogether, and therefore, the precision optimization step becomes the sole responsible for precision loss. Figure 1 gives a pictorial impression of the word-length optimization process and divides the precision optimization step into four interacting components, namely the optimization engine, the cost estimation, the constraint selection and the error estimation.

- The *optimization engine* basically consists of an algorithm that iteratively converges to the best word-length assignment. It has been shown that the constraint space is non-convex in nature [29]—it is actually possible to have a lower quantization error at a system output by reducing the word-length at an

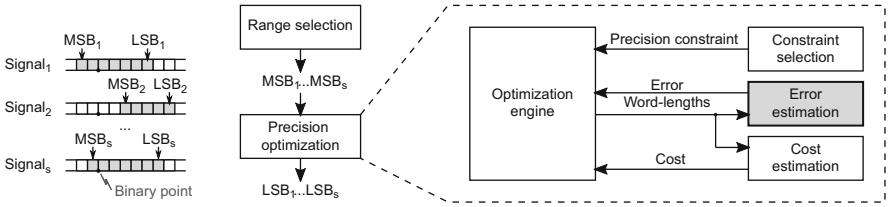


Fig. 1 Basic components of a word-length optimizaton process

internal node—, and that the optimization problem is NP-hard [35]. Accordingly, existing practical approaches are of a heuristic nature [21, 22, 32].

- A precise *cost estimation* of each word-length assignment hypothesis leads to impractical optimization times as such heuristic optimization algorithms involve a great number of cost and error evaluations. Instead, word-length optimization processes use fast abstract cost models, such as the hardware cost library introduced in the chapter [132] of this book or the fast models proposed by Clarke et al. [28] to estimate the power consumed in the arithmetic components and routing wires.
- The *precision constraint selection* block is responsible of reducing the abstract sentence "the maximum precision loss tolerable by the application" into a magnitude that can be measured by the error estimation. Practical examples have been proposed for audio [103] or wireless applications [109].
- Existing approaches for *error estimation* can be divided into simulation-based and analytical methods. Simulation-based methods are suitable for any type of application but are generally very slow. Alternatively, analytical error estimation methods can be significantly faster but often restrict the domain of application (e.g., only linear time-invariant systems [32]). There are also hybrid methods [122] that aim at combining the benefits of each method.

While the chapter presented in [132] covers in breadth most of the blocks in Fig. 1, this chapter takes a complementary in-depth approach and focuses on arguably the most important block in the word-length optimization process: the error estimation. The latter is crucial to ensure correctly behaving fixed-point systems and has received considerable attention in the research literature. Thus, in this chapter, we survey the main approaches proposed to model quantization errors. To understand their similarities and differences, we present a classification of the reviewed approaches based on their assumptions and coverage. We believe that this chapter will shed some light on the word-length optimization process as a whole and help readers choose the most convenient available approach to model quantization errors in their word-length optimization process.

The rest of the chapter is organized as follows. Section 2 introduces the main concepts regarding quantization. The next section deals with signal quantization. Noise metrics and both simulation-based and analytical techniques for the evaluation of quantization noise are explained. Regarding the analytical evaluation, this

covers both the estimation of noise power and noise bound. Section 4 addresses the quantization of coefficients. The different measurement parameters used to evaluate coefficient quantization are explained, with special emphasis on the use of the L_2-sensitivity. System stability is described in Sect. 5, again focusing on simulation-based and analytical approaches. Finally, a summary is presented in the last section.

2 Background

A typical *Digital Signal Processing (DSP)* design flow begins with a design specification and follows a number of steps to produce a satisfactory implementation as illustrated in Fig. 2. The original specification serves as a functional reference and is typically implemented in frameworks that prioritize software productivity, such as MATLAB, in floating-point or double precision. For instance to illustrate, such a specification can include a 64-point *Discrete Fourier Transform (DFT)*. Firstly, a skillful designer will reduce the algorithmic complexity in the *algorithmic refinement* step. The DFT matrix can be factorized into products of sparse factors (i.e., Fast Fourier Transform), which reduces the complexity from $O(n^2)$ to $O(n \log n)$. Additionally, the algorithmic refinement step can make use of approximations to further reduce the complexity—e.g., the *Maximum Likelihood (ML)* detector is approximated by a near-ML detector [109]. Once the algorithm structure is fixed, operators and signals are defined in the subsequent *algebraic transformation* and *static data formatting* steps, respectively. An algebraic approximation can for instance reduce a reciprocal square root operator to a scaled linear function [109]. Finally, the static data formatting step is the responsible of finalizing the bit-true

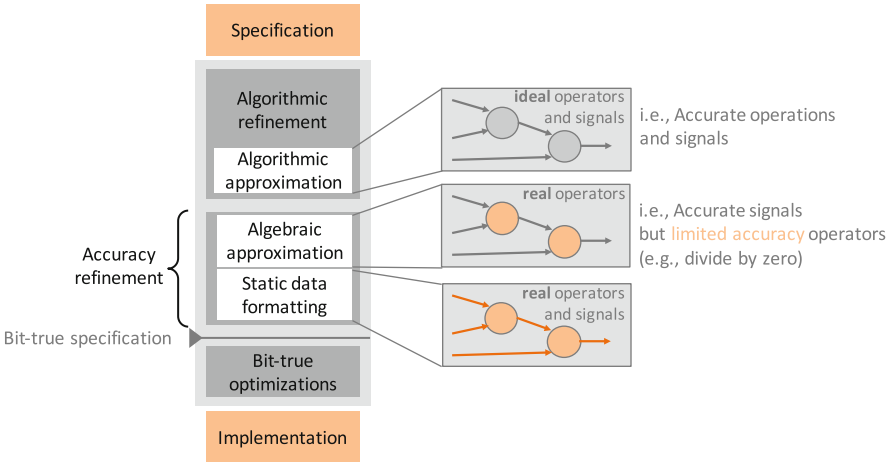


Fig. 2 Basic DSP design flow

specification that will constrain all succeeding (bit-true) optimizations, such as loop transformations, resource binding, scheduling, etc.

Algorithmic and algebraic approximations are integrating parts of what is known as *approximate computing* [107]. Instead, data formatting is equivalent to the word-length optimization process introduced in the previous section. Although some prior work targets implementations that do not add quantization error to those of the inputs [9, 84, 130], *lossy* static data formatting [34]—i.e., reduction of implementation cost by introducing additional quantization noise in intermediate nodes—is the common practice and the main focus of this chapter.

2.1 Floating-Point vs. Fixed-Point Arithmetic

The IEEE-754 standard [60] for *floating-point (FlP)* arithmetic—particularly the 64 bit double-precision format—is commonly used in implementations requiring high mathematical precision. However, many applications tolerate the use of less precise arithmetic modules in both FxP [34, 120] and non-standard FlP [51] formats. As introduced in Chapter [132], the FlP format represents numbers by means of two variables: an exponent e and a mantissa m. Given the pair (m, e), the value of the represented FlP number, V_{FlP}, is

$$V_{FlP} = m \cdot 2^e. \tag{2}$$

The combined use of mantissa and exponent provides the finest level of scaling: each number includes its own scaling factor. Thereby, FlP digital systems can effectively operate numbers with a very wide dynamic range. However, FlP arithmetic often involves overheads in terms of area, delay and energy consumption. Firstly, FlP requires wider bit-widths than FxP arithmetic to operate with equivalent precision on variables with low to moderate dynamic range [57], which is the typical case in most applications. Furthermore, FlP operators are more complex as they implement in hardware the alignment of the fractional point of the operands and the normalization of the output besides the actual operator.

Alternatively, FxP arithmetic constrains the exponent e to be a design time constant. Equation (2) remains valid but only the mantissa m changes at run time—and thus needs to be stored in memory. Accordingly, describing an implementation employing FxP arithmetic is more complex and tedious as the designer is responsible of handling explicitly in the source code the scaling of variables.

2.2 Finite Word-Length Effects

Quantized systems suffer from two types of errors: *overflow* and *precision* errors. On the one hand, overflow errors result from variable values growing beyond the limits of the word-length (WL). They are related to the lack of scaling and

saturation and wrap-around [97, 116, 119] are the most common techniques used to handle them at the operator output. Saturation employs extra hardware to detect and reduce overflow error. Instead, wrap-around is hardware-free but leads to intolerably huge errors in underdimensioned word-lengths. On the other hand, precision errors are due to the unavoidable limited precision of quantized digital implementations [97, 116, 119]. Rounding and truncation are the most common techniques used to handle precision errors at the operator output. Rounding employs extra hardware to reduce the maximum error magnitude resulting from the removal of LSBs. Instead, truncation is hardware-free but often accumulates larger precision errors. The technique leading to the best implementation is application dependent: even though rounding requires more complex operators, they can generally operate shorter word-lengths to achieve the same precision error as truncation [98].

The limited precision effects of the DSP realizations have been studied extensively since the raise of digital systems, particularly in *Linear Time Invariant (LTI)* systems [97, 116, 119]. They are commonly divided in four different types: round-off noise, coefficient quantization, limit cycles and system stability. *Round-Off Noise (RON)* refers to the probabilistic deviation of the results of a quantized implementation with respect to the error-free reference [97, 116, 119]. *Coefficient Quantization (CQ)* refers to the deterministic deviation of the parameters of the transfer function [71, 97, 119]. *Limit Cycles (LC)* are the parasitic oscillations that appear in quantized system under constant or zero inputs due to the propagation of the quantization errors through feedback loops [27, 119]. Finally, in the case of digital filters, the coefficient quantization modifies the position of the poles of the transfer function, which might jeopardize the system stability when approached carelessly [110]. Table 1 summarizes the classification of these effects attending to linearity and whether they result from the quantization of signals or coefficients.

RON is the prominent finite precision effect during normal operation of FxP systems [71, 97, 116, 119]. It introduces stochastic variations around the system's nominal operation point. Complementary, CQ effects modify the actual nominal operation point of the system and can lead to instability when such deviation is not carefully conducted. While RON and CQ effects apply to any FxP system, LCs effects are only relevant to particular types of systems (e.g., DSP filters) as they are the result of correlated quantization errors in feedback loops [116, 119]. For this reason, in this chapter we focus mainly on RON (most of Sect. 3) and CQ effects (Sect. 4) while also covering LCs for the sake of completeness but in much less detail (end of Sect. 3).

Table 1 Classification of the finite WL quantization effects

Type of effect	Quantization object	Name of effect
Linear	Signals	Round-off noise (RON)(Section III)
	Coefficients	Coefficient quantization (CQ) (Section IV)
Nonlinear	Signals	Limit cycle oscillations
	Coefficients	System instability

3 Effect of Signal Quantization

Finite precision arithmetic leads to unavoidable deviations of the finite precision values from the infinite precision ones. Such deviations, due to signal quantizations, modify the quality of the application output. Thus, they must be evaluated and maintained within reasonable bounds. In most cases these deviations are accurately modeled as additive white noise, or quantization noise. The quantization noise can be evaluated through analytical or fixed-point simulation based approaches. In the case of analytical approaches, a mathematical expression of a metric is determined. Computing an expression of an quality metric for every kind of application is generally an issue. Thus, the quality degradations are not analyzed directly in the quantization process, but an intermediate metric measuring the fixed-point accuracy is used instead.

Word-length optimization is split into two main steps. Firstly, a computational accuracy constraint is determined according to application quality and, secondly, the word-length optimization is carried out using this constraint. Interestingly, fixed-point simulation approaches enable the direct evaluation of the effect of quantization on application quality. But, in many cases, an intermediate accuracy metric is used because less samples are required to estimate this metric in contrast to directly computing or simulating application quality under quantization effects.

The different approaches available to analyze quantization noise effects that are covered in this section are displayed in Fig. 3. The techniques are first divided into the three main major groups: simulation-based, analytical and mixed (that combines the two previous ones) approaches. The graph include all techniques covered in the subsequent subsections and also the main related publications.

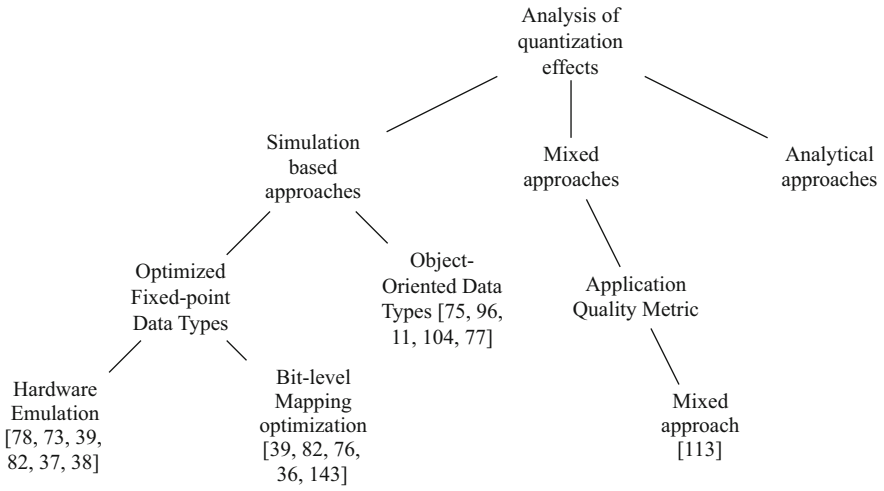


Fig. 3 Classification of the different approaches to analyze the quantization noise effects

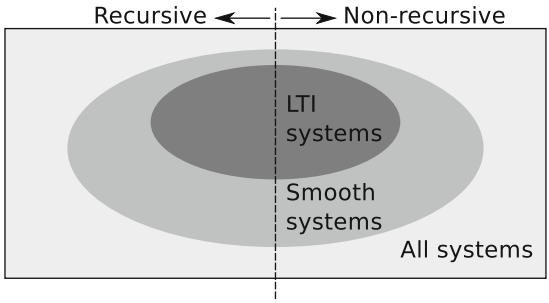


Fig. 4 Classification of systems targeted by RON evaluation techniques

Figure 4 shows the main classification of systems used by the different techniques devoted to RON evaluation: LTI systems, *smooth* systems and *all* systems. *Smooth* systems are those whose operations are differentiable and can be linearized without committing a significant error. This classification also distinguishes between recursive systems—systems with loops or cyclic—and non-recursive systems—systems without loops or acyclic. The different regions displayed in the graph are related to different techniques that are only able to handle a particular type of systems.

Section 3.1 introduces the different noise metrics used. Section 3.2 covers the analytical evaluation of the quantization noise effect, embracing both the noise power and noise bound computation. Then the techniques based on fixed-point simulation and the hybrid techniques are presented in Sect. 3.3.

3.1 Error Metrics

Different metrics can be used to measure the accuracy of a fixed-point realization. This accuracy can be evaluated through the bounds of the quantization errors [2, 43], the number of significant bits [24], or the power of the quantization noise [18, 102, 126]. The shape of the power spectral density (PSD) of the quantization noise is used as metric in [7] or in [31] for the case of digital filters. In [20], a more complex metric able to handle several models is proposed.

Regarding the metric that computes the bounds of the quantization errors, the maximum deviation between the exact value and the finite precision value is determined. This metric is used for critical systems when it is necessary to ensure that the error will not surpass a maximum deviation. In this case, the final quality has to be numerically validated.

As for the noise power computation, the error is modeled as a noise, and the second order moment is computed. This metric analyzes the dispersion of the finite precision values around the exact value and the mean behaviour of the error. The noise power metric is used in applications which tolerate sporadic high-value errors that do not affect the overall quality. In this case, the system design is based on a trade-off between application quality and implementation cost.

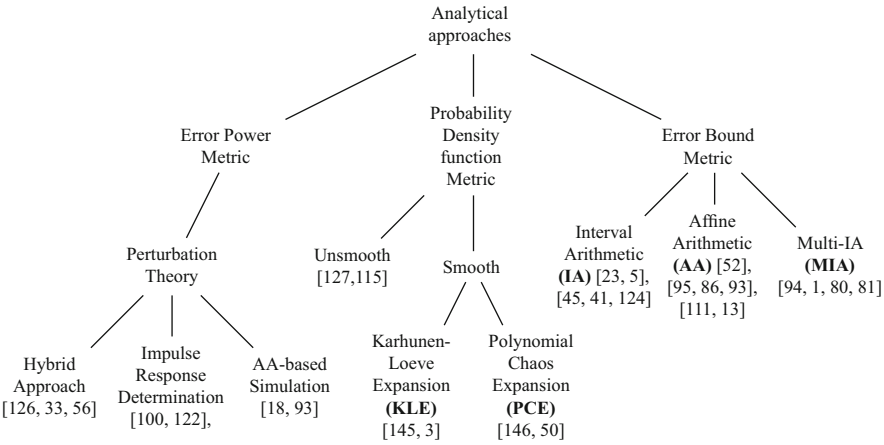


Fig. 5 Classification of the different analytical approaches to analyze the quantization noise effects

3.2 Analytical Evaluation of the Round-Off Noise

The aim of analytical approaches is to determine a mathematical expression of the fixed-point error metric. The error metric function depends on the word-length of the different data inside the application. The main advantage of these approaches is the short time required for the evaluation of the accuracy metric for a given set of word-lengths. The time required to generate this analytical function can be more or less important but this process is done only once, before the optimization process. Then, each evaluation of the accuracy metric for a given WL sets corresponds to the computation of a mathematical expression. The main drawback of these analytical approaches is that they do not support all kinds of systems. Figure 5 depicts a classification of existing analytical approaches to analyze the quantization noise effects. This classification depends on the type of metric used (bound, power or probability density function), on the smooth/unsmooth nature of the noise, and on the technique used. In this section, we review the different analytical approaches for computing: RON bounds, RON power, and the effect of RON on any quality metric in the presence of unsmooth operators.

3.2.1 Quantization Noise Bounds

There are a number of techniques and methods that have been suggested in the literature to measure the bounds of the quantization noise. Since the numerical techniques typically lead to exceedingly long computation times, different alternatives have been proposed to obtain results faster.

Table 2 shows the most relevant techniques related to the evaluation of noise bounds. The first column indicates the name of the technique. The second column displays the main characteristics of the technique, while the third column

Table 2 Techniques for the evaluation of the quantization noise bounds

General features	Particular features	System	Loops	Speed	References
Interval arithmetic and range propagation					
Forward-backward propagation: reduces some overestimation but the results are still oversized	Combines three methods to reduce oversize: number of bits, range of each variable, and logic value of each bit. Integrated in the *Bitwise* tool	All	No	Fast	Stephenson [130]
	Inspired by Stephenson et al. [130], combines constraint propagation, simulation, range evaluation and slack analysis. Integrated in the *Précis* tool	All	No	Medium	Chang [23]
Forward propagation	User annotations. Integrated in the *Match* compiler and the *AccelFPGA* tool	All	No	Medium	Nayak [108] Banerjee [4, 5]
	Precision analysis stage based on error propagation	All	No	Medium	Doi [45]
IA overestimation reduction	Integrated in the *Gappa* tool	All	No		De Dine-chin [42]
Multi-interval arithmetic					
More accurate results than IA, but still oversized (splitting does not solve the dependency problem)	Evaluates the propagation of the intervals due to the quantization operations through the feedback loops. Integrated in the *Abaco* set of tools	LTI	Yes	Very fast	Lopez [94]
	Symbolic Noise Analysis (SNA) by splitting the intervals. They take into account the probabilities in the propagation of the error	LTI	No	Fast	Ahmadi [1]
	Based on the Satisfiability Modulo Theory (SMT) the intervals are iteratively reduced by splitting them and selecting which parts are valid	All	Yes	Medium	Kinsman [79–81]

Affine arithmetic					
More accurate results than IA and MIA	It provides guaranteed bounds	LTI	No	Very fast	Fang [53]
	It provides estimates of the bounds. Integrated in the *Abaco* tool	LTI	Yes	Very fast	Lopez [92, 93, 95]
	It provides guaranteed bounds. Implemented on *Minibit* and *Lengthfinder* tools	Polynomial	No	Medium Fast	Lee [84]
Sensitivity analysis					
Based in automatic differentiation. It provides fast results	It computes the maximum deviation for each noise source and performs propagation by means of signal derivatives. It provides guaranteed bounds, yet oversized	Smooth	No	Very fast	Gaffar [58]
Arithmetic transformations					
Analytical approach that follows a similar concept to the Taylor Models. AT provides a canonical representation of the propagation functions	The output is described as a polynomial function of the inputs. The WLs are optimized by considering the imprecision allowed for the quantizations	Polynomial	No	Fast	Pang [112, 124, 125]
	AA is used for range analysis, and (AT, IA) for WL analysis and optimization. Small overestimation	LTI Polynomial	Yes	Very fast Fast	Sarbishei [124, 125]

shows particular features of the cited approaches. The next three columns contain information about the type of systems that the approaches can be applied to (all, polynomial, based on smooth operations and LTI systems), the existence of loops and the computational speed of the approach.

The analytical techniques used to evaluate the noise bounds can be classified in two major groups: (1) interval-based computation (Interval Arithmetic (IA), Multi-IA (MIA), Affine Arithmetic (AA) and satisfiability modulo theory) and (2) polynomial representation with interval remainders (sensitivity analysis and Arithmetic Transformations (AT)). Principal techniques are described in the following paragraphs.

Interval-Based Computations

In the last decade, *interval-based computations* have emerged as an alternative to simulation-based techniques. A high number of simulations are required in order to cover a significant set of possible values of the inputs, so traditional simulation-based techniques imply very long computation times. As an alternative, interval-based methods have been suggested to speedup the computation process. The results are obtained much faster, but they have to deal with the continuous growth of the intervals (oversizing) through the sequence of operations. Thus, these techniques are restricted to a limited subset of systems (mostly LTI or quasi-LTI), or combined with other techniques to reduce the oversize.

The most classical approach is the computation using interval arithmetic (IA), also called *forward propagation*, *value propagation* or *range propagation* techniques. Given the ranges of the inputs of a system, represented by intervals, IA computes the guaranteed ranges of the outputs. The main drawback of these techniques is the so-called *dependency problem*, which is produced when the same variable is used in several places within the algorithms under analysis, since IA is not able to track dependency between variables, ranges are overestimated. To alleviate this situation, some authors have suggested splitting the intervals in a number of sections, generating a *Multi-IA* approach.

One of the earliest works that applied value propagation to the computation of the noise bounds was developed by Stephenson et al. in the *Bitwise* project [130]. They perform forward and backward range propagation, and combine three different types of analysis to optimize the WLs with guaranteed accuracy: analysis of the number of bits, the ranges of the operands, and the logic value of each bit. The analysis of the number of bits provides larger WLs than the analysis of ranges, but limits the LSB of the result. In combination with backward propagation, the evaluation of the logic values of the operands enables some optimization, but it is not significant in the general case. Since the oversizing of these techniques rapidly increases along the sequence of operations, this approach does not provide practical results in complex systems. However, it provides fast and guaranteed results for smaller blocks.

Chang et al. have applied a similar approach in the *Précis* tool [23]. By including fixed-point annotations in Matlab code, they perform fixed-point simulation, range analysis, forward and backward propagation, and slack analysis. The annotations are based on the routine *fixp*, which allows modelling different integer and fractional

WLs, as well as overflow and underflow quantization strategies. They indicate that the combined application of range analysis (MSB) and propagation analysis (LSB) provides accurate WLs, and that the propagation based on the number of bits is more conservative than range analysis for the MSBs. Slack analysis uses the difference between these two results to provide an ordered list of signals that provide better results when their LSBs are optimized [23].

Nayak [108] and Banerjee et al. [4, 5] have applied the propagation techniques to the computation of the noise bounds. They have developed an automatic quantization environment that has been included in the *Match* project and the *AccelFPGA* tool.

In [45], Doi et al. present a WL optimization method that estimates the optimum WLs using noise propagation. They propagate the noise ranges using IA, and apply it in combination with a nonlinear programming solver to estimate the optimum WLs in LTI blocks without loops. Due to the oversizing of the interval-based computations, the bounds provided in this process are conservative in most cases, but the difference with the optimum result is not significant in blocks without loops.

The *Gappa* tool [41, 42] uses a different approach to deal with the oversizing associated to the interval computations. It creates a set of theorems to rewrite the most common expressions into similar ones that are less affected by the correlations in the interval computations. This approach provides guaranteed and accurate results, but up to now its application is limited to systems without loops and branches [41], and requires a very good knowledge of the target system [42].

Multi-IA (MIA) has also been applied by several authors to reduce the width of the bounds of the quantization noise. In [94], the authors suggest a method to reduce the overestimation of IA and use it to provide refined bounds in the impulse response and the transfer function of an Infinite impulse response (IIR) filter. Although MIA provides less conservative bounds than IA, MIA does not solve the dependency problem and is therefore not a good option for systems with loops [95].

The *Symbolic Noise Analysis* (SNA) method presented in [1] splits the noise intervals into smaller parts and performs IA propagation of each part. At the output, intervals are combined according to their probabilities to provide the histogram of the output noise. When there is small or no oversizing, this approach provides accurate estimates of the PDF of the output noise. However, in the general case, this only provides bounds associated to each part, and less conservative global bounds than IA or range propagation methods.

Kinsman and Nicolici [80, 81] propose to use *Satisfiability Modulo Theory* (SMT). This approach initially performs IA propagation of the values of all the signals and noise sources, and provides an initial (conservative) estimate of the bounds at the output. After that, all the sources are successively split using the bisection method to provide less conservative ranges in each iteration. The process finishes after reaching a given constraint or when all the intervals have zero width (degenerated intervals). The authors indicate that this method is particularly useful in presence of discontinuities (such as in systems with divisions or inverse functions) and that it provides more accurate results than AA in non-linear systems [79]. In a later work, the authors have generalized this idea to handle floating- and fixed-point

descriptions using the same solver [80] and have introduced vectors to reduce the amount of terms in the splitting process [81].

Affine Arithmetic (AA) [131] was proposed to optimize the bounds of signals and noise sources in LTI fixed-point realizations [53]. The authors propose to apply AA for feed-forward systems to obtain guaranteed bounds and also to obtain a practical estimation based on a confidence interval. Moreover, an iterative method is proposed for systems with feedback and is proved to always converge although the bounds are overestimated. A more detailed analysis about the application of AA to characterize quantized LTI systems has been carried out in [92, 93, 95]. The authors have evaluated the source and propagation models of AA in fixed-point LTI systems with feedback loops, and have concluded that AA propagates the exact results in systems described by sequences of affine operations (i.e., LTI systems). In [92] and [95], they propose a variation of the description of the quantization operations of AA that provides more accurate estimates of the noise bounds. A comparison between IA, MIA, AA and the proposed approach shows that IA and MIA are affected by the dependency problem in most LTI systems with feedback loops (whenever the filter has complex poles), and do not provide useful results [95]. In [93], the expressions for the generation of the affine sources, the propagation of the noise terms, and the computation of the output results are provided. Although they are oriented to the computation of the MSE statistics, the derivation of the corresponding expressions to obtain the minimum guaranteed bounds is very easily obtained.

AA has also been suggested in combination with Adaptive Simulated Annealing (ASA) to perform WL optimization of fixed-point systems without feedback loops in the tool *Minibit* [85].

Polynomial Representations with Interval Remainders

The *polynomial representations with interval remainders* are based on the perturbation theory and follow a similar idea to the Taylor Models. They perform a polynomial Taylor series decomposition and the smallest uncertainties can be merged in one or more terms, or simply they can be neglected. These approaches have been suggested, in particular in recent years, to perform efficient evaluation of polynomial sequences of operations.

Perturbation theory is based on a Taylor series decomposition of a given order and can include intervals to provide guaranteed bounds of the results. This idea was first presented by Wadekar and Parker [140], but the implementation details of the computation were not given. The most relevant contributions are those based on sensitivity analysis (using first-order derivatives) and arithmetic transformations (canonical polynomial representations with an error interval remainder). Handelman representations [12] can handle more detailed representations of the internal descriptions, they are out of the scope of this paper since their application so far is to floating-point systems.

Gaffar et al. [58] have suggested an approach based on an *automatic differentiation* method and have applied it to linear or quasi-linear systems. The noise bounds are computed as the sum of the maximum deviation of each noise signal multiplied by its corresponding sensitivity. The main advantage of this approach is

that the bounding expression is very easily obtained, since in this type of systems the sensitivities are the operands of the multiplications and the other terms of the Taylor series are considered negligible. However, since it is aimed at providing guaranteed bounds of the results, the provided WLs are usually overestimated even for small blocks [58].

Another interesting approach which acquired relevance in the latest years is the optimization of systems using *Arithmetic Transformations* (AT) [112, 124, 125]. ATs are polynomials that represent pseudo-boolean functions. Their extensions also include word-level inputs and sequential variables in the representations. AT representations are canonical, so the propagation of the polynomial terms is guaranteed to be accurate. In addition, due to their origin, they are particularly well suited to describe and optimize the operations of a given circuit.

In [112], authors distinguish three sources of error: approximation by the finite-order polynomial, quantization of the input signals, and optimization of the WLs of coefficients and result [112]. The combination of these three sources must be less than the specified error bound to provide a valid implementation. They initially determine the order of the Taylor series and the amount of input quantization. After that, a branch and bound algorithm, tuned for this application and guided by the sensitivity, is used for the optimization process [112]. In [125] and [124], the authors extend this approach to evaluate systems containing feedback loops. In [125], they provide the analytical expressions for the analysis of IIR filters, taking into account both MSE statistics and bounds as the target measurements. In [124], they extend this analysis to polynomial systems with loops, and show that AT paired with IA is more efficient than AA to provide the noise bounds. One of the main features of this approach is that it does not require numerical simulations, unlike other similar approaches.

3.2.2 Round-Off Noise Power

Existing approaches to compute the analytical expression of the quantization noise power are based on perturbation theory, which models finite precision values as the addition of the infinite precision values and a small perturbation. At node i, a quantization error signal b_i is generated when some bits are eliminated during a fixed-point format conversion (quantization). This error is assimilated to an additive noise which propagates inside the system. This noise source contributes to the output quantization noise b_y through the gain α_i, as shown in Fig. 6.

The aim of this approach is to define the output noise b_y power expression according to the noise source b_i parameters and the gains α_i between the output and a noise source.

Table 3 summarizes the main techniques to compute the RON power. The first column indicates the type of technique used. The second column displays the main characteristic of the technique, while the next column shows particular features of the cited approaches. The next three columns contain information about the type of systems that the approaches handle (*All*, based on *smooth* operations and *LTI*), the

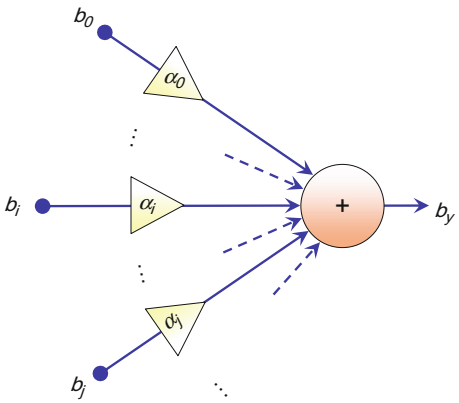


Fig. 6 Model for the computation of output RON power based on noise sources b_i and gains α_i

existence of loops and the computational speed of the approach. The last columns shows the references to the published works.

The next paragraphs focus on the model used for the quantization process, which has three phases: (1) *noise generation*, (2) *noise propagation*, and (3) *noise aggregation*.

Noise Generation

In finite precision arithmetic, signal quantization leads to an unavoidable error. A commonly used model for the continuous-amplitude signal quantization has been proposed in [141] and refined in [129]. The quantization of signal x is modeled by the sum of this signal and a random variable b (quantization noise). This additive noise b is a uniformly distributed white noise that is uncorrelated with signal x and any other quantization noise present in the system (due to the quantization of other signals). The validity conditions of the quantization noise properties have been defined in [129]. These conditions are based on characteristic function of the signal x, which is the Fourier transform of the probability density function (PDF). This model is valid when the dynamic range of signal x is sufficiently greater than the quantum step size and the signal bandwidth is large enough.

This model has been extended to include the computation noise in a system resulting from some bit elimination during a fixed-point format conversion. More especially, the round-off error resulting from the multiplication of a constant by a discrete amplitude signal has been studied in [6]. This study is based on the assumption that the PDF is continuous. However, this hypothesis is no longer valid when the number k of bits eliminated during a quantization operation is small. Thus, in [30], a model based on a discrete PDF is suggested and the first and second-order moments of the quantization noise are given. In this study, the probability value of each eliminated bit to be equal to 0 or 1 is assumed to be $1/2$.

Noise Propagation

Each noise source b_i propagates to the system output and contributes to the noise b_y at the output. The propagation noise model is based on the assumption that the

Table 3 Techniques for the analytical evaluation of the quantization noise power

General features	Particular features	System	Loops	Speed	References
Hybrid techniques					
Based on statistical expressions. Requires large matrix computations.	Coefficients K_i and L_{ij} are computed using fixed-point simulations and then substituted in the statistical matrix equations	Smooth	Yes	Medium	Shi [126]
		Smooth	Yes	Medium	Constantinides [33]
		Smooth	Yes	Medium	Fiore [56]
Impulse response determination					
Based on system transformations. Provides fast results.	Coefficients K_i and L_{ij} are computed from the impulse response between the noise sources and the output. Integrated in the *ID.Fix* tool	LTI	Yes	Very fast	Menard [100]
		Smooth	Yes	Fast	Rocher [122]
Affine arithmetic simulations					
Based on AA simulations. Provides fast results	Coefficients K_i and L_{ij} are computed from the results of the AA simulations. Integrated in the *Abaco* and *Quasar* tools	LTI	Yes	Very fast	Lopez [93]
		Smooth	Yes	See note[a]	Caffarena[18]
Combines MAA and PCE	Provides accurate results in strongly nonlinear systems	Polynomial	Yes	Medium/fast	Esteban [50]

[a]Fast for LTI & non-linear acyclic systems and slow for non-linear cyclic systems

quantization noise is sufficiently small compared to the signal to consider. Thus, the finite precision values can be modeled by using the addition of the infinite precision values and a small perturbation. A first-order Taylor approximation [33, 121] is used to linearize the operation behavior around the infinite precision values. This approach allows obtaining a time-varying linear expression of the output noise according to the input noise [99]. In [126], a second-order Taylor approximation is used directly on the expression of the output quantization noise. In [93] and [18], affine arithmetic is used to model the propagation of the quantization noise inside the system. Affine expression allows obtaining directly a linear expression of the output noise according to the input noises. For non-affine operations, a first order Taylor approximation is used to obtain a linear behaviour. These models, based on the perturbation theory, are only valid for smooth operations. An operation is considered to be smooth if the output is a continuous and differentiable function of its inputs.

Noise Aggregation

Finally, the output noise b_y is the sum of all the noise source contributions. The second order moment of b_y can be expressed as a weighted sum of the statistical parameters of the noise source:

$$E(b_y^2) = \sum_{i=1}^{N_e} K_i \sigma_{b_i}^2 + \sum_{i=1}^{N_e} \sum_{j=1}^{N_e} L_{ij} \mu_{b_i} \mu_{bj} \tag{3}$$

where μ_{b_i} and $\sigma_{b_i}^2$ are respectively the mean and the variance of noise source b_i, and N_e is the total number of error sources. These terms depends on the fixed-point formats and are determined during the evaluation of the accuracy analytical expression. The terms K_i and L_{ij} are constant and depend on the computation graph between b_i and the output. Thus, these terms are computed only once for the evaluation of the accuracy analytical expression. These constant terms can be considered as the gain between the noise source and the output.

For the case of Linear Time-Invariant systems, the expressions of K_i and L_{ij} are given in [101]. The coefficient L_{ij} can now be computed by the multiplication of terms L_i and L_j, which can be calculated independently. The coefficients K_i and L_{ij} are determined from the transfer function $H_i(z)$ or the impulse response $h_i(n)$ of the system having b_i as input and b_y as output. In [100, 102], a technique is proposed to compute these coefficients from the SFG (Signal Flow Graph) of the application. The recurrent equation of the output contribution of b_i is computed by traversing the SFG representing the application at the noise level. To support recursive systems, for which the SFG contains cycles, this SFG is transformed into several Directed Acyclic Graphs (DAG). The recurrent equations associated to each DAG are computed and then merged together after a set of variable substitutions. The different transfer functions are determined from the recurrent equations by applying a Z transform.

In [18], AA is used to keep track of the propagation of every single noise contribution along the datapath, and from this information the coefficients K_i and L_i are extracted. The method has been proposed for LTI in [93] and for non-LTI systems in [18]. An affine form, defined by a central value and an uncertainty term (error term in this context), is assigned to each noise source. These terms depend on the mean and variance of the noise source. Then, the central value and the uncertainty terms associated to each noise source are propagated inside the system through an affine arithmetic based simulation. The values of the coefficients K_i and L_{ii} are extracted from the affine form of the output noise. In the case of recursive systems, it is necessary to use a large number of iterations to ensure that the results converge to stable values. In some cases, this may lead to large AA error terms and therefore to long computation time.

In the method proposed in [122], an analytical expression of the coefficients K_i and L_{ij} is determined. For each noise source b_i, the recurrent equation of the output contribution of b_i is determined automatically from the application SFG with the technique presented in [100]. A time-varying impulse response h_i is computed from each recurrent equation. The output quantization noise b_y is the sum of the noise source b_i convolved with its associated time varying impulse response. The second-order moment of b_y is determined. The expression of the coefficients is proposed in [122]. These coefficients can be computed directly from their expression by approximating an infinite sum, or a linear prediction approach can be used to obtain more quickly the value of these coefficients. The statistical parameters of the signal terms involved in the expression of the coefficients are computed from a single floating-point simulation, leading to reduced computation times. The analysis to compute coefficients K_i and L_{ij} is done on an SFG representing the application and where the control flow has been removed. To avoid loop unrolling which can lead to huge graph, a method based on polyhedral analysis has been proposed in [44].

Different hybrid techniques [33, 56, 126] that combine simulations and analytical expressions have been proposed to compute the coefficients K_i and L_{ij} from a set of simulations. In [126], these $N_e(N_e + 1)$ coefficients are obtained by solving a linear system in which K_i and L_{ij} are the variables. The way to proceed is to carry out several fixed-point simulations where a range of values for σ_{b_i} and μ_{b_i} is covered for each noise source. The fixed-point parameters of the system are set carefully to control each quantizer and to analyze its influence on the output. For each simulation, the statistical parameters of each noise source b_i are known from the fixed-point parameter and the output noise power is measured. At least $N_e(N_e + 1)$ fixed-point simulations are required to be able to solve the system of linear equations. A similar approach is used in [56] to obtain the coefficients by simulation. Each quantizer is perturbed to analyze its influence at the output to determine K_i and L_{ii}. To obtain the coefficients L_{ij} with $i \neq j$, the quantizers are perturbed in pairs. This approach requires again $N_e(N_e+1)$ simulations to compute the coefficients, which requires long computation times.

During the last 15 years, numerous works on analytical approaches for RON power estimation have been conducted and interesting progresses have been made for the automation of this process. These approaches allow for the evaluation of the

RON power and are very fast compared to simulation-based approaches. Theoretical concepts have been established enabling the development of automatic tools to generate the expression of the RON power. The limit of the proposed methods have been identified. Analytical approaches based on perturbation theory are valid for systems made-up of only smooth operations.

3.2.3 Probability Density Function

The probability density function (PDF) of the quantization noise has been used as a metric to analyze the effect of signal quantization. This metric provides more information than the quantization error bounds or the quantization noise power. They are of special interest if applied to the analysis of unsmooth operations since error bounds or noise power are mainly suitable for differentiable operations.

There are two types of measures used to optimize quantized systems: statistical analysis of the quantization noise, and guaranteed bounds of the results. In most cases, statistical analysis techniques only compute the mean and variance of the quantization noise (or, alternatively, the noise power) at the output signal. Since the number of noise sources is usually high, these techniques assume that the Central Limit Theorem is valid, and the output noise follows a Gaussian distribution. Consequently, these two parameters fully characterize the distribution of the quantization noise. However, in systems with non-linear blocks (such as slicers) the Central Limit Theorem can no longer be valid, and a more detailed analysis is required. In this sense, some work focused on evaluating the PDF of the quantization noise.

In the context of guaranteed bounds, the objective is to ensure that the maximum distortion introduced in the quantization process is below a given constraint. Some techniques select the WLs and perform the computations to ensure that the bounds of the quantization noise are below this constraint. Other techniques focus on ensuring that the output of the quantized system is equal to a valid reference (e.g., the floating-point one). In both cases, to obtain efficient implementations, it is important to ensure that the provided bounds are close to the numerical ones, and that the oversizing included in the process (if any) is small.

Stochastic approaches, based on Karhunen-Loève Expansion (KLE) and Polynomial Chaos Expansion (PCE), have been used to model the quantization noise at the output of a system. The output quantization noise PDF can be extracted from the coefficients of the KLE or PCE. In the domain of fixed-point system design, these techniques have been previously proposed to determine the signal dynamic range in LTI [145] and non-LTI systems [146]. In [3], a stochastic approach using KLE is used to determine the quantization noise PDF of an LTI system output. The KLE coefficients associated to a noise source are propagated to the output by means of the impulse response between the noise source and the system output. In [50], a stochastic approach based on a combination of Modified Affine Arithmetic (MAA) and Polynomial Chaos Expansion (PCE) is proposed to determine the output quantization noise PDF. Compared to KLE based approach, PCE allows supporting

non-LTI systems. This technique is based on decomposing the random variables into weighted sums of Legendre orthogonal polynomials. The Legendre polynomial bases are well suited to represent uniformly distributed random variables, thus, they are very efficient to model quantization noise.

The determination of the PDF is required to handle unsmooth operations. In [127], the effect of quantization noise on the signum function is analyzed. This work has been extended in [115] to handle more complex decision operations which have specific contours like in QAM (Quadrature Amplitude Modulation) constellation diagrams. These two models are defined for one single unsmooth operation. Handling systems with several unsmooth operations is still an open issue for purely analytical approaches.

3.3 Simulation-Based and Mixed Approaches

3.3.1 Fixed-Point Simulation-Based Evaluation

The quantization error can be obtained by extracting the difference between the outputs of simulation when the system has a very large precision (e.g. simulation with double-precision floating-point) and when there is quantization (bit-true fixed-point simulation), as shown in Fig. 7. Floating-point simulation is considered to be the reference given that the associated error is definitely much smaller than the error associated to fixed-point computation. Different error metrics can be computed from the quantization error obtained from this simulation. The main advantage of simulation-based approaches is that every kind of application can be supported. Fixed-point simulation can be performed using tools such as [40, 47, 75, 96].

Different `C++` classes, to emulate the fixed-point mechanisms have been proposed, such as `sc_fixed` (*SystemC*) [11], `ac_fixed` (*Algorithm C Data Types*) [104] or `gFix` [77]. The `C++` class attributes define the fixed-point parameters associated to the data: integer and fractional word-lengths, overflow and quantization modes, signed/unsigned operations. For `ac_fixed`, the fixed-point attributes can be parametrized through template parameters. For `sc_fixed`, these attributes can be static to obtain fast simulations or dynamic so they can be modified at run-time. Bit-true operations are performed by overloading the different arithmetic operators. During the execution of a fixed-point operation, the data range is analyzed and the

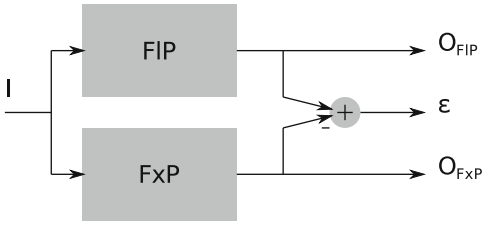


Fig. 7 Simulation-based computation of quantization error

overflow mode is applied if required. Then, the data is cast with the appropriate quantization mode. Thus, for a single fixed-point operation, several processing steps are required to obtain a bit true simulation. Therefore, these techniques suffer from a major drawback which is the extremely long simulation time [39]. This becomes a severe limitation when these methods are used in the data word-length optimization process where multiple simulations are needed. The simulations are made on floating-point machines and the extra-code used to emulate fixed-point mechanisms increases the execution time between one to two orders of magnitude compared to traditional simulations with native floating-point data types [36, 76]. Besides, to obtain an accurate estimation of the statistical parameters of the quantization error, a great number of samples must be taken for the simulation. This large number of samples combined with the fixed-point mechanism emulation lead to very long simulation time.

Different techniques have been proposed to reduce this overhead. The execution time of the fixed-point simulation can be reduced by using more efficient fixed-point data types. In [77], the aim is to reduce the execution time of the fixed-point simulation by using efficiently the floating-point units of the host computer. The mantissa is used to compute the integer operations. Thus, the word-length of the data is limited to 53 bits for double data types. The execution time is one order of magnitude greater than the one required for a fixed-point simulation. This technique is also used in *SystemC* [11] for the fast fixed-point data types.

