Embedded Systems

Joachim Keinert
Jürgen Teich

Design of Image Processing Embedded Systems Using Multidimensional Data Flow

Springer

Embedded Systems

Series Editors

Nikil D. Dutt
Peter Marwedel
Grant Martin

For further volumes:
http://www.springer.com/series/8563

Joachim Keinert · Jürgen Teich

Design of Image Processing Embedded Systems Using Multidimensional Data Flow

Springer

Joachim Keinert
Michaelstraße 40
D-90425 Nürnberg
Germany
joachim.keinert@yahoo.de

Jürgen Teich
Department of Computer Science 12
University of Erlangen-Nuremberg
Am Weichselgarten 3
D-91058 Erlangen
Germany
teich@informatik.uni-erlangen.de

ISBN 978-1-4419-7181-4 e-ISBN 978-1-4419-7182-1
DOI 10.1007/978-1-4419-7182-1
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2010937183

Preface

Overview of This Book

With the availability of chips offering constantly increasing computational performance and functionality, design of more and more complex applications becomes possible. This is particularly true for the domain of image processing, which is characterized by huge computation efforts. Unfortunately, this evolution risks to be stopped by the fact that employed design methodologies remain on a rather low level of abstraction. The resulting design gap causes increasing development costs or even project failure and thus threatens the technical progress.

Consequently, new design methodologies are urgently required. A corresponding review about the state of the art reveals that different approaches are competing in order to solve the above-mentioned challenges. The proposed techniques range from behavioral compilers accepting standard C or Matlab code as input, over block-based design methods such as Simulink and SystemC, to data flow specifications and polyhedral analysis. Each of them offers important benefits, such as quick and easy hardware prototyping, higher levels of abstractions, and enhanced system and algorithm analysis on different levels of granularity. However, a solution combining the advantages of all these approaches is still missing. As a consequence, system level design of image processing applications still causes various challenges. Corresponding examples are the lack to handle the resulting system complexity or to cover important algorithmic properties. Equally, the synthesis of high-performance hardware implementations is still difficult.

Fortunately, recent research is able to demonstrate that multidimensional data flow seems to be a promising technique solving these drawbacks, because it combines the advantages of block-based specification, data flow-related system analysis, and polyhedral optimization on different levels of granularity. These benefits enable, for instance, the verification of the application specification on a very high level of abstraction, the calculation of required memory sizes for correct algorithm implementation considering different design tradeoffs, and the synthesis of high-performance communication infrastructures and algorithm implementations.

However, despite these advantages, multidimensional data flow still lives quite in the shadows and is rarely adopted in both commercial and academic systems. Consequently, this book aims to give an encompassing description of the related techniques in order to demonstrate how multidimensional data flow can boost system implementation. In particular, this book identifies some of the requirements for system level design of image processing algorithms and gives an encompassing review in how far they are met by different approaches found in literature and industry. Next, a multidimensional data flow model of computation is intro-

duced that is particularly adapted for image processing applications. Its ability to represent both static and data-dependent point, local, and global algorithms as well as the possibility for seamless interaction with already existing one-dimensional models of computation permit the description of complex systems. Based on these foundations, it is shown how system analysis and synthesis can be simplified by automatic tool support. In particular, it is explained in detail, how the amount of memory required for correct implementation can be derived by means of polyhedral analysis and how communication primitives for high-speed multidimensional communication can be generated. Application to different examples such as a lifting-based wavelet transform, JPEG2000 encoding, JPEG decoding, or multi-resolution filtering illustrates the concrete application of these techniques and demonstrates the capability to deliver better results in shorter time compared to related approaches while offering more design flexibility.

Target Audience

As a consequence of the encompassing description of a system level design methodology using multidimensional data flow, the book addresses particularly all those active or interested in the research, development, or deployment of new design methodologies for data-intensive embedded systems. These are intended to process huge amounts of data organized in form of array streams. Image processing applications are particular prominent examples of this algorithm class and are thus in the focus of this book.

In addition to this primary target audience, the book is also useful for system design engineers by describing new technologies for inter-module communication as well as design tradeoffs that can be exploited in embedded systems. And finally, the book wants to promote multidimensional data flow and makes it more accessible for education and studies by giving an encompassing description of related techniques, use cases, and applications.

Prerequisites

Since this book bridges different technologies such as data flow modeling, polyhedral analysis, and hardware synthesis, important concepts necessary in the context of multidimensional data flow are shortly summarized before their application. By this means it is avoided to unnecessarily complicate understanding of the presented material. Nevertheless, it is assumed that the reader is skilled in fundamental maths such as vector spaces and matrix multiplication. Integer linear optimization is used in both memory analysis and communication synthesis. While the fundamentals are shortly summarized in this book, additional knowledge can deliver more detailed insights. Furthermore, familiarity with basic concepts of image processing helps in understanding the presented material. For a clearer picture of the overall concepts, some experiences in software and hardware implementation are helpful, although not strictly necessary.

How the Book Is Organized

The content of this book can be categorized in four major parts as depicted in Fig. 1:

- Introductory background information
- Related techniques
- Multidimensional modeling
- Analysis and synthesis

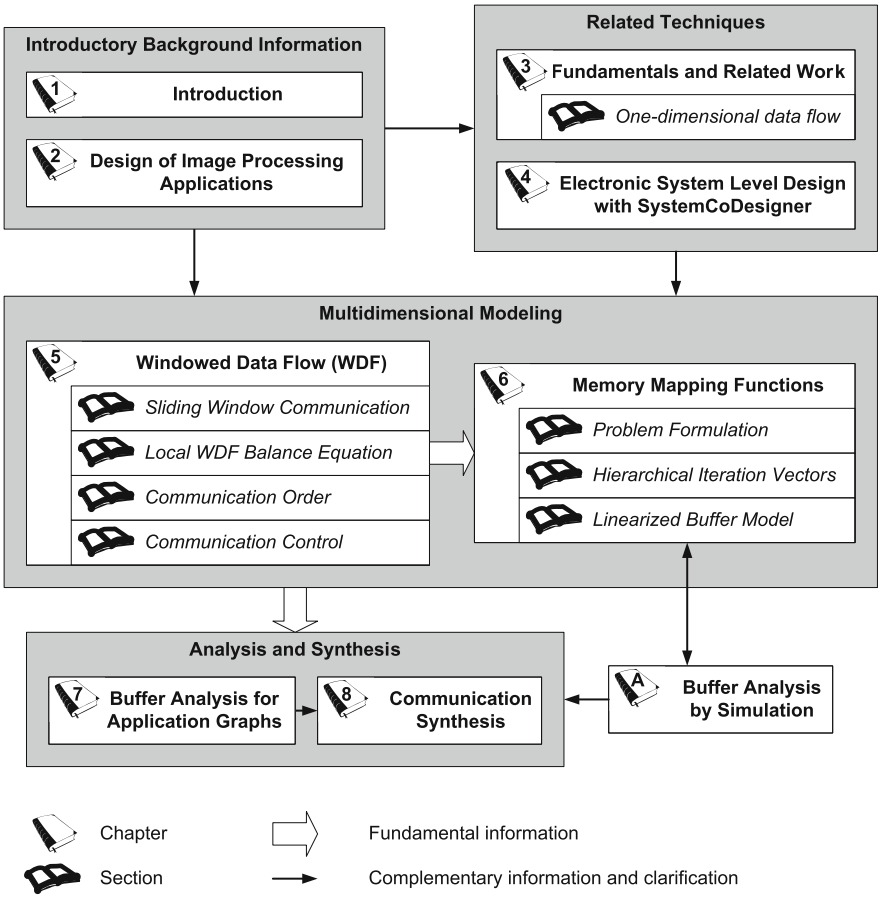


Fig. 1 *Arrows* depict concepts and information used in the corresponding target book parts. *Bold arrows* define information that is fundamental for understanding the chapter which the arrow points to

The first part containing the *introductory background information* basically aims to clarify the purpose and the main focus of the described technologies. To this end, Chapter 1 (Introduction) explains the need for new design technologies and overviews the overall design flow described in this book. Chapter 2 (Design of Image Processing Applications) adds some general considerations about the design of image processing embedded systems and exemplifies a JPEG2000 encoder in order to clarify the type of algorithms that are addressed in this book. The insights gained during its manual development have been formulated into a corresponding set of requirements that shall be taken into account in the rest of this monograph.