The fixed-point simulation can be accelerated by executing it on a more adequate machine like a fixed-point DSP [37, 39, 73, 78, 82] or an FPGA [38] through hardware acceleration. In the case of hardware implementation, the operator word-length, the supplementary elements for overflow and quantization modes are adjusted to comply exactly with the fixed-point specification which has to be simulated. In the case of software implementation, the operator and register word-lengths are fixed. When the word-length of the fixed-point data is lower than the data word-length supported by the target machine, different degrees of freedom are available to map the fixed-point data into the target storage elements. In [39], to optimize this mapping, the execution time of the fixed-point simulation is minimized. The cost integrates the data alignment and the overflow and quantization mechanism. This combinatorial optimization problem is solved by a divide and conquer technique and several heuristics to limit the search space are used. In [82] a technique is proposed to minimize the execution time due to scaling operations according to the shift capabilities of the target architecture. In the same way, the aim of the Hybris simulator [36, 76] is to optimize the mapping of the fixed-point data described with *SystemC* into the target architecture register. All compile-time information are used to minimize the number of operations required to carry-out the fixed-point simulation. The overflow and quantization operations are implemented by conditional structures, a set of shift operations or bit mask operations. Nevertheless, to obtain fast simulation, some quantization modes are not supported. In [143], the binary point alignment is formulated as a combinatorial optimization problem and an integer linear programming approach is used to solve it. But, this approach is limited to simple applications to obtain reasonable

optimization times. These methods reduce the execution time of the fixed-point simulation but, this optimization needs to be performed every time that the fixed point configuration changes. Accordingly, it might not compensate for the execution time gain of the fixed-point simulation when involving complex optimizations.

3.3.2 Mixed Approach

To handle systems made-up of unsmooth operations, a mixed approach which combines analytical evaluations and simulations has been proposed in [113, 114]. The idea is to evaluate directly the application performance metric with fixed-point simulation and to accelerate drastically the simulation with analytical models. In this technique the analytical approach is based on the perturbation theory and the simulation is used when the assumptions associated with perturbation theory are no longer valid (i.e. when a decision error occurs). In this case, the quantization noise at the unsmooth operation input can modify the decision at the operation output compared to the one obtained with infinite precision.

This technique selectively simulates parts of the system only when an decision error occurs [114]. Given that decision errors are rare event the simulation time is not so important as for classical fixed-point simulations. The global system is divided into smooth clusters made-up of smooth operations. These smooth clusters are separated by unsmooth operations. The single source noise model [103] is used to capture the statistical behavior of quantization noise accurately at the output of each smooth cluster. In [103], The authors propose to model the output quantization noise of a LTI system with a weighted sum of a Gaussian random variable and a uniform random variable. In [123], the output quantization noise of a smooth system is modeled by a generalized Gaussian random variable, whose parameters define the shape of the PDF. These parameters are analytically determined from the output quantization noise statistics (mean, variance and kurtosis). The general expression of the noise moments are given in [123], and are computed from the impulse responses between the noise sources and the system output.

4 Effect of Coefficient Quantization

Coefficient Quantization (CQ) is the part of the implementation process that describes the degradation of the system operation due to the finite WL representation of the constant values of a system. Especially this problematic is of high importance for LTI systems with the quantization of the coefficients. Opposite to RON, CQ modifies the impulse and frequency responses for LTI system and the functionality for other systems. In the analysis of the quantization effects for LTI systems, this parameter is the first to be determined, since it involves two major tasks: (1) the selection of the most convenient filter structure to perform the required operation, and (2) the determination of the actual values of the coefficients associated to it.

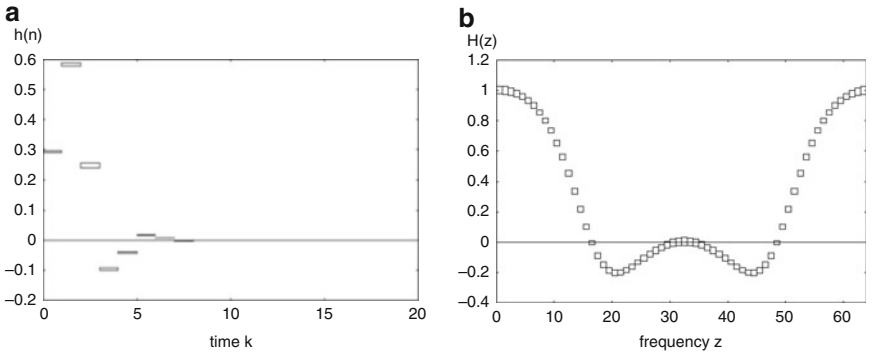


Fig. 8 Effect of CQ on a given filter realization: (**a**) Evolution in time of the impulse response of the differences in the output response. (**b**) Distribution of the effects in the frequency domain. The intervals represent the deviation between the quantized and unquantized samples of the impulse response and the transfer function

Figure 8 illustrates the amount of deviation due to CQ by means of interval simulations. A butterworth filter has been realized in DFIIt (Direct Form II transposed) form, and each coefficient has been replaced by a small interval that describes the difference between the ideal coefficient and the quantized one using seven fractional bits. Figure 8a shows the impulse response of the realization, where the size of each interval reveals how sensitive is each sample to this quantization of coefficients. Figure 8b shows the transfer function associated to it, where in this case the intervals reveal the most sensitive frequencies to the same set of quantizations.

In LTI systems, CQ has been traditionally measured using the so-called Coefficient Sensitivity (CS). Although this parameter was originally defined for LTI systems, whose operation is described by $H(z)$, its current use has also been extended to non-linear systems.

Table 4 summarizes the most important techniques and groups related to the computation of the CS. The first column indicates the type of technique used to compute this parameter (residues, geometric sum of matrices, Lyapunov equations, perturbation theory). The second and third columns respectively provide the most important work in this area, and the most relevant features in each case. The last two columns provide the main advantages and disadvantages of the different approaches. First, an overview of the different parameters used in the literature to measure the CS is presented, before discussing in more detail the L_2-sensitivity. Second, the most commonly-used L_2-sensitivity computation procedures are described. Finally, several analytical techniques are described.

Table 4 Measurement techniques for the computation of the Coefficient Sensitivity (CS)

Features		Advantages	Disadvantages	References
Evaluation of the residues				
General analytical procedure based on complex mathematical equality		General method. Provides exact results	Very complex to develop. Different analysis for each structure	Roberts [119]
Geometric sum of matrices				
Analytical procedure that approximates S_{L12} by using infinite sums in state-space realizations.		The analytical expressions easier to obtain	Limited to state-space realizations. Provides an upper bound	Hinamoto [66]
Lyapunov equations				
Provides the analytical expression for families of filter structures, mainly state-space realizations.		Fast and exact results (without infinite sums)	Iterative method. Limited to certain filter structures	Li [89] Hilaire [64]
Perturbation theory				
Compute the sum of deviations of all the coefficients.	Analytical approach based on Lyapunov equation	Extremely fast, if the analytical expression is obtained	Limited to state-space realizations.	Xiao [147]
	Interval-based procedure	Fast and automatic. Valid for all types of systems	Approximated value. Requires interval computations support	Lopez [91]

4.1 Measurement Parameters

A number of procedures have been initially suggested to minimize the degradation of $H(z)$ with respect to the quantization of all coefficients of the realization under different constraints [133–135]. In these procedures, the coefficients of the realization have been obtained by minimizing the so-called L_1/L_2-*sensitivity*, $S_{L_{12}}$ [59, 67–69, 133–135, 144]. The main feature of this parameter is that its upper bound is easily obtained [59, 88, 144]. However, two different norms are applied to obtain the result. Therefore, its physical interpretation is not clear.

Instead, it is more natural to measure the deviations of $H(z)$ using only the L_2-norm [68, 88]. For this reason, the so-called L_2-*sensitivity*, S_{L2}, is currently applied [67, 68]. The main feature of this parameter is that it is proportional to the variance of the deviation of $H(z)$ due to the quantization of all the coefficients of the realization [59, 67, 68, 144]. However, the computation of its analytical expression requires performing extremely complex mathematical operations [68, 89, 144]. Due

to this fact the computation of the L_2-sensitivity has been limited to simple linear structures, typically SSR (State-Space Representation) forms. Since each analytical expression only characterizes one family of filter structures, it requires developing a new mathematical expression to optimize or compare each new structure. The most recent work in this area are focused on minimizing the L_2-sensitivity of two-dimensional (2-D) SSR filter structures [67, 68], and of structures based on the generalized delta operator [89, 148].

In [136–138], the authors have compared the performance of the filter structures by computing the maximum value of the magnitude sensitivity, S_{mag}, or the relative sensitivity, S_{mag}. The main feature of S_{mag} and S_{rel} is that their numerical values are more easily computed than the analytical expressions of $S_{L_{12}}$ or S_{L2}. For this reason, they have been used in combination with simulated annealing or genetic algorithms that perform automated search of the most robust structures against the quantization of coefficients [136–138]. However, S_{mag} and S_{rel} only provide information of the maximum deviations of $H(z)$. In contrast, the L_2-sensitivity provides global information about the deviations of $H(z)$. For this reason, this parameter is widely preferred [59, 66, 68, 88, 89].

In [64], the authors introduce a unified algebraic description able to represent the most widely used families of filter realizations. They focus on the fixed- and floating-point deviation of the transfer function and pole measures using CS parameters. They apply Adaptive Simulated Annealing to obtain the optimal realization among these structures. In particular, they introduce the S_{L2}^{W} measure, which considers the individual quantization of coefficients into the traditional L_2-sensitivity parameter. This work has been further expanded in [62] to include L_2-scaling constraints, and in [63] to include the evaluation of MIMO filters and controllers.

Table 5 summarizes the parameters introduced in this Section. In each column, the representation of the different parameters, the main references associated to them, and their most important features, advantages and disadvantages are also briefly outlined.

4.2 L_2-Sensitivity

Since the L_2-Sensitivity is much more commonly used than the others CQ measurement parameters, in this section its mathematical definition and physical interpretation are described in more detail.

Definition The L_2-sensitivity is the parameter that quantitatively measures the influence of the variations of all the coefficients of the realization in the transfer function. Its mathematical definition is as follows

$$S_{L_2} = \sum_{i=1}^{n_c} S_{c_i} = \sum_{i=1}^{n_c} \left\| \frac{\partial H(z)}{\partial c_i} \right\|_2^2 \tag{4}$$

Table 5 Measurement parameters for coefficient quantization

Parameter	Features	Advantages	Disadvantages	References
S_{L12}	Initial measure of the coefficient sensitivity, based on the L_1 and the L_2-norms	It has a simple expression in some filter structures	Only provides an upper bound, based on two different norms	[69, 133–135]
S_{L2}	Advanced measurement, based only on the L_2-norm. Development of the expressions associated to each filter structure	Global measurement. It has statistical meaning	Complex to develop	[59, 67, 68, 88, 89, 144, 148]
S_{mag}, S_{rel}	Information about the magnitude of the quantizations	Computationally simple	Only provides information about the maximum deviations	[136–138]
S_{L2}^{W}	Measures the actual deviations of coefficients with different amount of quantizations	More accurate than S_{L2}	Requires complex analytical developments	[62–64]

where S_{c_i} is the sensitivity of the transfer function with respect to coefficient c_i, and $\|X(z)\|_2^2$ represents the L_2-norm of $X(z)$ [66, 89]. This definition considers that all the coefficients of the set $i = \{1, \ldots, n_c\}$ are affected by quantization [45]. Coefficients not affected by quantization operations (i.e., those that are exactly represented with the assigned number of bits) are excluded from this set.

Statistical Interpretation Using a first-order approximation of the Taylor series, the degradation of $H(z)$ due to the quantization of the coefficients follows

$$\Delta H(z) = H_{Q_c}(z) - H(z) = \sum_{i=1}^{n_c} \frac{\partial H(z)}{\partial c_i} \Delta c_i \tag{5}$$

where $H_{Q_c}(z)$ is the transfer function of the realization with quantized coefficients. From a statistical point of view, the variance of the degradation of $H(z)$ due to these quantization operations is given by

$$\sigma_{\Delta H}^2 = \sum_{i=1}^{n_c} \left\| \frac{\partial H(z)}{\partial c_i} \right\|_2^2 \sigma_{\Delta c_i}^2 = \sum_{i=1}^{n_c} S_{c_i} \sigma_{\Delta c_i}^2 \tag{6}$$

When all the coefficients are quantized to the same number of bits, $\sigma_{\Delta c_i}^2$ is equal to the common value $\sigma_{\Delta c}^2$. In this case, Eq. (6) is simplified to

$$\sigma_{\Delta H}^2 = \sum_{i=1}^{n_c} S_{c_i} \sigma_{\Delta c}^2 = S_{L_2} \sigma_{\Delta c}^2 \tag{7}$$

where $\sigma^2_{\Delta c}$ is the variance of the coefficients affected by the quantization operations.

Therefore, S_{L2} provides a global measure of the degradation of $H(z)$ with respect to the quantization of all the coefficients of the realization. Consequently, in the comparison of the different filter structures, the L_2-sensitivity indicates the most robust realizations against the quantization of coefficients. However, it must be noted that once the final realization has been chosen, the quantization of coefficients has deterministic effects on the computation of the output samples, and the behaviour of the filter structure is completely determined by $H_{Q_c}(z)$.

4.3 Analytical Approaches to Compute the L_2-Sensitivity

The analytical computation of the L_2-sensitivity is based on calculating the individual sensitivities of the coefficients of the realization. There are three different types of techniques: (1) evaluation of residues, (2) geometric series of matrices, or (3) Lyapunov equations. However, since all of them are based on developing expressions for the different realizations, they are only valid for particular structures, mainly SSR (State-Space Realization) and DFIIt (Direct Form II transposed) forms.

Evaluation of the Residues The reference procedure to compute the value of S_{L2} is to analytically develop the expressions of the derivatives of $H(z)$ [119]. This approach separately computes the L_2-norms of the sensitivities of the coefficients. The derivatives involved in this process are extremely complex, even in simple LTI systems. Therefore, this procedure is only applicable to compute the reference values in some low-complexity LTI systems.

Geometric Series of Matrices (GSM) In this case, the expression to compute the S_{L2} is transformed into an equivalent expression that computes the sensitivity of all the coefficients of the same group [66]. This procedure computes an upper bound of S_{L2}, which is equal to the real value if all the coefficients of the SSR filter are quantized [147]. Its main advantage is that it is easily extended to n-D filters [66]. However, it has two important drawbacks: (1) its application to non-SSR structures or sparse realizations has not been defined; and (2) due to the infinite sums involved, the results are only approximated up to a given degree of accuracy. The approximations can be made as accurate as required by adding a large number of terms, but in such cases the computation times involved to provide the results can be very high.

Lyapunov Equations (LEs) In this procedure, the computation of the infinite sum of matrices of the GSM method is replaced by the computation of the solutions of their associated LEs. This procedure is very accurate and fast, but requires performing iterative computations, and the involved equations must be solved for each non-zero coefficient [147]. Its main drawback is that these expressions are only applicable to 1-D SSR filters. This procedure has also been used in [89] to develop the expressions of the L_2-sensitivity of DFIIt structures with generalized

delta operators, and in [64, 65] to include different amounts of quantization in each coefficient of the realization.

Perturbation Methods The existing analytical techniques to compute the S_{L2} have the drawback of being only valid for each family of filter structures, and the required expressions are in most cases very difficult to develop. Moreover, these techniques cannot be extended to evaluate the sensitivity of a given signal in non-linear systems.

In [147], the author suggests an analytical approach based on an improved S_{L2} measure that separately computes the sensitivities of all the coefficients of the realization. Using this improved measure, an analytical expression to compute the S_{L2} based on LEs for state-space realizations is derived. This measure is more accurate, and the computation of S_{L2} as the sum of contributions of the individual coefficients facilitates the automatization. The author also develops the analytical expressions for the state-space realizations, but these expressions cannot be generalized.

5 System Stability Due to Signal Quantization

Although most of existing techniques to evaluate the quantization effects are based on substituting the quantizers by additive noise sources, this approximation is only valid under certain assumptions (see Sect. 3.2) [6, 10, 71, 117, 142]. In particular, when the quantization operations in the feedback loops significantly affect the behavior of the system, oscillations of a given frequency and amplitude may appear, provoking an unstable behaviour at the output. These oscillations are called Limit Cycles [27, 97, 106, 116, 119].

Figure 9 shows an example of the existence of LCs. In unquantized systems, the output response tends to zero, since it is a requirement of the stability of the LTI systems (Fig. 9a). In quantized systems, due to the nonlinear effect of the quantization operations, the output response may present self-sustained oscillations of a given amplitude and frequency (Fig. 9b). These two parameters vary according to the quantized realization and the values of the input signals, although certain conditions have been provided in the literature to keep them under a given limit.

To detect the oscillations, the actual behavior of the quantizers must be evaluated, instead of substituting them by their respective equivalent linear models (i.e. noise sources) [6, 27, 117, 119]. In LTI systems these oscillations have been extensively analyzed in the second-order sections [16, 25–27, 87], and sufficient conditions that ensure the absence of LCs have also been developed [46, 72, 128], particularly in regular filters structures [17, 27, 48, 49, 54, 55, 61, 70, 116, 119].

In this Section, a classification of the procedures most commonly used to guarantee the absence of LCs in digital filters is first presented, followed by a description of the automatic techniques to detect LCs.

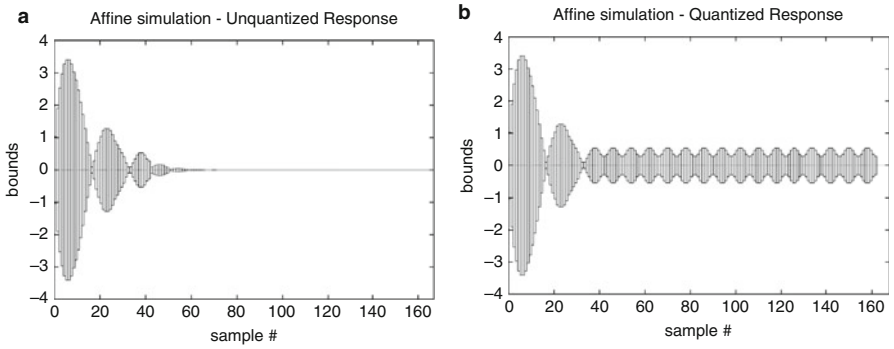


Fig. 9 Detection of LCs in a filter using AA-based computations. The joint simulation of all the input values allows fast detection of system instabilities and self-sustained oscillations: (**a**) The unquantized response of the reference interval [−1,1] at sampled time k=0 tends to zero. (**b**) The quantized system generates LCs due to the nonlinear effects of the quantization operations and the feedback loops

5.1 Analysis of Limit Cycles in Digital Filters

Limit Cycles (LCs) are self-sustained oscillations that appear due to the propagation of the non-linear effects of the quantization operations through the feedback loops [97, 106, 116, 119]. The techniques aimed to detect LCs primarily intend to bound the maximum amplitudes of these oscillations [19], and, in particular, their effect at the output signal.

Similarly to the computation of the RON and the CQ, the techniques used to detect and bound the LCs are classified into analytical and simulation-based. *Analytical techniques* provide three different types of results [19]: (1) they give sufficient conditions to ensure asymptotic stability of filters after quantization [14, 15, 105, 119]; (2) they present requirements for the absence of LCs [139]; or (3) they describe strategies to eliminate zero-input and constant-input LCs [83]. These techniques have been used to select realizations where the absence of LCs is guaranteed. However, they are not able to evaluate all the possible values of the coefficients, so in general they must be combined with simulation-based procedures for a detailed analysis of the target structure. Moreover, these techniques have focused on obtaining the analytical expressions of the coefficients of the second-order sections and SSR filters, but there are few results about factored-SSR filters of arbitrary order [119], and they do not consider arbitrary number of quantizers. Consequently, this type of technique is not suitable to perform automated analysis of LCs of generic filter structures.

Simulation-based techniques perform exhaustive evaluations of all the possible sets of values of the state variables [8, 19, 74, 90, 92, 118]. They provide precise results, but they require exceedingly long computation times [8, 74, 118]. Consequently, this type of technique allows automated analysis of LCs in generic filter structures, but requires a bounding stage to perform these computations in realistic computation times.

The application of AA-based simulations reduces by several orders of magnitude the computation time required to bound the LCs of generic filter structures. Moreover, they can be used in combination with numerical simulations to detect the presence or to guarantee the absence of LCs [90, 92].

5.2 Simulation-Based LC Detection Procedures

Existing simulation-based LC detection procedures perform the computation in two stages [8, 74, 118]: (1) they compute the bounds of the maximum amplitude and frequency of the LCs; and (2) they perform exhaustive search for LCs among all the possible combinations of values of the state variables (SVs) contained within these bounds. Since the SVs have a finite number of bits, the number of possible combinations of values of the SVs is also finite, i.e.,

$$n_{st} = \prod_i (2^{q_i}), \qquad i = 1, \ldots, n_{SV} \tag{8}$$

where n_{SV} is the number of SVs of the target structure, and q_i is the number of bits of state variable i.

From (8), it is clear that the number of combinations, n_{st}, is huge even for small-order filters. Consequently, the aim of the first step is to reduce the number of combinations to be tested for LCs. This reduction is obtained by limiting the maximum values of the SVs, M, or the maximum period of oscillation, T_{max} [74, 118]. The expressions of M and T_{max} are difficult to obtain, and they are dependent on the filter structure. The interested reader is referred to [118] for the expressions of these parameters in SSR filters, and to [74] for their expressions in second-order DFIIt forms with delta operators.

The exhaustive search is performed by evolving the values of the SVs. In each iteration, four possible cases may occur [74]: (1) The state vector is repeated, which means that a LC is found. (2) The state converge to a point that produces zero output. This situation occurs when the values of the SVs are below a given threshold. (3) The state vector has grown out of the search space. (4) The maximum number of steps has expired. If none of these situations occur, the state vector evolves to the values of the next iteration. The most recent algorithms make use of alternative procedures to speed up the required computations, but they still follow the basic principles explained above [74]. They consider that: (a) the large values of the SVs do not need to be tested due to condition (3); (b) the small values of the SVs converge to zero output in short time; and (c) most LCs have short period of oscillation, so they are quickly identified.

In summary, the existing simulation-based procedures are based on performing exhaustive searches among the values of the SVs but they need a binding stage, which depends on the target structure. This type of procedure can be accelerated in combination with AA [90, 92], since it is capable of evaluating a large number of states in a single execution of the algorithm.

6 Summary

Fixed-point design plays a major role in the VLSI implementation of state-of-the-art multimedia and communication applications. This chapter surveys the major works related to the automated evaluation of fixed-point quantization effects, focusing on signal quantization, coefficient quantization and system stability. The main approaches in the field have been explained and classified covering simulation-based, analytical and hybrid techniques. The chapter is intended to provide digital designers with a useful guide while facing the design of fixed-point systems.

When assessing the effect of signal quantization the designer can use general approaches such as simulation-based techniques but at the expense of expending a long time in the quantization process. For particular types of systems it is possible to apply analytical and hybrid automatic techniques that reduce computation time considerably. As a general remark, all the available techniques are not suitable to the optimization of high-complexity systems, so a system-level approach to quantization is most needed.

Regarding, coefficient quantization the designer has to check the impact of finite WL coefficient on the system properties (i.e. frequency response). The majority of the available techniques are system-specific and require the manual development of analytical expressions, so there are still research opportunities in this problem.

Finally, the detection of LCs, the main approaches are based on simulations and exhaustive search, so the computation time can be high for complex systems. Also, the starting condition of the algorithms are system dependant so the results depend on user experience. A fast approach for LC detection is needed, so there is still room for research in this area.

Quantization is an intriguing field of research which has been open for more than 30 years, and the most impacting contributions are still to come, as no general solution exists yet in practice.

References

1. A. Ahmadi and M. Zwolinski. Symbolic noise analysis approach to computational hardware optimization. In *IEEE/ACM Design Automation Conference, 2008. DAC 2008*, pages 391–396, 2008.
2. G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. Academic Press, New York, 1983.
3. A. Banciu, E. Casseau, D. Menard, and T. Michel. Stochastic modeling for floating-point to fixed-point conversion. In *IEEE International Workshop on Signal Processing Systems, (SIPS)*, Beirut, October 2011.
4. P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, and R. Uribe. Automatic conversion of floating point matlab programs into fixed point fpga based hardware design. In *Field-Programmable Custom Computing Machines (FCCM)*, pages 263–264, 2003.
5. P. Banerjee, M. Haldar, A. Nayak, V. Kim, V. Saxena, S. Parkes, D. Bagchi, S. Pal, N. Tripathi, D. Zaretsky, R. Anderson, and J. Uribe. Overview of a compiler for synthesizing matlab programs onto fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):312–324, 2004.

6. C. Barnes, B. N. Tran, and S. Leung. On the Statistics of Fixed-Point Roundoff Error. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 33(3):595–606, June 1985.
7. B. Barrois, K. Parashar, and O. Sentieys. Leveraging Power Spectral Density for Scalable System-Level Accuracy Evaluation. In *IEEE/ACM Conference on Design Automation and Test in Europe (DATE)*, page 6, Dresden, Germany, Mar. 2016.
8. P. Bauer and L.-J. Leclerc. A computer-aided test for the absence of limit cycles in fixed-point digital filters. *IEEE Transactions on Signal Processing*, 39(11):2400–2410, 1991.
9. A. Benedetti and P. Perona. Bit-Width Optimization for Configurable DSPs by Multi-interval Analysis. In *IEEE Asilomar Conf. on Signals, Systems and Computers*, 2000.
10. W. Bennett. Spectra of quantized signals. *Bell System Tech. J.*, 27:446–472, 1948.
11. F. Berens and N. Naser. *Algorithm to System-on-Chip Design Flow that Leverages System Studio and SystemC 2.0.1*. Synopsys Inc., march 2004.
12. D. Boland and G. Constantinides. Bounding variable values and round-off effects using Handelman representations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(11):1691–1704, 2011.
13. D. Boland and G. Constantinides. A scalable precision analysis framework. *IEEE Transactions on Multimedia*, 15(2):242–256, 2013.
14. T. Bose and M. Chen. Overflow oscillations in state-space digital filters. *IEEE Transaction on Circuits and Systems*, 38(7):807–810, 1991.
15. T. Bose and M. Chen. Stability of digital filters implemented with two's complement truncation quantization. *IEEE Transaction on Signal Process.*, 40(1):24–31, 1992.
16. H. Butterweck, A. van Meer, and G. Verkroost. New second-order digital filter sections without limit cycles. *IEEE Transactions on Circuits and Systems*, 31(2):141–146, 1984.
17. M. Buttner. Elimination of limit cycles in digital filters with very low increase in the quantization noise. *IEEE Transactions on Circuits and Systems*, 24(6):300–304, 1977.
18. G. Caffarena, C. Carreras, J. Lopez, and A. Fernandez. SQNR Estimation of Fixed-Point DSP Algorithms. *Int. J. on Advances in Signal Processing*, 2010:1–11, 2010.
19. J. Campo, F. Cruz-Roldan, and M. Utrilla-Manso. Tighter limit cycle bounds for digital filters. *IEEE Signal Processing Letters*, 13(3):149–152, 2006.
20. M. Cantin, Y. Savaria, D. Prodanos, and P. Lavoie. A Metric for Automatic Word-Length Determination of Hardware Datapaths. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):2228–2231, October 2006.
21. M.-A. Cantin, Y. Savaria, and P. Lavoie. A comparison of automatic word length optimization procedures. In *IEEE International Symposium on Circuits and Systems*, volume 2, pages II–612–II–615 vol.2, 2002.
22. F. Catthoor, H. de Man, and J. Vandewalle. Simulated-annealing-based optimization of coefficient and data word-lengths in digital filters. *International Journal of Circuit Theory and Applications*, I:371–390, 1988.
23. M. Chang and S. Hauck. Precis: a usercentric word-length optimization tool. *IEEE Design Test of Computers*, 22(4):349–361, 2005.
24. J.-M. Chesneaux, L.-S. Didier, and F. Rico. Fixed CADNA library. In *Real Number Conference (RNC)*, pages 215–221, Lyon, France, September 2003.
25. T. Claasen and L. Kristiansson. Necessary and sufficient conditions for the absence of overflow phenomena in a second-order recursive digital filter. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 23(6):509–515, 1975.
26. T. Claasen, W. Mecklenbrauer, and J. Peek. Second-order digital filter with only one magnitude-truncation quantizer and having practically no limit-cycles. *Electronics Letters*, 9(22):531–532, 1973.
27. T. Claasen, W. Mecklenbrauker, and J. Peek. Effects of quantization and overflow in recursive digital filters. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 24(6): 517–529, 1976.
28. J. A. Clarke, G. A. Constantinides, and P. Y. K. Cheung. Word-length selection for power minimization via nonlinear optimization. *ACM Transactions on Design Automation of Electronic Systems*, 14(3): 1–28, 2009.

29. G. Constantinides. *High Level Synthesis and Wordlength Optimization of Digital Signal Processing Systems*. PhD thesis, Electr. Electron. Eng., Univ. London, 2001.
30. G. Constantinides, P. Cheung, and W. Luk. Truncation Noise in Fixed-Point SFGs. *IEE Electronics Letters*, 35(23):2012–2014, 1999.
31. G. Constantinides, P. Cheung, and W. Luk. Roundoff-noise shaping in filter design. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 57–60, Geneva, May 2000.
32. G. Constantinides, P. Cheung, and W. Luk. Wordlength optimization for linear digital signal processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(10):1432– 1442, October 2003.
33. G. A. Constantinides. Word-length optimization for differentiable nonlinear systems. *ACM Transactions on Design Automation of Electronic Systems*, 11(1):26–43, 2006.
34. G. A. Constantinides, P. Y. K. Cheung, and W. Luk. Wordlength Optimization for Linear Digital Signal Processing. *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, 22(10):1432–1442, 2003.
35. G. A. Constantinides and G. J. Woeginger. The complexity of multiple wordlength assignment. *Applied Mathematics Letters*, 15(2):137–140, 2002.
36. M. Coors, H. Keding, O. Luthje, and H. Meyr. Fast Bit-True Simulation. In *ACM/IEEE Design Automation Conference (DAC)*, pages 708–713, Las Vegas, june 2001.
37. M. Coors, H. Keding, O. Luthje, and H. Meyr. Integer Code Generation For the TI TMS320C62x. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Sate Lake City, May 2001.
38. L. D. Coster. *Bit-True Simulation of Digital Signal Processing Applications*. PhD thesis, KU Leuven, 1999.
39. L. D. Coster, M. Ade, R. Lauwereins, and J. Peperstraete. Code Generation for Compiled Bit-True Simulation of DSP Applications. In *IEEE International Symposium on System Synthesis (ISSS)*, pages 9–14, Hsinchu, December 1998.
40. Coware. Coware SPW. Technical report, Coware, 2010.
41. M. Daumas and G. Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Softw.*, 37(1):2:1–2:20, Jan. 2010.
42. F. de Dinechin, C. Q. Lauter, and G. Melquiond. Assisted verification of elementary functions using gappa. In *Applied computing*, SAC '06, pages 1318–1322, New York, NY, USA, 2006. ACM.
43. L. de Figueiredo and J. Stolfi. Affine arithmetic: Concepts and applications. *Numerical Algorithms*, 37(1):147–158, 2004.
44. G. Deest, T. Yuki, O. Sentieys, and S. Derrien. Toward scalable source level accuracy analysis for floating-point to fixed-point conversion. In *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '14, pages 726–733, Piscataway, NJ, USA, 2014. IEEE Press.
45. N. Doi, T. Horiyama, M. Nakanishi, and S. Kimura. Minimization of fractional wordlength on fixed-point conversion for high-level synthesis. In *Asia and South Pacific Design Automation Conference, 2004*. Pages 80 – 85, 27-30 2004.
46. P. Ebert, J. Mazo, and M. Taylor. Overflow oscillations in digital filters. *Bell System Tech. J.*, 48:2999–3020, 1969.
47. J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming Heterogeneity, the Ptolemy Approach. *Proceedings of the IEEE*, 91, 2003.
48. K. Erickson and A. Michel. Stability analysis of fixed-point digital filters using computer generated Lyapunov functions- part i: Direct form and coupled form filters. *IEEE Transactions on Circuits and Systems*, 32(2):113–132, 1985.
49. K. Erickson and A. Michel. Stability analysis of fixed-point digital filters using computer generated Lyapunov functions- part ii: Wave digital filters and lattice digital filters. *IEEE Transactions on Circuits and Systems*, 32(2):132–142, 1985.
50. L. Esteban, J. Lopez, E. Sedano, S. Hernandez-Montero, and M. Sanchez. Quantization analysis of the infrared interferometer of the tj-ii for its optimized fpga-based implementation. *IEEE Transactions on Nuclear Science*, page accepted, 2013.

51. C. Fang, T. Chen, and R. Rutenbar. Lightweight Floating-Point Arithmetic: Case Study of Inverse Discrete Cosine Transform. *EURASIP J. on Applied Signal Processing*, 2002(2002):879–892, 2002.
52. C. Fang, T. Chen, and R. Rutenbar. Floating-point error analysis based on affine arithmetic. In *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, 2:561–564, 2003.
53. C. Fang, R. Rutenbar, and T. Chen. Fast, accurate static analysis for fixed-point finite-precision effects in dsp designs. In *Int. Conf. on Computer-Aided Design, 2003 (ICCAD '03).*, pages 275–282, 2003.
54. A. Fettweis. Some principles of designing digital filters imitating classical filter structures. *IEEE Transactions on Circuits and Systems*, 18(2):314–316, 1971.
55. A. Fettweis. Wave digital filters: Theory and practice. *Proceedings of the IEEE*, 74:270–327, 1986.
56. P. Fiore. Efficient Approximate Wordlength Optimization. *IEEE Transactions on Computers*, 57(11):1561 –1570, November 2008.
57. A. Gaffar, O. Mencer, and W. Luk. Unifying Bit-Width Optimisation for Fixed-Point and Floating-Point Designs. In *IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 79–88, 2004.
58. A. Gaffar, O. Mencer, W. Luk, P. Cheung, and N. Shirazi. Floating-point bitwidth analysis via automatic differentiation. In *IEEE International Conference on Field-Programmable Technology, 2002. (FPT)*, pages 158–165, 2002.
59. M. Gevers and G. Li. *Parametrizations in control, estimation, and filtering problems : accuracy aspects*. Communications and control engineering series. Springer-Verlag, London ; New York, 1993. Michel Gevers and Gang Li.
60. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.
61. A. Gray and J. Markel. Digital lattice and ladder synthesis. *IEEE Trans. Audio Electroacoust.*, 21:491–500, 1973.
62. T. Hilaire. Low-parametric-sensitivity realizations with relaxed L_2-dynamic-range-scaling constraints. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 56(7):590–594, 2009.
63. T. Hilaire and P. Chevrel. Sensitivity-based pole and input-output errors of linear filters as indicators of the implementation deterioration in fixed-point context. *EURASIP Journal on Advances in Signal Processing*, 2011(1):893760, 2011.
64. T. Hilaire, P. Chevrel, and J. Whidborne. A unifying framework for finite wordlength realizations. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 54(8):1765–1774, 2007.
65. T. Hilaire, D. Menard, and O. Sentieys. Bit Accurate Roundoff Noise Analysis of Fixed-point Linear Controllers. In *IEEE International Conference on Computer-Aided Control Systems (CACSD)*, pages 607–612, September 2008.
66. T. Hinamoto, K. Iwata, and W.-S. Lu. l_2-sensitivity minimization of one- and two- dimensional state-space digital filters subject to l_2-scaling constraints. *IEEE Transactions on Signal Processing*, 54(5):1804–1812, 2006.
67. T. Hinamoto, H. Ohnishi, and W.-S. Lu. Minimization of l_2-sensitivity for state-space digital filters subject to l_2-dynamic-range scaling constraints. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 52(10):641–645, 2005.
68. T. Hinamoto, S. Yokoyama, T. Inoue, W. Zeng, and W.-S. Lu. Analysis and minimization of l_2-sensitivity for linear systems and two-dimensional state-space filters using general controllability and observability gramians. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 49(9):1279–1289, 2002.
69. L. Jackson. Roundoff noise bounds derived from coefficient sensitivities for digital filters. *IEEE Transactions on Circuits and Systems*, 23(8):481–485, 1976.
70. L. Jackson. Limit cycles in state-space structures for digital filters. *IEEE Transactions on Circuits and Systems*, 26(1):67–68, 1979.

71. L. Jackson. *Digital Filters and Signal Processing*. Kluwer Academic Publishers, Boston, 1986. by Leland B. Jackson. ill. ; 25 cm. Includes index.
72. E. Jury and B. Lee. The absolute stability of systems with many nonlinearities. *Automat. Remote Contr.*, 26:943–961, 1965.
73. J. Kang and W. Sung. Fixed-Point C Compiler for TMS320C50 Digital Signal Processor. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Munich, April 1997.
74. J. Kauraniemi. Analysis of limit cycles in the direct form delta operator structure by computer-aided test. *International Conference on Acoustics, Speech, and Signal Processing, 1997*. 3:2177–2180 vol3, 1997.
75. H. Keding. Pain Killers for the Fixed-Point Design Flow. Technical report, Synopsys, 2010.
76. H. Keding, M. Willems, M. Coors, and H. Meyr. FRIDGE: A Fixed-Point Design and Simulation Environment. In *Design, Automation and Test in Europe*, pages 429–435, Paris, France, 1998.
77. S. Kim, K.-I. Kum, and W. Sung. Fixed-point optimization utility for C and C++ based digital signal processing programs. *IEEE Transactions on Circuits and Systems II - Analog and Digital Signal Processing*, 45(11):1455 –1464, Nov 1998.
78. S. Kim and W. Sung. A Floating-point to Fixed-point Assembly program Translator for the TMS 320C25. *IEEE Transactions on Circuits and Systems*, 41(11):730–739, Nov. 1994.
79. A. Kinsman and N. Nicolici. Bit-width allocation for hardware accelerators for scientific computing using sat-modulo theory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(3):405–413, 2010.
80. A. Kinsman and N. Nicolici. Automated range and precision bit-width allocation for iterative computations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(9):1265–1278, 2011.
81. A. Kinsman and N. Nicolici. Computational vector-magnitude-based range determination for scientific abstract data types. *IEEE Transactions on Computers*, 60(11):1652–1663, 2011.
82. K. Kum, J. Kang, and W. Sung. AUTOSCALER for C: An optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Transactions on Circuits and Systems II - Analog and Digital Signal Processing*, 47(9):840–848, Sept. 2000.
83. T. Laakso, P. Diniz, I. Hartimo, and J. Macedo, T.C. Elimination of zero-input and constant-input limit cycles in single-quantizer recursive filter structures. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 39(9):638–646, 1992.
84. D.-U. Lee, A. Gaffar, R. Cheung, W. Mencer, O. Luk, and G. Constantinides. Accuracy-Guaranteed Bit-Width Optimization. *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, 25(10):1990–2000, 2006.
85. D.-U. Lee, A. Gaffar, O. Mencer, and W. Luk. Minibit: bit-width optimization via affine arithmetic. In *Design Automation Conference, 2005.*, pages 837–840, 2005.
86. D.-U. Lee and J. Villasenor. A bit-width optimization methodology for polynomial-based function evaluation. *IEEE Transactions on Computers*, 56(4):567–571, 2007.
87. A. Lepschy, G. Mian, and U. Viaro. Stability analysis of second-order direct-form digital filters with two roundoff quantizers. *IEEE Transaction on Circuits Syst.*, 33(8):824–826, 1986.
88. G. Li, M. Gevers, and Y. Sun. Performance analysis of a new structure for digital filter implementation. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 47(4):474–482, 2000.
89. G. Li and Z. Zhao. On the generalized dfiit structure and its state-space realization in digital filter implementation. *IEEE Transaction on Circuits and Systems I: Regular Papers*, 51(4):769–778, 2004.
90. J. Lopez. *Evaluacion de los Efectos de Cuantificacion en las Estructuras de Filtros Digitales Utilizando Tecnicas de Cuantificacion Basadas en Extensiones de Intervalos*. PhD thesis, Univ. Politecnica de Madrid, Madrid, 2004.
91. J. Lopez, G. Caffarena, and C. Carreras. Fast and accurate computation of the l_2-sensitivity in digital filter realizations. Technical report, Univ. Politecnica de Madrid, 2006.

92. J. Lopez, G. Caffarena, C. Carreras, and O. Nieto-Taladriz. Analysis of limit cycles by means of affine arithmetic computer-aided tests. In *12th European Signal Processing Conference EUSIPCO'04*, pages 991–994, Vienna (Austria), 2004.
93. J. Lopez, G. Caffarena, C. Carreras, and O. Nieto-Taladriz. Fast and accurate computation of the roundoff noise of linear time-invariant systems. *IET Circuits, Devices and Systems*, 2(4):393–408, August 2008.
94. J. Lopez, C. Carreras, G. Caffarena, and O. Nieto-Taladriz. Fast characterization of the noise bounds derived from coefficient and signal quantization. In *International Symposium on Circuits and Systems (ISCAS '03).*, volume 4, pages IV–309–IV–312 vol4, 2003.
95. J. A. Lopez, C. Carreras, and O. Nieto-Taladriz. Improved interval-based characterization of fixed-point lti systems with feedback loops. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(11):1923–1933, 2007.
96. Mathworks. *Fixed-Point Blockset User's Guide (ver. 2.0)*, 2001.
97. J. McClellan, C. Burrus, A. Oppenheim, T. Parks, R. Schafer, and H. Schuessler. *Computer-Based Exercises for Signal Processing Using Matlab 5*. Matlab Curriculum Series. Prentice Hall, New Jersey, 1998.
98. D. Menard, D. Novo, R. Rocher, F. Catthoor, and O. Sentieys. Quantization Mode Opportunities in Fixed-Point System Design. In *European Signal Processing Conference (EUSIPCO)*, pages 542–546, Aalborg, August 2010.
99. D. Menard, R. Rocher, P. Scalart, and O. Sentieys. SQNR Determination in Non-Linear and Non-Recursive Fixed-Point Systems. In *European Signal Processing Conference*, pages 1349–1352, 2004.
100. D. Menard, R. Rocher, and O. Sentieys. Analytical Fixed-Point Accuracy Evaluation in Linear Time-Invariant Systems. *IEEE Transactions on Circuits and Systems I: Regular Papers,*, 55(1), November 2008.
101. D. Menard and O. Sentieys. A methodology for evaluating the precision of fixed-point systems. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Orlando, May 2002.
102. D. Menard and O. Sentieys. Automatic Evaluation of the Accuracy of Fixed-point Algorithms. In *Design, Automation and Test in Europe (DATE)*, Paris, march 2002.
103. D. Menard, R. Serizel, R. Rocher, and O. Sentieys. Accuracy Constraint Determination in Fixed-Point System Design. *EURASIP Journal on Embedded Systems*, 2008:12, 2008.
104. Mentor Graphics. *Algorithmic C Data Types*. Mentor Graphics, v.1.3 edition, march 2008.
105. W. Mills, C. Mullis, and R. Roberts. Digital filter realizations without overflow oscillations. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(4):334–338, 1978.
106. S. K. Mitra. *Digital signal processing laboratory using MATLAB*. WCB/McGraw-Hill, Boston, 1999. Sanjit K. Kumar. ill. ; 24 cm. + 1 computer disk. System requirements for computer disk: IBM pc or compatible, or Macintosh power pc; Windows 3.11 or higher; MATLAB Version 5.2 or higher; Signal Processing Toolbox Version 4.2 or higher.
107. S. Mittal. A survey of techniques for approximate computing. *ACM Computer Survey*, 48(4):62:1–62:33, Mar. 2016.
108. A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee. Precision and error analysis of matlab applications during automated hardware synthesis for fpgas. In *Design, Automation and Test in Europe, 2001*, pages 722–728, 2001.
109. D. Novo, N. Farahpour, U. Ahmad, F. Catthoor, and P. Ienne. Energy efficient mimo processing: A case study of opportunistic run-time approximations. In *Design, automation and test in Europe*, pages 1–6. ACM, 2014.
110. A. V. Oppenheim and R. W. Schafer. *Discrete-Time Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ, 1987.
111. W. G. Osborne, J. Coutinho, R. C. C. Cheung, W. Luk, and O. Mencer. Instrumented multi-stage word-length optimization. In *International Conference on Field-Programmable Technology, 2007. ICFPT 2007*, pages 89–96, 2007.
112. Y. Pang, K. Radecka, and Z. Zilic. Optimization of imprecise circuits represented by taylor series and real-valued polynomials. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(8):1177–1190, 2010.

113. K. Parashar, D. Menard, R. Rocher, O. Sentieys, D. Novo, and F. Catthoor. Fast Performance Evaluation of Fixed-Point Systems with Un-Smooth Operators. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Jose, 11 2010.
114. K. Parashar, D. Menard, and O. Sentieys. Accelerated performance evaluation of fixed-point systems with un-smooth operations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(4):599–612, April 2014.
115. K. Parashar, R. Rocher, D. Menard, and O. Sentieys. Analytical Approach for Analyzing Quantization Noise Effects on Decision Operators. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 1554–1557, Dallas, march 2010.
116. K. K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. Wiley, New York, 1999. Keshab K. Parhi. ill. ; 25 cm. "A Wiley-Interscience publication.".
117. S. Parker and P. Girard. Correlated noise due to roundoff in fixed point digital filters. *IEEE Transactions on Circuits and Systems*, 23(4):204–211, 1976.
118. K. Premaratne, E. Kulasekere, P. Bauer, and L.-J. Leclerc. An exhaustive search algorithm for checking limit cycle behavior of digital filters. *IEEE Transactions on Signal Processing*, 44(10):2405–2412, 1996.
119. R. A. Roberts and C. T. Mullis. *Digital Signal Processing*. Addison-Wesley series in electrical engineering. Addison-Wesley, Reading, Mass., 1987. Richard A. Roberts, Clifford T. Mullis. ill. ; 24 cm. Includes index.
120. R. Rocher, D. Menard, N. Herve, and O. Sentieys. Fixed-Point Configurable Hardware Components. *EURASIP Journal on Embedded Systems*, 2006:Article ID 23197, 13 pages, 2006.
121. R. Rocher, D. Menard, P. Scalart, and O. Sentieys. Analytical accuracy evaluation of Fixed-Point Systems. In *European Signal Processing Conference (EUSIPCO)*, Poznan, September 2007.
122. R. Rocher, D. Menard, P. Scalart, and O. Sentieys. Analytical approach for numerical accuracy estimation of fixed-point systems based on smooth operations. *IEEE Transactions on Circuits and Systems I: Regular Papers*, PP(99):1 –14, 2012.
123. R. Rocher and P. Scalart. Noise probability density function in fixed-point systems based on smooth operators. In *Conference on Design and Architectures for Signal and Image Processing (DASIP 2012)*, pages 1–8, Oct. 2012.
124. O. Sarbishei and K. Radecka. On the fixed-point accuracy analysis and optimization of polynomial specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):831–844, 2013.
125. O. Sarbishei, K. Radecka, and Z. Zilic. Analytical optimization of bit-widths in fixed-point lti systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(3):343–355, 2012.
126. C. Shi and R. Brodersen. A Perturbation Theory on Statistical Quantization Effects in Fixed-Point DSP with Non-Stationary Inputs. In *IEEE Int. Conf. on Circuits and Systems*, volume 3, pages 373–376 Vol.3, 2004.
127. C. Shi and R. Brodersen. Floating-point to fixed-point conversion with decision errors due to quantization. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Montreal, May 2004.
128. V. Singh. An extension to jury-lee criterion for the stability analysis of fixed point digital filters designed with two's complement arithmetic. *IEEE Transactions on Circuits and Systems*, 33(3):355, 1986.
129. A. Sripad and D. L. Snyder. A Necessary and Sufficient Condition for Quantization Error to be Uniform and White. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 25(5):442–448, October 1977.
130. M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *SIGPLAN conference on Programming Language Design and Implementation*, pages 108–120, 2000.
131. J. Stolfi and L. d. Figueiredo. Self-validated numerical methods and applications. In *21st Brazilian Mathematics Colloquium, IMPA*, Rio de Janeiro, Brazil, 1997.

132. W. Sung. Optimization of number representations. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*. Springer, third edition, 2018.
133. V. Tavsanoglu and L. Thiele. Optimal design of state - space digital filters by simultaneous minimization of sensitivity and roundoff noise. *IEEE Transactions on Circuits and Systems*, 31(10):884–888, 1984.
134. L. Thiele. Design of sensitivity and round-off noise optimal state-space discrete systems. *Int. J. Circuit Theory Appl.*, 12:39–46, 1984.
135. L. Thiele. On the sensitivity of linear state-space systems. *IEEE Transactions on Circuits and Systems*, 33(5):502–510, 1986.
136. K. Uesaka and M. Kawamata. Synthesis of low coefficient sensitivity digital filters using genetic programming. In *IEEE International Symposium on Circuits and Systems, ISCAS '99*, volume 3, pages 307–310 vol3, 1999.
137. K. Uesaka and M. Kawamata. Heuristic synthesis of low coefficient sensitivity second-order digital filters using genetic programming. *IEEE Proceedings Circuits, Devices and Systems*, 148(3):121–125, 2001.
138. K. Uesaka and M. Kawamata. Evolutionary synthesis of digital filter structures using genetic programming. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 50(12):977–983, 2003.
139. P. Vaidyanathan and V. Liu. An improved sufficient condition for absence of limit cycles in digital filters. *IEEE Transactions on Circuits and Systems*, 34(3):319–322, 1987.
140. S. Wadekar and A. Parker. Accuracy sensitive word-length selection for algorithm optimization. In *International Conference on Computer Design: VLSI in Computers and Processors, 1998*, pages 54–61, 1998.
141. B. Widrow. Statistical Analysis of Amplitude Quantized Sampled-Data Systems. *Transaction on AIEE, Part. II: Applications and Industry*, 79:555–568, 1960.
142. B. Widrow, I. Kollar, and M.-C. Liu. Statistical theory of quantization. *IEEE Transactions on Instrumentation and Measurement*, 45(2):353–361, 1996.
143. M. Willems. *A Methodology for the Efficient Design of Fixed-Point Systems*. PhD thesis, Aachen University of Technology, German, 1998.
144. N. Wong and T.-S. Ng. A generalized direct-form delta operator-based iir filter with minimum noise gain and sensitivity. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 48(4):425–431, 2001.
145. B. Wu, J. Zhu, and F. Najm. An analytical approach for dynamic range estimation. In *ACM/IEEE Design Automation Conference (DAC)*, pages 472–477, San Diego, june 2004.
146. B. Wu, J. Zhu, and F. Najm. Dynamic range estimation for nonlinear systems. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 660–667, 2004.
147. C. Xiao. Improved l_2-sensitivity for state-space digital system. *IEEE Transactions on Signal Processing*, 45(4):837–840, 1997.
148. Z. Zhao and G. Li. Roundoff noise analysis of two efficient digital filter structures. *IEEE Transactions on Signal Processing*, 54(2):790–795, 2006.

Models of Architecture for DSP Systems

Maxime Pelcat

Abstract Over the last decades, the practice of representing digital signal processing applications with formal Model of computations (MoCs) has developed. Formal MoCs are used to study application properties (liveness, schedulability, parallelism...) at a high level, often before implementation details are known. Formal MoCs also serve as an input for Design Space Exploration (DSE) that evaluates the consequences of software and hardware decisions on the final system. The development of formal MoCs is fostered by the design of increasingly complex applications requiring early estimates on a system's functional behavior.

On the architectural side of digital signal processing system development, heterogeneous systems are becoming ever more complex. Languages and models exist to formalize performance-related information of a hardware system. They most of the time represent the topology of the system in terms of interconnected components and focus on time performance. However, the body of work on what we will call MoAs in this chapter is much more limited and less neatly delineated than the one on MoCs. This chapter proposes and argues a definition for the concept of an MoA and gives an overview of architecture models and languages that draw near the MoA concept.

1 Introduction

In computer science, system performance is often used as a synonym for real-time performance, i.e. adequate processing speed. However, most DSP systems must, to fit their market, be efficient in many of their aspects and meet at the same time

M. Pelcat (✉)
Institut Pascal, Aubière, France
IETR/INSA, Rennes, France
e-mail: mpelcat@insa-rennes.fr

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_30

several efficiency constraints, including high performance, low cost, and low power consumption. These systems are referred to as high performance embedded systems [38] and include for instance medical image processing systems [34], wireless transceivers [32], and video compression systems [6].

The holistic optimisation of a system in its different aspects is called Design Space Exploration (DSE) [31]. Exploring the design space consists in creating a Pareto chart such as the one in Fig. 1 and choosing solutions on the Pareto front, i.e. solutions that represent the best alternative in at least one dimension and respect constraints in the other dimensions. As an example, p_1 on Fig. 1 can be energy consumption and p_2 can be response time. Figure 1 illustrates in two dimensions a problem that, in general, has many more dimensions. In order to make system-level design efficient, separation of concerns is desirable [16]. Separation of concerns refers to forcing decisions on different design concerns to be (nearly) independent. The separation of concerns between application and architecture design makes it possible to generate many points for the Pareto by varying separately application and architecture parameters and observing their effects on system efficiency.

For example, the designer can build an application, test its efficiency on different platform architectures and, if constraints are not met by any point on the Pareto, iterate the process until reaching a satisfactory efficiency. This process is illustrated on Fig. 2 and leads to Pareto points in Fig. 1. Taking the hypothesis that a unique constraint is set on the maximum m_{p1} of property p_1, the first six generated systems in Fig. 2 led to $p_1 > m_{p_1}$ (corresponding to points over the dotted line in Fig. 1) and different values of p_2, p_3, etc. The seventh generated system has $p_1 \leq m_{p1}$ and thus respects the constraint. Further system generations can be performed to optimize p_2, p_3, etc. and generate the points under the dotted line in Fig. 1. Such a design effort is feasible only if application and architecture can be played with efficiently. On the application side, this is possible using Model of computations (MoCs) that represent the high-level aspects (e.g. parallelism, exchanged data, triggering events...) of an application while hiding its detailed implementation. Equivalently on the architectural side, Models of Architecture (MoAs) can be used to extract the fundamental elements affecting efficiency while ignoring the details of circuitry.

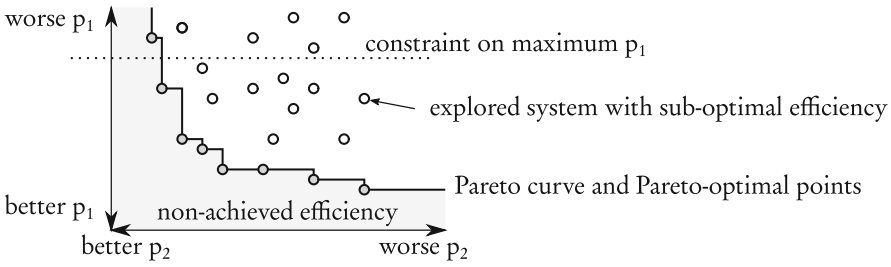


Fig. 1 The problem of Design Space Exploration (DSE) illustrated on a 2-D Pareto chart with efficiency metrics p_1 and p_2

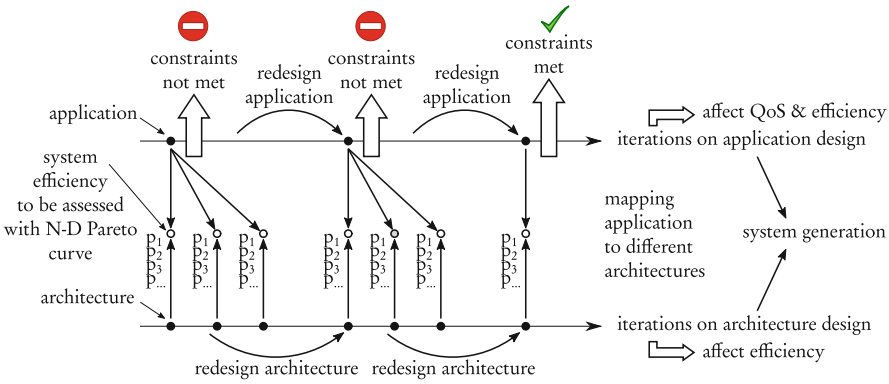


Fig. 2 Example of an iterative design process where application is refined and, for each refinement step, tested with a set of architectures to generate new points for the Pareto chart

This chapter aims at reviewing languages and tools for modeling architectures and precisely defining the scope and capabilities of MoAs.

The chapter is organised as follows. The context of MoAs is first explained in Sect. 2. Then, definitions of an MoA and a quasi-MoA are argued in Sect. 3. Sections 4 and 5 give examples of state of the art quasi-MoAs. Finally, Sect. 6 concludes this chapter.

2 The Context of Models of Architecture

2.1 Models of Architecture in the Y-Chart Approach

The main motivation for developing Models of Architecture is for them to formalize the specification of an architecture in a Y-chart approach of system design. The Y-chart approach, introduced in [18] and detailed in [2], consists in separating in two independent models the application-related and architecture-related concerns of a system's design.

This concept is refined in Fig. 3 where a set of applications is mapped to a set of architectures to obtain a set of efficiency metrics. In Fig. 3, the application model is required to conform to a specified MoC and the architecture model is required to conform to a specified MoA. This approach aims at separating *What* is implemented from *How* it is implemented. In this context, the application is qualified by a *Quality of Service (QoS)* and the architecture, offering resources to this application, is characterized by a given *efficiency* when supporting the application. For the discussion not to remain abstract, next section illustrates the problem on an example.

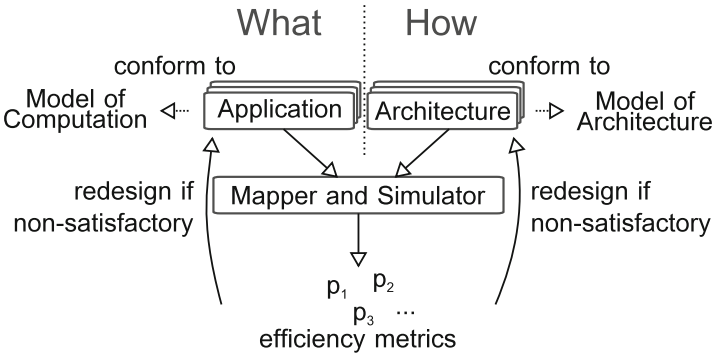


Fig. 3 MoC and MoA in the Y-chart [18]

2.2 *Illustrating Iterative Design Process and Y-Chart on an Example System*

QoS and efficiency metrics are multi-dimensional and can take many forms. For a signal processing application, QoS may be the Signal-to-Noise Ratio (SNR) or the Bit Error Rate (BER) of a transmission system, the compression rate of an encoding application, the detection precision of a radar, etc. In terms of architectural decisions, the obtained set of efficiency metrics is composed of some of the following Non-Functional Properties (NFPs):

- over time:
  - *latency* (also called response time) corresponds to the time duration between the arrival time of data to process and the production time of processed data,
  - *throughput* is the amount of processed data per time interval,
  - *jitter* is the difference between maximal and minimal latency over time,
- over energy consumption:
  - *energy* corresponds to the energy consumed to process an amount of data,
  - *peak power* is the maximal instantaneous power required on alimentation to process data,
  - *temperature* is the effect of dissipated heat from processing,
- over memory:
  - *Random Access Memory (RAM) requirements* corresponds to the amount of necessary read-write memory to support processing,
  - *Read-Only Memory (ROM) requirements* is the amount of necessary read-only memory to support processing,

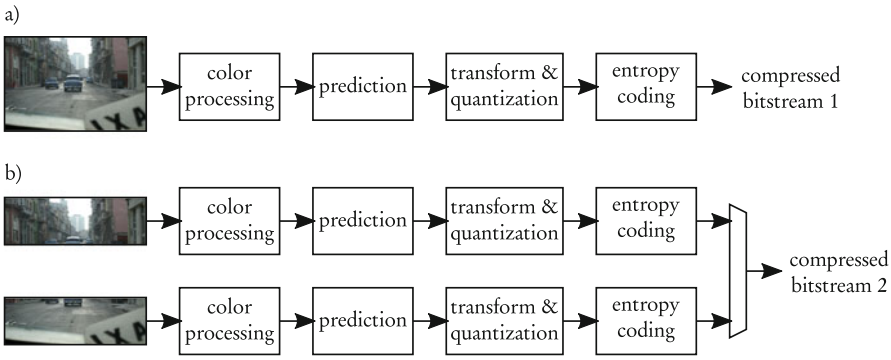


Fig. 4 Illustrating designer's freedom on the application side with a video compression example. (**a**) Original video compression application. (**b**) Redesigned video compression application forcing data parallelism

- over security:
  - *reliability* is $1 - p_f$ with p_f the probability of system failure over time,
  - *electromagnetic interference* corresponds to the amount of non-desired emitted radiations,
- over space:
  - *area* is the total surface of semiconductor required for a given processing,
  - *volume* corresponds to the total volume of the built system.
  - *weight* corresponds to the total weight of the built system.
- and *cost* corresponds to the monetary cost of building one system unit under the assumption of a number of produced units.

The high complexity of automating system design with a Y-chart approach comes from the extensive freedom (and imagination) of engineers in redesigning both application and architecture to fit the efficiency metrics, among this list, falling into their applicative constraints. Figure 4 is an illustrating example of this freedom on the application side. Let us consider a video compression system, to be ported on a platform. As shown in Fig. 4a, the application initially has only pipeline parallelism. Assuming that all four tasks are equivalent in complexity and that they receive and send at once a full image as a message, pipelining can be used to map the application to a multicore processor with four cores, with the objective to rise throughput (in frames per second) when compared to a monocore execution. However, latency will not be reduced because data will have to traverse all tasks before being output. In Fig. 4b, the image has been split into two halves and each half is processed independently. The application QoS in this second case will be lower, as the redundancy between image halves is not used for compression. The compression rate or image quality will thus be degraded. However, by accepting QoS reduction, the designer has created data parallelism that offers new opportunities for latency reduction, as processing an image half will be faster than processing a whole image.

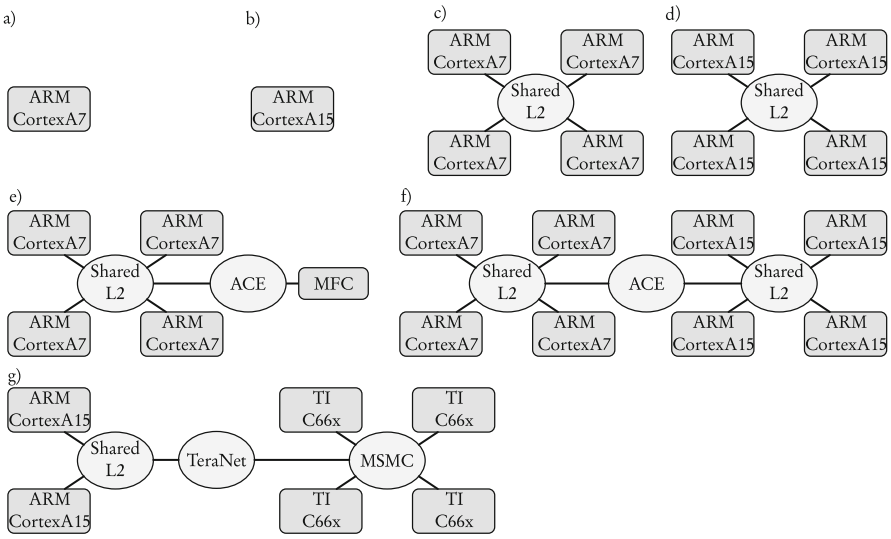


Fig. 5 Illustrating designer's freedom on the architecture side with some current ARM-based and Digital Signal Processor-based multi-core architectures. (**a**) Monocore energy-efficient. (**b**) Monocore high-performance. (**c**) Quad-core energy-efficient. (**d**) Quad-core high-performance. (**e**) Quad-core energy-efficient + accelerator. (**f**) Octo-core big.LITTLE. (**g**) multi-ARM + multi-DSP processor from Texas Instruments

In terms of architecture, and depending on money and design time resources, the designer may choose to run some tasks in hardware and some in software over processors. He can also choose between different hardware interconnects to connect these architecture components. For illustrative purpose, Fig. 5 shows different configurations of processors that could run the applications of Fig. 4. rounded rectangles represent Processing Elements (PEs) performing computation while ovals represent Communication Nodes (CNs) performing inter-PE communication. Different combinations of processors are displayed, leveraging on high-performance out-of-order ARM Cortex-A15 cores, on high-efficiency in-order ARM Cortex-A7 cores, on the Multi-Format Codec (MFC) hardware accelerator for video encoding and decoding, or on Texas Instruments C66x Digital Signal Processing cores. Figure 5g corresponds to a 66AK2L06 Multicore DSP+ARM KeyStone II processor from Texas Instruments where ARM Cortex-A15 cores are combined with C66x cores connected with a Multicore Shared Memory Controller (MSMC) [36]. In these examples, all PEs of a given type communicate via shared memory with either hardware cache coherency (*Shared L2*) or software cache coherency (*MSMC*), and with each other using either the Texas Instruments *TeraNet* switch fabric or the ARM AXI Coherency Extensions (ACE) with hardware cache coherency [35].

Each architecture configuration and each *mapping* and *scheduling* of the application onto the architecture leads to different efficiencies in all the previously listed NFPs. Considering only one mapping per application-architecture couple, models

from Figs. 4 and 5 already define $2 \times 7 = 14$ systems. Adding mapping choices of tasks to PEs, and considering that they all can execute any of the tasks and ignoring the order of task executions, the number of possible system efficiency points in the Pareto Chart is already roughly 19,000,000. This example shows how, by modeling application and architecture independently, a large number of potential systems is generated which makes automated multi-dimensional DSE necessary to fully explore the design space.

2.3 *On the Separation Between Application and Architecture Concerns*

Separation between application and architectural concerns should not be confused with software (SW)/hardware (HW) separation of concerns. The software/hardware separation of concerns is often put forward in the term *HW/SW co-design*. Software and its languages are not necessarily architecture-agnostic representations of an application and may integrate architecture-oriented features if the performance is at stake. This is shown for instance by the differences existing between the C++ and CUDA languages. While C++ builds an imperative, object-oriented code for a processor with a rather centralized instruction decoding and execution, CUDA is tailored to GPGPUs with a large set of cores. As a rule of thumb, software qualifies what may be reconfigured in a system while hardware qualifies the static part of the system.

The separation between application and architecture is very different in the sense that the application may be transformed into software processes and threads, as well as into hardware Intellectual Property cores (IPs). Software and Hardware application parts may collaborate for a common applicative goal. In the context of DSP, this goal is to transform, record, detect or synthetize a signal with a given QoS. MoCs follow the objective of making an application model agnostic of the architectural choices and of the HW/SW separation. The architecture concern relates to the set of hardware and software support features that are not specific to the DSP process, but create the resources handling the application.

On the application side, many MoCs have been designed to represent the behavior of a system. The Ptolemy II project [7] has a considerable influence in promoting MoCs with precise semantics. Different families of MoCs exist such as finite state machines, process networks, Petri nets, synchronous MoCs and functional MoCs. This chapter defines MoAs as the architectural counterparts of MoCs and presents a state-of-the-art on architecture modeling for DSP systems.

2.4 Scope of This Chapter

In this chapter, we focus on architecture modeling for the performance estimation of a DSP application over a complex distributed execution platform. We keep functional testing of a system out of the scope of the chapter and rather discuss the early evaluation of system non-functional properties. As a consequence, virtual platforms such as QEMU [3], gem5 [4] or Open Virtual Platforms simulator (OVPsim), that have been created as functional emulators to validate software when silicon is not available, will not be discussed. MoAs work at a higher level of abstraction where functional simulation is not central.

The considered systems being dedicated to digital signal processing, the study concentrates on signal-dominated systems where control is limited and provided together with data. Such systems are called *transformational*, as opposed to *reactive* systems that can, at any time, react to non-data-carrying events by executing tasks.

Finally, the focus is put on system-level models and design rather than on detailed hardware design, already addressed by large sets of existing literature. Next section introduces the concept of an MoA, as well as an MoA example named Linear System-Level Architecture Model (LSLA).

3 The Model of Architecture Concept

The concept of MoA is evoked in 2002 in [19] where it is defined as "a formal representation of the operational semantics of networks of functional blocks describing architectures". This definition is broad, and allows the concepts of MoC and MoA to overlap. As an example, a Synchronous Dataflow (SDF) graph [14, 24] representing a system fully specialized to an application may be considered as a MoC, because it formalizes the application. It may also be considered as an MoA because it fully complies with the definition from [19]. Definition 4 of this chapter, adapted from [30], is a new definition of an MoA that does not overlap with the concept of MoC. The LSLA model is then presented to clarify the concept by an example.

3.1 Definition of an MoA

Prior to defining MoA, the notion of application activity is introduced that ensures the separation of MoC and MoA. Figure 6 illustrates how application activity provides intermediation between application and architecture. Application activity models the computational load handled by the architecture when executing the application.

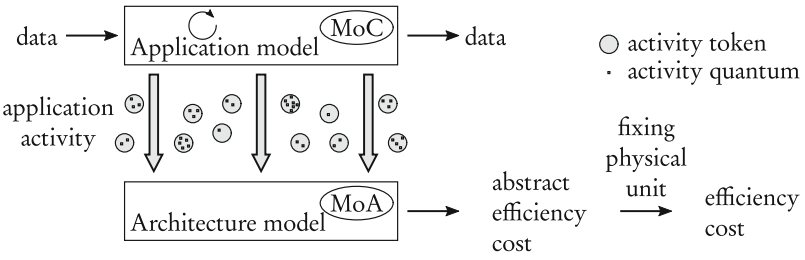


Fig. 6 Application activity as an intermediate model between application and architecture

Definition 1 Application activity A corresponds to the amount of processing and communication necessary for accomplishing the requirements of the considered application during the considered time slot. Application activity is composed of processing and communication *tokens*, themselves composed of *quanta*.

Definition 2 A quantum q is the smallest unit of application activity. There are two types of quanta: processing quantum q_P and communication quantum q_C.

Two distinct processing quanta are equivalent, thus represent the same amount of activity. Processing and communication quanta do not share the same unit of measurement. As an example, in a system with a unique clock and byte-addressable memory, 1 cycle of processing can be chosen as the processing quantum and 1 byte as the communication quantum.

Definition 3 A token $\tau \in T_P \cup T_C$ is a non-divisible unit of application activity, composed of a number of quanta. The function $size : T_P \cup T_C \rightarrow \mathbb{N}$ associates to each token the number of quanta composing the token. There are two types of tokens: processing tokens $\tau_P \in T_P$ and communication tokens $\tau_C \in T_C$.

The activity $\mathscr{A}$ of an application is composed of the set:

$$\mathscr{A} = \{T_P, T_C\} \tag{1}$$

where $T_P = \{\tau_P^1, \tau_P^2, \tau_P^3 \ldots\}$ is the set of processing tokens composing the application processing and $T_C = \{\tau_C^1, \tau_C^2, \tau_C^3 \ldots\}$ is the set of communication tokens composing the application communication.

An example of a processing token is a run-to-completion task with always identical computation. All tokens representing the execution of this task enclose the same number N of processing quanta (e.g. N cycles). An example of a communication token is a message in a message-passing system. The token is then composed of M communication quanta (e.g. M Bytes). Using the two levels of granularity of a token and a quantum, an MoA can reflect the cost of managing a quantum, and the additional cost of managing a token composed of several quanta.

Definition 4 A Model of Architecture (MoA) is an abstract efficiency model of a system architecture that provides a unique, reproducible cost computation, unequivocally assessing an architecture efficiency cost when supporting the *activity* of an application described with a specified MoC.

This definition makes three aspects fundamental for an MoA:

- *reproducibility*: using twice the same MoC and activity computation with a given MoA, system simulation should return the exact same efficiency cost,
- *application independence*: the MoC alone carries application information and the MoA should not comprise application-related information such as the exchanged data formats, the task representations, the input data or the considered time slot for application observation. *Application activity* is an intermediate model between a MoC and an MoA that prevents both models to intertwine. An *application activity* model reflects the computational load to be handled by architecture and should be versatile enough to support a large set of MoCs and MoAs, as demonstrated in [30].
- *abstraction*: a system efficiency cost, as returned by an MoA, is not bound to a physical unit. The physical unit is associated to an efficiency cost outside the scope of the MoA. This is necessary not to redefine the same model again and again for energy, area, weight, etc.

Definition 4 does not compel an MoA to match the internal structure of the hardware architecture, as long as the generated cost is of interest. An MoA for energy modeling can for instance be a set of algebraic equations relating application activity to the energy consumption of a platform. To keep a reasonably large scope, this chapter concentrates on graphical MoAs defined hereafter:

Definition 5 A graphical MoA is an MoA that represents an architecture with a graph $\Lambda = \langle M, L, t, p \rangle$ where M is a set of "black-box" components and $L \subseteq M \times M$ is a set of links between these components.

The graph Λ is associated with two functions t and p. The *type* function $t : M \times L \mapsto T$ associates a type $t \in T$ to each component and to each link. The type dedicates a component for a given service. The *properties* function $p : M \times L \times \Lambda \mapsto \mathrm{P}(\Pi)$, where P represents powerset, gives a set of properties $p_i \in P$ to each component, link, and to the graph Λ itself. Properties are features that relate application activity to implementation efficiency.

When the concept of MoA is evoked throughout this chapter, a graphical MoA is supposed, respecting Definition 5. When a model of a system architecture is

evoked that only partially compels with this definition, the term *quasi-MoA* is used, equivalent to *quasi-moa* in [30] and defined hereafter:

Definition 6 A quasi-MoA is a model respecting some of the aspects of Definition 4 of an MoA but violating at least one of the three fundamental aspects of an MoA, i.e. *reproducibility*, *application independence*, and *abstraction*.

All state-of-the-art languages and models presented in Sects. 4 and 5 define quasi-MoAs. As an example of a graphical quasi-MoAs, the graphical representation used in Fig. 5 shows graphs $\Lambda = \langle M, L \rangle$ with two types of components (PE and CN), and one type of undirected link. However, no information is given on how to compute a cost when associating this representation with an application representation. As a consequence, *reproducibility* is violated. Next section illustrates the concept of MoA through the LSLA example.

3.2 Example of an MoA: The Linear System-Level Architecture Model (LSLA)

The LSLA model computes an additive reproducible cost from a minimalistic representation of an architecture [30]. As a consequence, LSLA fully complies with Definition 5 of a graphical MoA. The LSLA composing elements are illustrated in Fig. 7. An LSLA model specifies two types of components: Processing Elements and Communication Nodes, and one type of link. LSLA is categorized as linear because the computed cost is a linear combination of the costs of its components.

Definition 7 The Linear System-Level Architecture Model (LSLA) is a Model of Architecture (MoA) that consists of an undirected graph $\Lambda = (P, C, L, cost, \lambda)$ where:

- P is a set of Processing Elements (PEs). A PE is an abstract processing facility with no assumption on internal parallelism, Instruction Set Architecture (ISA), or internal memory. A processing token τ_P from application activity must be mapped to a PE $p \in P$ to be executed.

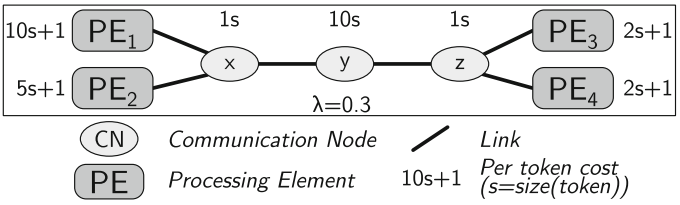


Fig. 7 LSLA MoA semantics elements

- C is the set of architecture Communication Nodes (CNs). A communication token τ_C must be mapped to a CN $c \in C$ to be executed.
- $L = \{(n_i, n_j) | n_i \in C, n_j \in C \cup P\}$ is a set of undirected links connecting either two CNs or one CN and one PE. A link models the capacity of a CN to communicate tokens to/from a PE or to/from another CN.
- $cost$ is a property function associating a cost to different elements in the model. The cost unit is specific to the non-functional property being modeled. It may be in mJ for studying energy or in mm^2 for studying area. Formally, the generic unit is denoted ν.

On the example displayed in Fig. 7, PE_{1-4} represent Processing Elements (PEs) while x, y and z are Communication Nodes (CNs). As an MoA, LSLA provides reproducible cost computation when the activity A of an application is *mapped* onto the architecture. The cost related to the management of a token τ by a PE or a CN n is defined by:

$$\begin{aligned} cost : T_P \cup T_C \times P \cup C &\rightarrow \mathbb{R} \\ \tau, n &\mapsto \alpha_n . size(\tau) + \beta_n, \\ \alpha_n \in \mathbb{R}, \beta_n \in \mathbb{R} & \end{aligned} \tag{2}$$

where α_n is the fixed cost of a quantum when executed on n and β_n is the fixed overhead of a token when executed on n. For example, in an energy modeling use case, α_n and β_n are respectively expressed in $energy/quantum$ and $energy/token$, as the cost unit ν represents energy. A token communicated between two PEs connected with a chain of CNs $\Gamma = \{x, y, z \ldots\}$ is reproduced $card(\Gamma)$ times and each occurrence of the token is mapped to 1 element of Γ. This procedure is illustrated in Fig. 8. In figures representing LSLA architectures, the size of a token $size(\tau)$ is abbreviated into s and the affine equations near CNs and PEs (e.g. $10s+1$) represent the cost computation related to Eq. (2) with $\alpha_n = 10$ and $\beta_n = 1$.

A token not communicated between two PEs, i.e. internal to one PE, does not cause any cost. The cost of the execution of application activity A on an LSLA graph Λ is defined as:

$$\begin{aligned} cost(\mathrm{A}, \Lambda) = &\sum_{\tau \in T_P} cost(\tau, map(\tau)) + \\ &\lambda \sum_{\tau \in T_C} cost(\tau, map(\tau)) \end{aligned} \tag{3}$$

where $map : T_P \cup T_C \rightarrow P \cup C$ is a surjective function returning the mapping of each token onto one of the architecture elements.

- $\lambda \in \mathbb{R}$ is a Lagrangian coefficient setting the Computation to Communication Cost Ratio (CCCR), i.e. the cost of a single communication quantum relative to the cost of a single processing quantum.

Similarly to the SDF MoC [24], the LSLA MoA does not specify relations to the outside world. There is no specific PEs type for communicating with non-modeled parts of the system. This is in contrast with Architecture Analysis and Design

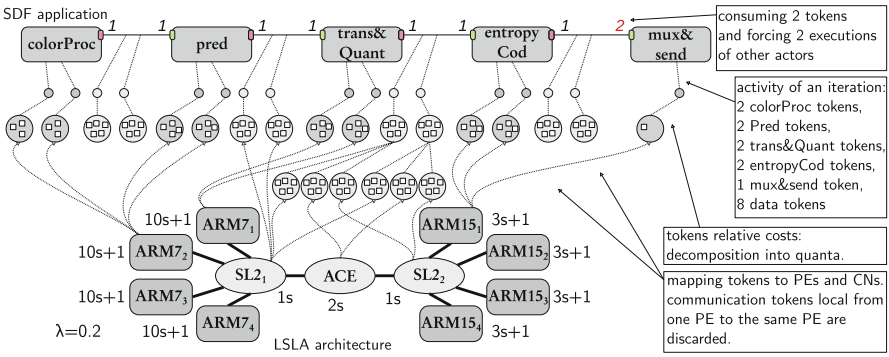


Fig. 8 Computing cost of executing an SDF graph on an LSLA architecture. The cost for 1 iteration is (looking first at processing tokens then at communication tokens from left to right) $31 + 31 + 41 + 41 + 41 + 41 + 13 + 13 + 4 + 0.2 \times (5 + 5 + 5 + 10 + 5 + 5 + 10 + 5) = 266$ ν (Eq. (3))

Language (AADL) `processors` and `devices` that separate I/O components from processing components (Sect. 4.1). The Definition 1 of activity is sufficient to support LSLA and other types of additive MoAs. Different forms of activities are likely to be necessary to define future MoAs. Activity Definition 1 is generic to several families of MoCs, as demonstrated in [30].

Figure 8 illustrates cost computation for a mapping of the video compression application shown in Fig. 4b, described with the SDF MoC onto the big.LITTLE architecture of Fig. 5f, described with LSLA. The number of tokens, quanta and the cost parameters are not representative of a real execution but set for illustrative purpose. The natural scope for the cost computation of a couple (SDF, LSLA), provided that the SDF graph is consistent, is one SDF graph iteration [30].

The SDF application graph has five actors $colorProc$, $pred$, $trans\&Quant$, $entropyCod$, and $mux\&Send$ and the four first actors will execute twice to produce the two image halves required by $mux\&Send$. The LSLA architecture model has 8 PEs $ARMj_k$ with $j \in \{7, 15\}$ and $k \in \{1, 2, 3, 4\}$, and 3 CNs $SL2_1$, ACE and $SL2_2$. Each actor execution during the studied graph iteration is transformed into one processing token. Each dataflow token transmitted during one iteration is transformed into one communication token. A token is embedding several quanta (white squares), allowing a designer to describe heterogeneous tokens to represent executions and messages of different weight.

In Fig. 8, each execution of actors $colorProc$ is associated with a cost of 3 quanta and each execution of other actors is associated to a cost of 4 quanta except $mux\&Send$ requiring 1 quantum. Communication tokens (representing one half image transfer) are given five quanta each. These costs are arbitrary here but should represent the relative computational load of the task/communication.

Each processing token is mapped to one PE. Communication tokens are "routed" to the CNs connecting their producer and consumer PEs. For instance, the fifth and sixth communication tokens in Fig. 8 are generating three tokens each mapped to

$SL2_1$, ACE and $SL2_2$ because the data is carried from $ARM7_1$ to $ARM15_1$. It is the responsibility of the mapping process to verify that a link $l \in L$ exists between the elements that constitute a communication route. The resulting cost, computed from Eqs. (2) and (3), is 266ν. This cost is reproducible and abstract, making LSLA an MoA.

LSLA is one example of an architecture model but many such models exist in literature. Next sections study different languages and models from literature and explain the quasi-MoAs they define.

4 Architecture Design Languages and Their Architecture Models

This section studies the architecture models provided by three standard ADLs targeting architecture modeling at system-level: AADL, MCA SHIM, and UML MARTE.

While AADL adopts an abstraction/refinement approach where components are first roughly modeled, then refined to lower levels of abstraction, UML MARTE is closer to a Y-Chart approach where the application and the architecture are kept separated and application is mapped to architecture.

For its part, MCA SHIM describes an architecture with "black box" processors and communications and puts focus on inter-PE communication simulation. All these languages have in common the implicit definition of a quasi-MoA (Definition 6). Indeed, while they define parts of graphical MoAs, none of them respect the three rules of MoA Definition 4.

4.1 The AADL Quasi-MoA

Architecture Analysis and Design Language (AADL) [9] is a standard language released by SAE International, an organization issuing standards for the aerospace and automotive sectors. The AADL standard is referenced as AS5506 [33] and the last released version is 2.2. Some of the most active tools supporting AADL are Ocarina[1] [21] and OSATE[2] [9].

[1] https://github.com/OpenAADL/ocarina.

[2] https://github.com/osate.

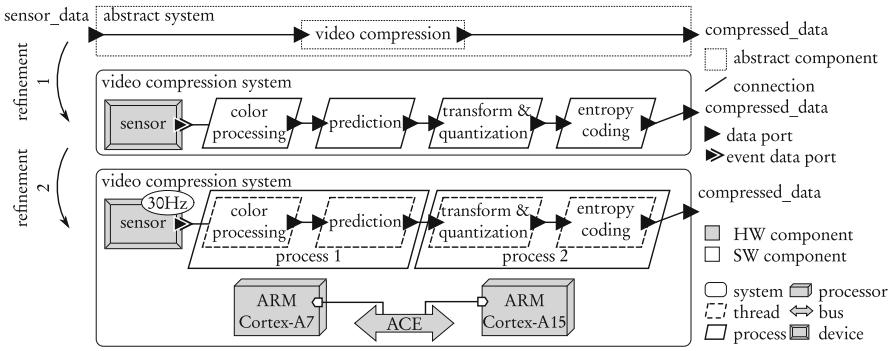


Fig. 9 The AADL successive refinement system design approach

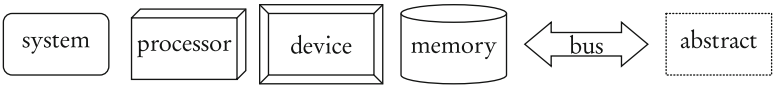


Fig. 10 The basic components for describing a hardware architecture in AADL

4.1.1 The Features of the AADL Quasi-MoA

AADL provides semantics to describe a software application, a hardware platform, and their combination to form a system. AADL can be represented graphically, serialized in XML or described in a textual language [10]. The term *architecture* in AADL is used in its broadest sense, i.e. a whole made up of clearly separated elements. A design is constructed by successive refinements, filling "black boxes" within the AADL context. Figure 9 shows two refinement steps for a video compression system in a camera. Blocks of processing are split based on the application decomposition of Fig. 4a. First, the system is abstracted with external data entering a video compression `abstract component`. Then, four software `processes` are defined for the processing. Finally, processes are transformed into four `threads`, mapped onto two processes. The platform is defined with two cores and a bus and application threads are allocated onto platform components. The allocation of threads to processors is not displayed. Sensor data is assigned a rate of 30 Hz, corresponding to 30 frames per second. Next sections detail the semantics of the displayed components.

Software, hardware and systems are described in AADL by a composition of `components`. In this chapter, we focus on the hardware platform modeling capabilities of AADL, composing an implicit graphical quasi-MoA. Partly respecting Definition 5, AADL represents platform with a graph $\Lambda = \langle M, L, t, p \rangle$ where M is a set of components, L is a set of links, t associates a type to each component and link and p gives a set of properties to each component and link. As displayed in Fig. 10, AADL defines six types of platform components with specific graphical representations. The AADL component type set is such that $t(c \in M) \in \{$`system`,

`processor`, `device`, `bus`, `memory`, `abstract`}. There is one type of link $t(l \in L) \in$ {`connection`}. A `connection` can be set between any two components among software, hardware or system. Contrary to the Y-chart approach, AADL does not separate application from architecture but makes them coexist in a single model.

AADL is an extensible language but defines some *standard component properties*. These properties participate to the definition of the quasi-MoA determined by the language and make an AADL model portable to several tools. The AADL standard set of properties targets only the time behavior of components and differs for each kind of component. AADL tools are intended to compute NFP costs such as the total minimum and maximum execution latency of an application, as well as the jitter. An AADL representation can also be used to extract an estimated bus bandwidth or a subsystem latency [20].

Processors are sequential execution facilities that must support thread scheduling, with a protocol fixed as a property. AADL platform components are not merely hardware models but rather model the combination of hardware and low-level software that provides services to the application. In that sense, the architecture model they compose is conform to MoA Definition 4. However, what is mapped on the platform is *software* rather than an *application*. As a consequence, the separation of concerns between application and architecture is not supported (Sect. 2.3). For instance, converting the service offered by a software thread to a hardware IP necessitates to deeply redesign the model. A `processor` can specify a *Clock_Period*, a *Thread_Swap_Execution_Time* and an *Assign_Time*, quantifying the time to access memory on the processor. Time properties of a processor can thus be precisely set.

A `bus` can specify a fixed *Transmission_Time* interval representing best- and worst-case times for transmitting `data`, as well as a *PerByte Transmission_Time* interval representing throughput. The time model for a message is thus an affine model w.r.t. message size. Three models for transfer cost computation are displayed in Fig. 11: linear, affine, and stair. Most models discussed in the next sections use

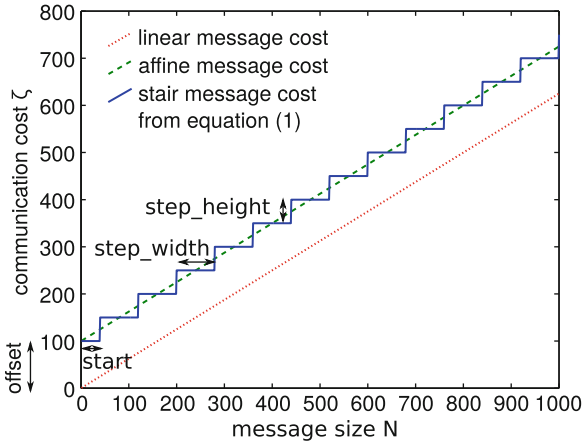


Fig. 11 Examples of different data transfer cost computation functions (in arbitrary units): a linear function (with one parameter), an affine function (with two parameters) and a step function (with four parameters)

one of these three models. The interpretation of AADL time properties is precisely defined in [9] Appendix A, making AADL time computation reproducible.

A `memory` can be associated to a *Read_Time*, a *Write_Time*, a *Word_Count* and a *Word_Size* to characterize its occupancy rate. A `device` can be associated to a *Period*, and a *Compute_Execution_Time* to study sensors' and actuators' latency and throughput. Platform components are defined to support a software application. The next section studies application and platform interactions in AADL.

4.1.2 Combining Application and Architecture in AADL

AADL aims at analyzing the time performance of a system's architecture, manually exploring the mapping (called `binding` in AADL) of software onto hardware elements. AADL quasi-MoA is influenced by the supported software model. AADL is adapted to the currently dominating software representation of Operating Systems (OS), i.e. the process and thread representation [9]. An application is decomposed into process and thread components, that are purely software concepts. A process defines an address space and a thread comes with scheduling policies and shares the address space of its owner process. A process is not executable by itself; it must contain a least one thread to execute. AADL `Threads` are sequential, preemptive entities [9] and requires scheduling by a `processor`. `Threads` may specify a *Dispatch_Protocol* or a *Period* property to model a periodic behavior or an event-triggered callback or routine.

A values or interval of *Compute_Execution_Time* can be associated to a `thread`. However, in real world, execution time for a thread firing depends on both the code to execute and the platform speed. *Compute_Execution_Time* is not related to the binding of the thread to a `processor` but a *Scaling_Factor* property can be set on the `processor` to specify its relative speed with regards to a reference processor for which `thread` timings have been set. This property is precise when all threads on a processor undergo the same *Scaling_Factor*, but this is not the case in general. For instance, if a thread compiled for the ARMv7 instruction set is first executed on an ARM Cortex-A7 and then on an ARM Cortex-A15 processor, the observed speedup depends much on the executed task. Speedups between $1.3\times$ and $4.9\times$ are reported in this context in [30].

AADL provides constructs for data message passing through *port* features and data memory-mapped communication through *require data access* features. These communications are bound to `busses` to evaluate their timings.

A *flow* is neither a completely software nor a completely hardware construct. It specifies an end-to-end flow of data between sensors and actuators for steady state and transient timing analysis. A *flow* has timing properties such as *Expected_Latency* and *Expected_Throughput* that can be verified through simulation.

4.1.3 Conclusions on the AADL Quasi-MoA

AADL specifies a graphical quasi-MoA, as it does define a graph of platform components. AADL violates the *abstraction* rule because cost properties are explicitly time and memory. It respects the *reproducibility* rule because details of timing simulations are precisely defined in the documentation. Finally, it violates the *application independence* rule because AADL does not conform to the Y-chart approach and does not separate application and architecture concerns.

AADL is a formalization of current best industrial practices in embedded system design. It provides formalization and tools to progressively refine a system from an abstract view to a software and hardware precise composition. AADL targets all kinds of systems, including transformational DSP systems managing data flows but also reactive system, reacting to sporadic *events*. The thread MoC adopted by AADL is extremely versatile to reactive and transformational systems but has shown its limits for building deterministic systems [23, 37]. By contrast, the quasi-MoAs presented in Sect. 5 are mostly dedicated to transformational systems. They are thus all used in conjunction with process network MoCs that help building reliable DSP systems. The next section studies another state-of-the-art language: MCA SHIM.

4.2 The MCA SHIM Quasi-MoA

The Software/Hardware Interface for Multicore/Manycore (SHIM) [12] is a hardware description language that aims at providing platform information to multicore software tools, e.g. compilers or runtime systems. SHIM is a standard developed by the Multicore Association (MCA). The most recent released version of SHIM is 1.0 (2015) [27]. SHIM is a more focused language than AADL, modeling the platform properties that influence software performance on multicore processors.

SHIM components provide timing estimates of a multicore software. Contrary to AADL that mostly models hard real-time systems, SHIM primarily targets best-effort multicore processing. Timing properties are expressed in clock cycles, suggesting a fully synchronous system. SHIM is built as a set of UML classes and the considered NFPs in SHIM are time and memory. Timing performances in SHIM are set by a `shim::Performance` class that characterizes three types of software activity: instruction executions for instructions expressed in the LLVM instruction set, memory accesses, and inter-core communications. LLVM [22] is used as a portable assembly code, capable of decomposing a software task into instructions that are portable to different ISAs.

SHIM does not propose a chart representation of its components. However, SHIM defines a quasi-MoA partially respecting Definition 5. A `shim::System-Configuration` object corresponds to a graph $\Lambda = \langle M, L, t, p \rangle$ where M is the set of components, L is the set of links, t associates a type to each component and link and p gives a set of properties to each component and link. A SHIM architecture description is decomposed into three main sets of elements: `Components`, `Address Spaces` and `Communications`. We group

and rename the components (referred to as "objects" in the standard) to makes them easier to compare to other approaches. SHIM defines two types of platform components. The component types $t(c \in M)$ are chosen among:

- `processor` (*shim::MasterComponent*), representing a core executing software. It internally integrates a number of `cache` memories (*shim::Cache*) and is capable of specific data access types to memory (*shim::AccessType*). A `processor` can also be used to represent a Direct Memory Access (DMA),
- `memory` (*shim::SlaveComponent*) is bound to an address space (*shim::AddressSpace*).

Links $t(l \in L)$ are used to set performance costs. They are chosen among:

- `communication` between two `processors`. It has three subtypes:
  - `fifo` (*shim::FIFOCommunication*) referring to message passing with buffering,
  - `sharedRegister` (*shim::SharedRegisterCommunication*) referring to a semaphore-protected register,
  - `event` (*shim::EventCommunication* for polling or *shim::InterruptCommunication* for interrupts) referring to inter-core synchronization without data transfer.
- `memoryAccess` between a `processor` and a `memory` (modeled as a couple *shim::MasterSlaveBinding*, *shim::Accessor*) sets timings to each type of data read/write accesses to the memory.
- `sharedMemory` between two `processors` (modeled as a triple *shim::SharedMemoryCommunication*, *shim::MasterSlaveBinding*, and *shim::Accessor*) sets timing performance to exchanging data over a shared memory,
- `InstructionExecution` (modeled as a *shim::Instruction*) between a processor and itself sets performance on instruction execution.

Links are thus carrying all the performance properties in this model. Application activity on a link l is associated to a `shim::Performance` property, decomposed into *latency* and *pitch*. *Latency* corresponds to a duration in cycles while *pitch* is the inverse (in cycles) of the throughput (in cycles^{-1}) at which a SHIM object can be managed. A latency of 4 and a pitch of 3 on a communication link, for instance, mean that the first data will take 4 cycles to pass through a link and then 1 data will be sent per 3 cycles. This choice of time representation is characteristic of the SHIM objective to model the average behavior of a system while AADL targets real-time systems. Instead of specifying time intervals $[min..max]$ like AADL, SHIM defines triplets $[min, mode, max]$ where mode is the statistical mode. As a consequence, a richer communication and execution time model can be set in SHIM. However, no information is given on how to use these performance properties present in the model. In the case of a communication over a shared memory for instance, the decision on whether to use the performance of this link or to use the performance of the shared memory data accesses, also possible to model, is left to the SHIM supporting tool.

4.2.1 Conclusions on MCA SHIM Quasi-MoA

MCA SHIM specifies a graphical quasi-MoA, as it defines a graph of platform components. SHIM violates the *abstraction* rule because cost properties are limited to time. It also violates the *reproducibility* rule because details of timing simulations are left to the interpretation of the SHIM supporting tools. Finally, it violates the *application independence* rule because SHIM supports only software, decomposed into LLVM instructions.

The modeling choices of SHIM are tailored to the precise needs of multicore tooling interoperability. The two types of tools considered as targets for the SHIM standard are Real-Time Operating Systems (RTOSs) and auto-parallelizing compilers for multicore processors. The very different objectives of SHIM and AADL have led to different quasi-MoAs. The set of components is more limited in SHIM and communication with the outside world is not specified. The communication modes between processors are also more abstract and associated to more sophisticated timing properties. The software activity in SHIM is concrete software, modeled as a set of instructions and data accesses while AADL does not go as low in terms of modeling granularity. To complement the study on a third language, the next section studies the different quasi-MoAs defined by the UML Modeling And Analysis Of Real-Time Embedded Systems (MARTE) language.

4.3 The UML MARTE Quasi-MoAs

The UML Profile for Modeling And Analysis Of Real-Time Embedded Systems (MARTE) is standardized by the Object Management Group (OMG) group. The last version is 1.1 and was released in 2011 [28]. Among the ADLs presented in this chapter, UML MARTE is the most complex one. It defines hundreds of UML classes and has been shown to support most AADL constructs [8]. MARTE is designed to coordinate the work of different engineers within a team to build a complex real-time embedded system. Several persons, expert in UML MARTE, should be able to collaborate in building the system model, annotate and analyze it, and then build an execution platform from its model. Like AADL, UML MARTE is focused on hard real-time application and architecture modeling. MARTE is divided into four *packages*, themselves divided into *clauses*. Three of these clauses define four different quasi-MoAs. These quasi-MoAs are named $QMoA^{i}_{MARTE} \mid i \in \{1, 2, 3, 4\}$ in this chapter and are located in the structure of UML MARTE clauses illustrated by the following list:

- The *MARTE Foundations* package includes:
  - the *Core Elements* clause that gathers constructs for inheritance and composition of abstract objects, as well as their invocation and communication.
  - the *Non-Functional Property (NFP)* clause that describes ways to specify non-functional constraints or values (Sect. 2.2), with a concrete type.

  - the *Time* clause, specific to the time NFP.
  - the *Generic Resource Modeling (GRM)* clause that offers constructs to model, at a high level of abstraction, both software and hardware elements. It defines a generic component named `Resource`, with `clocks` and `non-functional properties`. `Resource` is the basic element of UML MARTE models of architecture and application. The quasi-MoA $QMoA^{1}_{MARTE}$ is defined by GRM and based on `Resources`. It will be presented in Sect. 4.3.1.
  - the *Allocation Modeling* clause that relates higher-level `Resources` to lower-level `Resources`. For instance, it is used to allocate `Schedulable-Resources` (e.g. threads) to `ComputingResources` (e.g. cores).

- The *MARTE Design Model* package includes:
  - the *Generic Component Model (GCM)* clause that defines structured components, connectors and interaction ports to connect core elements.
  - the *Software Resource Modeling (SRM)* clause that details software resources.
  - the *Hardware Resource Modeling (HRM)* clause that details hardware resources and defines $QMoA^{2}_{MARTE}$ and $QMoA^{3}_{MARTE}$ (Sect. 4.3.2).
  - the *High-Level Application Modeling (HLAM)* clause that models real-time services in an OS.

- The *MARTE Analysis Model* package includes:
  - the *Generic Quantitative Analysis Modeling (GQAM)* clause that specifies methods to observe system performance during a time interval. It defines $QMoA^{4}_{MARTE}$.
  - the *Schedulability Analysis Modeling (SAM)* clause that refers to thread and process schedulability analysis. It builds over GQAM and adds scheduling-related properties to $QMoA^{4}_{MARTE}$.
  - the *Performance Analysis Modeling (PAM)* clause that performs probabilistic or deterministic time performance analysis. It also builds over GQAM.

- *MARTE Annexes* include *Repetitive Structure Modeling (RSM)* to compactly represent component networks, and the *Clock Constraint Specification Language (CCSL)* to relate clocks.

The link between application time and platform time in UML MARTE is established through clock and event relationships expressed in the CCSL language [25]. Time may represent a physical time or a logical time (i.e. a continuous repetition of events). Clocks can have causal relations (an event of clock A causes an event of clock B) or a temporal relations with type *precedence*, *coincidence*, and *exclusion*. Such a precise representation of time makes UML MARTE capable of modeling both asynchronous and synchronous distributed systems [26]. UML MARTE is capable, for instance, of modeling any kind of processor with multiple cores and independent frequency scaling on each core.

The UML MARTE `resource` composition mechanisms give the designer more freedom than AADL by dividing his system into more than two layers. For

instance, execution platform `resources` can be allocated to operating system `resources`, themselves allocated to application `resources` while AADL offers only a hardware/software separation. Multiple allocations to a single `resource` are either time multiplexed (*timeScheduling*) or distributed in space (*spatialDistribution*). Next sections explain the 4 quasi-MoAs defined by UML MARTE.

4.3.1 The UML MARTE Quasi-MoAs 1 and 4

The UML MARTE GRM clause specifies the $QMoA^{1}_{MARTE}$ quasi-MoA. It corresponds to a graph $\Lambda = \langle M, L, t, p \rangle$ where M is a set of `Resources`, L is a set of UML `Connectors` between these resources, t associates types to `Resources` and p gives sets of properties to `Resources`.

Seven types of resources are defined in GRM. Some inconsistencies between resource relations make the standard ambiguous on resource types. As an example, `CommunicationMedia` specializes `CommunicationResource` on standard p. 96 [28] while `CommunicationMedia` specializes `ProcessingResource` on standard p. 99. `SynchResource` disappears after definition and is possibly equivalent to the later `SwSynchronizationResource`. Considering the most detailed descriptions as reference, types of resources (illustrated in Fig. 12) are:

- a `Processing Resource`, associated to an abstract *speed Factor* property that can help the designer compare different `Processing Resources`. It has three subtypes: `Computing Resource` models a real or virtual PE storing and executing program code. It has no property. `Device Resource` communicates with the system environment, equivalently to an AADL device. It also has no property. `Communication Media` can represent a bus or a higher-level protocol over an interconnect. It has several properties: a mode among simplex, half-duplex, or full-duplex specifies whether the media is directed or not and the time multiplexing method for data. `Communication Media` transfers one data of *elementSize* bits per clock cycle. A *packet time* represents the time to transfer a set of elements. A *block time* represents the time before the media can transfer other packets. A *data rate* is also specified.
- a `Timing Resource` representing a clock or a timer, fixing a clock rate.
- a `Storage Resource` representing memory, associated with a unit size and number of units. Memory read and write occur in 1 clock cycle.

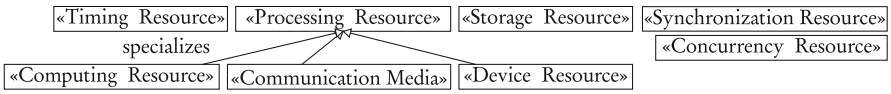


Fig. 12 Elements of the quasi-MoA define in UML MARTE Generic Resource Modeling (GRM)

- a `Concurrency Resource` representing several concurrent flows of execution. It is a generalization of `SchedulableResources` that model logical concurrency in threads and processes.

The communication time model of $QMoA^1_{MARTE}$, set by the `Communication Media`, is the affine model illustrated in Fig. 11. Precise time properties are set but the way to correctly compute a timing at system-level from the set of resource timings is not explicitly elucidated.

$QMoA^1_{MARTE}$ can be used for more than just time modeling. `ResourceUsage` is a way to associate physical properties to the usage of a resource. When events occur, amounts of physical resources can be specified as "consumed". A resource consumption amount can be associated to the following types of NFPs values: energy in Joules, message size in bits, allocated memory in bytes, used memory in bytes (representing temporary allocation), and power peak in Watts.

The Generic Quantitative Analysis Modeling (GQAM) package defines another quasi-MoA ($QMoA^4_{MARTE}$) for performing the following set of analysis: counting the repetitions of an event, determining the probability of an execution, determining CPU requirements, determining execution latency, and determining throughput (time interval between two occurrences). New resources named `GaExecHost` (ExecutionHost) and `GaCommHost` (CommunicationHost) are added to the ones of $QMoA^1_{MARTE}$ and specialize the `ProcessingResource` for time performance and schedulability analysis, as well as for the analysis of other NFPs. $QMoA^4_{MARTE}$ is thus close to $QMoA^1_{MARTE}$ in terms of resource semantics but additional properties complement the quasi-MoA. In terms of MoAs, $QMoA^1_{MARTE}$ and $QMoA^4_{MARTE}$ have the same properties and none of them clearly states how to use their properties.

4.3.2 The UML MARTE Quasi-MoAs 2 and 3

The UML MARTE Hardware Resource Modeling (HRM) defines two other, more complex quasi-MoAs than the previously presented ones: $QMoA^2_{MARTE}$ (logical view) and $QMoA^3_{MARTE}$ (physical view).

An introduction of the related software model is necessary before presenting hardware components because the HRM is very linked to the SRM software representation. In terms of software, the UML MARTE standard constantly refers to threads as the basic instance, modeled with a `swSchedulableResource`. The `swSchedulableResources` are thus considered to be managed by an RTOS and, like AADL, UML MARTE builds on industrial best practices of using preemptive threads to model concurrent applications. In order to communicate, a `swSchedulableResource` references specifically defined software communication and synchronization resources.

The `HW_Logical` subclause of HRM refers to five subpackages: `HW_Computing`, `HW_Communication`, `HW_Storage`, `HW_Device`, and

`HW_Timing`. It composes a complex quasi-MoA referred to as $QMoA^2_{MARTE}$ in this chapter. For brevity and clarity, we will not enter the details of this quasi-MoA but give some information on its semantics.

The UML MARTE $QMoA^2_{MARTE}$ quasi-MoA is, like AADL, based on a HW/SW separation of concerns rather than on an application/architecture separation. In terms of hardware, UML MARTE tends to match very finely the real characteristics of the physical components. UML MARTE HRM is thus torn between the desire to match current hardware best practices and the necessity to abstract away system specificities. A $QMoA^2_{MARTE}$ processing element for instance can be a processor, with an explicit Instruction Set Architecture (ISA), caches, and a Memory Management Unit (MMU), or it can be a Programmable Logic Device (PLD). In the description of a PLD, properties go down to the number of available Lookup Tables (LUTs) on the PLD. However, modern PLDs such as Field-Programmable Gate Arrays (FPGAs) are far too heterogeneous to be characterized by a number of LUTs. Moreover, each FPGA has its own characteristics and in the space domain, for instance, FPGAs are not based on a RAM configuration memory, as fixed in the MARTE standard, but rather on a FLASH configuration memory. These details show the interest of abstracting an MoA in order to be resilient to the fast evolution of hardware architectures.

`HW_Physical` composes the $QMoA^3_{MARTE}$ quasi-MoA and covers coarser-grain resources than $QMoA^2_{MARTE}$, at the level of a printed circuit board. Properties of resources include shape, size, position, power consumption, heat dissipation, etc.

Interpreting the technological properties of HRM quasi-MoAs $QMoA^2_{MARTE}$ and $QMoA^3_{MARTE}$ is supposed to be done based on designer's experience because the UML MARTE properties mirror the terms used for hardware design. This is however not sufficient to ensure the *reproducibility* of a cost computation.

4.3.3 Conclusions on UML MARTE Quasi-MoAs

When considering as a whole the 4 UML MARTE quasi-MoAs, the standard does not specify how the hundreds of NFP standard resource parameters are to be used during simulation or verification. The use of these parameters is supposed to be transparent, as the defined resources and parameters match current best practices. However, best practices evolve over time and specifying precisely cost computation mechanisms is the only way to ensure tool interoperability in the long run. UML MARTE quasi-MoAs do not respect the *abstraction* rule of MoAs because, while cost properties target multiple NFPs, each is considered independently without capitalizing on similar behaviors of different NFPs. Finally, $QMoA^1_{MARTE}$ and $QMoA^4_{MARTE}$ respect the *application independence* rule, and even extend it to the construction of more than two layers, while $QMoA^2_{MARTE}$ and $QMoA^3_{MARTE}$ rather propose a HW/SW decomposition closer to AADL.

4.4 Conclusions on ADL Languages

AADL and UML MARTE are both complete languages for system-level design that offer rich constructs to model a system. MCA SHIM is a domain-specific language targeted to a more precise purpose. While the three languages strongly differ, they all specify quasi-MoAs with the objective of modeling the time behavior of a system, as well as other non-functional properties. None of these three languages fully respects the three rules of MoA's Definition 4. In particular, none of them abstracts the studied NFPs to make generic the computation of a model's cost from the cost of its constituents. Abstraction is however an important feature of MoAs to avoid redesigning redundant simulation mechanisms.

To complement this study on MoAs, the next section covers four formal quasi-MoAs from literature.

5 Formal Quasi-MoAs

In this section, we put the focus on graphical quasi-MoAs that aim at providing system efficiency evaluations when combined with a model of a DSP application. The models and their contribution are presented chronologically.

5.1 The AAA Methodology Quasi-MoA

In 2003, an architecture model is defined for the *Adéquation Algorithm Architecture (AAA)* Y-chart methodology, implemented in the SynDEx tool [13]. The AAA architecture model is tailored to the needs of an application model that splits processing into tasks called *operations* arranged in a Directed Acyclic Graph (DAG) representing data dependencies between them.

The AAA architecture model is a graphical quasi-MoA $\Lambda = \langle M, L, t, p \rangle$, where M is a set of components, L is a set of undirected edges connecting these components, and t and p respectively give a type and a property to components. As illustrated in Fig. 13, there are three types $t \in T$ of components, each considered internally as a Finite State Machine (FSM) performing sequentially application management services: *memory*, *sequencer*, and *bus/multiplexer/demultiplexer (B/M/D)*. For their part, edges only model the capacity of components to exchange data.

In this model, a *memory* is a Sequential Access Memory (`SAM`) or a Random Access Memory (`RAM`). A `SAM` models a First In, First Out data queue (FIFO) for message passing between components. A `SAM` can be point-to-point or multipoint and support or not broadcasting. A `SAM` with broadcasting only pops a data when all readers have read the data. A `RAM` may store only data (RAM_D), only programs

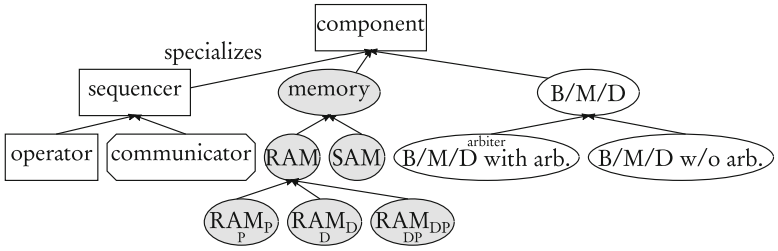


Fig. 13 Typology of the basic components in the AAA architecture model [13]. Leaf components are instantiable

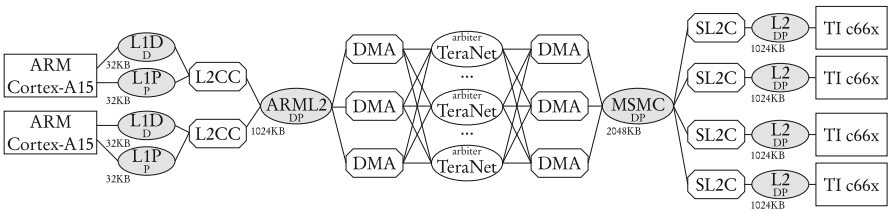


Fig. 14 Example of an architecture description with the AAA quasi-MoA

(RAM_P) or both (RAM_{DP}). When several sequencers can write to a memory, it has an implicit arbiter managing writing conflicts.

A *sequencer* is of type `operator` or `communicator`. An `operator` is a PE sequentially executing *operations* stored in a RAM_P or RAM_{DP}. An operation reads and writes data from/to a RAM_D or RAM_{DP} connected to the `operator`. A `communicator` models a DMA with a single channel that executes communications, i.e. operations that transfer data from a memory M_1 to a memory M_2. For the transfer to be possible, the communicator must be connected to M_1 and M_2.

A *B/M/D* models a bus together with its multiplexer and demultiplexer that implement time division multiplexing of data. As a consequence, a B/M/D represents a sequential schedule of transferred data. A B/M/D may require an arbiter, solving write conflicts between multiple sources. In the AAA model, the arbiter has a maximum bandwidth $BPMax$ that is shared between writers and readers.

Figure 14 shows an example, inspired by Grandpierre and Sorel [13], of a model conforming the AAA quasi-MoA. It models the 66AK2L06 processor [36] from Texas Instruments illustrated in Fig. 5g. `Operators` must delegate communication to `communicators` that access their data memory. The architecture has hardware cache coherency on ARM side (L2CC for L2 Cache Control) and software cache coherency on c66x side (SL2C for Software L2 Coherency). The communication between ARML2 and MSMC memories is difficult to model with AAA FSM components because it is performed by a Network-on-Chip (NoC) with complex topology and a set of DMAs so it has been represented as a network of B/M/Ds and communicators in Fig. 14.

Properties p on components and edges define the quasi-MoA. An `operator` Op has an associated function δ_{Op} setting a Worst Case Execution Time (WCET) duration to each operation $\delta_{Op}(o) \in \mathbb{R}_{\geq 0}$ where O is the set of all operations in the application. This property results from the primary objective of the AAA architecture model being the computation of an application WCET. Each edge of the graph has a maximum *bandwidth* B in bits/s. The aim of the AAA quasi-MoA is to feed a multicore scheduling process where application operations are mapped to `operators` and data dependencies are mapped to routes between `operators`, made of `communicators` and `busses`. Each `operator` and `communicator` being an FSM, the execution of operations and communications on a given sequencer is totally ordered. The application graph being a DAG, the critical path of the application is computed and represents the latency of one execution, i.e. the time distance between the beginning of the first operation and the end of the last operation. The computation of the latency from AAA application model and quasi-MoA in [13] is implicit. The behavior of the arbiter is not specified in the model so actual communication times are subject to interpretations, especially regarding the time quantum for the update of bandwidth utilization.

The AAA syntax-free quasi-MoA is mimicking the temporal behavior of a processing hardware in order to derive WCET information on a system. Many hardware features can be modeled, such as DMAs; shared memories and hardware FIFO queues. Each element in the model is sequential, making a coarse-grain model of an internally parallel component impossible. There is no cost abstraction but the separation between architecture model and application model is respected. The model is specific to dataflow application latency computation, with some extra features dedicated to memory requirement computation. Some performance figures are subject to interpretation and latency computation for a couple application/architecture is not specified.

The AAA model contribution is to build a system-level architecture model that clearly separates architecture concerns from algorithm concerns. Next section discusses a second quasi-MoA, named CHARMED.

5.2 The CHARMED Quasi-MoA

In 2004, the CHARMED co-synthesis framework [17] is proposed that aims at optimizing multiple system parameters represented in Pareto fronts. Such a multi-parameter optimization is essential for DSE activities, as detailed in [31].

In the CHARMED quasi-MoA $\Lambda = \langle M, L, t, p \rangle$, M is a set of PEs, L is a set of Communication Resources (CR) connecting these components, and t and p respectively give a type and a property to PEs and CRs. There is only one type of component so in this model, $t = PE$. Like in the AAA architecture model, PEs are abstract and may represent programmable microprocessors as well as hardware IPs. The PE vector of properties p is such that $p(PE \in M) = [\alpha, \kappa, \mu_d, \mu_i, \rho_{idle}]^T$

where α denotes the area of the PE, κ denotes the price of the PE, μ_d denotes the size of its data memory, μi denotes the instruction memory size and ρ_{idle} denotes the idle power consumption of the PE. Each CR edge also has a property vector: $p(CR \in L) = [\rho, \rho_{idle}, \theta]^T$ where ρ denotes the average power consumption per each unit of data to be transferred, ρ_{idle} denotes idle power consumption and θ denotes the worst case transmission rate or speed per each unit of data.

This model is close to the concept of MoA as stated by Definition 4. However, instead of abstracting the computed cost, it defines many costs altogether in a vector. This approach limits the scope of the approach and CHARMED metrics do not cover the whole spectrum on NFPs shown in Sect. 2.2. The CHARMED architecture model is combined with a DAG task graph of a stream processing application in order to compute costs for different system solutions. A task in the application graph is characterized by its required instruction memory μ, its Worst Case Execution Time $WCET$ and its average power consumption $\wp_{avg}$ while a DAG edge is associated with a data size δ. The cost for a system x has six dimensions: the area $\alpha(x)$, the price $\kappa(x)$, the number of used inter-processor routes $l_n(x)$, the memory requirements $\mu(x)$, the power consumption $\wp(x)$ and the latency $\tau(x)$. Each metric has an optional maximum value and can be set either as a constraint (all values under the constraint are equally good) or as an objective to maximize.

Cost computation is not fully detailed in the model. We can deduce from definitions that PEs are sequential units of processing where tasks are time-multiplexed and that a task consumes $\wp_{avg} \times WCET$ energy for each execution. The power consumption for a task is considered independent of the PE executing it. The latency is computed after a complete mapping and scheduling of the application onto the architecture. The price and area of the system are the sums of PE prices and areas. Memory requirements are computed from data and instruction information respectively on edges and tasks of the application graph. Using an evolutionary algorithm, the CHARMED framework produces a set of potential heterogeneous architectures together with task mappings onto these architectures.

For performing DSE, the CHARMED quasi-MoA has introduced a model that jointly considers different forms of NFP metrics. The next section presents a third quasi-MoA named System-Level Architecture Model (S-LAM).

5.3 *The System-Level Architecture Model (S-LAM) Quasi-MoA*

In 2009, the S-LAM model [29] is proposed to be inserted in the PREESM rapid prototyping tool. S-LAM is designed to be combined with an application model based on extensions of the Synchronous Dataflow (SDF) dataflow MoC [14] and a transformation of a UML MARTE architecture description into S-LAM has been conducted in [1].

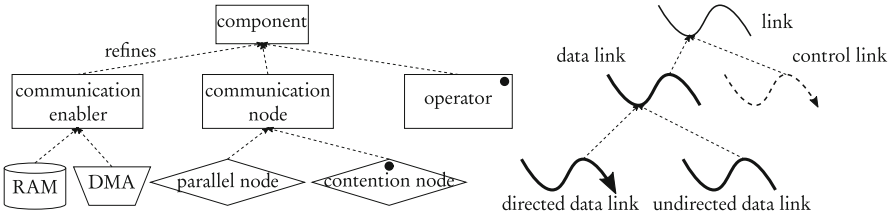


Fig. 15 Typology of the basic components in the S-LAM [29]. Leaf components are instantiable

S-LAM defines a quasi-MoA $\Lambda = \langle M, L, t, p \rangle$ where M is a set of components, L is a set of links connecting them, and t and p respectively give a type and a property to components. As illustrated in Fig. 15, there are five instantiable types of components: `operator`, `parallel node`, `contention node`, `RAM`, and `DMA`.

`Operators` represent abstract processing elements, capable of executing tasks (named actors in dataflow models) and of communicating data through links. Actors' executions are time-multiplexed over operators, as represented by the black dot on the graphical view, symbolizing scheduling. There are also `data links` and `control links`. A `data link` represents the ability to transfer data between components. `Control links` specify that an `operator` can program a `DMA`. Two actors cannot be directly connected by a data link. A route must be built, comprising at least one `parallel node` or one `contention node`. A `parallel node` N_p virtually consists of an infinite number of data channels with a given speed $\sigma(N_p)$ in Bytes/s. As a consequence, no scheduling is necessary for the data messages sharing a `parallel node`. A `contention node` N_c represents one data channels with speed $\sigma(N_c)$. Messages flowing over a `contention node` need to be scheduled, as depicted by the black dot in its representation. This internal component parallelism is the main novelty of S-LAM w.r.t. the AAA model. When transferring a data from operator O_1 to operator O_2, three scenarios are considered:

1. *direct messaging*: the sender operator itself sends the message and, as a consequence, cannot execute code simultaneously. It may have direct access to the receiver's address space or use a messaging component.
2. *DMA messaging*: the sender delegates the communication to a DMA. A `DMA` component must then be connected by a `data link` to a communication node of the route between O_1 and O_2 and a `control link` models the ability of the sender operator to program the DMA. In this case, the sender is free to execute code during message transfer.
3. *shared memory*: the message is first written to a shared memory by O_1, then read by O_2. To model this, a `RAM` component must be connected by a `data link` to a communication node of the route between O_1 and O_2.

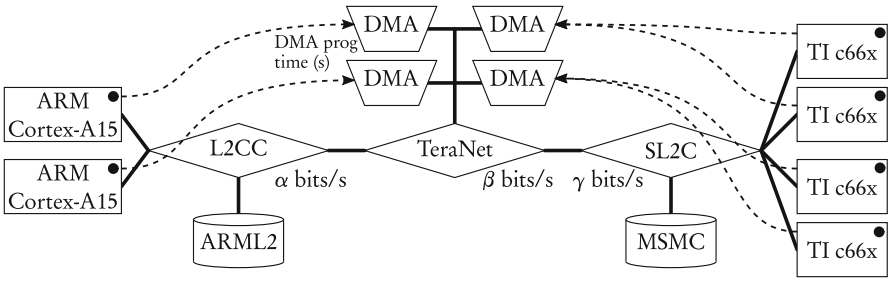


Fig. 16 Example of an architecture model with the S-LAM quasi-MoA

An S-LAM representation of an architecture can be built where different routes are possible between two operators O_1 and O_2 [29]. The S-LAM model has for primary purpose system time simulation. An S-LAM model can be more compact than an AAA model because of internal component parallelism. Indeed, there is no representation of a bus or bus arbiter in S-LAM and the same communication facility may be first represented by a `parallel node` to limit the amount of necessary message scheduling, then modeled as one or a set of`contention nodes` with or without DMA to study the competition for bus resources. Moreover, contrary to the AAA model, operators can send data themselves. Figure 16 illustrates such a compact representation on the same platform example than in Fig. 14. Local PE memories are ignored because they are considered embedded in their respective `operator`. The TeraNet NoC is modeled with a `parallel node`, modeling it as a bus with limited throughput but with virtually infinite inter-message parallelism.

The transfer latency of a message of M Bytes over a route $R = (N_1, N_2, \ldots, N_K)$, where N_i are communication nodes, is computed as $l(M) = min_{N \in R}(\sigma(N)) * M$. It corresponds in the linear model presented in Fig. 11 where the slope is determined by the slowest communication node. If the route comprises contention nodes involved in other simultaneous communications, the latency is increased by the time multiplexing of messages. Moreover, a `DMA` has an *offset* property and, if a `DMA` drives the transfer, the latency becomes $l(M) = offset + min_{NinR}(\sigma(N)) * M$, corresponding to the affine message cost in Fig. 11.

As in the AAA model, an S-LAM `operator` is a sequential PE. This is a limitation if a hierarchical architecture is considered where PEs have internal observable parallelism. S-LAM operators have an operator ISA type (for instance ARMv7 or C66x) and each actor in the dataflow application is associated to an execution time cost for each operator type. S-LAM clearly separates algorithm from architecture but it does not specify cost computation and does not abstract computation cost.

S-LAM has introduced a compact quasi-MoA to be used for DSP applications. The next section presents one last quasi-MoA from literature.

5.4 The MAPS Quasi-MoA

In 2012, a quasi-MoA is proposed in [5] for programming heterogeneous Multi-processor Systems-on-Chips (MPSoCs) in the MAPS compiler environment. It combines the multi-modality of CHARMED with a sophisticated representation of communication costs. The quasi-MoA serves as a theoretical background for mapping multiple concurrent transformational applications over a single MPSoC. It is combined with Kahn Process Network (KPN) application representations [2, 15] and is limited to the support of software applications.

The MAPS quasi-MoA is a graph $\Lambda = \langle M, L, t, p \rangle$ where M is a set of PEs, L is a set of named edges called Communication Primitives (CPs) connecting them, and t and p respectively give a type and a property to components. Each PE has properties $p(PE \in M) = (CM^{PT}, X^{PT}, V^{PT})$ where CM^{PT} is a set of functions associating NFP costs to PEs. An example of NFP is ζ^{PT} that associates to a *task* T_i in the application an execution time $\zeta^{PT}(T_i)$. X^{PT} is a set of PE attributes such as context switch time of the OS or some resource limitations, and V^{PT} is a set of variables, set late after application mapping decisions, such as the processor scheduling policy. A CP models a software Application Programming Interface (API) that is used to communicate among *tasks* in the KPN application. A CP has its own set of cost model functions CM^{CP} associating costs of different natures to communication volumes. A function $\zeta^{CP} \in CM^{CP}$ is defined. It associates a communication time $\zeta^{CP}(N)$ to a message of N bytes. Function ζ^{CP} is a stair function modeling the message overhead and performance bursts frequently observed when transferring data for instance with a DMA and packetization. This function, displayed in Fig. 11, is expressed as:

$$\zeta^{CP} : N \mapsto = \begin{cases} offset & if \quad N < start \\ offset + scale\_height\rceil & \\ \times \lceil (N - start + 1)/scale\_width & otherwise, \end{cases} \tag{4}$$

where $start$, $offset$, $scale\_height$ and $scale\_width$ are 4 CP parameters. The primary concern of the MAPS quasi-MoA is thus time. No information is given on whether the sender or the receiver PE can compute a task in parallel to communication. A CP also refers to a set of Communication Resources (CRs), i.e. a model of a hardware module used to implement the communication. A CRs has two attributes: the number of logical channels and the amount of available memory in the module. For example, a CR may model a shared memory, a local memory, or a hardware communication queue.

This quasi-MoA does not specify any cost computation procedure from the data provided in the model. Moreover, the MAPS architecture model, as the other architecture models presented in this Section, does not abstract the generated costs. Next section summarizes the results of studying the four formal architecture models.

5.5 *Evolution of Formal Architecture Models*

The four presented models have inspired the Definition 4 of an MoA. Theses formal models have progressively introduced the ideas of:

- architecture abstraction by the AAA quasi-MoA [13],
- architecture modeling for multi-dimensional DSE by CHARMED [17],
- internal component parallelism by S-LAM [29],
- complex data transfer models by MAPS [5].

The next section concludes this chapter on MoAs for DSP systems.

6 Concluding Remarks on MoA and Quasi-MoAs for DSP Systems

In this chapter, the notions of Model of Architecture (MoA) and quasi-MoA have been defined and several models have been studied, including fully abstract models and language-defined models. To be an MoA, an architecture model must capture efficiency-related features of a platform in a reproducible, abstract and application-agnostic fashion.

The existence of many quasi-MoAs and their strong resemblance demonstrate the need for architecture modeling semantics. Table 1 summarizes the objectives and properties of the different studied models. As explained throughout this chapter,

Table 1 Properties (from Definition 4) and objectives of the presented MoA and quasi-MoAs

Model	Reproducible	Appli. Agnostic	Abstract	Main objective
AADL quasi-MoA	✓	✗	✗	HW/SW codesign of hard RT system
MCA SHIM quasi-MoA	✗	✗	✗	Multicore performance simulation
UML MARTE quasi-MoAs	✗	✓/✗	✗	Holistic design of a system
AAA quasi-MoA	✗	✓	✗	WCET evaluation of a DSP system
CHARMED quasi-MoA	✗	✓	✗	DSE of a DSP system
S-LAM quasi-MoA	✗	✓	✗	Multicore scheduling for DSP
MAPS quasi-MoA	✗	✓	✗	Multicore scheduling for DSP
LSLA MoA	✓	✓	✓	System-level modeling of a NFP

LSLA is, to the extent of our knowledge, the only model to currently comply with the three rules of MoA definition (Definition 4).

LSLA is one example of an MoA but many types of MoAs are imaginable, focusing on different modalities of application activity such as concurrency or spatial data locality. A parallel with MoCs on the application side of the Y-chart motivates for the creation of new MoAs. MoCs have the ability to greatly simplify the system-level view of a design, and in particular of a DSP design. For example, and as discussed by several chapters in this Handbook, MoCs based on Dataflow Process Networks (DPNs) are able to simplify the problem of system verification by defining globally asynchronous systems that synchronize only when needed, i.e. when data moves from one location to another. DPN MoCs are naturally suited to modeling DSP applications that react upon arrival of data by producing data. MoAs to be combined with DPN MoCs do not necessarily require the description of complex relations between data clocks. They may require only to assess the efficiency of "black box" PEs, as well as the efficiency of transferring, either with shared memory or with message passing, some data between PEs. This opportunity is exploited in the semantics of the four formal languages presented in Sect. 5 and can be put in contrast with the UML MARTE standard that, in order to support all types of transformational and reactive applications, specifies a generic clock relation language named CCSL [25].

The three properties of an MoA open new opportunities for system design. While abstraction makes MoAs adaptable to different types of NFPs, cost computation reproducibility can be the basis for advanced tool compatibility. Independence from application concerns is moreover a great enabler for Design Space Exploration methods.

Architecture models are also being designed in other domains than Digital Signal Processing. As an example in the High Performance Computing (HPC) domain, the Open MPI Portable Hardware Locality (hwloc) [11] models processing, memory and communication resources of a platform with the aim of improving the efficiency of HPC applications by tailoring thread locality to communication capabilities. Similarly to most of the modeling features described in this chapter, the hwloc features have been chosen to tackle precise and medium-term objectives. The convergence of all these models into a few generic MoAs covering different aspects of design automation is a necessary step to manage the complexity of future large scale systems.

Acknowledgements I am grateful to François Berry and Jocelyn Sérot for their valuable advice and support during the writing of this chapter.

This work was partially supported by the CERBERO (Cross-layer modEl-based fRamework for multi-oBjective dEsign of Reconfigurable systems in unceRtain hybRid envirOnments) Horizon 2020 Project, funded by the European Union Commission under Grant 732105.

List of Acronyms

AAA	Adéquation algorithm architecture
AADL	Architecture analysis and design language
ADL	Architecture design language
API	Application programming interface
BER	Bit error rate
B/M/D	bus/multiplexer/demultiplexer
CCCR	Computation to communication cost ratio
CCSL	Clock constraint specification language
CN	Communication node
CP	Communication primitive
CPU	Central processing unit
CR	Communication resource
DAG	Directed acyclic graph
DMA	Direct memory access
DPN	Dataflow process network
DSE	Design space exploration
DSP	Digital signal processing
EDF	Earliest deadline first
FIFO	First in, first out data queue
FPGA	Field-programmable gate array
FSM	Finite state machine
GALS	Globally asynchronous locally synchronous
GCM	Generic component model
GPP	General purpose processor
GQAM	Generic quantitative analysis modeling
GRM	Generic resource modeling
HLAM	High-level application modeling
HPC	High performance computing
HRM	Hardware resource modeling
hwloc	Portable hardware locality
IP	Intellectual property core
ISA	Instruction set architecture
KPN	Kahn process network
LSLA	Linear system-level architecture model
LUT	Lookup table
MARTE	Modeling and analysis of real-time embedded systems
MCA	Multicore association
MMU	Memory management unit
MoA	Model of architecture
MoC	Model of computation
MPSoC	Multiprocessor system-on-chip
MSMC	Multicore shared memory controller

NFP	Non-functional property
NoC	Network-on-chip
OMG	Object management group
OS	Operating system
OSI	Open systems interconnection
PAM	Performance analysis modeling
PE	Processing element
PLD	Programmable logic device
PT	Processor type
PU	Processing unit
QoS	Quality of service
RAM	Random access memory
RM	Rate monotonic
ROM	Read-only memory
RSM	Repetitive structure modeling
RTOS	Real-time operating system
SAM	Sequential access memory
SAM	Schedulability analysis modeling (UML MARTE)
SDF	Synchronous dataflow
SHIM	Software/hardware interface for multicore/manycore
S-LAM	System-level architecture model
SMP	Symmetric multiprocessing
SNR	Signal-to-noise ratio
SRM	Software resource modeling
TLM	Transaction-level modeling
TU	Transfer unit
UML	Unified modeling language
WCET	Worst case execution time

References

1. Ammar M, Baklouti M, Pelcat M, Desnos K, Abid M (2016) Automatic generation of s-lam descriptions from uml/marte for the dse of massively parallel embedded systems. In: Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing 2015, Springer, pp 195–211
2. Bacivarov I, Haid W, Huang K, Thiele L (2018) Methods and tools for mapping process networks onto multi-processor systems-on-chip. In: Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) Handbook of Signal Processing Systems, 3rd edn, Springer
3. Bellard F (2005) QEMU, a Fast and Portable Dynamic Translator. In: USENIX Annual Technical Conference, FREENIX Track, pp 41–46
4. Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, others (2011) The gem5 simulator. ACM SIGARCH Computer Architecture News 39(2):1–7, URL http://dl.acm.org/citation.cfm?id=2024718
5. Castrillon Mazo J, Leupers R (2014) Programming Heterogeneous MPSoCs. Springer International Publishing, Cham, URL http://link.springer.com/10.1007/978-3-319-00675-8

6. Chen Y, Chen L (2013) Video compression. In: Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) Handbook of Signal Processing Systems, 2nd edn, Springer
7. Eker J, Janneck JW, Lee E, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y, et al (2003) Taming heterogeneity-the ptolemy approach. Proceedings of the IEEE 91(1):127–144
8. Faugere M, Bourbeau T, De Simone R, Gerard S (2007) Marte: Also an uml profile for modeling aadl applications. In: Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on, IEEE, pp 359–364
9. Feiler PH, Gluch DP (2012) Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language. Addison-Wesley
10. Feiler PH, Gluch DP, Hudak JJ (2006) The architecture analysis & design language (AADL): An introduction. Tech. rep., DTIC Document
11. Goglin B (2014) Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc). In: High Performance Computing & Simulation (HPCS), 2014 International Conference on, IEEE, pp 74–81
12. Gondo M, Arakawa F, Edahiro M (2014) Establishing a standard interface between multi-manycore and software tools-SHIM. In: COOL Chips XVII, 2014 IEEE, IEEE, pp 1–3
13. Grandpierre T, Sorel Y (2003) From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In: Formal Methods and Models for Co-Design, 2003. MEMOCODE'03. Proceedings. First ACM and IEEE International Conference on, IEEE, pp 123–132
14. Ha S, Oh H (2013) Decidable dataflow models for signal processing: Synchronous dataflow and its extensions. In: Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) Handbook of Signal Processing Systems, 2nd edn, Springer
15. Kahn G (1974) The semantics of a simple language for parallel programming. In Information Processing 74:471–475
16. Keutzer K, Newton AR, Rabaey JM, Sangiovanni-Vincentelli A (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE transactions on computer-aided design of integrated circuits and systems 19(12):1523–1543
17. Kianzad V, Bhattacharyya SS (2004) CHARMED: A multi-objective co-synthesis framework for multi-mode embedded systems. In: Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on, IEEE, pp 28–40
18. Kienhuis B, Deprettere E, Vissers K, van der Wolf P (1997) An approach for quantitative analysis of application-specific dataflow architectures. In: Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on, IEEE, pp 338–349
19. Kienhuis B, Deprettere EF, Van Der Wolf P, Vissers K (2002) A methodology to design programmable embedded systems. In: Embedded processor design challenges, Springer, pp 18–37
20. Larsen M (2016) Modelling field robot software using aadl. Technical Report Electronics and Computer Engineering 4(25)
21. Lasnier G, Zalila B, Pautet L, Hugues J (2009) Ocarina: An environment for aadl models analysis and automatic code generation for high integrity applications. In: International Conference on Reliable Software Technologies, Springer, pp 237–250
22. Lattner C, Adve V (2004) Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, IEEE Computer Society, p 75
23. Lee EA (2006) The problem with threads. Computer 39(5):33–42
24. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proceedings of the IEEE 75(9)
25. Mallet F, André C (2008) Uml/marte ccsl, signal and petri nets. PhD thesis, INRIA
26. Mallet F, De Simone R (2009) Marte vs. aadl for discrete-event and discrete-time domains. In: Languages for Embedded Systems and Their Applications, Springer, pp 27–41
27. Multicore Association (2015) Software/Hardware Interface for Multicore/Manycore (SHIM) - http://www.multicore-association.org/workgroup/shim.php/ (accessed 03/2017)

28. OMG (2011) UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Object Management Group, Needham, MA
29. Pelcat M, Nezan JF, Piat J, Croizer J, Aridhi S (2009) A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems. In: Proceedings of DASIP conference
30. Pelcat M, Mercat A, Desnos K, Maggiani L, Liu Y, Heulot J, Nezan JF, Hamidouche W, Menard D, Bhattacharyya SS (2017) Reproducible evaluation of system efficiency with a model of architecture: From theory to practice. Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)
31. Pimentel AD (2017) Exploring exploration: A tutorial introduction to embedded systems design space exploration. IEEE Design & Test 34(1):77–90
32. Renfors M, Juntti M, Valkama M (2018) Signal processing for wireless transceivers. In: Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) Handbook of Signal Processing Systems, 3rd edn, Springer
33. SAE International (2012) Architecture analysis and design language (aadl) - http://standards.sae.org/as5506c/ (accessed 03/2017)
34. Shekhar R, Walimbe V, Plishker W (2013) Medical image processing. In: Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) Handbook of Signal Processing Systems, 2nd edn, Springer
35. Stevens A (2011) Introduction to AMBA 4 ACE and big.LITTLE Processing Technology
36. Texas Instruments (2015) 66AK2L06 Multicore DSP+ARM KeyStone II System-on-Chip (SoC) - SPRS930. Texas Instruments, URL http://www.ti.com/lit/pdf/sprs866e (accessed 03/2017)
37. Van Roy P, et al (2009) Programming paradigms for dummies: What every programmer should know. New computational paradigms for computer music 104
38. Wolf M (2014) High-performance embedded computing: applications in cyber-physical systems and mobile computing. Newnes

Optimization of Number Representations

Wonyong Sung

Abstract In this section, automatic scaling and word-length optimization procedures for efficient implementation of signal processing systems are explained. For this purpose, a fixed-point data format that contains both integer and fractional parts is introduced, and used for systematic and incremental conversion of floating-point algorithms into fixed-point or integer versions. A simulation based range estimation method is explained, and applied to automatic scaling of C language based digital signal processing programs. A fixed-point optimization method is also discussed, and optimization examples including a recursive filter and an adaptive filter are shown.

1 Introduction

Although some embedded processors equip floating-point units, it is needed to process fixed-point data with reduced word-length like 8 or 16 bits to lower the energy consumption. But, integer or fixed-point versions can suffer from overflows and quantization effects. Converting a floating-point program to an integer version requires scaling of data, which is known to be difficult and time-consuming. VLSI implementation of digital signal processing algorithms demands fixed-point arithmetic for reducing the chip area, circuit delay, and power consumption. With fixed-point arithmetic, it is possible to use the fewest number of bits possible for each signal and save the chip area. However, if the number of bits is too small, quantization noise will degrade the system performance to an unacceptable level. Thus, fixed-point optimization that minimizes the hardware cost while meeting the fixed-point performance is very needed.

In Sect. 2, the data format for representing a fixed-point data is presented. This format contains both integer and fractional parts for representing a data. Thus, this

W. Sung (✉)
Department of Electrical and Computer Engineering, Seoul National University, Gwanak-gu, Seoul, Republic of Korea
e-mail: wysung@snu.ac.kr

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_31

format is very convenient for data conversion between floating-point and fixed-point data types. Section 3 contains the range estimation methods that are necessary for integer word-length determination and scaling. A simulation based range estimation method is explained, which can be applied to not only linear but also non-linear and time-varying systems. In Sect. 4, a floating-point to integer C program conversion process is shown. This code conversion process is especially useful for C program language based implementation of signal processing systems. Section 5 presents the word-length optimization flow for signal processing programs, which should be important for VLSI or 16-bit programmable digital signal processor (DSP) based implementations. In Sect. 6, the summary and related works are described.

2 Fixed-Point Data Type and Arithmetic Rules

A fixed-point data type does not contain an exponent term, which makes hardware for fixed-point arithmetic much simpler than that for floating-point arithmetic. However, fixed-point data representation only allows a limited dynamic range, hence scaling is needed when converting a floating-point algorithm into a fixed-point version. In this section, fixed-point data formats, fixed-point arithmetic rules, and a simple floating-point to fixed-point conversion example will be shown. The two's complement format is used when representing negative numbers.

2.1 Fixed-Point Data Type

A widely used fixed-point data format is the integer format. In this format, the least significant bit (LSB) has the weight of 1, thus the maximum quantization error can be as large as 0.5 even if the rounding scheme is used. As a result, small numbers cannot be faithfully represented with this format. Of course, there can be overflows even with the integer format because an N-bit signed integer has a value that is between -2^{N-1} and $2^{N-1}-1$. Another widely used format is the fractional format, in which the magnitude of a data cannot exceed 1. This format seems convenient for representing a signal whose magnitude is bounded by 1, but it suffers from overflow or saturation problems when the magnitude exceeds the bound. Figure 1 shows two different interpretations for a binary data '1001000.'

With either the integer or the fractional format, an expert can design an optimized digital signal processing system by incorporating proper scaling operations, which is, however, very complex and difficult to manage. This conversion flow is not easy because all of the intermediate variables or constants should be represented with only integers or fractions whose represented values are usually much different from those of the corresponding floating-point data. The difference of the representation format also hinders incremental conversion from a floating-point design to a fixed-

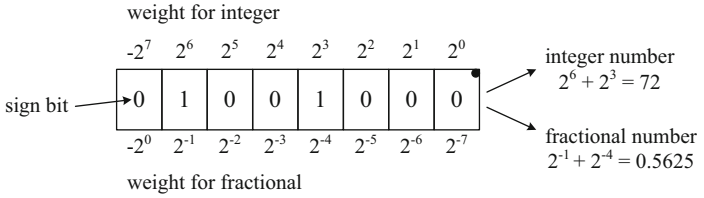


Fig. 1 Integer and fractional data formats

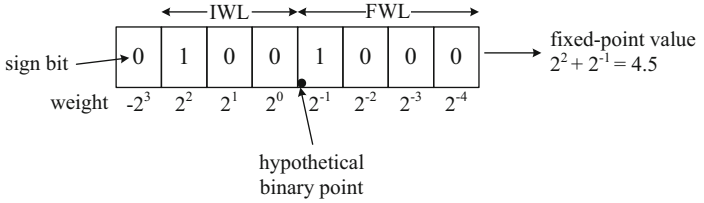


Fig. 2 Generalized fixed-point data format

point one. For seamless floating-point to integer or fixed-point conversion, the semantic gap between the floating-point and fixed-point data formats needs to be eliminated.

To solve these problems, a generalized fixed-point data-type that contains both integer and fractional parts can be used [17]. This fixed-point format contains the attributes specified as follows.

$$< wordlength,\ integer\ wordlength,\ sign\ overflow\ quantization\ mode > \tag{1}$$

The word-length (WL) is the total number of bits for representing a fixed-point data. The integer word-length (IWL) is the number of bits to the left of the (hypothetical) binary-point. The fractional word-length (FWL) is the number of bits to the right of the (hypothetical) binary point. The sign is not included in IWL, and can be either unsigned('u') or two's complement('t'). Thus, the word-length (WL) corresponds to 'IWL+FWL+1' for signed data, and is 'IWL+FWL' for unsigned data. If the fractional word-length is 0, the data with this format can be represented with integers. At the same way, it becomes the fractional format when the IWL is 0. Note that the IWL or FWL can be even larger than the WL; in this case, the other part has a minus word-length. Figure 2 shows an interpretation of an 8-bit binary data employing the fixed-point format with the IWL of 3.

The overflow and quantization modes are needed for arithmetic or quantization operations. The overflow mode specifies whether no treatment ('o') or saturation ('s') scheme is used when overflow occurs, and the quantization mode denotes whether rounding ('r') or truncation ('t') is employed when least significant bits are quantized.

Most of all, this fixed-point data representation is very convenient for translating a floating-point algorithm into a fixed-point version because the data values is not limited to integers or fractions. The range (R) and the quantization step (Q) are dependent on the IWL and FWL, respectively: $-2^{IWL} \leq R < 2^{IWL}$ and $Q = 2^{-FWL} = 2^{-(WL-1-IWL)}$ for the signed format. Assigning a large IWL to a variable can prevent overflows, but it increases the quantization noise. Thus, the optimum IWL for a variable should be determined according to its range or the possible maximum absolute value. The minimum IWL for a variable x, $IWL_{min}(x)$, can be determined according to its range, $R(x)$, as follows.

$$IWL_{min}(x) = \lceil \log_2 R(x) \rceil, \tag{2}$$

where $\lceil x \rceil$ denotes the smallest integer which is equal to or greater than x. Note that preventing overflow and saturation is very critical in fixed-point arithmetic because the magnitude of the error caused by them is usually much larger than that produced by quantization.

2.2 Fixed-Point Arithmetic Rules

Since the generalized fixed-point data format allows a different integer word-length, two variables or constants that do not have the same integer word-length cannot be added or subtracted directly. Let us assume that $x1$ is '01001000' with the IWL of 3 and $x2$ is '00010000' with the IWL of 2. Since the interpreted value of $x1$ is 4.5 and that of $x2$ is 0.5, the result should be a number that corresponds to 5.0 or a close one. However, direct addition of 01001000 ($x1$) and 00010000 ($x2$) does not yield the expected result. This is because the two data have different integer word-lengths. The two fixed-point data should be added after aligning their hypothetical binary points. The binary point can be moved, or the integer word-length can be changed, by using arithmetic shift operations. Arithmetic right shift by one bit increases the integer word-length by one, while arithmetic left shift decreases the integer word-length. The number of shifts required for addition or subtraction can easily be obtained by comparing the integer word-lengths of the two input data format. In the above example, $x2$, with the IWL of 2, should be shifted right by 1 bit before performing the integer addition to align the binary-point locations. As illustrated in Fig. 3, this results in a correct value of 5.0 when the output is interpreted with the IWL of 3. Note that the result of addition or subtraction sometimes needs an increased IWL. If the IWL of the added result is greater than those of two input operands, the inputs should be scaled down to prevent overflows. Subtraction can be treated the same way with addition. The scaling rules for addition and subtraction are shown in Table 1, where Ix and Iy are the IWL's of two input operands x and y, respectively, and Iz is that of the result, z.

In fixed-point multiplication, the word-length of the product is equal to the sum of two input word-lengths. In two's complement multiplication, two identical

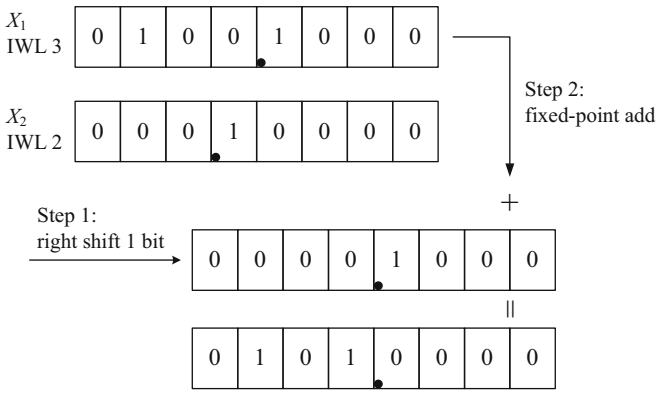


Fig. 3 Fixed-point addition with different IWL's

Table 1 Fixed-point arithmetic rules

	Floating-point	Fixed-point			Result IWL
		$I_x > I_y, I_z$	$I_y > I_x, I_z$	$I_z > I_x, I_y$	
Assignment	$x = y$	$x = y >> (I_x - I_y)$	$x = y << (I_y - I_x)$	–	I_x
Addition/subtraction	$x + y$	$x + (y >> (I_x - I_y))$	$(x >> (I_y - I_x)) + y$	$(x >> (I_z - I_x)) + y >> (I_z - I_y)$	$max(I_x, I_y, I_z)$
Multiplication	$x * y$	$mulh(x, y)$			$I_x + I_y + 1$ or $I_x + I_y$

z: a variable storing the result

sign bits are generated except for the case that both input data correspond to the negative minimum, '$100 \cdots 0$.' Ignoring this case, the IWL of the two's complement multiplied result becomes $Ix + Iy + 1$. Figure 4 shows the multiplied result of two 8-bit fixed-point numbers. By assuming the IWL of 5, we can obtain the interpreted value of 2.25.

2.3 Fixed-Point Conversion Examples

To illustrate the fixed-point conversion process, a floating-point version of the recursive filter shown in Fig. 5a is transformed to a fixed-point hardware system. Assume that the input signal has the range of 1, which implies that it is between -1 and 1. The output signal is also known to be between -5.3 and 5.3. The output signal range is obtained from floating-point simulation results. The coefficient is 0.9, and is unsigned. Hence, the range of the multiplied signal, $z[n]$, will be 4.77 (=0.9*5.3).

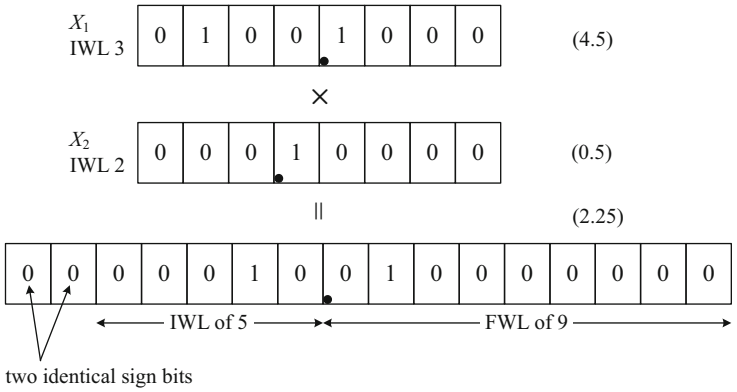


Fig. 4 Fixed-point multiplication

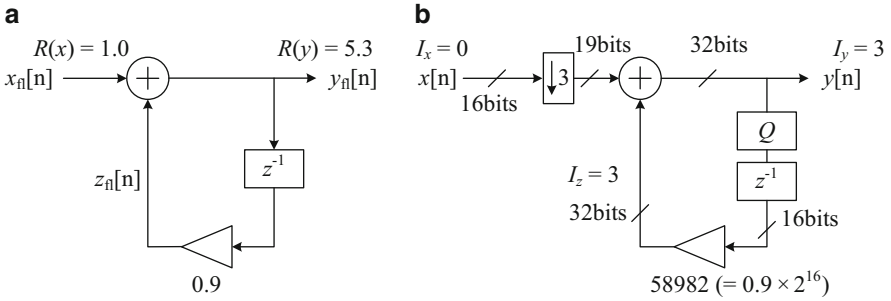


Fig. 5 Floating-point to fixed-point conversion of a recursive filter. (**a**) Floating-point filter. (**b**) Fixed-point filter

From the given range information, we can assign the IWL's of 0 for $x[n]$, 3 for $y[n]$, 3 for $z[n]$ and 0 for the coefficient. The coefficient a is coded as '58982,' which corresponds to the unsigned number 0.9×2^{16}. Since the multiplication of $y[n]$ and a is conducted between signed and unsigned numbers, the IWL of $z[n]$ is 3, which is $I_y + I_a$. If the coefficient a is coded with the two's complement format, the IWL of $z[n]$ would be 4 due to the extra sign generated in the multiplication process. Since the precision of the hardware multiplier is 16-bit, only the upper 16 bits, including the sign bit, of $y[n]$ is used for the multiplication. The quantizer (Q) in this figure takes the upper 16 bits among 32 bits of $y[n]$. Since the difference between I_x and I_z is 3, $x[n]$ is scaled down or arithmetic shift-righted by 3 bits, as the hardware in Fig. 5b shows.

There are a few different fixed-point implementations. One example is a fixed-point implementation without needing shift operations. Note that no shift operation is needed when adding or subtracting two fixed-point data with the same integer word-length. In this case, the IWL of 3 is assigned to the input $x[n]$, even though the range of $x[n]$ is 1.0. This means that the input $x[n]$ is un-normalized, and the

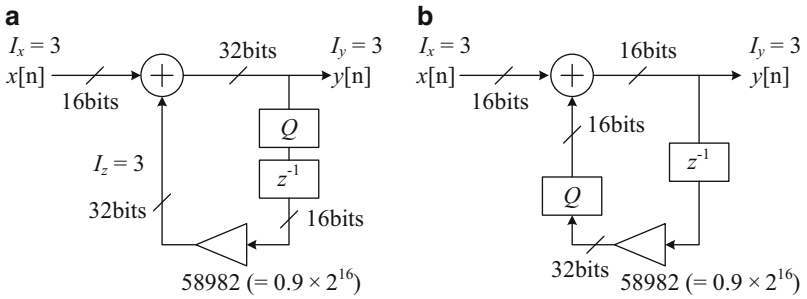


Fig. 6 Fixed-point filters with reduced complexity. (**a**) Fixed-point filter without shift. (**b**) Fixed-point filter with a 16 bit adder

upper 3 bits next to the sign bit are unused. Since the IWL's of $x[n]$ and $z[n]$ are the same, there is no need of inserting a shifter. Figure 6a shows the resulting hardware. The SQNR (Signal to Quantization Noise Ratio) of the input is obviously lowered by employing the un-normalized scheme. Another fixed-point implementation in Fig. 6b shows the hardware using a 16-bit adder. In this case, the quantizer (Q) is moved to the output of the multiplier. Note that the SQNR of this scheme is even lower than that of Fig. 6a.

In the above example, the range of 1 is assumed to the input $x[n]$, which is from the floating-point design. However, assuming the range of 2 as for the input $x[n]$ does not change the resultant hardware because the output range should be doubled in this case.

3 Range Estimation for Integer Word-Length Determination

The floating-point to fixed-point conversion examples in the previous section shows that estimating the ranges of all of variables is most crucial for this conversion process. There are two different approaches for range estimation. One is to calculate the L1-norm of the system and the other is using the simulation results of floating-point systems [12, 17].

3.1 L1-Norm Based Range Estimation

The L1-norm of a linear shift-invariant system is the maximum value of the output when the absolute value of the input is bounded to 1. If the unit-pulse response of a system is $h[n]$, where $n = 0, 1, 2, 3, \cdots \infty$, the L1-norm of this system is defined as:

$$L1\text{-}norm(h[n]) = \sum_{n=0}^{\infty} |h[n]| \tag{3}$$

Obviously, the L1-norm can easily be estimated for an FIR system. There are also several analytical methods that compute the L1-norm of an IIR (infinite impulse response) system [12]. Since the unit-pulse response of an IIR system usually converges to zero when thousands of time-steps elapse, it is practically possible to estimate the L1-norm of an IIR system with a simple C code or a Matlab program that sums up the absolute value of the unit-pulse response, instead of conducting a contour integration [11, 12]. Since L1-norm cannot easily be defined to time-varying or non-linear systems, the L1-norm based range estimation method is hardly applicable to systems containing non-linear and time-varying blocks. Another characteristic of the L1-norm is that it is a very conservative estimate, which means that the range obtained with the L1-norm is the largest one for any set of the given input, and hence the result can be an over-estimate. For example, the L1-norm of the first order recursive system shown in Fig. 5a is 10, which corresponds to the case that the input is a DC signal with the maximum value of 1. For example, if we design a speech processing system, the input with this characteristic is not likely to exist. With an over-estimated range, the data should be shift-down by more bits, which will increase the quantization noise level. For a large scale system, the L1-norm based scaling can be impractical because accumulation of extra-bits at each stage may seriously lower the accuracy of the output. However, if a very reliable system that should not experience any overflow is needed, the L1-norm based scaling can be considered. The L1-norm based scaling is limited in use for real applications because most practical systems contain time-varying or non-linear blocks.

3.2 Simulation Based Range Estimation

The simulation based method estimates the ranges by simulation of floating-point design while applying realistic input signal-samples [17]. This method is especially useful when there is a floating-point simulation model, which can be a C program or a CAD system based design. This method can be applied to general, including non-linear and time-varying, systems. Thus, provided that there is a floating-point version of a designed system and various input files for simulation, a CAD tool can convert a floating-point design to a fixed-point version automatically. One drawback of this method is that it needs extensive simulations with different environmental parameters and various input signal files. The scaling with this approach is not conservative, thus there can be overflows if the statistics of the real input signal differ much from the ones used for the range estimation. Therefore, it is needed to employ various input files for simulation or give some additional integer bits, called

the guard-bits, to secure overflow-free design. This simulation based method can also be applied to word-length optimization.

For unimodal and symmetric distributions, the range can be effectively estimated by using the mean and the standard deviation, which are obtained from simulation results, as follows.

$$R = |\mu| + n \times \sigma, \quad n \propto k \tag{4}$$

Specifically, we can use n as $k + 4$, where k is the kurtosis [17]. However, the above rule is not satisfactory for other distributions, which may be multimodal, non symmetric, or non zero mean. As an alternate rule, we can consider

$$R = \hat{R}_{99.9\%} + g \tag{5}$$

where g is a guard for the range. A partial maximum, $\hat{R}_{P\%}$, indicates a sub-maximum value, which covers $P\%$ of the entire samples. Note that various sub-maxima are collected during the simulation. The more different $\hat{R}_{100\%}$ and $\hat{R}_{99.9\%}$ are, the larger guard value is needed.

3.3 C++ Class Based Range Estimation Utility

A range estimation utility for C language based digital signal processing programs is explained, which is freely available [3]. This range estimation utility is not only essential for automatic integer C code generation, but also useful for determining the number of shifts in assembly programming of fixed-point DSPs [16]. With this utility, users develop a C application program with floating-point arithmetic. The range estimator then finds the statistics of internal signals throughout floating-point simulation using real inputs, and determines the integer word-lengths of variables. Although we can develop a separate program that traces the range information during simulation, this approach may demand too much program modification. The developed range estimation class uses the operator overloading characteristics of C++ language, thus a programmer does not need to change the floating-point code significantly for range estimation.

To record the statistics during simulation, a new data class for tracing the possible maximum value of a signal, i.e., the range, has been developed and named as `fSig`. In order to prepare a range estimation model of a C or C++ digital signal processing program, it is only necessary to change the type of variables, from `float` to `fSig`. The `fSig` class not only computes the current value, but also keeps the records of a variable using private members for it. When the simulation is completed, the ranges of the variables declared as `fSig` class are readily available from the records stored in the class.

The `fSig` class has several private members including `Data`, `Sum`, `Sum2`, `Sum3`, `Sum4`, `AMax`, and `SumC`. `Data` keeps the current value, while `Sum` and

Sum2 record the summation and the square summation of past values, respectively. Sum3 and Sum4 store the third and fourth moments, respectively. They are needed to calculate the statistics of a variable, such as mean, standard deviation, skewness, and kurtosis. AMax stores the absolute maximum value of a variable during the simulation. The class also keeps the number of modifications during the simulation in SumC field.

The fSig class overloads arithmetic and relational operators. Hence, basic arithmetic operations, such as addition, subtraction, multiplication, and division, are conducted automatically for fSig variables. This property is also applicable to relational operators, such as '==,' '!=,' '>,' '<,' '>=,' and '<=.' Therefore, any fSig instance can be compared with floating-point variables and constants. The contents, or private members, of a variable declared by the fSig class is updated when the variable is assigned by one of the assignment operators, such as '=,' '+=,' '−=,' '∗=,' and '/=.' The *range estimator* is executed after preparing the simulation model that only modifies the variable declaration. After the simulation is completed, the mean (μ), standard deviation (σ), skewness (s), and kurtosis (k) can be calculated using Sum, Sum2, Sum3, Sum4 and SumC information. Then, the statistical range of fSig variable x can be estimated. The integer word-lengths of all signals are then obtained from their ranges.

As an example, let us consider a first order digital IIR filter. The original C++ program for the filter and a translated version for range estimation are given in Fig. 7.

As shown in Fig. 7, it is only necessary for developing a range estimation program to modify the declaration part of the original floating-point version. In this example, a white noise sequence uniformly distributed between -1 and $+1$ is used for the input data, and four times of standard deviation is used for estimating the ranges. Note that the integer word-length for X_{in} is known, 0. The range estimation result is shown in Fig. 8.

a

```
void
iir1(short argc, char *argv[])
{
  float Xin;
  float Yout;    // fSig()
  float Ydly;    // fSig()
  float Coeff;

  Coeff = 0.9;
  Ydly = 0.;
  for( i = 0; i < 1000; i++ ) {
    infile >> Xin ;
    Yout = Coeff * Ydly + Xin ;
    Ydly = Yout ;
    outfile << Yout << '\n';
  }
}
```

b

```
void
iir1(short argc, char *argv[])
{
  float Xin;
  static fSig Yout("iir1/Yout");
  static fSig Ydly("iir1/Ydly");
  float Coeff;

  Coeff = 0.9;
  Ydly = 0.;
  for( i = 0; i < 1000; i++ ) {
    infile >> Xin ;
    Yout = Coeff * Ydly + Xin ;
    Ydly = Yout ;
    outfile << Yout << '\n';
  }
}
```

Fig. 7 C++ programs for a first order IIR filter. (**a**) The original C++ program. (**b**) A version for the range estimator

```
Statistics:
  VarName     Mean   StdDev Skewness Kurtosis  R99.9%   R100% Update
iir1/Ydly -0.1133 +1.3076  +0.0220  -0.0258 +4.2638 +4.4214   3001
iir1/Yout -0.1134 +1.3078  +0.0220  -0.0268 +4.2638 +4.4214   3000

Integer word-lengths:
      VarName          Range      IWL
    iir1/Ydly      +5.309891       +3
    iir1/Yout      +5.309515       +3
```

Fig. 8 The result of the range estimator for the IIR filter

The elements of an array variable are assumed to have the same IWL for simple code generation. If it is not, the scaled integer codes need to check the array index, which can slow down program execution significantly. For a pointer variable, the IWL is defined as that of the pointed variables. For example, when the pointer variables p and q have the IWL of 2 and 3, respectively, the expression $*q = *p$ can be converted to $*q = *p >> 1$. Since the IWL of a pointer variable is not changed at runtime, a pointer cannot support two variables having different IWL's. In this case, the IWL's of these pointers are equalized automatically at the integer C code generation step that will be described in the next section.

4 Floating-Point to Integer C Code Conversion

C language is most frequently used for developing digital signal processing programs. Although C language is very flexible for describing algorithms with complex control flows, it does not support fixed-point data formats. In this section, a floating-point to integer C program conversion procedure is explained [13, 24]. As shown in Fig. 9, the conversion flow utilizes the simulation based range estimation results for determining the number of shift operations for scaling. In addition, the number of shift operations is minimized by equalizing the IWL's of corresponding variables or constants for the purpose of reducing the execution time.

4.1 Fixed-Point Arithmetic Rules in C Programs

As summarized in Table 1, the addition or subtraction of two input data with different IWL's needs arithmetic shift before conducting the operation. Fixed-point multiplication in C language needs careful treatment because integer multiplication in ANSI C only stores the lower-half of the multiplied result, while fixed-point multiplication needs the upper-half part. Integer multiplication is intended to prevent any loss of accuracy in multiplication of small numbers, and hence it can generate an overflow when large input data are applied. However, for signal processing purpose, the upper part of the result is needed to prevent overflows and keep accuracy. Integer and fixed-point multiplication operations are compared in Fig. 10a, b [14, 15].

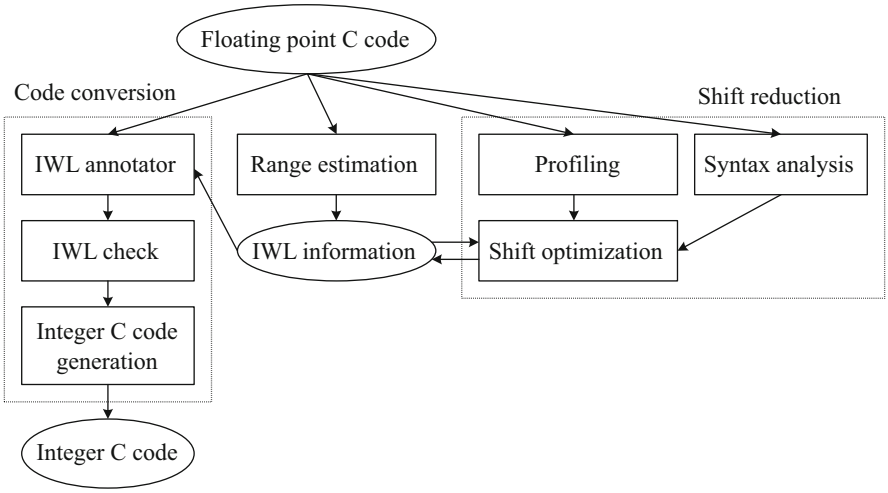


Fig. 9 Fixed-point addition with different IWL's

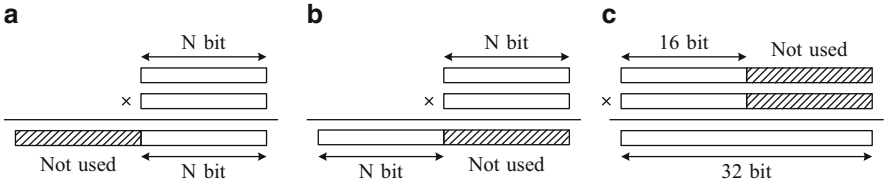


Fig. 10 Integer and fixed-point multiplications. (**a**) ANSI C integer multiplication. (**b**) Fixed-point multiplication. (**c**) MPYH instruction of TMS320C60

In traditional C compilers, a double precision multiplication operation followed by a double to single conversion is needed to obtain the upper part, which is obviously very inefficient [28]. However, in C compilers for some DSPs such as Texas Instruments' TMS320C55 ('C55), the upper part of the multiplied result can be obtained by combining multiply and shift operations [6]. In the case of TMS320C60 ('C60), which has 16 by 16-bit multipliers as well as 32-bit registers and ALU's, the multiplication of the upper 16-bit parts of two 32-bit operands is efficiently supported by C intrinsics as depicted in Fig. 10c [7]. If there is no support for obtaining the upper part of the multiplied result in the C compiler level, an assembly level implementation of fixed-point multiplication is useful. For the Motorola 56000 processor, fixed-point multiplication is implemented with a single instruction using inline assembly coding [5]. Note that, in Motorola 56000, the IWL of the multiplication result is $I_x + I_y$, because the output of the multiplier is one bit left shifted in hardware. The implementation of the macro or inline function for fixed-point multiplication, *mulh()*, is dependent on the compiler of a target processor as illustrated in Table 2.

Table 2 Implementation of fixed-point multiplication

Target processor	Implementation
TMS320C50	`#define mulh(x,y) ((x)*(y)>>16)`
TMS320C60	`#define mulh(x,y) _mpyh(x,y)`
Motorola 56000	`__inline int mulh(int x, int y) { int z; __asm("mpy %1,%2,%0":"=D"(z):"R"(x),"R"(y)); return z; }`

Fig. 11 An example of expression conversion

```
y=(a+b)*c;   ⇨   tmp = a+b;
                 y=tmp*c;
```

4.2 Expression Conversion Using Shift Operations

The most frequently used expression in digital signal processing is the accumulation of product terms, which can be generally modeled as follows.

$$x_i = \sum_{j,k} x_j \times x_k + \sum_{l} x_l \tag{6}$$

Complex expressions in C programs are converted to several expressions having this form. Figure 11 shows one example.

Assuming that there is no shifter at the output of the adder, the IWL of the added result is determined by the maximum value of two input operands and the result, as shown in Table 1. From this, the IWL of the right hand side expression, I_{rhs}, is represented by the maximum IWL of the terms as shown in Eq. (7).

$$I_{rhs} = \max_{j,k,l}(I_{x_j} + I_{x_k} + 1, I_{x_l}, I_{x_i}), \tag{7}$$

where $I_x + I_y + 1$ is used for the IWL of the multiplied results. The number of scaling shifts for the product, addition, or assignment, which is represented as, $s_{j,k}$, s_l or s_i, respectively, is determined as follows.

$$s_{j,k} = I_{rhs} - (I_{x_j} + I_{x_k} + 1) \tag{8}$$

$$s_l = I_{rhs} - I_{x_l} \tag{9}$$

$$s_i = I_{rhs} - I_{x_i} \tag{10}$$

Equation (6) is now converted to the scaled expression as follows.

$$x_i = \{\sum_{j,k}((x_j \times x_k) >> s_{j,k}) + \sum_{l}(x_l) >> s_l)\} << s_i \tag{11}$$

4.3 Integer Code Generation

The IWL information file generated in the range estimation step includes the scope of a variable that indicates whether it is global or local, the variable name and its IWL. In the automatic scaling integer program converter for C, this information is attached to the symbol table of the floating-point program. The conversion of types and expression trees is conducted in the integer C code generation stage. The symbol tables are modified to replace floating-point types with integer types. Not only the float type but also the float-based types such as pointers to float, float arrays, and float functions are converted to corresponding integer-based types. The expression tree conversion that inserts scaling shifts uses the fixed-point arithmetic rules shown in Table 1. It is performed from the bottom to the top of a parse tree, and the IWL information of each tree node is also propagated in the same way. In this step, the pointer operations that involve different IWL's are also checked.

4.3.1 Shift Optimization

In many programmable DSP's, an implementation that needs no or less scaling shift operations is not only faster but also requires a smaller code size. Since no shift operation is needed for addition or assignment of operands having the same IWL, the number of scaling shifts can be reduced by equalizing the IWL's of relevant variables. An example implementation with shift reduction is illustrated in Fig. 6a. Note that it is only allowed to increase the initial IWL's that are determined according to Eq. (2), thus the equalization can increase the quantization noise level. Scaling shift reduction requires global optimization because IWL modification of a variable in an expression can incur additional scaling shifts in other expressions. Shift optimization also depends on the architecture of a DSP. For example, if a DSP has a barrel shifter, the number of bits for one scaling shift, unless it is zero, does not affect the number of execution cycles. However, if it has no barrel shifter and should conduct the scaling by employing one-bit shift operations, the shift cost is also affected by the number of bits for one scaling operation. It is also needed for minimizing the execution time to reduce the number of scaling operations that are inside a long loop. Thus, this optimization requires program-profiling results.

The IWL modification that minimizes the overhead for scaling is conducted as follows. First, the number of shifts for each expression is formulated with the IWL's of the relevant variables and constants. Second, the cost function that corresponds to the total overhead of scaling shifts is made based on the results of the first step, the target DSP architecture, and the program-profiling information. Finally, the cost function is minimized by modifying the IWL's using the integer linear programming or the simulated annealing algorithms. Note that shift reduction using a DSP architecture without a barrel shifter can be modeled as an integer linear programming problem. The simulated annealing algorithm is a general optimization method, but the optimization can take much time. Detailed methods for shift reduction can be found in [13].

4.4 Implementation Examples

A fourth order IIR filter is implemented for TI's 'C50, 'C60 and Motorola 56000 using the developed scaling method. Floating-point C code for a fourth order IIR filter shown in Fig. 12 is given in Fig. 13a.

After the range estimation, the original IWL's are determined as shown in Table 3. With these initially determined IWL's, the integer C code shown in Fig. 13b is generated using the IWL model of $I_x + I_y + 1$ for the multiplied results. At first, the coefficients `a1`, `a2`, `b1`, and `b2` are all converted with the two's complement format having the IWL of 1. Although some coefficients, such as 0.35 or −0.75504, can be encoded with the IWL of 0, the same IWL is given to all the coefficients to reduce the number of shifts for scaling. As a result, the coefficient value of 1 is converted to 2^{30} (= 1073741824), the value of 0.355407 is translated to 0.355407×2^{30} (= 381615360), and so on. The expression `x1 = 0.01 * *x` is converted as follows. The constant '0.01' is changed to a two's complement integer '1374389534' (= 0.01×2^{37}) because the decimal number 0.01 is represented in binary as '0.0 0000 0101 0001 1110 1011 ⋯,' where there are six zeroes below the binary point. Thus, the IWL of −6 is given to this constant, and translates 0.01 to $0.01 \times 2^{(31+6)}$. The input `x` is read from the file and has the IWL of 17 with the two's complement format. The multiplied result of `a` and `x` has the IWL of 12 (= −6 + 17 + 1), but `x1` has the IWL of 10. Thus, there needs two bit left shift before assigning the multiplied result to `x1` and the expression becomes `x1=sll(mulh(1374389534, *x), 2)`. Next, since the IWL of `t1` or `d1` is 12, the IWL of `b1[0]*d1[0]` and `b1[1]*d1[1]` is 14 (= 1+12+1). At the same way, since the IWL of `x1` is 10, and that of `mulh(*b1, *d1)` or `mulh(b1[1], d1[1])` is 14, `x1` needs to be right shifted by 4 bits for addition. The added result is again assigned to `t1`, which has the IWL of 12, thus it needs two bit left shift before assigning to `t1`, and forms the equation of `t1 = sll( (x1»4)+ mulh(*b1, *d1) + mulh(b1[1], d1[1]), 2)`. Because the IWL of `t1` and `d1` is 12, the multiplication with the coefficients `a1` produces the result having the IWL of 14 (= 12 + 1 +1), and the multiplied result can be assigned to `y1` without any shift. The rest of the code can be interpreted at the same way.

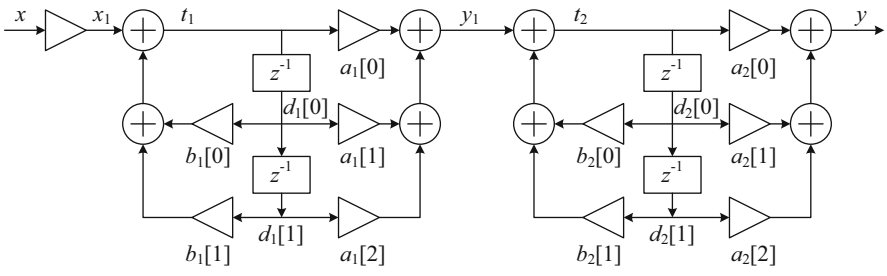


Fig. 12 A fourth order IIR filter

a

```
float a1[3] = { 1, 0.355407, 1.0 };
float a2[3] = { 1, -1.091855, 1.0 };
float b1[2] = { 1.66664, -0.75504 };
float b2[2] = { 1.58007, -0.92288 };
float d1[2], d2[2];
void iir4(float *x, float *y)
{
        float x1, y1, t1, t2;
        x1 = 0.01* *x;
        t1 = x1 + b1[0]*d1[0] + b1[1]*d1[1];
        y1 = a1[0]*t1 + a1[1]*d1[0] + a1[2]*d1[1];
        d1[1] = d1[0];
        d1[0] = t1;
        t2 = y1 + b2[0]*d2[0] + b2[1]*d2[1];
        *y = a2[0]*t2 + a2[1]*d2[0] + a2[2]*d2[1];
        d2[1] = d2[0];
        d2[0] = t2;
}
```

b

```
#define sll(x,y) ((x)<<(y))
int a1[3] =  { 1073741824, 381615360, 1073741824  };
int a2[3] =  { 1073741824, -1172370379, 1073741824  };
int b1[2] =  { 1789541073, -810718027  };
int b2[2] =  { 1696587243, -990934855  };
int d1[2];
int d2[2];
extern void iir4(int *x, int *y)
{
    int x1;
    int y1;
    int t1;
    int t2;
    x1 = sll(mulh(1374389534, *x), 2);
    t1 = sll((x1 >> 4) + mulh(*b1, *d1) + mulh(b1[1], d1[1]), 2);
    y1 = mulh(*a1, t1) + mulh(a1[1], *d1) + mulh(a1[2], d1[1]);
    d1[1] = *d1;
    *d1 = t1;
    t2 = sll((y1 >> 4) + mulh(*b2, *d2) + mulh(b2[1], d2[1]), 2);
    *y = sll(mulh(*a2, t2) + mulh(a2[1], *d2)
         + mulh(a2[2], d2[1]), 3);
    d2[1] = *d2;
    *d2 = t2;
}
```

c

```
#define sll(x,y) ((x)<<(y))
int a1[3] =  { 134217728, 47701920, 134217728  };
int a2[3] =  { 1073741824, -1172370379, 1073741824  };
int b1[2] =  { 1789541073, -810718027  };
int b2[2] =  { 1696587243, -990934855  };
int d1[2];
int d2[2];
extern void iir4(int *x, int *y)
{
    int x1;
    int y1;
    int t1;
    int t2;
    x1 = mulh(1374389534, *x);
    t1 = sll(x1 + mulh(*b1, *d1) + mulh(b1[1], d1[1]), 2);
    y1 = mulh(*a1, t1) + mulh(a1[1], *d1) + mulh(a1[2], d1[1]);
    d1[1] = *d1;
    *d1 = t1;
    t2 = sll(y1 + mulh(*b2, *d2) + mulh(b2[1], d2[1]), 2);
    *y = mulh(*a2, t2) + mulh(a2[1], *d2) + mulh(a2[2], d2[1]);
    d2[1] = *d2;
    *d2 = t2;
}
```

Fig. 13 The C codes for the fourth order IIR filter. (**a**) The floating-point C code. (**b**) The integer C code before shift reduction. (**c**) The integer C code after shift reduction

Table 3 IWL determined by the range estimation for the fourth order filter

Variable	Original IWL	Optimized IWL	IWL increment
x	17	20	3
y	15	18	3
x1	10	15	5
y1	14	18	4
t1, d1	12	13	1
t2, d2	16	16	0
b1, b2, a2	1	1	0
a1	1	4	3

Table 4 Performance comparison for the fourth order IIR filter

	# of cycles			SQNR	
	Floating-p.	Integer	Speed-up	Floating-p.	Integer (dB)
'C50	2980	100	29.8	–	49.3
'C60	3659	9	406.6	–	57.9
56000	26,282	921	28.5	–	78.5

Shift reduction is performed with a DSP architecture that employs a barrel shifter, and the integer C code shown in Fig. 13c is generated by controlling the IWL's. When considering the expression of `x1 = sll(mulh(1374389545, *x), 2)`, we can eliminate the shift left operation by increasing the IWL of `x1` by 2 bits. Note that the IWL reduction process requires simultaneous IWL modification of several variables or constants. Table 3 gives the optimized IWL's and Fig. 13c shows the optimized integer C code. Although there can be a loss of bit resolution, we can find that several redundant shift operations are successfully eliminated. In the optimized code, left shift operations are eliminated when calculating `x1` and `y`. We can also find that 4bit right shift operations are removed when using `x1` and `y1`.

In the fourth order IIR filter, the speed-up, which is the ratio in the execution time of the integer to the floating-point versions, was 29.8, 406, and 28.5 for 'C50, 'C60, and Motorola 56000, respectively, as shown in Table 4. The remarkable speed-up of 'C60 is mainly due to the deeply pipelined VLIW architecture having a large register file and the efficient C compiler. This machine can execute up to eight integer operations in one cycle and store all the variables of a small loop kernel in the registers, but needs a large number of no-operation cycles for floating-point function calls to flush pipeline registers. The compiler for 'C60 is very efficient because it has several compiler friendly components, such as large general purpose register files, an orthogonal instruction set and a VLIW scheduler [7]. The developed shift reduction technique is applied to this example. The number of shift operations in the converted C code is reduced from 7 to 2 without imposing an IWL upper bound for TI's 'C50 and 'C60. The number of shifts in the C code for Motorola 56000 is different from that of TI's DSP's, because the IWL of multiplication results is different as described in the previous section. The cycle counts of the shift reduced

Table 5 Shift reduction results of the fourth order IIR filter

IWL increment upper bound		0 (no shift reduction)	3	Infinite
# of shifts in C codes		*7*	*4*	*2*
'C50	# of cycles	100	96	94
	Speedup	–	4%	6%
	SQNR	49.3 dB	51.2 dB	54.1 dB
'C60	# of cycles	9	6	8
	speedup	–	33%	11%
	SQNR	57.9 dB	57.1 dB	54.2 dB
# of shifts in C codes		*5*	*3*	*2*
56000	# of cycles	921	675	577
	Speedup	–	27%	37%
	SQNR	78.5 dB	78.5 dB	78.5 dB

codes are shown in Table 5. As shown in this Table, 'C60 achieves 33% of speed-up increase using the shift optimization. 'C50 shows a relatively low speed-up because the shifts can be performed by load-store instructions with no additional cycle in 'C50. For the Motorola 56000, high speed-up can be achieved because its shift cost is much higher than that of the other DSP's employing barrel shifters. The SQNR of the fixed-point implementations was measured as 49.3 dB, 57.9 dB and 78.5 dB for 'C50, 'C60 and Motorola 56000, respectively. Note that 'C50 uses a 16-bit word-length for internal memory, while 'C60 supports a native 32-bit word-length, although both machines have only 16-bit multipliers. In 'C60, the upper 16 bits of the 32-bit data are multiplied to produce a 32-bit result. Thus, the 'C50 based implementation generates more quantization noise because the internal data-path truncates some of the 32-bit multiplied result. Motorola 56000 uses a 24-bit data type for both addition and multiplication. The upper 24 bits of the multiplier output are used for the multiplication results. When the IWL's are increased for the shift reduction, the fixed-point performance is slightly degraded in the 'C60 examples because the FWL's are decreased, but the 'C50 example results do not agree with our expectation. It is because the quantization noises due to the scale down shifts are eliminated. In this example, the input is scaled instead of the internal signals to reduce the scaling shifts, and it results in less quantization noise at the output signal. Since the results of the optimization as a function of the IWL increase are not simple, we need to try a few different upper bounds to find the best one.

5 Word-Length Optimization

VLSI implementation of digital signal processing algorithms requires fixed-point arithmetic for the sake of circuit area minimization, speed, and low-power consumption. Word-length optimization is also used for 16-bit programmable DSP and SIMD

processor based implementations because some SIMD arithmetic instructions for embedded processors employ shortened word-length, 8-bit or 16-bit, data for increasing the data-level parallelism. In the word-length optimization, it is necessary to reduce the quantization effects to an acceptable level without increasing the hardware cost too much. If the number of bits assigned is too small, quantization noise will degrade the system performance too much; on the other hand, if the number of bits is too large, the hardware cost will become too high. Word-length optimization for linear time-invariant systems may be conducted by analytical methods [9]. However, the simulation based approach is preferred because these methods allow not only linear but also non-linear and time-varying blocks [23]. In this sub-section, the finite word-length effects will be described, a C++ class library based fixed-point simulation tool will be presented, and the simulation based word-length optimization method will be explained.

5.1 Finite Word-Length Effects

Fixed-point arithmetic introduces quantization noise, and by which the final system performance is inevitably degraded in most cases. The *Analysis of Finite Word-Length Effects in Fixed-Point Systems* chapter of this handbook treats this problem in detail. The needed performance of a system with fixed-point arithmetic should be defined first for word-length optimization. Finite word-length effects for implementing a digital filter can be classified into coefficient and signal quantization [8]. The coefficient quantization changes the system transfer function, thus its effect can be observed by displaying the frequency response of the transfer function. Frequency responses of an FIR filter with floating-point, 12-bit, 8-bit, and 4-bit coefficients are shown in Fig. 14. The filter structure also affects the quantization effects. When implementing a high order recursive filters, the second order cascaded forms usually show better results when compared to the direct forms.

The signal quantization can be considered as adding noise, instead of distorting the system transfer function. One of the most widely used fixed-point performances is the SQNR that is defined as Eq. (12), where $y_{fl}[n]$ is the floating-point result and $y[n]$ is the fixed-point result.

$$SQNR = \frac{P_{signal}}{P_{noise}} = \frac{E[{y_{fl}}^2]}{E[(y_{fl} - y)^2]} \tag{12}$$

The SQNR is a convenient measure, but it can usually be applied to linear time-invariant systems where the output quantization noise can be modeled as additive terms.

Figure 15 shows a simple additive noise model for the first order filter shown in Fig. 5a.

Note that the quantization noise can be roughly modeled as uniformly distributed between $-\delta/2$ and $\delta/2$ when the rounding scheme is employed, where δ corresponds

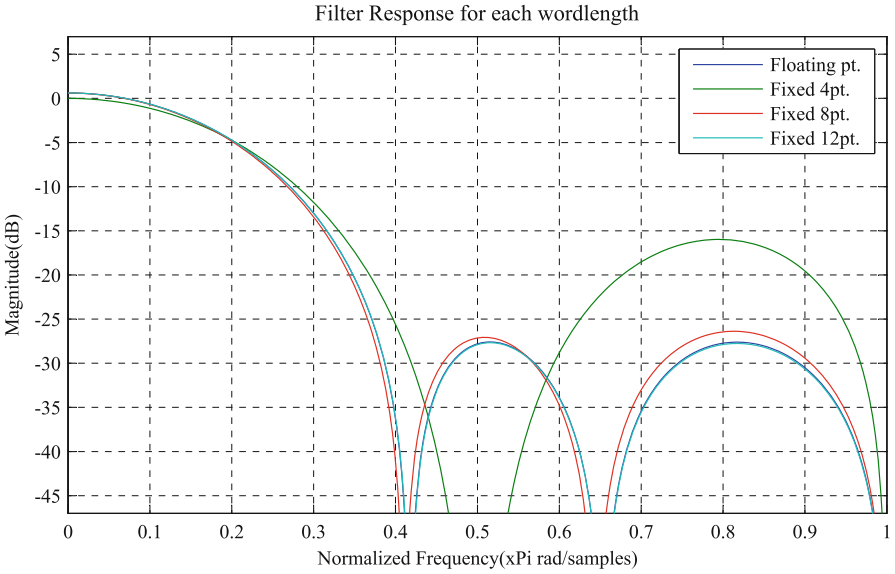
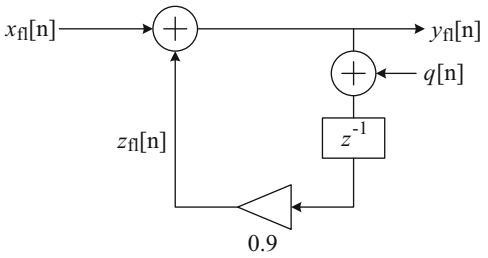


Fig. 14 Coefficient quantization effects of an FIR filter

Fig. 15 An additive noise model



to 2^{-FWL}. Thus the maximum noise amplitude is halved when increasing the FWL by 1 bit. The quantization scheme that simply eliminates the low significant bits in word-length reduction creates DC biased quantization noise, which can result in a serious problem when an accumulation circuit that has a very high DC gain is followed. Thus, as the FWL is decreased by 1 bit, the output noise power becomes quadrupled, and the SQNR is decreased by 6 dB. At the same way, the increase of the FWL by 1 bit increases the SQNR by 6 dB. For more complex systems having multiple of quantization noise sources, the additive quantization noise model can also be built in a similar way, and the output noise is the sum of all the quantization noise sources [12]. Therefore, the increase of all the fractional word-lengths by 1 bit also raises the output SQNR by 6 dB in this case, too. The SQNR as the fixed-point performance measure can also be used for waveform coders, such as the adaptive delta modulators(ADMs) or CELP vocoders; however the increase of all the fractional word-lengths by 1 bit does not reduce the output noise power

by 6 dB because these systems are not linear with respect to the quantization noise sources. For the case of an adaptive filter, word-length reduction causes slow convergence and higher steady state noise power. Thus, the fixed-point performance for optimizing an adaptive filter can be modeled as the noise power after some time-off period [22].

5.2 *Fixed-Point Simulation Using C++ `gFix` Library*

Although several analytical methods for evaluating the fixed-point performance of a digital signal processing algorithm have been developed by using the statistical model of quantization noise, they are not easily applicable to practical systems containing non-linear and time-varying blocks[26, 27]. The analysis is more complicated when a specific kind of input signal, such as speech, is required for the evaluation. In order to relieve these problems, simulation tools can be used for evaluating the fixed-point characteristics of a digital signal processing algorithm. There are a few commercially available fixed-point simulation tools for signal processing. The SPD (Signal Processing Designer) of CoWare and the MATLAB of MathWorks provide fixed-point simulation libraries [1, 4]. Mixed simulation of floating-point and fixed-point blocks is allowed with these libraries. The fixed-point block can be a simple adder or a quite complex one, such as FFT or digital filtering. In order to assign a fixed-point format for each block, it is just needed to open a block by mouse clicking and edit the fixed-point attributes for the block, such as the word-length, integer or fractional word-length, overflow or saturation mode, rounding or quantization mode, and so on.

However, the widely used C programming language does not support fixed-point arithmetic. Fixed-point simulation of a C program for signal processing can be conducted with a fixed-point class that also uses the operator overloading characteristic of the C++ language. A fixed-point data class, `gFix`, and its operators are developed to prepare a fixed-point version of a floating-point program, and to know its finite word-length and scaling effects by simulation [17]. The `gFix` class follows the generalized fixed-point format. For example, `gFix` (10, 2, "tsr") represents a format with the WL of 10, IWL of 2, and the two's complement, saturation for overflow handling, and rounding for quantization. The `gFix` class supports all of the assignment and arithmetic operations supported in C or C++ languages. There is no loss of accuracy during the fixed-point add or multiply operations. However, arithmetic right shift or arithmetic left shift may cause loss of accuracy or overflows. The assignment operator, '=,' converts the input data according to the fixed-point format of the left side variable, and assigns the format converted data to this variable. If the given format of the left side variable does not have an enough precision for representing the input data, the data is modified according to the attributes of the left side variable, such as saturation, rounding, or truncation. Since the format conversion is occurred only at the assignment operator, two programs shown in Fig. 16a, b can have different fixed-point results.

a

```
gFix a(12,0,"tsr");
gFix b(12,0,"tsr");
gFix c(12,0,"tsr");
gFix d(10,1,"tsr");

d = a + b + c;
```

b

```
gFix a(12,0,"tsr");
gFix b(12,0,"tsr");
gFix tmp(10,1,"tsr");
gFix c(12,0,"tsr");
gFix d(10,1,"tsr");

tmp = a + b;
d = tmp + c;
```

Fig. 16 Three operand addition using different architectures

Fig. 17 A fixed-point C++ program for a first order IIR filter

```
void
iir1(short argc, char *argv[])
{
  gFix Xin(12,0);
  gFix Yout(16,3);
  gFix Ydly(16,3);
  gFix Coeff(10,0);

  Coeff = 0.9;
  Ydly = 0.;
  for( i = 0; i < 1000; i++ ) {
    infile >> Xin ;
    Yout = Coeff * Ydly + Xin ;
    Ydly = Yout ;
    outfile << Yout << '\n';
  }
}
```

In Fig. 16b, the result of $a + b$ is format converted to 10 bit data, and then added to the operand c, and then format converted again. The fixed-point performance of different implementations with the same algorithm can be compared by utilizing the above characteristics.

A fixed-point simulation model of a simple IIR filter converted from the floating-point C program is shown in Fig. 17. Note that only the type of variables is converted to `gFix`, but the other parts of the program are not changed. Similar fixed-point data types are now supported with SystemC.

5.3 Word-Length Optimization Method

Word-length optimization usually tries to find the word-length vector that minimizes the hardware cost while meeting the system performance with fixed-point arithmetic. Since the optimum word-length is dependent on the desired fixed-point performance of a system, a performance measurement block has to be included as a part of a system set-up. The performance measurement block must generate a positive result (or pass) when the quantization effects are acceptable. A hardware cost library is also needed to estimate the total complexity when implementing the system with the given word-length vector. Note that not only the algorithm but also the system architecture affects the hardware cost.

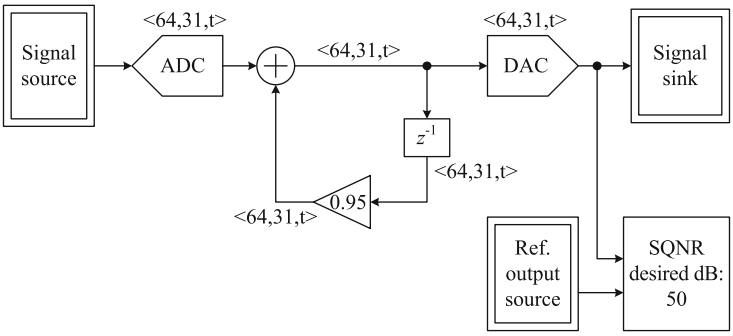


Fig. 18 Fixed-point setup for a first order filter

In the simplest case, only one word-length is used for all arithmetic operations, which is called the uniform word-length optimization. In the uniform word-length optimization, fixed-point simulation with a shorter (or longer) word-length than the optimum one should yield a fixed-point performance which is lower (or higher) than the needed performance. Thus, it is possible to arrive at the optimum word-length by increasing (or decreasing) the word-length when the obtained performance is lower (or higher) than the needed fixed-point performance. In the case of linear time-invariant systems, it is possible to reduce the number of simulations by considering that the SQNR becomes higher by 6 dB with the word-length increase of one bit.

Usually, there are multiple word-lengths to optimize in implementing a fixed-point system. As the number of word-lengths to optimize increases, optimization of them should take a longer time. In other words, minimizing the number of variables is very important for reducing the optimization time. In this optimization method, the number of different word-lengths is reduced by signal grouping that assigns the same word-length to signals, for example, connected with a delay or a multiplexer block. The word-length sensitivity of a signal needs to be considered for optimization. Some signals are very sensitive to quantization, thus they need a long word-length. The minimum bound of the word-length for each signal group is in inverse relation with the sensitivity, and can be used to reduce the search space. Finally, the optimization of different word-lengths requires a hardware cost model. The word-length optimization method in this section consists of four steps: signal grouping, sign and integer word-length determination, minimum word-length determination, and cost optimum word-length search.

As an example, Fig. 18 shows the setup for fixed-point optimization of a first order filter. The SQNR is used as the measure of the fixed-point performance.

5.3.1 Signal Grouping

The optimization method preprocesses the netlist of a signal flow block diagram to group the signals that can have the same fixed-point attributes and, as a result,

to minimize the number of variables for optimization. Both automatic and manual grouping functions are employed. The automatic grouping rules are as follows.

1. Signals connected by a delay, a multiplexer, or a switch are grouped.
2. Input and output signals of an adder or a subtractor are grouped.
3. Signals connected by a multiplier, a quantizer, or a format converter can have different fixed-point data formats unless these signals are grouped together via the other path in a signal flow block diagram.

In [18], more grouping rules were implemented to reduce the number of groups as much as possible. But, for the Fixed Point Optimizer of SPW, a manual grouping mechanism is developed instead of eliminating complex grouping rules. By manually adding a prefix to each fixed-point attribute parameter, we can combine signals into one group that would otherwise be assigned to different groups. For example, the ADC and the DAC in Fig. 7 can have the same fixed-point data format if they are manually grouped.

In high-level synthesis, it is necessary to bind operations in order to reduce the number of hardware components [20]. In this case, we can use the binding results for grouping.

5.3.2 Determination of Sign and Integer Word-Length

If the minimum value for a signal is not negative, the sign can be 'u' (unsigned). Otherwise, it should be 't' (two's complement). The integer word-length for a signal can be determined from the range of a signal, $R(x)$, using Eq. (2).

Although each signal can have a separate integer word-length, one common integer word-length is assigned to all the signals in the same group in order to lower the use of shifters for implementation. All the signs and the integer word-lengths are determined in just one simulation.

5.3.3 Determination of the Minimum Word-Length for Each Group

Assume a word-length vector $\boldsymbol{w}$, whose component is the word-length in each group.

$$\boldsymbol{w} = (w_1, w_2, \cdots, w_N), \tag{13}$$

where N is the number of groups.

The performance of a fixed-point system, such as SQNR or negative of mean squared error, is represented by $p(\boldsymbol{w})$, and the hardware cost, usually the number of gates, is $c(\boldsymbol{w})$. Then, the optimum word-length vector, $\boldsymbol{w}_{opt}$, should have the minimum value of $c(\boldsymbol{w})$ while $p(\boldsymbol{w})$ is larger than $p_{desired}$. We assume the following relation between the word-length vector and the fixed-point performance.

$$p((w_1, w_2, \cdots, w_i, \cdots, w_N)) \geq p((w_1, w_2, \cdots, w_i - 1, \cdots, w_N)), \tag{14}$$

where $1 \leq i \leq N$. The above equation represents that reducing a word-length of a group decreases, or at least does not increase, the fixed-point performance of a system. Then, the number of simulations required for the search can be reduced greatly by decomposing the procedure into two steps: the minimum word-length and the cost optimum word-length determination.

The minimum word-length for a group, $\boldsymbol{w}_{i,min}$, is the smallest word-length that satisfies the fixed-point performance of a system when the word-lengths of all other groups are very large, typically a 64 bit fixed-point or the floating-point type. By the assumption shown in Eq. (14), this minimum word-length is not larger than the optimum word-length for the group. The minimum word-length determination procedure for the first order digital filter is illustrated in Table 6. The word-length vector for this filter consists of three components, which are 'ADC,' 'DAC' and filter word-lengths. The uniform word-length that satisfies the fixed-point performance is first determined. The uniform word-length is not only useful for some architecture, such as bit-serial implementations, but also the upper limit of the minimum word-length according to the assumption in Eq. (14). Thus, the search for finding the minimum word-length for each group goes downward from the determined uniform word-length. The number of simulations required for the minimum word-

Table 6 Search sequence for the first order recursive filter

Word-length			Simulation result	Hardware cost	Comment
ADC	Filter	DAC			
16	16	16	Pass		Uniform word-length
14	14	14	Fail		Determination
15	15	15	Fail		
14	64	64	Pass		Group ADC minimum
12	64	64	Pass		Word-length determination
10	64	64	Fail		
11	64	64	Pass		
64	14	64	Fail		Group filter minimum
64	15	64	Fail		Word-length determination
64	64	14	Pass		Group DAC minimum
64	64	12	Pass		Word-length determination
64	64	10	Fail		
64	64	11	Pass		
11	16	11	Fail	8054	Minimum word-length
12	16	11	Fail	8254	Exhaustive search
11	16	12	Fail	8256	
11	17	11	Fail	8281	
13	16	11	Fail	8454	
12	16	12	Fail	8456	
11	16	13	Fail	8458	
12	17	11	Pass	8481	Determined word-length

length determination procedure is typically two to four times the number of groups. The minimum word-length determination does not require the hardware cost model.

5.3.4 Determination of the Minimum Hardware Cost Word-Length Vector

The word-length vector that satisfies the system performance while requiring the minimum hardware cost is determined at the next step. If the simulation using the minimum word-length vector, $\boldsymbol{w}_{min}$, satisfies the desired performance, the minimum word-length vector $\boldsymbol{w}_{min}$ becomes the optimum word-length vector, $\boldsymbol{w}_{opt}$. If not, the word-length is increased, and the fixed-point system performance is measured by simulation. Three search strategies, an exhaustive and two heuristic methods, are developed for this step. The search sequences for the first order recursive filter will be illustrated as an example. Usually, the optimum word-length for each group is, at most, two bits larger than the minimum word-length.

The hardware cost of the components for implementing a DSP system, such as ADC, adder, multiplier, and delay, has to be known for the minimum cost word-length optimization of a system. For example, when the price of an ADC is high, it may be justified to minimize the word-length of the ADC by increasing the word-lengths for multipliers and adders. Adders, constant coefficient multipliers, and delays are modeled to consume the hardware resources in proportion to their word-lengths. The number of gates for a multiplier is dependent on the product of two input word-lengths. Note that the hardware cost model is affected not only by the process technology but also by the implementation architecture. When a signal processing algorithm is implemented in a time-domain multiplexed mode, the per-bit cost of arithmetic blocks that are shared can be lowered. Since the hardware cost model is stored at an external file, a designer can assign an appropriate value to each component according to the implementation architecture and the ASIC library. In this example, the hardware cost model using VLSI Technologies' cell library is used, and the hardware cost of 200 and 202 is assumed for each bit of ADC and DAC, respectively. According to these models, the hardware cost increase for each bit is 200 for group 'ADC,' 227 for group 'filter,' and 202 for group 'DAC.'

In the exhaustive search algorithm, the word-length vector that requires the least cost is selected in a priority for the fixed-point performance measurement starting from the minimum word-length vector. For example, the word-length for group 'ADC' will be increased first as shown in Table 6. If the test is failed, the word-length for group 'DAC' will be increased instead of group 'ADC.' The search sequence using the exhaustive algorithm is shown in Table 6. The minimum cost word-length vector determined is (12, 17, 11), and the hardware cost required is 8481. Although the result is the minimum cost solution, this method demands many fixed-point performance measurements. The search procedure is a combinatorial algorithm as a function of the number of groups and is practical only when the number of groups is small, usually less than 6.

Table 7 Search sequences of performance/cost directed heuristic method

Word-length			Performance	Hardware	
ADC	Filter	DAC	SQNR (dB)	cost	Comment
11	16	11	34.27	8054	Minimum word-length
12	16	11	38.56	8254	Selected
11	17	11	34.83	8281	
11	16	12	34.59	8256	
13	16	11	39.65	8454	
12	17	11	40.36	8481	Determined word-length
12	16	12	37.43	8456	

In the first heuristic search algorithm, all the word-lengths are increased by one, e.g. the word-length vector is increased to (12, 17, 12) from the minimum word-length vector (11, 16, 11), and the fixed-point performance is measured. If the result is not satisfactory, the above procedure is repeated. This step requires typically one to two simulations. After then, the word-length for a group whose cost saving is the greatest is decreased. If it does not satisfy the performance, the word-length is restored. This procedure is conducted for all the group, and, as a result, requires N simulations. The maximum number of simulations required for the heuristic search based word-length optimization, including the minimum word-length determination, is only linearly proportional to the number of groups, N. The word-length vector determined for the first order digital filter shown in Fig. 7 is (12, 17, 11), which is the same as the minimum cost word-length. According to the experiments using eight examples, the additional hardware cost is usually less than 5% of that required for the exhaustive search algorithm.

In the second heuristic search algorithm, the word-length vector that shows the best ratio of performance increment to cost increase is selected. The word-length of each group is increased by one bit, and the fixed-point performance is measured using simulations for each case, and the best result is selected for the next iteration until the desired fixed-point performance is satisfied. Since the fixed-point performance increment is used in this method, this heuristic can be applied when the fixed-point performance can be quantified such as SQNR. When the fixed-point performance is tested using binary decision, e.g. pass or fail, this method cannot be used. The search sequences of this heuristic method for the first order recursive filter is shown in Table 7.

5.4 Optimization Example

The impulse response of a channel can be identified by using an adaptive digital filter which adjusts the coefficients to minimize the channel modeling error. When the word-length of the filter coefficients or the input data is not sufficient, the system

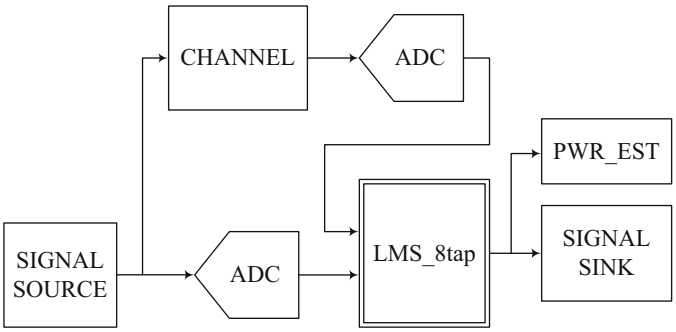
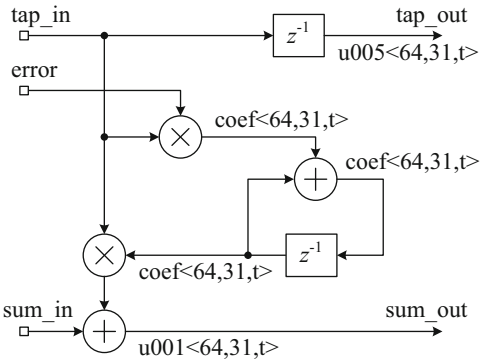


Fig. 19 An LMS channel identification system

Fig. 20 Grouping result for one tap of LMS adaptive filter



not only converges slowly but also the magnitude of error becomes large. Thus, as shown in Fig. 19, a power estimation block that measures the average power of the error signal after some time-off period is used for the fixed-point performance measure [22].

When we apply the automatic grouping rules, the coefficient in each tap becomes a separate group. In order to reduce the number of groups, the coefficients are manually grouped by assigning a fixed-point attribute of 'coef.' The grouping result for each tap is shown in Fig. 20.

The 'tap_in' and 'tap_out' signals are automatically grouped because they are connected by a delay, and assigned to 'u005.' The 'sum_in' and 'sum_out' signals are also grouped, and assigned to the 'u001' group. The integer word-length, minimum word-length, and optimum word-length for each group of signals are shown in Table 8. The optimization results are compared for three search algorithms: uniform word-length, exhaustive search, heuristic search based on uniformly increasing the minimum word-length vector. The uniform word-length optimization requires 13 bits for all the signals and the hardware cost required is 96,265 gates. The exhaustive and heuristic search algorithms, both are based on the minimum cost criterion, need the cost of 46,278 and 49,520 gates, respectively. The hardware cost required using the minimum cost optimization is just 48% of

Table 8 Determined fixed-point attributes for the LMS adaptive filter

Group	Signal	Integer word-length	Minimum word-length	Optimum word-length
u001	Sum	3	12	12
u005	Tap	2	7	7
u002	Error	−4	5	5
coef	Coefficient	1	13	13
u004	ADC (channel)	3	8	10
u006	ADC (source)	2	7	13

that needed for the uniform word-length determination in this example. The number of simulations for searching the optimum word-length vector from the minimum word-length vector is 27 and 5 for the exhaustive and heuristic search algorithms, respectively.

6 Summary and Related Works

Fixed-point hardware or integer arithmetic based implementation of digital signal processing algorithms is important for not only hardware cost minimization but also power consumption reduction. The conversion of a floating-point algorithm into a fixed-point or an integer version has been considered very time-consuming; it often takes more than 50% of the algorithm to hardware or software implementation procedure [25]. The fixed-point format discussed in Sect. 2 bridges the gap between the floating-point and fixed-point data types, and allows seamless and incremental conversion. Note that incremental conversion can help a designer very much because the effects of each small conversion can readily be verified. There are other fixed-point formats used for similar purposes, and one of them is the Q format [2]. In the Q format, the integer and fractional bits are given, while the sign is usually implied as the two's complement. For example, 'Q1,30' describes a fixed-point data with 1 integer bit and 30 fractional bits stored as a 32-bit two's complement number. The Q format has been used widely for assembly programming of Texas Instruments digital signal processors.

The simulation based range estimation and automatic conversion process becomes more and more popular as the computing power for simulation becomes cheaper. The FRIDGE, a fixed-point design and simulation environment, supports interpolative approach to reduce the range estimation overhead [25, 29]. This tool also supports integer and VHDL code generation path from a floating-point C program. In these days, application oriented language, such as Matlab from MathWorks, is frequently used for signal processing application development. The Simulink package for Matlab and SPD (Signal Processing Designer) of CoWare support easy to use, GUI (Graphic User Interface) supported, fixed-point arithmetic [1, 4]. With these tools, it is possible to convert a floating-point design into a fixed-point version in an incremental and convenient way.

For the word-length optimization of VLSI and FPGA based design, several different search methods have been studied recently [10, 21]. For linear systems, the word-length optimization is converted to an integer programming problem by exploiting the additive quantization noise model [9]. A fixed-point optimization flow that conducts both high-level synthesis and fixed-point optimization simultaneously has been developed [19]. As one of the future works, integer code generation for SIMD (Single Instruction Multiple Data) architecture is needed; this work requires combined processing of scaling, word-length optimization, and automatic vectorization. Also, fast word-length optimization of very large systems is an interesting problem. The word-length optimization or scaling software needs to be integrated into widely used CAD tools. The capability of fixed-point optimization tools currently supported by commercial CAD software can hardly meet users' expectation.

References

1. URL http://www.coware.com/products/signalprocessing.php
2. URL http://en.wikipedia.org/wiki/Q_(number_format)
3. Fixed-Point C++ class. URL http://msl.snu.ac.kr/fixim/
4. Simulink. URL http://www.mathworks.com/products/simulink/
5. DSP56KCC User's Manual. Motorola Inc. (1992)
6. TMS320C2x/C2xx/C5x Optimizing C Compiler (Version 6.60). Texas Instruments Inc., TX (1995)
7. TMS320C6x Optimizing C Compiler. Texas Instruments Inc., TX (1997)
8. Catthoor, F., Vandewalle, J., Man, H.D.: Simulated Annealing based Optimization of Coefficient and Data Word-Lengths in Digital Filters. Int. J. Circuit Theory and Applications **16**, 371–390 (1988)
9. Constantinides, G., Cheung, P., Luk, W.: Wordlength optimization for linear digital signal processing. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **22**(10), 1432–1442 (2003).
10. Han, K., Evans, B.L.: Optimum wordlength search using sensitivity information. EURASIP J. Appl. Signal Process. **2006**, 76–76 (January).
11. Han, K., Olson, A., Evans, L.: Automatic floating-point to fixed-point transformations. In: Signals, Systems and Computers, 2006. ACSSC '06. Fortieth Asilomar Conference on, pp. 79–83 (2006).
12. Jackson, L.B.: On the Interaction of Roundoff Noise and Dynamic Range in Digital Filters. The Bell System Technical Journal pp. 159–183 (1970)
13. K. Kum, J.K., Sung, W.: AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors. IEEE Trans. Circuits and Systems-II: Analog and Digital Signal Processing **47**(9), 840–848 (2000)
14. Kang, J., Sung, W.: Fixed-point C language for digital signal processing. In: Proc. of the 29th Annual Asilomar Conference on Signals, Systems and Computers, vol. 2, pp. 816–820 (1995)
15. Kang, J., Sung, W.: Fixed-point C compiler for TMS320C50 digital signal processors. In: Proc. of 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 707–710 (1997)
16. Kim, S., Sung, W.: A Floating-point to Fixed-point Assembly Program Translator for the TMS 320C25. IEEE Trans. on Circuits and Systems **41**(11), 730–739 (1994)

17. Kim, S., Sung, W.: Fixed-point optimization utility for C and C++ based digital signal processing programs. IEEE Trans. on Circuits and Systems (will be published)
18. Kum, K.I., Sung, W.: VHDL based Fixed-point Digital Signal Processing Algorithm Development Software. In: Proceeding of International Conference on VLSI and CAD '93, pp. 257–260. Korea (1993)
19. Kum, K.I., Sung, W.: Combined word-length optimization and high-level synthesis of digital signal processing systems. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **20**(8), 921–930 (2001).
20. Micheli, G.D.: Synthesis and Optimization of Digital Circuits. McGraw-Hill, Inc., NJ (1994)
21. Shi, C., Brodersen, R.: Automated fixed-point data-type optimization tool for signal processing and communication systems. In: Design Automation Conference, 2004. Proceedings. 41st, pp. 478–483 (2004)
22. Sung, W., Kum, K.I.: Word-Length Determination and Scaling Software for a Signal Flow Block Diagram. In: Proceeding of the International Conference on Acoustics, Speech, and Signal Processing '94, vol. 2, pp. 457–460. Adelaide, Australia (1994)
23. Sung, W., Kum, K.I.: Simulation-Based Word-Length Optimization Method for Fixed-Point Digital Signal Processing Systems. IEEE Trans. on Signal Processing **43**(12), 3087–3090 (1995)
24. Willems, M., Bürsgens, V., Grötker, T., Meyr, H.: FRIDGE: An interactive code generation environment for HW/SW codesign. In: Proc. of 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 287–290 (1997)
25. Willems, M., Bürsgens, V., Meyr, H.: FRIDGE: Floating-point programming of fixed-point digital signal processors. In: Proc. of the International Conference on Signal Processing Applications and Technology (1997)
26. Wong, P.W.: Quantization and roundoff noises in fixed-point FIR digital filters. IEEE Trans. Signal Processing **39**, 1552–1563 (1991)
27. Yun, I.D., Lee, S.U.: On the fixed-point error analysis of several fast DCT algorithms. IEEE Trans. Circuits and Systems for Video Technology **3**(1), 27–41 (1993)
28. Zivŏjnovic, V.: Compilers for Digital Signal Processors. DSP & Multimedia Technology **4**(5), 27–45 (1995)
29. H. Keding: Pain killers for fixed-point design flow. Technical Report, Synopsys (2010)

Dynamic Dataflow Graphs

Bart D. Theelen, Ed F. Deprettere, and Shuvra S. Bhattacharyya

Abstract Much of the work to date on dataflow models for signal processing system design has focused on decidable dataflow models. This chapter reviews more general dataflow modeling techniques targeted to applications that include dynamic dataflow behavior. The complexity in such applications demands for increased degrees of agility and flexibility in dataflow models. With the application of dataflow techniques addressing these challenges, interest in classes of more general dataflow models has risen correspondingly. We first provide a motivation for dynamic dataflow models of computation, and review a number of specific methods that have emerged in this class of models. The dynamic dataflow models covered in this chapter are Boolean Dataflow, CAL, Parameterized Dataflow, Enable-Invoke Dataflow, Scenario-Aware Dataflow, and Dynamic Polyhedral Process Networks.

1 Motivation for Dynamic DSP-Oriented Dataflow Models

The decidable dataflow models covered in [30] are useful for their predictability, strong formal properties, and amenability to powerful optimization techniques. However, for many signal processing applications, it is not possible to represent all of the functionality in terms of purely decidable dataflow representations. For example, functionality that involves conditional execution of dataflow subsystems

B. D. Theelen (✉)
Océ Technologies B.V., Venlo, The Netherlands
e-mail: bart.theelen@oce.com

Ed F. Deprettere
Leiden Embedded Research Center, Leiden University Leiden Institute Advanced Computer Science, Leiden, The Netherlands
e-mail: edd@liacs.nl

S. S. Bhattacharyya
Department of ECE and UMIACS, University of Maryland, College Park, MD, USA

Laboratory for Pervasive Computing, Tampere University of Technology, Tampere, Finland
e-mail: ssb@umd.edu

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_32

or actors with dynamically varying production and consumption rates can in general not be expressed with decidable dataflow models.

The need for expressive power beyond what decidable dataflow techniques provide is becoming increasingly important in the design and implementation of signal processing systems. This is due to the increasing levels of application dynamics that must be supported in such systems. Examples include the need to support multi-standard and other forms of multi-mode signal processing operation; variable data rate processing; and complex forms of adaptive signal processing behaviors.

Intuitively, *dynamic dataflow* models can be viewed as dataflow modeling techniques in which the production and consumption rates of actors can vary in ways that are not entirely predictable at compile time. It is possible to define dynamic dataflow modeling formats that are decidable. For example, by restricting the types of dynamic dataflow actors, and by restricting the usage of such actors to a small set of graph patterns or "schemas". Gao, Govindarajan, and Panangaden defined the class of *well-behaved dataflow graphs*, which provides a dynamic dataflow modeling environment that is amenable to compile-time bounded memory verification [18].

Most existing DSP-oriented dynamic dataflow modeling techniques do not provide decidable dataflow modeling capabilities. In other words, in exchange for the increased modeling flexibility (expressive power), one must typically give up guarantees on compile-time buffer underflow (deadlock) and overflow validation. In dynamic dataflow environments, analysis techniques may succeed in guaranteeing avoidance of buffer underflow and overflow for a significant subset of specifications. However, in general, specifications may exist that "break" these analysis techniques in the sense that compile-time analysis gives inconclusive results.

Dynamic dataflow techniques can be divided into two general classes:

- Models formulated explicitly in terms of interacting combinations of state machines and dataflow graphs. In this case, the dataflow dynamics are represented directly in terms of transitions within one or more underlying state machines
- Models where the dataflow dynamics are represented using alternative means

The separation in this dichotomy can become somewhat blurry for models that have a well-defined state structure governing the dataflow dynamics, but whose design interface does not expose this structure directly to the programmer. Dynamic dataflow techniques in the first category are covered in [30]—in particular, those based on explicit interactions between dataflow graphs and finite state machines. This chapter focusses on the second category.[1] Specifically, dynamic dataflow modeling techniques that involve different kinds of modeling abstractions, apart from state transitions, as the key mechanisms for capturing dataflow behaviors and their potential for run-time variation.

[1] Except for the Scenario Aware Dataflow model in Sect. 6.

Numerous dynamic dataflow modeling techniques have evolved over the past couple of decades. A comprehensive coverage of these techniques, even after excluding the "state-centric" ones, is out of the scope this chapter. The objective is to provide a representative cross-section of relevant dynamic dataflow techniques. The emphasis is on techniques for which useful forms of compile-time analysis methods have been developed. Such techniques can be important for exploiting the specialized properties exposed by these models, and improving predictability and efficiency when deriving simulations or implementations.

2 Boolean Dataflow

The *Boolean Dataflow* (BDF) model of computation extends *Synchronous Dataflow* (SDF) with a class of dynamic dataflow actors in which production and consumption rates on actor ports can vary as two-valued functions of *control tokens*. Such control tokens are consumed from or produced onto designated *control ports* of dynamic dataflow actors. An actor input port is referred to as a *conditional input port* if its consumption rate can vary in such a way. Similarly an output port with a dynamically varying production rate under this model is referred to as a *conditional output port*. Given a conditional input port p of a BDF actor A, there is a corresponding input port C_p, called the *control input* for p. The consumption rate on C_p is statically fixed at one token per invocation of A. The number of tokens consumed from p during a given invocation of A is a two-valued function of the data value that is consumed from C_p during the same invocation. The dynamic dataflow behavior for a conditional output port is characterized in a similar way, except that the number of tokens produced on such a port can be a two-valued function of a token consumed from a control input port or of a token produced onto a *control output* port. If a conditional output port q is controlled by a control output port C_q, then the production rate on the control output is statically fixed at one token per actor invocation. The number of tokens produced on q during a given invocation is a two-valued function of the data value produced onto C_q during the same invocation of the actor.

Two fundamental dynamic dataflow actors in BDF are the *switch* and *select* actors, which are illustrated in Fig. 1a. The switch actor has two input ports, a control input port w_c and a *data* input port w_d, and two output ports w_x and w_y. The port w_c accepts Boolean valued tokens, and the consumption rate on w_d is statically fixed at one token per actor invocation. On a given invocation of a switch actor, the data value consumed from w_d is copied to a token that is produced on either w_x or w_y depending on the Boolean value consumed from w_c. If this Boolean value is `true`, then the value from the data input is routed to w_x, and no token is produced on w_y. Conversely, if the control token value is `false`, then the value from w_d is routed to w_y with no token produced on w_x.

A BDF select actor has a single control input port s_c, two additional input ports (*data input ports*) s_x and s_y, and a single output port s_o. Similar to the control port

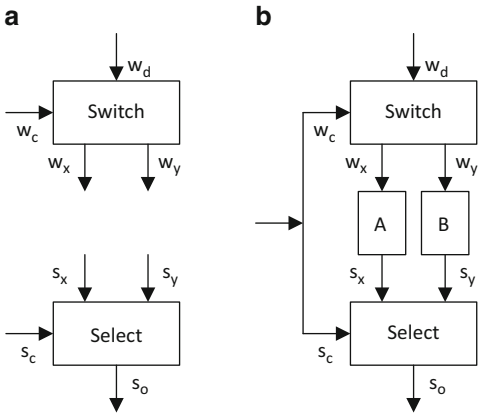


Fig. 1 (**a**) Switch and select actors in Boolean dataflow, and (**b**) An if-then-else construct expressed in terms of Boolean dataflow

of the switch actor, port s_c accepts Boolean valued tokens, and the production rate on s_o is statically fixed at one token per invocation. On each invocation of the select actor, data is copied from a single token from either s_x or s_y to s_o depending on whether the corresponding control token value is `true` or `false` respectively.

Switch and select actors can be integrated along with other actors in various ways to express different kinds of control constructs. For example, Fig. 1b illustrates an if-then-else construct, where the actors A and B are applied conditionally based on a stream of control tokens. Here A and B are SDF actors that each consume one token and produce one token on each invocation.

Buck has developed scheduling techniques to automatically derive efficient control structures from BDF graphs under certain conditions [9]. Buck also showed that BDF is Turing complete, and furthermore, that SDF augmented with just switch and select (and no other dynamic dataflow actors) is also Turing complete. This latter result provides a convenient framework to demonstrate Turing completeness for other kinds of dynamic dataflow models, such as the *Enable-Invoke Dataflow* (EIDF) model described in Sect. 5. In particular, if a given model of computation can express all SDF actors as well as the functionality associated with the BDF switch and select actors, then such a model can be shown to be Turing complete.

3 CAL

In addition to providing a dynamic dataflow model of computation that is suitable for signal processing system design, *CAL* provides a complete programming language. It is furthermore supported by a growing family of development tools for hardware and software implementation. The name "CAL" is derived as a self-referential acronym for the *CAL Actor Language*. CAL was developed by Eker and Janneck at U. C. Berkeley [13]. It has since evolved into an actively-developed, widely-investigated language for design and implementation of embedded software

and field programmable gate array applications (e.g., see [29, 56, 78]). One of the most notable developments to date in the evolution of CAL has been its adoption as part of the recent MPEG standard for *Reconfigurable Video Coding* (RVC) [6].

A CAL program is specified as a network of CAL actors, where each actor is a dataflow component that is expressed in terms of a general underlying form of dataflow. This general form of dataflow admits both static and dynamic behaviors, and even non-deterministic behaviors. Like typical actors in any dataflow programming environment, a CAL actor in general has a set of input ports and a set of output ports that define interfaces to the enclosing dataflow graph. A CAL actor also encapsulates its own private state, which can be modified by the actor as it executes but cannot be modified directly by other actors.

The functional specification of a CAL actor is decomposed into a set of *actions*, where each action can be viewed as a template for a specific class of firings or invocations of the actor. Each firing of an actor corresponds to a specific action and executes based on the code that is associated with that action. The core functionality of actors therefore is embedded within the code of the actions. Actions can in general consume tokens from actor input ports, produce tokens on output ports, modify the actor state, and perform computation in terms of the actor state and the data obtained from any consumed tokens.

The number of tokens produced and consumed by each action with respect to each actor output and input port, respectively, is declared up front as part of the declaration of the action. An action need not consume data from all input ports nor must it produce data on all output ports. However, the ports with which the action exchanges data, and the associated rates of production and consumption must be constant for the action. Across different actions, however, there is no restriction of uniformity in production and consumption rates. This flexibility enables the modeling of dynamic dataflow in CAL.

A CAL actor A can be represented as a sequence of four elements $\sigma_0(A)$, $\Sigma(A)$, $\Gamma(A)$, $pri(A)$, where $\Sigma(A)$ represents the set of all possible values that the state of A can take. $\sigma_0(A) \in \Sigma(A)$ represents the initial state of the actor, before any actor in the enclosing dataflow graph has started execution. $\Gamma(A)$ represents the set of actions of A. Finally, $pri(A)$ is a partial order relation, called the *priority relation* of A, on $\Gamma(A)$ that specifies relative priorities between actions.

Actions execute based on associated *guard conditions* as well as the priority relation of the enclosing actor. More specifically, each actor has an associated guard condition, which can be viewed as a Boolean expression in terms of the values of actor input tokens and actor state. An actor A can execute whenever its associated guard condition is satisfied (*true*-valued), and no higher-priority action (based on the priority relation $pri(A)$) has a guard condition that is also satisfied.

In summary, CAL is a language for describing dataflow actors in terms of ports, actions (firing templates), guards, priorities, and state. This finer, *intra-actor* granularity of formal modeling within CAL allows for novel forms of automated analysis for extracting restricted forms of dataflow structure. Such restricted forms of structure can be exploited with specialized techniques for verification or synthesis to derive more predictable or efficient implementations.

An example of the capability for *specialized region detection* in CAL programs is the technique of deriving and exploiting so-called *Statically Schedulable Regions* (SSRs) [29]. Intuitively, an SSR is a collection of CAL actions and ports that can be scheduled and optimized statically using the full power of static dataflow techniques, such as those available for SDF, and integrated into the schedule for the overall CAL program through a top-level dynamic scheduling interface.

SSRs can be derived through a series of transformations that are applied on intermediate graph representations. These representations capture detailed relationships among actor ports and actions, and provide a framework for effective *quasi-static scheduling* of CAL-based dynamic dataflow representations. Quasi-static scheduling is the construction of dataflow graph schedules in which a significant proportion of overall schedule structure is fixed at compile-time. Quasi-static scheduling has the potential to significantly improve predictability, reduce run-time scheduling overhead, and as discussed above, expose subsystems whose internal schedules can be generated using purely static dataflow scheduling techniques.

A further discussion of CAL can be found in [43], which presents the application of CAL to reconfigurable video coding.

4 Parameterized Dataflow

Parameterized Dataflow is a meta-modeling approach for integrating dynamic parameters and run-time adaptation of parameters in a structured way into the class of dataflow models of computations that have a well-defined concept of a graph *iteration* [4]. For example, SDF and *Cyclo-Static* SDF (CSDF), which are discussed in [30], and *Multi-Dimensional* SDF (MDSDF), which is discussed in [39], have well defined concepts of iterations based on solutions to the associated forms of balance equations. Each of these models can be integrated with Parameterized Dataflow to provide a dynamically parameterizable form of the original model.

When Parameterized Dataflow is applied to generalize a specialized dataflow model such as SDF, CSDF, or MDSDF, the specialized model is referred to as the *base model*. The resulting dynamically parameterizable form of the base model is referred to as *parameterized* X, where X denotes the base model. For example, when Parameterized Dataflow is applied to SDF as the base model, the resulting model of computation is called *Parameterized Synchronous Dataflow* (PSDF). PSDF is significantly more flexible than SDF as it allows arbitrary parameters of SDF graphs to be modified at run-time. Furthermore, PSDF provides a useful framework for quasi-static scheduling, where fixed-iteration looped schedules—such as single appearance schedules [5] for SDF graphs—can be replaced by *parameterized looped schedules* [4, 41]. In such parameterized schedules, loop iteration counts are represented as symbolic expressions in terms of variables whose values can be adapted dynamically through computations derived from the enclosing PSDF specification.

Intuitively, Parameterized Dataflow allows arbitrary attributes of a dataflow graph to be parameterized, with each parameter characterized by an associated domain of admissible values that the parameter can take on at any given time. Graph attributes that can be parameterized include scalar or vector attributes of individual actors, such as the coefficients of a finite impulse response filter or the block size associated with an FFT. Also edge attributes, like the delay of an edge or the data type associated with tokens transferred across the edge, can be parameterized. A final parameterization example are graph attributes related to numeric precision, which may be passed down to selected subsets of actors and edges within the given graph.

The Parameterized Dataflow representation of a computation involves three cooperating dataflow graphs, which are referred to as the *body*, *subinit*, and *init* graphs. The body graph typically represents the functional "core" of the computation, while the subinit and init graphs are dedicated to managing the parameters of the body graph. In particular, each output port of the subinit graph is associated with a body graph parameter such that data values produced at the output port are propagated as new parameter values of the associated parameter. Similarly, output ports of the init graph are associated with parameter values in the subinit and body graphs.

Changes to body graph parameters, which occur based on new parameter values computed by the init and subinit graphs, cannot occur at arbitrary points in time. Instead, once the body graph begins execution it continues uninterrupted through a graph iteration, where the specific notion of an iteration in this context can be specified by the user in an application-specific way. For example, in PSDF, the most natural and general definition for a body graph iteration would be a single SDF *iteration* of the body graph (as defined by the SDF repetitions vector [30]).

An iteration of the body graph can however also be defined as some constant number of iterations, e.g., the number of iterations required to process a fixed-size block of input data samples. Furthermore, parameters that define the body graph iteration can be used to parameterize the body graph or the enclosing PSDF specification at higher levels of the model hierarchy. In this way, the processing that is defined by a graph iteration can itself be dynamically adapted as the application executes. For example, the duration (or block length) for fixed-parameter processing may be based on the size of a related sequence of contiguous network packets, where the sequence size determines the extent of the associated graph iteration.

Body graph iterations can even be defined to correspond to individual actor invocations. This can be achieved by defining an individual actor as the body graph of a parameterized dataflow specification, or by simply defining the notion of iteration for an arbitrary body graph to correspond to the *next actor firing* in the graph execution. Thus, when modeling applications with parameterized dataflow, designers have significant flexibility to control the windows of execution that define the boundaries at which graph parameters can be changed.

A combination of cooperating body, init, and subinit graphs is called a PSDF *specification*. PSDF specifications can be abstracted as PSDF actors in higher level PSDF graphs, and in this way, PSDF specifications can be integrated hierarchically.

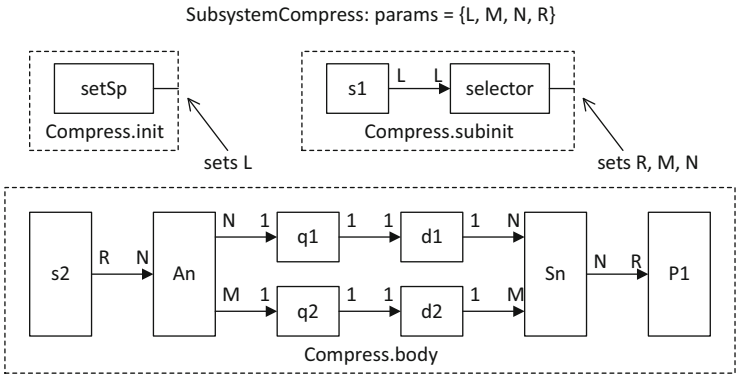


Fig. 2 An illustration of a speech compression system that is modeled using PSDF semantics. This illustration is adapted from [4]

Figure 2 illustrates a PSDF specification for a speech compression system. This illustration is adapted from [4]. Here, *setSp* ("set speech") is an actor that reads a header packet from a stream of speech data, and configures L, which is a parameter representing the length of the next speech instance to process. The *s1* and *s2* actors are input interfaces that inject successive samples of the current speech instance into the dataflow graph. The actor *s2* zero-pads each speech instance to a length R ($R \geq L$) so that the resulting length is divisible by N, which is the speech segment size. The *An* ("analyze") actor performs linear prediction on speech segments, and produces corresponding auto-regressive (AR) coefficients (in blocks of M samples), and residual error signals (in blocks of N samples) on its output edges. The actors *q1* and *q2* represent quantizers, and complete the modeling of the transmitter component of the body graph. The receiver side functionality is modeled in the body graph starting with the actors *d1* and *d2*, which represent dequantizers. The actor *Sn* ("synthesize") reconstructs speech instances using corresponding blocks of AR coefficients and error signals. Actor *P1* ("play") represents an output interface for playing or storing the resulting speech instances.

The model order (number of AR coefficients) M, speech segment size N, and zero-padded speech segment length R are determined on a per-segment basis by the *selector* actor in the subinit graph. Existing techniques, such as the Burg segment size selection algorithm and AIC order selection criterion [32] can be used for this.

The model of Fig. 2 can be optimized to eliminate the zero padding overhead (modeled by the parameter R). This optimization can be performed by converting the design to a *Parameterized Cyclo-Static Dataflow* (PCSDF). In PCSDF, Parameterized Dataflow is integrated with CSDF as the base model instead of SDF.

Further details on the exemplified speech compression application and its representations in PSDF and PCSDF, the semantics of Parameterized Dataflow and PSDF, and quasi-static scheduling techniques for PSDF can be found in [4]. Parameterized Cyclo-Static Dataflow (PCSDF), the integration of Parameterized Dataflow meta-modeling with Cyclo-Static Dataflow, is explored further in [57].

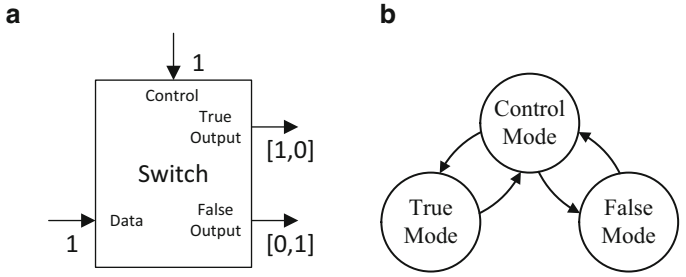
The exploration of different models of computation, including PSDF and PCSDF, for modeling software defined radio systems is described in [3]. In [38], Kee et al. explore the application of PSDF techniques to field programmable gate array implementation of the physical layer for 3GPP-Long Term Evolution (LTE). The integration of concepts related to parameterized dataflow in language extensions for embedded streaming systems is described in [42]. General techniques for analysis and verification of hierarchically reconfigurable dataflow graphs are explored in [47].

5 Enable-Invoke Dataflow

Enable-Invoke Dataflow (EIDF) is another DSP-oriented dynamic dataflow modeling technique [51]. The applicability of EIDF has been demonstrated in the context of behavioral simulation, FPGA implementation, and prototyping of different scheduling strategies [49–51]. This latter capability—prototyping of scheduling strategies—is particularly important in analyzing and optimizing embedded software. The importance and complexity of carefully analyzing scheduling strategies are high, even for the restricted SDF model where scheduling decisions have a major impact on key implementation metrics [7]. The incorporation of dynamic dataflow features makes the scheduling problem more critical since application behaviors are less predictable and more difficult to understand through analytical methods.

EIDF is based on a formalism in which actors execute through dynamic transitions among *modes*. Although each mode has constant production/consumption rate behavior, different modes can have different dataflow rates. Unlike other forms of mode-oriented dataflow specification, SDF-integrated starcharts [30], SysteMoc [14], and CAL (see Sect. 3), EIDF imposes a strict separation between fireability checking (checking whether or not the next mode has sufficient data to execute), and mode execution (carrying out the execution of a given mode). This allows for lightweight fireability checking since it is completely separated from mode execution. Furthermore, the approach improves predictability of mode executions since there is no waiting for data (blocking reads)—the time required to access input data is not affected by scheduling decisions or global dataflow graph state.

For a given EIDF actor, the specification for each mode of the actor includes the number of tokens that is consumed on each input port, the number of tokens that is produced on each output port, and the computation (the *invoke function*) that is to be performed when the actor is invoked in the given mode. The specified computation must produce the designated number of tokens on each output port. The invoke function must also produce a value for the *next mode* of the actor, which determines the number of input tokens required for and the computation to be performed during the next actor invocation. The next mode can in general depend on the current mode as well as the input data that is consumed as the mode executes.



c

Mode	Control Input	Data Input	True Output	False Output
Control Mode	1	0	0	0
True Mode	0	1	1	0
False Mode	0	1	0	1

Fig. 3 An illustration of the design of a `switch` actor in CFDF. (**a**) Switch actor. (**b**) Mode transition diagram. (**c**) Production and consumption behavior of modes

At any given time between mode executions (actor invocations), an enclosing scheduler can query the actor using the *enable function* of the actor. The enable function can only examine the number of tokens on each input port (without consuming any data), and based on these "token populations", the function returns a Boolean value indicating whether or not the next mode has enough data to execute to completion without waiting for data on any port.

The set of possible next modes for a given actor at a given point in time can in general be empty or contain one or multiple elements. If the next mode set is empty (i.e., it is `null`), then the actor cannot be invoked again before it is somehow reset or re-initialized from environment that controls the enclosing dataflow graph. A `null` next mode is therefore equivalent to a transition to a mode that requires an infinite number of tokens on an input port. The provision for multi-element sets of next modes allows for natural representation of non-determinism in EIDF specifications.

When the set of next modes for a given actor mode is restricted to have at most one element, the resulting model of computation, called *Core Functional Dataflow* (*CFDF*), is a deterministic, Turing complete model [51]. CFDF semantics underlie the *Functional DIF* simulation environment for behavioral simulation of signal processing applications. Functional DIF integrates CFDF-based dataflow graph specification using the *Dataflow Interchange Format* (*DIF*). DIF is a textual language for DSP-oriented dataflow graphs and Java-based specifications of intra-actor functionality covering enable and invoke functions, and next mode computations [51].

Figures 3 and 4 illustrate, respectively, the design of a CFDF actor and its implementation in functional DIF. This actor provides functionality that is equivalent to the Boolean Dataflow switch actor described in Sect. 2.

a

```
public CFSwitch() {
    _control       = addMode("control");
    _control_true  = addMode("control_true");
    _control_false = addMode("control_false");

    _data_in       = addInput("data_in");
    _control_in    = addInput("control_in");
    _true_out      = addOutput("true_out");
    _false_out     = addOutput("false_out");

    _control.setConsumption(_control_in, 1);
    _control_true.setConsumption(_data_in, 1);
    _control_true.setProduction(_true_out, 1);
    _control_false.setConsumption(_data_in, 1);
    _control_false.setProduction(_false_out, 1);
}
```

b

```
public boolean enable(CoreFunctionMode mode) {
    if (_control == mode) {
        if (peek(_control_in) > 0) {
            return true;
        }
        return false;
    } else if (_control_true == mode) {
        if (peek(_data_in) > 0) {
            return true;
        }
        return false;
    } else if (_control_false == mode) {
        if (peek(_data_in) > 0) {
            return true;
        }
        return false;
    }
    return false;
}
```

c

```
public CoreFunctionMode invoke(CoreFunctionMode mode) {
    if (_init == mode) {
        return _control;
    }
    if (_control == mode) {
        if ((Boolean)pullToken(_control_in)) {
            return _control_true;
        } else {
            return _control_false;
        }
    }
    if (_control_true == mode) {
        Object obj = pullToken(_data_in);
        pushToken(_true_out, obj);
        return _control;
    }
    if (_control_false == mode) {
        Object obj = pullToken(_data_in);
        pushToken(_false_out, obj);
        return _control;
    }
}
```

Fig. 4 An implementation of the `switch` actor design of Fig. 3 in the functional DIF environment. (**a**) Constructor (defines modes and dataflow behavior). (**b**) Enable Function (determines whether firing condition is met). (**c**) Invoke function (performs action and determines next mode)

6 Scenario-Aware Dataflow

This section discusses *Scenario-Aware Dataflow* (SADF), which is a generalization of dataflow models with strict periodic or static behavior. Like many dataflow models, SADF is primarily a coordination language that highlights how actors (which are potentially executed in parallel) interact. To express dynamism, SADF distinguishes data and control explicitly. The control-related coherency between the behavior (and hence, the resource requirements) of different parts of a signal processing application can be captured with so-called *scenarios* [25]. The scenarios commonly coincide with dissimilar (but within themselves more static) modes of operation originating, for example, from different parameter settings, sample rate conversion factors, or the signal processing operations to perform. Scenarios are typically defined by clustering operational situations with similar resource requirements [25]. The scenario-concept in SADF allows for more precise (quantitative) analysis results compared to applying SDF-based analysis techniques. Moreover, common subclasses of SADF can be synthesized into efficient implementations [35, 66].

6.1 SADF Graphs

We introduce SADF by some examples from the multi-media domain. We first consider the MPEG-4 video decoder for the Simple Profile from [67, 71]. It supports video streams consisting of *Intra* (I) and *Predicted* (P) frames. For an image size of 176×144 pixels (QCIF), there are 99 macro blocks to decode for I frames and no motion vectors. For P frames, such motion vectors determine the new position of certain macro blocks relative to the previous frame. The number of motion vectors and macro blocks to process for P frames ranges between 0 and 99. The MPEG-4 decoder clearly shows variations in the functionality to perform and in the amount of data to communicate between the operations. This leads to large fluctuations in resource requirements [52]. The order in which the different situations occur strongly depends on the video content and is generally not periodic.

Figure 5 depicts an SADF graph for the MPEG-4 decoder in which nine different scenarios are identified. SADF distinguishes two types of actors: *kernels* (solid vertices) model the data processing parts, whereas *detectors* (dashed vertices) control the behavior of actors through scenarios.[2] Moreover, *data* channels (solid edges) and *control* channels (dashed edges) are distinguished. Control channels communicate scenario-valued tokens that influence the control flow. Data tokens do not influence the control flow. The availability of tokens in channels is shown with a dot. Here, such dots are labeled with the number of tokens in the channel. The start and end points of channels are labeled with *production* and *consumption rates* respectively.

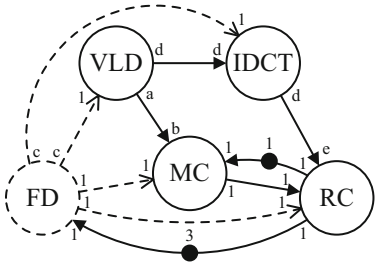
[2] In case of one detector, SADF literature may not show the detector and control channels explicitly.

They refer to the number of tokens atomically produced respectively consumed by the connected actor upon its *firing*. The rates can be fixed or scenario-dependent, similar as in PSDF. Fixed rates are positive integers. Parameterized rates are valued with non-negative integers that depend on the scenario. The parameterized rates for the MPEG-4 decoder are listed in Fig. 5b. A value of 0 expresses that data dependencies are absent or that certain operations are not performed in those scenarios. Studying Fig. 5b reveals that for any given scenario, the rate values yield a consistent SDF graph. In each of these scenario graphs, detector FD has a repetition vector entry of 1 [71], which means that scenario changes as prescribed by the behavior of FD may only occur at iteration boundaries of each scenario graph. This is not necessarily true for SADF in general as discussed below.

SADF specifies execution times of actors (from a selected time domain, see Sect. 6.2) per scenario. Figure 5c lists the worst-case execution times of the MPEG-4 decoder for an ARM7TDMI processor. The tables in Fig. 5 show that the worst-case communication requirements occur for scenario P_{99}, in which all actors are active and production/consumption rates are maximal. Scenario P_{99} also requires maximal execution times for VLD, IDCT, and MC, while for RC, it is scenario I in which the worst-case execution time occurs. Traditional SDF-based approaches need to combine these worst-case requirements into one (unrealistically) conservative model, which yields too pessimistic analysis results.

An important aspect of SADF is that sequences of scenarios are made explicit by associating *state machines* to detectors. The dynamics of the MPEG-4 decoder

a



b

Rate	(Sub)Scenario		
	I	P_0	P_x
a	0	0	1
b	0	0	x
c	99	0	x
d	I	0	1
e	99	0	x

$x \in \{30, 40, 50, 60, 70, 80, 99\}$

c

Actor	(Sub)Scenario	E (kCycles)
VLD	P_0	0
	All except P_0	40
IDCT	P_0	0
	All except P_0	17
MC	I, P_0	0
	P_{30}	90
	P_{40}	145
	P_{50}	190
	P_{60}	235
	P_{70}	265
	P_{80}	310
	P_{99}	390
RC	I	350
	P_0	0
	P_{30}, P_{40}, P_{50}	250
	P_{60}	300
	P_{70}, P_{80}, P_{99}	320
FD	All	0

Fig. 5 Modeling the MPEG-4 decoder with SADF. (**a**) Actors and channels. (**b**) Parameterized rates. (**c**) Worst-case execution times

originate from control-flow code that (implicitly or explicitly) represents a state-machine with video stream content dependent guards on the transitions between states. One can think of if-statements that distinguish processing I frames from processing P frames. For the purpose of compile-time analysis, SADF abstracts from the content of data tokens (similar to SDF and CSDF) and therefore also from the concrete conditions in control-flow code. Different types of state machines can be used to model the occurrences of scenarios, depending on the compile-time analysis needs as presented in Sect. 6.2. The dynamics of the MPEG-4 decoder can be captured by a state-machine of 9 states (one per scenario) associated to detector FD.

The operational behavior of actors in SADF follows two steps, similar to the *switch* and *select* actors in BDF (see Sect. 2) and to EIDF (see Sect. 5). The first step covers the control part which establishes the mode of operation. The second step is like the traditional data flow behavior of SDF actors[3] in which data is consumed and produced. Kernels establish their scenario in the first step when a scenario-valued token is available on their control inputs. The operation mode of detectors is established based on external and internal forces. We use *subscenario* to denote the result of the internal forces affecting the operation mode. External forces are the scenario-valued tokens available on control inputs (similar as for kernels). The combination of tokens on control inputs for a detector determine its scenario,[4] which (deterministically) selects a corresponding state-machine. A transition is made in the selected state machine, which establishes the subscenario. Where the scenario determines values for parameterized rates and execution time details for kernels, it is the subscenario that determines these aspects for detectors. Tokens produced by detectors onto control channels are scenario-valued to coherently affect the behavior of controlled actors, which is a key feature of SADF. Actor firings in SADF block until sufficient tokens are available. Hence, the execution of different scenarios can overlap in a pipelined fashion. For example, in the MPEG-4 decoder, IDCT is always ready to be executed immediately after VLD, which may already have accepted a control token with a different scenario value from FD. The ability to express such so-called *pipelined reconfiguration* is another key feature of SADF.

We now turn our attention to the MP3 audio decoder example from [67] depicted in Fig. 6. It illustrates that SADF graphs can contain multiple detectors, which may even control each other's behavior. MP3 decoding transforms a compressed audio bitstream into pulse code modulated data. The stream is partitioned into frames of 1152 mono or stereo frequency components, which are divided into two granules of 576 components structured in blocks [58]. MP3 distinguishes three frame types: Long (*L*), Short (*S*) and Mixed (*M*), and two block types: Long (*BL*) and Short (*BS*). A Long block contains 18 frequency components, while Short blocks include only 6 components. Long frames consist of 32 Long blocks, Short frames of 96

[3]Execution of the reflected function or program is enabled when sufficient tokens are available on all (data) inputs, and finalizes (after a certain execution time) with producing tokens on the outputs.

[4]If a detector has no control inputs, it operates in a default scenario ϵ and has one state machine.

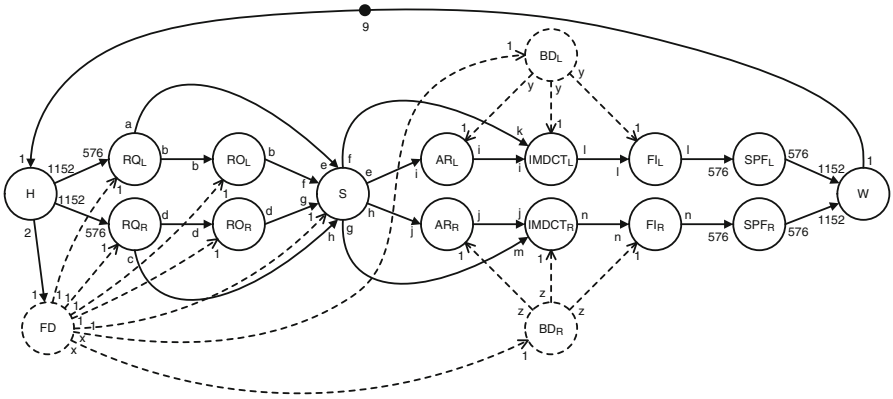


Fig. 6 Modeling an MP3 decoder with SADF using hierarchical control

Short blocks and Mixed frames are composed of 2 Long blocks, succeeded by 90 Short blocks. The frame type and block type together determine the operation mode. Neglecting that the frame types and specific block type sequences are correlated leads to unrealistic models. The sequences of block types is dependent on the frame types as reflected in the structure of source code of the MP3 audio decoder. SADF supports *hierarchical control* to intuitively express this kind of correlation between different aspects that determine the scenario.

Figure 7a lists the parameterized rates for the MP3 decoder. Only five combinations of frame types occur for the two audio channels combined. We use a two-letter abbreviation to indicate the combined fame type for the left and right audio channel respectively: *LL*, *SS*, *LS* and *SL*. Mixed frames *M* cover both audio channels simultaneously. Detector FD determines the frame type with a state machine of five states, each uniquely identifying a subscenario in {*LL*, *SS*, *LS*, *SL*, *M*}. The operation mode of kernel S depends on the frame types for both audio channels together and therefore it operates according to a scenario from this same set. The scenario of kernels RQ_L, RO_L and RQ_R, RO_R is only determined by the frame type for either the left or right audio channel. They operate in scenario *S*, *M* or *L* by receiving control tokens from FD, valued with either the left or right letter in *LL*, *SS*, *LS*, *SL* or with *M*.

Detectors BD_L and BD_R identify the appropriate number and order of Short and Long blocks based on the frame scenario, which they receive from FD as control tokens valued *L*, *S* or *M*. From the perspective of BD_L and BD_R, block types *BL* and *BS* are refinements (subscenarios) of the scenarios *L*, *S* and *M*. Figure 7b shows the three state machines associated with BD_L as well as BD_R. Each of their states implies one of the possible subscenarios in {LBL, SBS, MBL, MBS}. The value of the control tokens produced by BD_L and BD_R to kernels AR_L, $IMDCT_L$, FI_L and AR_R, $IMDCT_R$, FI_R in each of the four possible subscenarios matches the last two letters of the subscenario name (i.e., BL or BS). Although subscenarios LBL

a

Rate	Scenario		
	L	*S*	*M*
a, c	576	0	36
b, d	0	576	540

Rate	(Sub)Scenario				
	LL	*SS*	*LS*	*SL*	*M*
e	0	576	0	576	36
f	576	0	576	0	540
g	0	576	576	0	540
h	576	0	0	576	36
x	1	1	1	1	2

Rate	Scenario	
	BL	*BS*
i, j	18	0
k, m	0	6
l, n	18	6

Rate	SubScenario			
	LBL	*SBS*	*MBL*	*MBS*
y, z	32	96	2	90

b

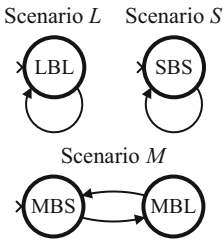


Fig. 7 Properties of the MP3 decoder model. (**a**) Parameterized rates. (**b**) State machines for BD_L and BD_R

and MBL both send control tokens valued BL, the difference between them is the number of such tokens (similarly for subscenarios SBS and MBS).

Consider decoding of a Mixed frame. It implies the production of two *M*-valued tokens on the control port of detector BD_L. By interpreting each of these tokens, the state machine for scenario *M* in Fig. 7b makes one transition. Hence, BD_L uses subscenario MBL for its first firing and subscenario MBS for its second firing. In subscenario MBL, BD_L sends 2 *BL*-valued to kernels AR_L, $IMDCT_L$ and SPF_L, while 90 *BS*-valued tokens are produced in subscenario MBS. As a result, AR_L, $IMDCT_L$ and SPF_L first process 2 Long blocks and subsequently 90 Short blocks as required for Mixed frames.

The example of Mixed frames highlights a unique feature of SADF: reconfigurations may occur *during* an iteration. An iteration of the MP3 decoder corresponds to processing frames, while block type dependent variations occur during processing Mixed frames. Supporting reconfiguration within iterations is fundamentally different from assumptions underlying other dynamic dataflow models, including for example PSDF. The concept is orthogonal to hierarchical control. Hierarchical control is also different from other dataflow models with hierarchy such as Heterogeneous Dataflow [26]. SADF allows *pipelined* execution of the controlling and controlled behavior together, while other approaches commonly prescribe that the controlled behavior must first finish completely before the controlling behavior may continue.

6.2 Analysis

Various analysis techniques exist for SADF, allowing the evaluation of both qualitative properties (such as consistency and absence of deadlock) and best/worst-case and average-case quantitative properties (like minimal and average throughput). We

briefly discuss consistency of SADF graphs. The MPEG-4 decoder is an example of a class of SADF graphs where each scenario is like a consistent SDF graph and scenario changes occur at iteration boundaries of these scenario graphs (although still pipelined). Such SADF graphs are said to be *strongly consistent* [71], which is easy to check as it results from structural properties only. The SADF graph of the MP3 decoder does not satisfy these structural properties (for Mixed frames), but it can still be implemented in bounded memory. The required consistency property is called *weak consistency* [22, 67]. Checking weak consistency requires taking the possible (sub)scenario sequences as captured by the state machines associated to detectors into account, which complicates a consistency check considerably.

Analysis of quantitative properties and the efficiency of the underlying techniques depend on the selected type of state machine associated to detectors as well as the chosen time model. For example, one possibility is to use non-deterministic state machines, which merely specify what sequences of (sub)scenarios *can* occur but not how often. This typically enables worst/best-case analysis. Applying the techniques in [19, 22, 23] then allows computing that a throughput of processing 0.253 frames per kCycle can be guaranteed for the MPEG-4 decoder. An alternative is to use probabilistic state machines (i.e., Markov chains), which also capture the occurrence probabilities of the (sub)scenario sequences to allow for average-case analysis as well. Assuming that scenarios I, P_0, P_{30}, P_{40}, P_{50}, P_{60}, P_{70}, P_{80} and P_{99} of the MPEG-4 decoder may occur in any order and with probabilities 0.12, 0.02, 0.05, 0.25, 0.25, 0.09, 0.09, 0.09 and 0.04 respectively, the techniques in [68] compute that the MPEG-4 decoder processes on average 0.426 frames per kCycle.

The semantics of SADF graphs where Markov chains are associated to detectors while assuming generic discrete execution time distributions[5] has been defined in [67] by using *Timed Probabilistic Systems* (TPS) as formal sematic model. Such transition systems operationalize the behavior with states and guarded transitions that capture events like the begin and end of each of the two steps in firing actors and progress of time. In case an SADF graph yields a TPS with finite state space, it is amenable to analysis techniques for (Priced) Timed Automata, Markov Decision Processes, and Markov Chains by defining reward structures as also used in (probabilistic or quantitative) model checking. In [68], for example, specific properties of dataflow models in general and SADF in particular are discussed that enable substantial state-space reductions during such analysis. The underlying techniques have been implemented in [69] in the SDF^3 tool kit [63], covering the computation of worst/best-case and average-case properties for SADF including throughput and various forms of latency and buffer occupancy metrics [69].

Other variants of Scenario-Aware Dataflow have been proposed that are supported by exact analysis techniques using formal sematic models. The techniques presented in [36, 37, 72] exploit *Interactive Markov Chains* (IMC) to combine the association of Markov chains to detectors with exponentially distributed execution times, which allows for instance computing the response time distribution of the

[5]This covers the case of constant execution times as so-called point distributions [67, 68].

MPEG-4 decoder to complete processing the first frame [72]. A further generalisation of the time model for Scenario-Aware Dataflow with Markov chains associated to detectors is proposed in [31]. This generalisation is based on the formal sematic model of *Stochastic Timed Automata* (STA) and allows for scenario-dependent cost annotations to compute for instance energy consumption.

When abstracting from the stochastic aspects of execution times and scenario occurrences, SADF is still amenable to worst/best-case analysis. Since SADF graphs are timed dataflow graphs, they exhibit *linear timing behavior* [19, 44, 77]. This property facilitates network-level worst/best-case analysis by considering the worst/best-case execution times for individual actors. For linear timed systems, this is known to lead to the overall worst/best-case performance. For the class of SADF graphs with a single detector (often called *FSM-based SADF*), very efficient performance analysis can be done based on a $(\max, +)$-algebraic interpretation of the operational semantics. It allows for worst-case throughput analysis, some latency analysis and can find critical scenario sequences without explicitly exploring the underlying state-space. Instead, the analysis is performed by means of state-space analysis and maximum-cycle ratio analysis of the equivalent but much smaller $(\max, +)$-automaton [19, 22, 23]. Reference [22] shows how this analysis can be extended for weakly-consistent SADF graphs. An alternative to using $(\max, +)$-algebra is proposed in [60], where the formal semantic model of *Timed Automata* (TA) is exploited to enable analyzing various qualitative and quantitative properties.

In case exact computation is hampered by state-space explosion, [69, 71] exploit an automated translation into process algebraic models expressed in the *Parallel Object-Oriented Specification Language* (POOSL) [70], which supports statistical model checking (simulation-based estimation) of various average-case properties.

6.3 Synthesis

FSM-based SADF graphs have been extensively studied for implementation on (heterogeneous) multi-processor platforms [35, 65]. Variations in resource requirements need to be exploited to limit resource usage without violating any timing requirements. The result of the design flow for FSM-based SADF implemented in the SDF^3 tool kit [63] is a set of Pareto optimal mappings that provide a trade-off in valid resource usages. For certain mappings, the application may use many computational resources and few storage resources, whereas an opposite situation may exist for other mappings. At run-time, the most suitable mapping is selected based on the available resources not used by concurrently running applications [59].

We highlight two key aspects of the design flow of [63, 65]. The first concerns mapping channels onto (possibly shared) storage resources. Like other dataflow models, SADF associates unbounded buffers with channels, but a complete graph may still be implemented in bounded memory. FSM-based SADF allows for efficient compile-time analysis of the impact that certain buffer sizes have on the timing of the application. Hence, a synthesized implementation does not require

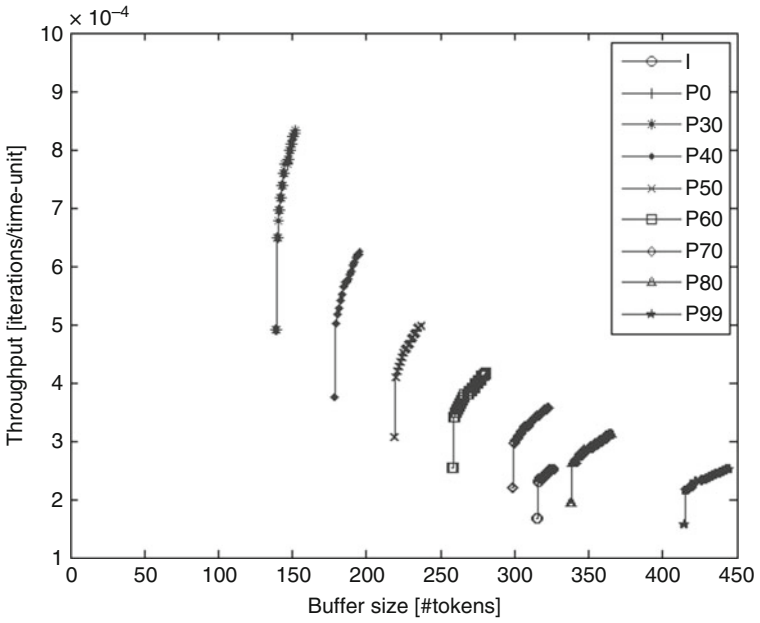


Fig. 8 Throughput/buffer size trade-off space for the MPEG-4 decoder

run-time buffer management, thereby making it easier to guarantee timing. The design flow in [65] dimensions the buffer sizes of all individual channels in the graph sufficiently large to ensure that timing (i.e., throughput) constraints are met, but also as small as possible to save memory and energy. It exploits the techniques of [64] to analyze the trade-off between buffer sizes and throughput for each individual scenario in the FSM-based SADF graph. After computing the trade-off space for all individual scenarios, a unified trade-off space for all scenarios is created. The same buffer size is assigned to a channel in all scenarios. Combining the individual spaces is done using Pareto algebra [21] by taking the free product of all trade-off spaces and selecting only the Pareto optimal points in the resulting space. Figure 8 shows the trade-off space for the individual scenarios in the MPEG-4 decoder. In this application, the set of Pareto points that describe the trade-off between throughput and buffer size in scenario P_{99} dominate the trade-off points of all other scenarios. Unifying the trade-off spaces of the individual scenarios therefore results in the trade-off space corresponding to scenario P_{99}. After computing the unified throughput/buffer trade-off space, the synthesis process in [65] selects a Pareto point with the smallest buffer size assignment that satisfies the throughput constraint as a means to allocate the required memory resources in the multiprocessor platform.

A second key aspect of the synthesis process is the fact that actors of the same or different applications may share resources. The set of concurrently active applications is typically unknown at compile-time. It is therefore not possible to construct a single static-order schedule for actors of different applications. The

design flow in [65] uses static-order schedules for actors of the same application, but sharing of resources between different applications is handled by run-time schedulers with TDMA policies. It uses a binary search algorithm to compute the minimal TDMA time slices ensuring that the throughput constraint of an application is met. By minimizing the TDMA time slices, resources are saved for other applications. Identification of the minimal TDMA time slices works as follows. In [1], it is shown that the timing impact of a TDMA scheduler can be modeled into the execution time of actors. This approach is used to model the TDMA time slice allocation it computes. Throughput analysis is then performed on the modified FSM-based SADF graph. When the throughput constraint is met, the TDMA time slice allocation can be decreased. Otherwise it needs to be increased. This process continues until the minimal TDMA time slice allocation satisfying the throughput constraint is found.

7 Dynamic Polyhedral Process Networks

The chapter on *Polyhedral Process Networks* (PPN) [74] deals with the automatic derivation of certain dataflow networks from *Static Affine Nested Loop Programs* (SANLP). An SANLP is a nested loop program in which loop bounds, conditions and variable index expressions are (quasi-)affine expressions in the iterators of enclosing loops and static parameters.[6] Because many signal processing applications are not static, there is a need to consider *dynamic affine nested loop programs* (DANLP) which differ from SANLPs in that they can contain

1. *if-the-else* constructs with no restrictions on the condition [61],
2. *loops* with no condition on the bounds [45],
3. *while* statements other than `while(1)` [46],
4. dynamic parameters [79].

Remark In all DANLP programs presented in subsequent Subsections, arrays are indexed by affine functions of static parameters and enclosing for-loop iterators. This is why the *A* is still in the name DANLP.

7.1 Weakly Dynamic Programs

Whereas condition statements in an SANLP must be affine in static parameters and iterators of enclosing loops, if conditions can be anything in a DANLP. Such programs have been called *Weakly Dynamic Programs* (WDP) in [61]. A simple

[6]The corresponding tool is called PNgen [75], and is part of the `Daedalus` design framework [48], http://daedalus.liacs.nl.

example of a WDP is shown in Fig. 9. The question here is whether the argument of function $F3$ originates from the output of function $F2$ or function $F1$.

In the case of an SANLP, the input-output equivalent PPN is obtained by:

1. Converting the SANLP—by means of an *array analysis* [15, 16]—into a *Single Assignment Code* (SAC) used in the compiler community and the systolic array community [33]
2. Deriving from the SAC a *Polyhedral Reduced Dependence Graph* (PRDG) [55]
3. Constructing the PPN from the PRDG [11, 40, 55]

While in an SAC every variable is written *only once*, in a *Dynamic Single Assignment Code* (dSAC) every variable is written *at most once*. For some variables, it is not known at compile time whether or not they will be read or written. For a WDP not all dependences are known at compile time and therefore, the analysis must be based on the so-called *Fuzzy Array Dataflow Analysis* (FADA) [17]. This approach allows the conversion of a WDP to a dSAC. The procedure to generate the dSAC is out of the scope. The dSAC for the WDP in Fig. 9 is shown in Fig. 10.

Parameter `C` in the dSAC of Fig. 10 is emerging from the if-statement in line 8 of the original program shown in Fig. 9. This if-statement also appears in the dSAC in line 14. The dynamic change of the value of `C` is accomplished by the lines 18 and 21 in Fig. 10. The control variable `ctrl(i)` in line 18 stores the iterations for which the data dependent condition that introduces `C` is `true`. Also, the variable `ctrl(i)` is used in line 21 to assign the correct value to `C` for the current iteration. See [61] for more details.

The dSAC can now be converted to two graph structures, namely the *Approximate Reduced Dependence Graph* (ADG), and the *Schedule Tree* (STree). The ADG is the dynamic counterpart of the static PRDG. Both the PRDG and the ADG are composed of processes N, input ports IP, output ports OP, and edges E [11, 55]. They contain all information related to the data dependencies between functions in the SAC and the dSAC, respectively. However, in a WDP some dependencies are not known at compile time, hence the name *approximate*. Because of this, the ADG has the additional notion of *Linearly Bounded Set* (LBS), as follows.

Let be given four sets of functions $S1 = \{f_x^1(i) \mid x = 1..|S1|,\ i \in Z^n\}$, $S2 = \{f_x^2(i) \mid x = 1..|S2|,\ i \in Z^n\}$, $S3 = \{f_x^3(i) \mid x = 1..|S3|,\ i \in Z^n\}$, $S4 =$

Fig. 9 Pseudo code of a simple Weakly Dynamic Program

```
1  %parameter N 8 16;
2
3  for i = 1:1:N,
4   [x(i), t(i)] = F1(...);
5  end
6
7  for i = 1:1:N,
8   if t(i) <= 0,
9    [x(i)] = F2( x(i) );
10  end
11  [...] = F3( x(i) );
12 end
```

Fig. 10 Dynamic Single Assignment Code for the example if Fig. 9

```
1  %parameter N 8 16;
2
3  for i = 1:1:N,
4    ctrl(i) = N+1;
5  end
6  for i = 1:1:N,
7   [out_0, out_1] = F1(...);
8   [x_1(i)] = opd(out_0);
9   [t_1(i)] = opd(out_1);
10 end
11
12 for i = 1:1:N,
13  [t_1(i)] = ipd(t_1(i));
14  if t_1(i) <= 0,
15   [in_0] = ipd(x_1(i));
16   [out_0] = F2(in_0);
17   [x_2(i)] = opd(out_0);
18   [ ctrl(i) ] = opd( i );
19  end
20
21  C = ipd( ctrl(i) );
22  if i = C,
23   [in_0] = ipd(x_2(C));
24  else
25   [in_0] = ipd(x_1(i));
26  end
27
28  [out_0] = F3(in_0);
29  [...] = opd(out_0);
30 end
```

$\{f_x^4(i) \mid x = 1..|S4|,\ i \in Z^n\}$, an integral $m \times n$ matrix A and an integral n-vector b. An LBS is a set of points $LBS = \{\ i \in Z^n \mid A.i \geq b,$

$$
\begin{aligned}
&if\ S1 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S1|},\ f_x^1(i) \geq 0,\\
&if\ S2 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S2|},\ f_x^2(i) \leq 0,\\
&if\ S3 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S3|},\ f_x^3(i) > 0,\\
&if\ S4 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S4|},\ f_x^4(i) < 0\ \}.
\end{aligned}
$$

The set of points $B = \{\ i \in Z^n \mid A.i \geq b\ \}$ is called *linear bound* of the LBS and the set $S = S1 \cup S2 \cup S3 \cup S4$ is called *filtering set*. Every $f_x^j(i) \in S$ can be an arbitrary function of i.

Consider the dSAC shown in Fig. 10. The exact iterations i are not known at compile time because of the dynamic condition at line 14 in the dSAC. That is why the notion of linearly bounded set is introduced, by which the unknown iterations i are approximated. So, ND_{N2} is the following LBS: $ND_{N2} = \{i \in Z \mid 1 \leq i \leq$

$N \wedge 8 \leq N \leq 16,\ t\_1(i) \leq 0\}$. The linear bound of this LBS is the polytope $B = \{1 \leq i \leq N \wedge 8 \leq N \leq 16\}$ that captures the information known at compile time about the bounds of the iterations i. The variable $t\_1(i)$ is interpreted as an unknown function of i called filtering function whose output is determined at run time.

The STree contains all information about the execution order amongst the functions in the dSAC. The STree represents one valid schedule between all these functions called *global schedule*. From the STree a local schedule between any arbitrary set of the functions in the dSAC can be obtained by pruning operations on the STree. Such a local schedule may for example be needed when two or more processes are merged [62]. The STree is obtained by converting the dSAC to a syntax tree using a standard syntax parser, after which all the nodes and edges that are not related to nodes Fi, i.e., $F1$, $F2$, and $F3$ in Fig. 10 [61]. Figure 11 depicts a summary.

The difference between the ADG in Fig. 11a and the transformed ADG in Fig. 11b is that an ADG may have several input ports connected to a single output port whilst in the transformed ADG every input port is connected to only one single output port (in accordance with the Kahn Process Network semantics [34]). Parsing the STree in Fig. 11c top-down from left to right generates a program that gives a valid execution order (global schedule) among the functions $F1$, $F2$ and $F3$ which is the original order given by the dSAC. The process network in Fig. 11d may be the result of a design space exploration, and some optimizations. For example, process $P2$ is constructed by grouping nodes $N1$ and $N3$ in the ADG in Fig. 11b. Because the behavior of process $P2$ is sequential (by default), it has to execute the functionality of nodes $N1$ and $N3$ in sequential order. This order is obtained from the STree in Fig. 11c. See [61] for details.

In a (static) PPN, there are two models of FIFO communication [73], namely *in-order communication* and *out-of-order communication*. In the first model, the order in which tokens are read from a FIFO channel is the same as the order in which they have been written to the channel. In the second model, that order is different. In a PPN that is input-output equivalent to a WDP, there are two more FIFO communication models, namely *in-order with coloring* and *out-of-order with coloring*. This is necessary because the number of tokens that will be written to a channel and read from that channel is not known at compile time [61].

Buffer sizes can be determined using the procedure given in [74, 75]. It however needs a conservative strategy (i.e., an over-estimation) due to the fact that the rate and the exact amount of data tokens transferred over a particular data channel is unknown at compile-time. Such over-estimation can be achieved by modifying the iteration domains of all input/output ports, such that all dynamic if-conditions defining any of these iteration domains always evaluate to `true`.

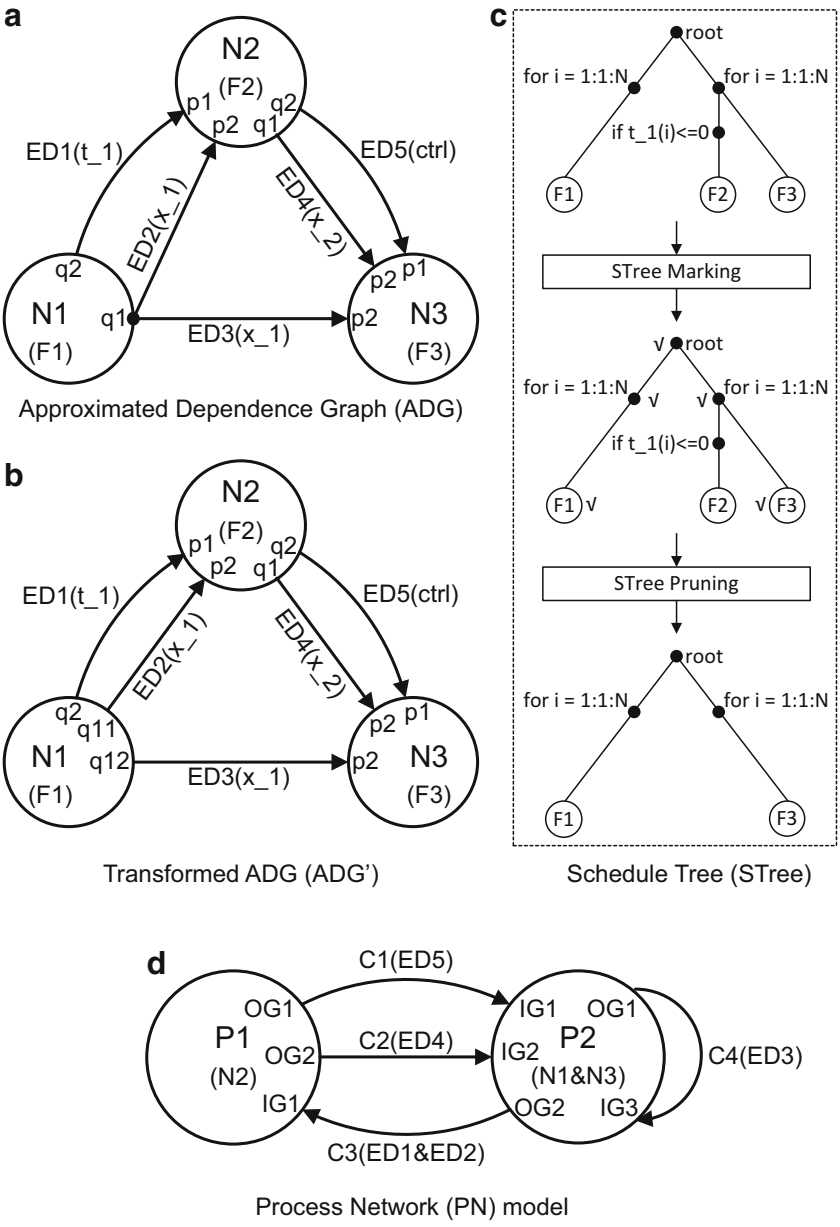


Fig. 11 Examples of (**a**) Approximated Dependence Graph (ADG) model; (**b**) Transformed ADG; (**c**) Schedule Tree and Transformations; (**d**) Process Network model

a

```
1  %parameter N 1 10;
2
3   for j = 1 to N,
4    for i = 1 to f(...),
5     y[ i ] = F1()
6    end
7   end
8   [...] = F2( y[5] ),
```

b

```
1  %parameter N 1 10;
2
3  for j = 1 to N,
4     X[j] = f(...)
5     for i = 1 to  max_f,
6       if i <=  X[j],
7       y[i] = F1()
8       end
9     end
10  end
11 [] = F2( y[5] )
```

Fig. 12 A `Dynloop` program and its equivalent WDP program. (**a**) Example of a `Dynloop` program. (**b**) Equivalent Weakly Dynamic Program

7.2 Dynamic Loop-Bounds

While loop bounds in an SANLP have to be affine functions of iterators of enclosing loops and static parameters, loop bounds in a DANLP program can be dynamic. Such programs have been called `Dynloop` programs in [45]. A simple example of a `Dynloop` program is shown in Fig. 12a.

A `Dynloop` program can be cast in the form of a WDP, see Sect. 7.1. The WDP corresponding to the `Dynloop` program in Fig. 12a is shown in Fig. 12b. The maximum value of $f()$, denoted by `max_f` in line 5 in Fig. 12b is substituted for the upper bound of the loop at line 4 in Fig. 12a. The value of `max_f` can be determined by studying the range of function $f()$.[7] As in Sect. 7.1, a dSAC can now be obtained by means of a FADA [17]. This analysis introduces parameters to deal with the dynamic structure in the WDP. The values of these parameters have to be changed dynamically. This is done by introducing for every such parameter, a control variable that stores the correct value of the parameter for every iteration. However, the straightforward introduction of control values as done in Sect. 7.1 violates the dSAC condition that every control variable is written *at most once*. To obtain a valid dSAC, an additional dataflow analysis for the control variables is necessary, resulting in additional control variables [45].

The final dSAC is shown in Fig. 13 where it has been assumed that the variable `y(5)` has been initialized to zero. The control variables must be initialized with values that are greater than the maximum value of the corresponding parameters. For the example at hand, parameter $c1 \in [1..\mathtt{N}]$, and $c2 \in [1..\mathtt{max\_f}]$. Therefore, the corresponding control variables are initialized as follows:

$$\forall \mathtt{i} : 1 \leq \mathtt{i} \leq \mathtt{max\_f} : \mathtt{ctrl\_c1[i]} = \mathtt{N} + 1,$$
$$\mathtt{ctrl\_c2[i]} = \mathtt{max\_f} + 1.$$

[7] If that is not possible, then an alternative way to estimate `max_f` is given in [45].

For brevity, the initialization is not shown in Fig. 13. After applying the standard *linearization* [73], and its extension of Sect. 7.1, and estimating buffer sizes as also described Sect. 7.1, the resulting PPN is as shown in Fig. 14.

7.3 Dynamic While-Loops

While only `while(1)` loops are allowed in an SANLP program, in a DANLP program any while-loop is acceptable. Such DANLP programs have been called *While-Loop Affine Programs* (WLAP) in [46]. There are a number of publications that address the problem of while loops parallelization [2, 8, 10, 24, 27, 28, 53, 54]. The approach presented here has the advantage that it

- supports both task-level and data-level parallelism,
- generates also parallel code for multi-processor systems with distributed memory,
- provides an automatic data-dependence analysis procedure,
- exposes and utilizes all available parallelism.

An example is shown in Fig. 15a. The question is again from where for example function $F7$ gets its scalar argument `x`. Because this is not known at compile-time,

Fig. 13 Final dSAC

```
1  %parameter N 1 10;

2  for j = 1 to N,
3    X[j] = f()
4    for i = 1 to max_f,
5      if i <= X[j],
6        y_1[j,i] = F1()
7        ctrl_c1[i] = j
8        ctrl_c2[i] = i
9      end
10     ctrl_c1_1[j,i] = ctrl_c1[i]
11     ctrl_c2_1[j,i] = ctrl_c2[i]
12   end
13 end
14 if max_f >= 5,
15   c1 = ctrl_c1_1[N, 5]
16   c2 = ctrl_c2_1[N, 5]
17 else
18   c1 = N + 1
19   c2 = max_f + 1
20 end
21 if c1 <= N & c2 == 5,
22   in_0 = y_1[c1,c2]
23 else
24   in_0 = 0
25 end
26 [...] = F2( in_0 )
```

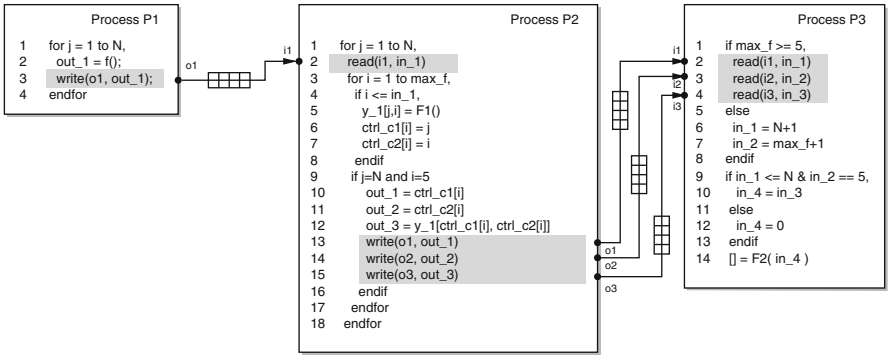


Fig. 14 The final PPN derived from the program in Fig. 13

a FADA analysis [17] is necessary to find all data dependencies. The approach to convert a WLAP program to an input-output equivalent PPN goes in four steps:

1. All data-dependency relations in the initial WLAP program have to be found by applying FADA analysis on it. Recall that the result of the analysis is approximated, i.e., it depends on parameters which values are determined at run-time.
2. Based on the results of the analysis, the initial WLAP is transformed into a dSAC representation, see Sect. 7.1. Parameters that are introduced by the FADA appear in the dSAC, and their values are assigned using control variables.
3. The control variables are generated in a way that extends the methods in Sects. 7.1 and 7.2 to be applicable for WLAP programs as well [46].
4. The topology of the corresponding PPN is derived as well as the code to be executed in the processes of the PPN.

The result of step 2 for the example in Fig. 15a is shown in Fig. 15b. The iterator w is associated with the while loop and is initialized with value 0, meaning that the while loop has never been executed. The parameter α captures the value of the for-loop iterator in the enclosing while-loop and is initialized to $N + 1$. The parameter β is the upper bound of the while-loop iterator w. Because $\alpha \in [1..N]$ and $\beta \geq 1$, the above initializations satisfy the condition that their values are never taken by the corresponding parameters. It follows from line 23 in Fig. 15b that control variable `ctrl_x_5` is initialized to `ctrl_x_5 = (N+1,0)` at line 3 in Fig. 15b.

Where does control variable `ctrl_x_5` come from? It comes from the construction of the dSAC. The procedure to derive the dSAC is largely based on [17] and its extension in Sect. 7.2. The problem is again that the dSAC resulting from the FADA analysis is not a proper dSAC because it violates the property that every variable is written *at most once*. The relation between writing to and reading from the control variables must be identified by performing a dataflow analysis for the control variables, where the writings to them occur inside a while-loop. To that end, an additional control variable `ctrl_x_5_` is introduced right *after* the while-loop,

a

```
1  &parameter EPS 0.005

2  for i = 1 to N,
3    y[i] = F1()
4    x = F2( y[i] )
5   while ( x >= EPS )
6     x = F3()
7     for j = i+1 to N+1,
8       y[j] = F4( y[j-1] )
9       x = F5( x, y[j] )
10    end
11     y[i] = F6( x )
12   end
13    out = F7( x )
14  end
```

b

```
1  %parameter EPS 0.005

2  w = 0
3  ctrl_x_5 = (N+1,0)
4  for i = 1 to N,
5    y_1[i] = F1()
6    in_2 = y_1[i]
7    x_2[i] = F2( in_2 )
8    while (in_w = σx(⟨W,(i,w)⟩) >= EPS),
9      w = w + 1
11     x_3[i,w] = F3()
11     for j = i+1 to N+1,
12       in_4 = σy(⟨S4,(i,w,j)⟩)
13       y_4[i,w,j] = F4( in_4 )
14       in_5_x = σx(⟨S5,(i,w,j)⟩)
15       in_5_y = y_4[i,w,j]
16       x_5[i,w,j] =
           F5( in_5_x, in_5_y )
17       ctrl_x_5 = (i,w)
18     end
19     in_6 = σx(⟨S6,(i,w)⟩)
20     y_6[i,w] = F6( in_6 )
21   end
22   ctrl_x_5_[i] = ctrl_x_5
23   (α,β) = ctrl_x_5_[i]
24   in_7 = σx(⟨S7,(i,α,β)⟩)
25   out = F7( in_7 )
26 end
```

Fig. 15 Example of a while-loop affine program and its corresponding dynamic single assignment program. (**a**) An example of a WLAP program. (**b**) The corresponding final dSAC

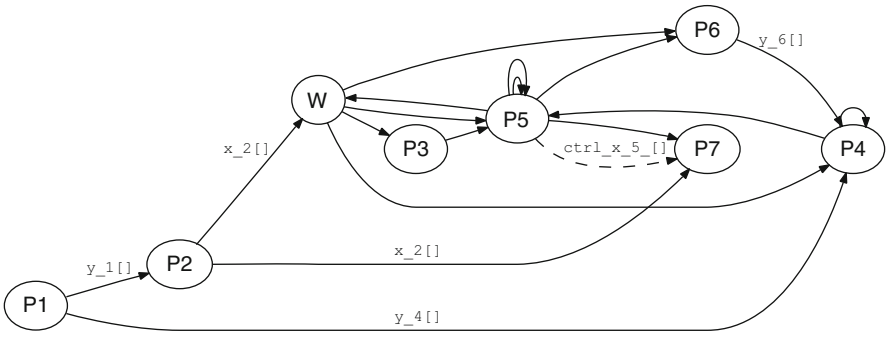


Fig. 16 The PPN for the program in Fig. 15

see line 22 in Fig. 15b. The new control variable is written at every iteration of for-loop `i` and takes the value either of control variable `ctrl_x_5` assigned on the last iteration of the while-loop, or its initial value, if the while-loop is not executed. A static *Exact Array Dataflow Analysis* (EADA) [15] can be performed on this new control variable `ctrl_x_5_`. This is possible because the new control variable is not surrounded by the dynamic while-loop, i.e., it is outside the while-loop.

a

```
1  %parameter EPS 0.005

2  w = 0
3  for i = 1 to N,
4    while(1),
5      w = w + 1
6      if (w > 2) then w = 2
7      if (w == 1),
8        read(P2, 1, in_w)
9      else
10       read(P5, 2, in_w)
11     end
12     out_w = (in_w >= EPS)
13     write(P3, 3, out_w)
14     write(P4, 4, out_w)
15     write(P5, 5, out_w)
16     write(P6, 6, out_w)
17     if (!out_w) <break>
18   end
19 end
```

b

```
1  w = 0
2  for i = 1 to N,
3    read(P5, 1, in_c)
4    if (in_c.β>=1 &&
       1<= in_c.α <= i),
5      read(P5, 2, in_7)
6    else
7      read(P2, 3, in_7)
8    end
9    out = F7( in_7 )
10 end
```

c

```
1  w = 0
2  ctrl_x_5 = (N+1,0)
3  for i = 1 to N,
4    while(1),
5      w = w + 1
6      if (w > 2) then w = 2
7      read(W, 1, in_w)
8      if (!in_w) <break>
9      for j = i+1 to N+1,
10       if (j == i+1),
11         if (w == 1),
12           read(P3, 2, in_5_x)
13         else
14           read(P5, 3, in_5_x)
15         en
16       else
17         read(P5, 4, in_5_x)
18       end
19       read(P4,5, in_5_y)
20       out_5 = F5( in_5_x, in_5_y )
21       ctrl_x_5 = (i,w)
22       if (j == N+1),
23         write(P5, 6, out_5)
24       else
25         write(P5, 7, out_5)
26       endif
27     end
28   end
29   out_5_c = ctrl_x_5
30   out_5_x = out_5
31   write(P7, 8, out_5_c)
32   write(P7, 9, out_5_x)
33 end
```

Fig. 17 Processes W, $P5$, and $P7$ after linearization. (**a**) Code of process W. (**b**) Code of process P7. (**c**) Code of process P5

Step 4 constructs the PPN from the dSAC. The PPN corresponding to the dSAC in Fig. 15b is depicted in Fig. 16. It consists of 8 processes and 18 channels. The processes $P1$–$P7$ correspond to the functions $F1$–$F7$ in Fig. 15. Process W corresponds to the while condition at line 8 of the dSAC in Fig. 15b.

The code for processes W, $P5$, and $P7$ is shown in Fig. 17. Process W is an example of a process detecting the termination of the while-loop at line 5 in Fig. 15a. Process $P5$ is an example of a process executing a function enclosed in the while-loop. Process $P7$ is an example of a process that runs a function *outside* the while-loop, and has a data dependency with a function *inside* the while-loop.

7.4 Parameterized Polyhedral Process Networks

Parameters that appear in an SANLP program are static. In a DANLP, parameters can be dynamic. A *Polyhedral Process Network* (PPN) that is input-output equivalent to a DANLP program is called a *Parameterized Polyhedral Process Network*, which is abbreviated to P^3N. The formal definition of a P^3N is given in [79], and is only slightly different from the definition in [74]. Although the consistency of a P^3N has to be checked at run-time, still some analysis can be done at compile-time.

Remark There are two assumptions here. First, dynamic conditions, dynamic loop bounds and dynamic while-loops are left out to focus only on dynamic parameters. Second, values of the dynamic parameters are obtained from the environment.

An example P^3N is shown in Fig. 18. Figure 18a is a static PPN of which process *P3* is shown in Fig. 18b. Figure 18c presents a P^3N version of the PPN in Fig. 18a. Process *P*3 of the P^3N in Fig. 18c is shown in Fig. 18d. The PPN and the P^3N have the same *dataflow* topology. Processes *P2* and *P3* in the P^3N in Fig. 18c are reconfigured by two parameters *M* and *N* whose values are updated from *the environment* at run-time using process *Ctrl* and FIFO channels *ch7*, *ch8*, and *ch9*. The P^3N in Fig. 18c may be derived from a sequential program, yet it can also be constructed from library elements as in [30] or using the approach of [12].

Reference [74] explains that a parametric polyhedron $\mathcal{P}(\mathbf{p})$ is defined as $\mathcal{P}(\mathbf{p}) = \{(w, x_1, \ldots, x_d) \in \mathbb{Q}^{d+1} \mid A \cdot (w, x_1, \ldots, x_d)^T \geq B \cdot \mathbf{p} + b\}$ with $A \in \mathbb{Z}^{m \times d}$, $B \in \mathbb{Z}^{m \times n}$ and $c \in \mathbb{Z}^m$. For nested loop programs, w is to be interpreted as the one-dimensional `while(1)` index, and d as the depth of a loop nest. For a particular value of w, the polyhedron gets closed, i.e., it becomes a polytope. The parameter vector $\mathbf{p}$ is bounded by a polytope $\mathcal{P}_\mathbf{p} = \{\mathbf{p} \in \mathbb{Q}^n \mid C \cdot \mathbf{p} \geq d\}$. The domain D_P of a process is defined as the set of all integral points in its underlying parametric polyhedron, i.e., $D_P = \mathcal{P}_P(\mathbf{p}) \cap \mathbb{Z}^{d+1}$. The domains D_{IP} and D_{OP} of an input port *IP* and an output port *OP*, respectively, of a process are subdomains of the domain of that process.

The following four notions play a role in the operational semantics of a P^3N:

- A **process iteration** of process P is a point $(w, x_1, \ldots, x_d) \in D_P$, where the following operations are performed sequentially: reading a token from each *IP* for which $(w, x_1, \ldots, x_d) \in D_{IP}$, executing process function F_P, and writing a token to each OP for which $(w, x_1, \ldots, x_d) \in D_{OP}$.
- A **process cycle** $CYC_P(\mathcal{S}, \mathbf{p}) \subset D_P$ is the set of lexicographically ordered points $\in D_P$ for a particular value of $w = \mathcal{S} \in \mathbb{Z}^+$. The lexical ordering is typically imposed by a loop nest.
- A **Process execution** E_P is a sequence of process cycles denoted by $CYC_P(1, \mathbf{p}_1) \rightarrow CYC_P(2, \mathbf{p}_2) \rightarrow \ldots \rightarrow CYC_P(k, \mathbf{p}_k)$, where $k \rightarrow \infty$.
- A point $Q_P(\mathcal{S}, \mathbf{p}_i) \in CYC_P(\mathcal{S}, \mathbf{p}_i)$ of process P is a **quiescent point** if $CYC_P(\mathcal{S}, \mathbf{p}_i) \in E_P$ and $\neg(\exists(w, x_1, \ldots, x_d) \in CYC_P(\mathcal{S}, \mathbf{p}_i) : (w, x_1, \ldots, x_d) \prec Q_P(\mathcal{S}, \mathbf{p}_\mathcal{S})$.

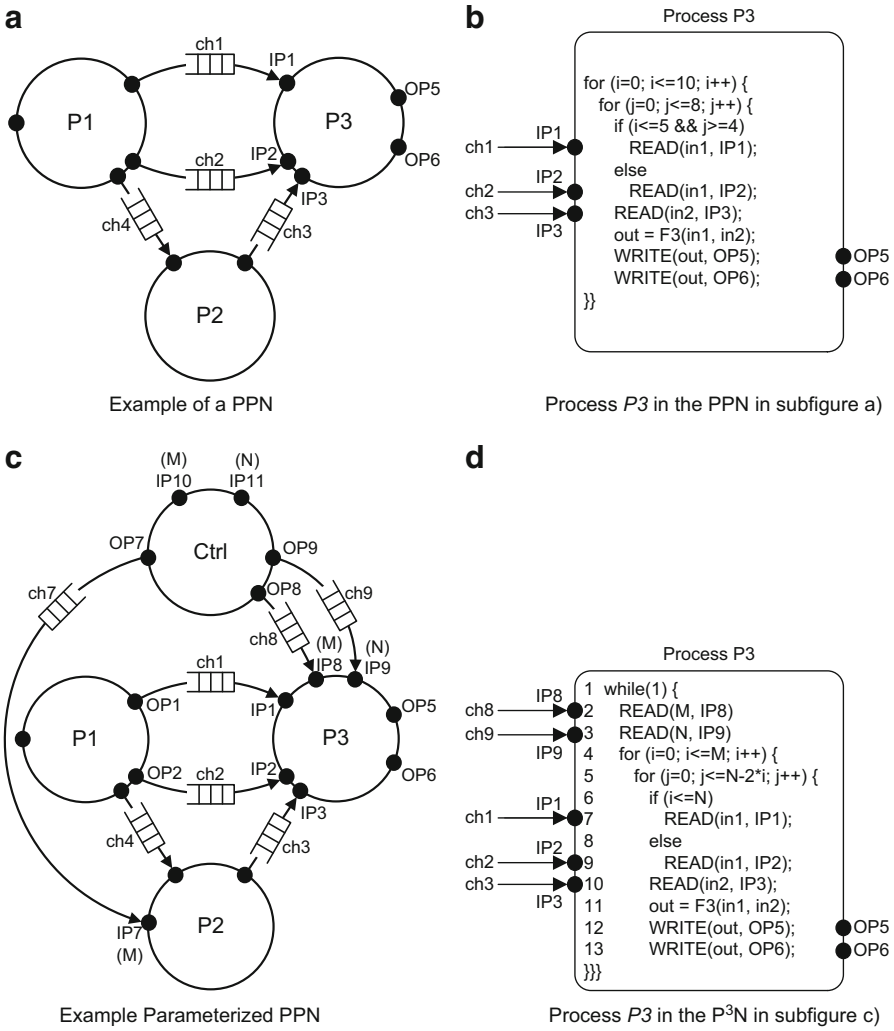


Fig. 18 (**a**) An example of a PPN, (**b**) process *P3* in the PPN, (**c**) an example of a P^3N, and (**d**) process *P3* in the P^3N

Thus, process *P* can change parameter values at the first process iteration of any process cycle during the execution. The notion of quiescent points as being the points at which values of the parameters **p** can change appears also in [47].

The behavior of the control process *Ctrl* is given in Fig. 19a. Process *Ctrl* starts with at least one valid parameter combination (lines 1-2) and then reads parameters from the environment (lines 3-4) every pre-specified time interval. For every incoming parameter combination, the process function `Eval` (line 5) checks whether the combination of parameter values is valid. The implementation of function `Eval` is given in Fig. 19b. If the combination is valid, then function `Eval`

returns the current parameter values (`M, N`). Otherwise, the last valid parameter combination (propagated through `M_new, N_new` in this example) is returned. After the evaluation of the parameter combination, process *Ctrl* writes the parameter values to output ports (lines 6-8) when all channels *ch7*, *ch8*, and *ch9* have at least one buffer place available. When at least one channel buffer is full, the incoming parameters combination is discarded and the control process continues to read the next parameters combination from the environment. Furthermore, the depth of the FIFOs of the control channels determines how many process cycles of the dataflow processes are allowed to overlap. Valid parameter values lead to the consistent execution of a P^3N, i.e., without deadlocks and with bounded memory (FIFOs with finite capacity).

To illustrate the consistency problem, consider channel *ch3* connecting processes *P2* and *P3* of the P^3N given in Fig. 18c. The access of processes *P2* and *P3* to channel *ch3* is depicted in Fig. 20. Consistency requires that, for each corresponding process cycle of both processes $CYC_{P2}(i, M_i)$ and $CYC_{P3}(i, M_i, N_i)$, the number of tokens produced by process *P2* to channel *ch3* must be equal to the number of tokens consumed by process *P3* from channel *ch3*. For example, if $(M, N) = (7, 8)$, *P2* produces 25 tokens to *ch3* and *P3* consumes 25 tokens from the same channel after one corresponding process cycle of both processes. It can be verified that *P2* produces 13 tokens to *ch3* while *P3* requires 20 tokens from it in a corresponding process cycle when $(M, N) = (3, 7)$. Thereby, in order to complete one execution cycle of *P3* in this case, it will read data from *ch3* which will be produced during the next execution cycle of *P2*. Evidently this leads to an incorrect execution of the P^3N. From this example, it is clearly seen that the incoming values of (M, N) must satisfy certain relation to ensure the consistent execution of the P^3N.

Although the consistency of a P^3N has to be checked at run-time, still some analysis can be done at design-time. This is because input ports and output ports of a process cycle are parametric polytopes. The number of points in a port domain equals the number of tokens that will be written to a channel or read from a channel

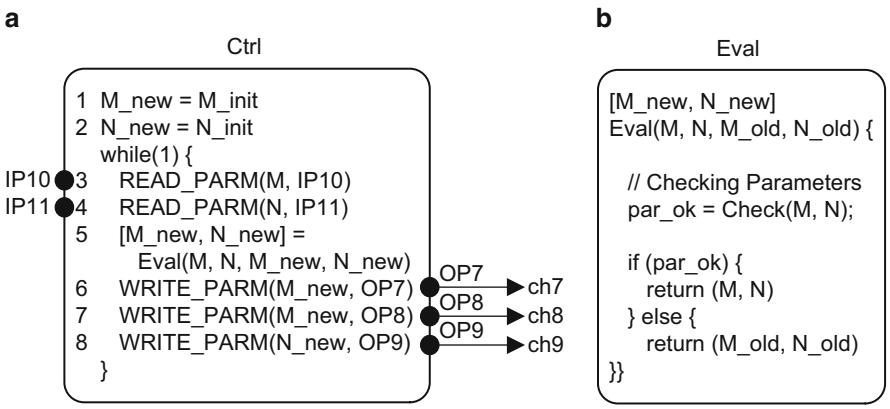


Fig. 19 (**a**) Control process *Ctrl* and (**b**) process function *Eval*

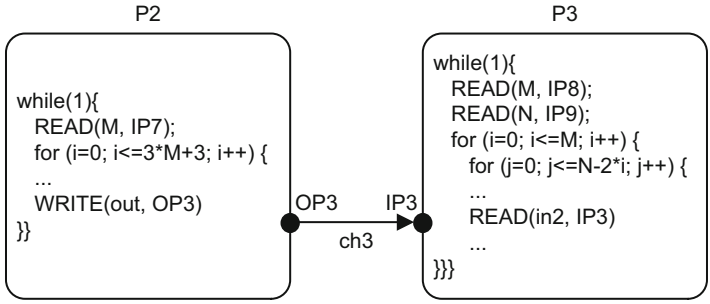


Fig. 20 Which combinations $(M,\ N)$ do ensure consistency of P^{3}N?

depending on whether the port is an output port or an input port, respectively. The condition $|D_{OP}^{CYC}| = |D_{IP}^{CYC}|$ can be checked by comparing the number of points in both port domains. The counting problem can be solved in polynomial time using the *Barvinok* library [74, 76]. In general, the number of points in domain $D_X = \mathcal{P}_X(\mathbf{p}) \cap \mathbb{Z}^{d+1}$, where X stand for either a process P, an input port IP, or an output port OP, is a set of quasi-polynomials [74].

For the example in Fig. 20, the difference $|D_{OP}^{CYC}| - |D_{IP}^{CYC}|$ is,

$$\begin{cases} (1 + N + N \cdot M - M^2) - (3M + 4) = 0 & \text{if } (M, N) \in C1 \\ (1 + \frac{3}{4}N + \frac{1}{4}N^2 + \frac{1}{4}N - \frac{1}{4} \cdot \{0, 1\}_N) - (3M + 4) = 0 & \text{if } (M, N) \in C2 \end{cases}$$

where $C1 = \{(M, N) \in \mathbb{Z}^2 \mid M \leq N \wedge 2M \geq 1 + N\}$, $C2 = \{(M, N) \in \mathbb{Z}^2 \mid 2M \leq N\}$, and $\{0, 1\}_N$ is a periodic coefficient with period 2.[8] If in this example the range of the parameters is $0 \leq M, N \leq 100$, then there are only 10 valid parameter combinations. If $0 \leq M, N \leq 1000$, then there are 34 valid parameter combinations, and if $0 \leq M, N \leq 10000$, then the number of valid combinations is 114.

The symbolic subtraction of the quasi-polynomials can result in constant zero, non-zero constant, or a quasi-polynomial. In the first case, consistency is always preserved for all parameters within the range. In the second case, all parameters within the range are invalid, because they violate the consistency condition. In the third case, a quasi-polynomial remains, and only some parameter combinations within the range are valid for the consistency condition. The equations can be solved at design time, and all valid parameter combinations are put in a table which is stored in a function *Check*. At run-time, the control process only propagates those incoming parameter combinations that match an entry in the table. Alternatively, function *Check* evaluates the difference between the two quasi-polynomials against zero with incoming parameter values at run-time. When using a table, the execution time of the P^{3}N is almost equal to the execution time of the corresponding PPN. On the other hand, evaluation the polynomials at run-time overlaps the dataflow

[8] $\{0, 1\}_N$ is `0` or `1` depending on whether N is even or odd, respectively.

processing. For medium and high workloads (execution latency of the processes) the overhead is negligible. See [79] for further details.

8 Summary

This chapter reviewed several DSP-oriented dataflow models of computation that focus on representing dynamic dataflow behavior. As signal processing systems are developed and deployed for more complex applications, exploration of such generalized dataflow modeling techniques is of increasing importance. This chapter complemented the discussion in [30], which focuses on the relatively mature class of decidable dataflow modeling techniques, and builds on the dynamic dataflow principles introduced in certain specific forms [14, 20].

Acknowledgements In this work, Bhattacharyya has been supported in part by the US Air Force Office of Scientific Research. The authors also thank Marc Geilen (m.c.w.geilen@tue.nl) and Sander Stuijk (s.stuijk@tue.nl), both from the Eindhoven University of Technology, for their contribution to Sect. 6.

References

1. Bekooij, M., Hoes, R., Moreira, O., Poplavko, P., Pastrnak, M., Mesman, B., Mol, J.D., Stuijk, S., Gheorghita, S.V., van Meerbergen, J.: Dataflow analysis for real-time embedded multiprocessor system design. In: P. van der Stok (ed.) Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, pp. 81–108. Springer (2005)
2. Benabderrahmane, M.W., Pouchet, L.N., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: R. Gupta (ed.) Proceedings of the International Conference on Compiler Construction, CC, pp. 283–303. Springer (2010)
3. Berg, H., Brunelli, C., Lucking, U.: Analyzing models of computation for software defined radio applications. In: Proceedings of the International Symposium on System-on-Chip, SoC, pp. 1–4 (2008)
4. Bhattacharya, B., Bhattacharyya, S.S.: Parameterized dataflow modeling for DSP systems. IEEE Transactions on Signal Processing **49**(10), 2408–2421 (2001)
5. Bhattacharyya, S.S., Buck, J.T., Ha, S., Lee, E.A.: Generating compact code from dataflow specifications of multirate signal processing algorithms. IEEE Transactions on Circuits and Systems — I: Fundamental Theory and Applications **42**(3), 138–150 (1995)
6. Bhattacharyya, S.S., Eker, J., Janneck, J.W., Lucarz, C., Mattavelli, M., Raulet, M.: Overview of the MPEG reconfigurable video coding framework. Journal of Signal Processing Systems (2010)
7. Bhattacharyya, S.S., Leupers, R., Marwedel, P.: Software synthesis and code generation for DSP. IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing **47**(9), 849–875 (2000)
8. Bijlsma, T., Bekooij, M.J.G., Smit, G.J.M.: Inter-task communication via overlapping read and write windows for deadlock-free execution of cyclic task graphs. In: Proceedings of the International Symposium on Systems, Architectures, Modeling, and Simulation, ICSAMOS, pp. 140–148 (2009)

9. Buck, J.T.: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley (1993)
10. Collard, J.F.: Automatic parallelization of while-loops using speculative execution. International Journal of Parallel Programming **23**(2), 191–219 (1995)
11. Deprettere, E.F., Rijpkema, E., Kienhuis, B.: Translating imperative affine nested loop programs to process networks. In: E.F. Deprettere, J. Teich, S. Vassiliadis (eds.) Embedded Processor Design Challenges, LNCS 2268, pp. 89–111. Springer (2002)
12. Desnos, K., Palumbo, F.: Dataflow modeling for reconfigurable signal processing systems. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
13. Eker, J., Janneck, J.W.: CAL language report, language version 1.0 — document edition 1. Tech. Rep. UCB/ERL M03/48, Electronics Research Laboratory, University of California at Berkeley (2003)
14. Falk, J., Neubauer, K., Haubelt, C., Zebelein, C., Teich, J.: Integrated modeling using finite state machines and dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
15. Feautrier, P.: Dataflow analysis of scalar and array references. International Journal of Parallel Programming **20**(1), 23–53 (1991)
16. Feautrier, P.: Automatic parallelization in the polytope model. In: The Data Parallel Programming Model, pp. 79–103 (1996)
17. Feautrier, P., Collard, J.F.: Fuzzy array dataflow analysis. Tech. rep., Ecole Normale Superieure de Lyon (1994). ENS-Lyon/LIP N^o 94-21
18. Gao, G.R., Govindarajan, R., Panangaden, P.: Well-behaved dataflow programs for dsp computation. In: Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, ICASSP, pp. 561–564 (1992)
19. Geilen, M.C.W.: Synchronous dataflow scenarios. ACM Transactions on Embedded Computing Systems **10**(2), 16:1–16:31 (2011)
20. Geilen, M.C.W., Basten, T.: Kahn process networks and a reactive extension. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
21. Geilen, M.C.W., Basten, T., Theelen, B.D., Otten, R.: An algebra of pareto points. Fundamenta Informaticae **78**(1), 35–74 (2007)
22. Geilen, M.C.W., Falk, J., Haubelt, C., Basten, T., Theelen, B.D., Stuijk, S.: Performance analysis of weakly-consistent scenario-aware dataflow graphs. Journal of Signal Processing Systems **87**(1), 157–175 (2017)
23. Geilen, M.C.W., Stuijk, S.: Worst-case performance analysis of synchronous dataflow scenarios. In: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, CODES/ISSS, pp. 125–134. ACM, New York, NY, USA (2010)
24. Geuns, S.J., Bekooij, M.J.G., Bijlsma, T., Corporaal, H.: Parallelization of while loops in nested loop programs for shared-memory multiprocessor systems. In: Proceedings of Design, Automation and Test in Europe, DATE, pp. 1–6 (2011)
25. Gheorghita, S.V., Stuijk, S., Basten, T., Corporaal, H.: Automatic scenario detection for improved WCET estimation. In: Proceedings of the Design Automation Conference, DAC, pp. 101–104 (2005)
26. Girault, A., Lee, B., Lee, E.: Hierarchical finite state machines with multiple concurrency models. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **18**(6), 742 –760 (1999)
27. Griebl, M., Collard, J.F.: Generation of synchronous code for automatic parallelization of while loops. In: S. Haridi, K. Ali, P. Magnusson (eds.) Proceedings of the International Conference on Parallel Processing, EURO-PAR, pp. 313–326. Springer (1995)
28. Griebl, M., Lengauer, C.: A communication scheme for the distributed execution of loop nests with while loops. International Journal of Parallel Programming **23** (1995)

29. Gu, R., Janneck, J.W., Raulet, M., Bhattacharyya, S.S.: Exploiting statically schedulable regions in dataflow programs. In: Proceedings of the International Conference on Acoustics, Speech and Signal Processing, ICASSP, pp. 565–568 (2009)
30. Ha, S., Oh, H.: Decidable signal processing dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
31. Hartmanns, A., Hermanns, H., Bungert, M.: Flexible support for time and costs in scenario-aware dataflow. In: Proceedings of the International Conference on Embedded Software, EMSOFT, pp. 3:1–3:10 (2016)
32. Haykin, S.: Adaptive Filter Theory. Prentice Hall (1996)
33. Hu, Y.H., Kung, S.Y.: Systolic arrays. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
34. Kahn, G.: The semantics of a simple language for parallel programming. In: Proceedings of Information Processing (1974)
35. van Kampenhout, R., Stuijk, S., Goossens, K.: Programming and analysing scenario-aware dataflow on a multi-processor platform. In: Proceedings of Design, Automation and Test in Europe, DATE, pp. 876–881 (2017)
36. Katoen, J.P., Wu, H.: Exponentially timed SADF: Compositional semantics, reductions, and analysis. In: Proceedings of the International Conference on Embedded Software, EMSOFT, pp. 1–10 (2014)
37. Katoen, J.P., Wu, H.: Probabilistic model checking for uncertain scenario-aware data flow. Transactions on Design Automation of Electronic Systems **22**(1), 15:1–15:27 (2016)
38. Kee, H., Bhattacharyya, S.S., Wong, I., Rao, Y.: FPGA-based design and implementation of the 3GPP-LTE physical layer using parameterized synchronous dataflow techniques. In: Proceedings of the International Conference on Acoustics, Speech and Signal Processing, ICASSP, pp. 1510–1513 (2010)
39. Keinert, J., Deprettere, E.F.: Multidimensional dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, pp. 1145–1175. Springer (2013)
40. Kienhuis, B., Rijpkema, E., Deprettere, E.F.: Compaan: Deriving process networks from Matlab for embedded signal processing architectures. In: Proceedings of the International Workshop on Hardware/Software Codesign, CODES, pp. 13–17 (2000)
41. Ko, M., Zissulescu, C., Puthenpurayil, S., Bhattacharyya, S.S., Kienhuis, B., Deprettere, E.F.: Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation. IEEE Transactions on Signal Processing **55**(6), 3126–3138 (2007)
42. Lin, Y., Choi, Y., Mahlke, S., Mudge, T., Chakrabarti, C.: A parameterized dataflow language extension for embedded streaming systems. In: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS, pp. 10–17 (2008)
43. Mattavelli, M., Janneck, J.W., Raulet, M.: MPEG reconfigurable video coding. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
44. Moreira, O.: Temporal analysis and scheduling of hard real-time radios running on a multi-processor. Ph.D. thesis, Eindhoven University of Technology (2012)
45. Nadezhkin, D., Nikolov, H., Stefanov, T.: Translating affine nested-loop programs with dynamic loop bounds into polyhedral process networks. In: Proceedings of the Symposium on Embedded Systems for Real-Time Multimedia, ESTIMedia, pp. 21–30 (2010)
46. Nadezhkin, D., Stefanov, T.: Automatic derivation of polyhedral process networks from while-loop affine programs. In: Proceedings of the Symposium on Embedded Systems for Real-Time Multimedia, ESTIMedia, pp. 102–111 (2011)
47. Neuendorffer, S., Lee, E.: Hierarchical reconfiguration of dataflow models. In: Proceedings of the International Conference on Formal Methods and Models for Co-Design, MEMOCODE, pp. 179–188 (2004)

48. Nikolov, H., Stefanov, T., Deprettere, E.F.: Systematic and automated multi-processor system design, programming, and implementation. IEEE Transactions on Computer-Aided Design **27**(3), 542–555 (2008)
49. Plishker, W., Sane, N., Bhattacharyya, S.S.: A generalized scheduling approach for dynamic dataflow applications. In: Proceedings of Design, Automation and Test in Europe, DATE, pp. 111–116 (2009)
50. Plishker, W., Sane, N., Bhattacharyya, S.S.: Mode grouping for more effective generalized scheduling of dynamic dataflow applications. In: Proceedings of the Design Automation Conference, DAC, pp. 923–926 (2009)
51. Plishker, W., Sane, N., Kiemb, M., Anand, K., Bhattacharyya, S.S.: Functional DIF for rapid prototyping. In: Proceedings of the International Symposium on Rapid System Prototyping, RSP, pp. 17–23 (2008)
52. Poplavko, P., Basten, T., van Meerbergen, J.: Execution-time prediction for dynamic streaming applications with task-level parallelism. In: Proceedings of the Euromicro Conference on Digital System Design Architectures, Methods and Tools, DSD, pp. 228–235 (2007)
53. Raman, E., Ottoni, G., Raman, A., Bridges, M.J., August, D.I.: Parallel-stage decoupled software pipelining. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO, pp. 114–123 (2008)
54. Rauchwerger, L., Padua, D.: Parallelizing while loops for multiprocessor systems. In: Proceedings of International Parallel Processing Symposium, IPDPS, pp. 347–356 (1995)
55. Rijpkema, E., Deprettere, E.F., Kienhuis, B.: Deriving process networks from nested loop algorithms. Parallel Processing Letters **10**(2), 165–176 (2000)
56. Roquier, G., Wipliez, M., Raulet, M., Janneck, J.W., Miller, I.D., Parlour, D.B.: Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study. In: Proceedings of the Workshop on Signal Processing Systems, SIPS, pp. 281–286 (2008)
57. Saha, S., Puthenpurayil, S., Bhattacharyya, S.S.: Dataflow transformations in high-level DSP system design. In: Proceedings of the International Symposium on System-on-Chip, SoC, pp. 131–136 (2006)
58. Shlien, S.: Guide to MPEG-1 audio standard. IEEE Transactions on Broadcasting **40**(4), 206 –218 (1994)
59. Shojaei, H., Ghamarian, A.H., Basten, T., Geilen, M.C.W., Stuijk, S., Hoes, R.: A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for CMP run-time management. In: Proceedings of the Design Automation Conference, DAC, pp. 917–922 (2009)
60. Skelin, M., Wognsen, E.R., Olesen, M.C., Hansen, R.R., Larsen, K.G.: Model checking of finite-state machine-based scenario-aware dataflow using timed automata. In: Proceedings of the International Symposium on Industrial Embedded Systems, SIES, pp. 1–10 (2015)
61. Stefanov, T., Deprettere, E.F.: Deriving process networks from weakly dynamic applications in system-level design. In: Proceedings of the International Conference on Hardware/Software Codesign and Systems Synthesis, CODES+ISSS, pp. 90–96 (2003)
62. Stefanov, T., Kienhuis, B., Deprettere, E.F.: Algorithmic transformation techniques for efficient exploration of alternative application instances. In: Proceedings of the International Symposium on Hardware/Software Codesign, CODES, pp. 7–12 (2002)
63. Stuijk, S., Geilen, M.C.W., Basten, T.: SDF^3: SDF For Free. In: Proceeding of the International Conference on Application of Concurrency to System Design, ACSD, pp. 276–278 (2006). URL http://www.es.ele.tue.nl/sdf3
64. Stuijk, S., Geilen, M.C.W., Basten, T.: Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. IEEE Transactions on Computers **57**(10), 1331–1345 (2008)
65. Stuijk, S., Geilen, M.C.W., Basten, T.: A predictable multiprocessor design flow for streaming applications with dynamic behaviour. In: Proceedings of the Euromicro Conference on Digital System Design: Architectures, Methods and Tools, DSD, pp. 548–555 (2010)

66. Stuijk, S., Geilen, M.C.W., Theelen, B.D., Basten, T.: Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS, pp. 404–411 (2011)
67. Theelen, B.D., Geilen, M.C.W., Stuijk, S., Gheorghita, S.V., Basten, T., Voeten, J.P.M., Ghamarian, A.H.: Scenario-aware dataflow. Tech. Rep. ESR-2008-08, Eindhoven University of Technology (2008)
68. Theelen, B.D., Geilen, M.C.W., Voeten, J.P.M.: Performance model checking scenario-aware dataflow. In: U. Fahrenberg, S. Tripakis (eds.) Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS, pp. 43–59. Springer (2011)
69. Theelen, B.D.: A performance analysis tool for scenario-aware streaming applications. In: Proceedings of the International Conference on Quantitative Evaluation of Systems, QEST, pp. 269–270 (2007)
70. Theelen, B.D., Florescu, O., Geilen, M.C.W., Huang, J., van der Putten, P.H.A., Voeten, J.P.M.: Software/hardware engineering with the parallel object-oriented specification language. In: Proceedings of the International Conference on Formal Methods and Models for Codesign, MEMOCODE, pp. 139–148 (2007)
71. Theelen, B.D., Geilen, M.C.W., Basten, T., Voeten, J.P.M., Gheorghita, S.V., Stuijk, S.: A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In: Proceedings of the International Conference on Formal Methods and Models for Co-Design, MEMOCODE, pp. 185–194 (2006)
72. Theelen, B.D., Katoen, J.P., Wu, H.: Model checking of scenario-aware dataflow with CADP. In: Proceedings of Design, Automation and Test in Europe, DATE, pp. 653–658 (2012)
73. Turjan, A., Kienhuis, B., Deprettere, E.F.: Realizations of the Extended Linearization Model. in Domain-Specific Embedded Multiprocessors (Chapter 9), Marcel Dekker, Inc. (2003)
74. Verdoolaege, S.: Polyhedral process networks. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, pp. 1335–1375. Springer (2013)
75. Verdoolaege, S., Nikolov, H., Stefanov, T.: pn: a tool for improved derivation of process networks. EURASIP Journal on Embedded Systems (2007)
76. Verdoolaege, S., Seghir, R., Beyls, K., Loechner, V., Bruynooghe, M.: Counting integer points in parametric polytopes using Barvinok's rational functions. Algorithmica (2007)
77. Wiggers, M.: Aperiodic multiprocessor scheduling. Ph.D. thesis, University of Twente (2009)
78. Willink, E.D., Eker, J., Janneck, J.W.: Programming specifications in CAL. In: Proceedings of the OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture (2002)
79. Zhai, J.T., Nikolov, H., Stefanov, T.: Modeling adaptive streaming applications with parameterized polyhedral process networks. In: Proceedings of the Design Automation Conference, DAC, pp. 116–121 (2011)