The second part about *related techniques* summarizes concepts useful for system level design of image processing applications. To this end, Chapter 3 (Fundamentals and Related Work) reviews related approaches and evaluates their benefits for system level design of image processing applications. In particular, it investigates a huge amount of different specification techniques ranging from sequential languages enriched by communicating sequential processes up to multidimensional data flow models. Furthermore, it evaluates their ability to describe complex image processing applications. Additionally, Chapter 3 also summarizes the capacities of several behavioral compilers, buffer analysis techniques, and communication synthesis approaches. Furthermore, it discusses several system level design methodologies. Subsequently, Chapter 4 presents an overview on the ESL tool SystemCoDesigner, since it shall serve as an example how to combine multidimensional system design with available ESL techniques. Furthermore, a case study in form of a Motion-JPEG decoder demonstrates the potential of ESL design for image processing applications and discusses lessons learned for both application modeling and synthesis.

Both Chapters 3 and 4 are thus intended to provide further insights into system level design of embedded systems. In particular, they aim to clarify the benefits of multidimensional data flow and its interaction with existing technologies. Consequently, both chapters can be consulted as needed. The only exception represents Section 3.1.3 (One-Dimensional Data Flow) that is recommended for all those not being familiar with one-dimensional data flow models of computation.

Detailed discussion of multidimensional system level design then starts with the third and central book part about *multidimensional modeling*, subdivided into two chapters. Chapter 5 (Windowed Data Flow (WDF)) introduces the *windowed data flow (WDF)* model of computation used for application modeling in the remainder of this monograph. This includes both a theoretical discussion and the application to two concrete examples, namely the binary morphological reconstruction and the JPEG2000 lifting-based wavelet transform. In particular Sections 5.1 (Sliding Window Communication), 5.2 (Local WDF Balance Equation), 5.3 (Communication Order), and 5.4 (Communication Control) introduce fundamental concepts required in the remainder of this monograph. The same holds for Sections 6.1 (Problem Formulation), 6.2 (Hierarchical Iteration Vectors), and 6.3 (Memory Models). It discusses fundamental concepts of memory organization within multidimensional arrays as required in the remaining chapters. In particular, a study is performed that compares two different memory allocation functions in terms of memory efficiency.

Based on those multidimensional modeling concepts, the fourth part of this book then addresses system *analysis and synthesis*. More precisely, Chapter 7 (Buffer Analysis for Complete Application Graphs) is dedicated to the question of automatic buffer size determination required for correct system implementation. Taking the results of Chapter 6 into account, Chapter 7 presents a method for polyhedral buffer size requirement calculation in case of complex graph topologies. Application to several examples like the lifting-based wavelet transform, JPEG2000 block building, and multi-resolution image filtering demonstrates that the resulting analysis times are suitable for system level design of complex applications and competitive with alternative approaches. Furthermore, it will be shown that analytical methods deliver better solutions in shorter time compared to buffer analysis via simulation, while offering more design tradeoffs.

The so-derived buffer sizes can be directly used for efficient communication synthesis. To this end, Chapter 8 (Communication Synthesis) considers the derivation of high-speed hardware communication primitives from WDF specifications. This allows to interconnect hardware modules by a high-performance point-to-point communication. The corresponding technology can be used both for classical hardware design and for system level design with

multidimensional data flow. To this end, Chapter 8 presents all analysis steps required to transform a WDF edge into an efficient hardware implementation. Application to different examples originating from a JPEG2000 encoder and a JPEG decoder demonstrates the benefits of the methodology. Furthermore, Chapter 8 illustrates how the hardware communication primitive can be combined with a behavioral synthesis tool in order to handle overlapping sliding windows.

The book is concluded by Chapter 9. Appendix A (Buffer Analysis by Simulation) then delivers some supplementary information concerning a buffer analysis performed during simulation as applied in Chapter 6. Appendix B summarizes the abbreviations used within this book while Appendix C contains repeatedly used formula symbols.

Distinct Features and Benefits of This Book

Although combining the benefits of various design methodologies such as block-based system design, high-level simulation, system analysis, and polyhedral optimization, multidimensional data flow is still not very widely known. Whereas there exist several books discussing the one-dimensional counterparts, similar literature is not available for multidimensional modeling. Consequently, this book aims to provide a detailed insight into these design methodologies. Furthermore, it wants to provide an encompassing review on related work and techniques in order to show their relation to multidimensional data flow.

By these means, the book is intended to contribute to the promotion of multidimensional data flow in both academic and industrial projects. Furthermore, it aims to render the subject more accessible for education. In more detail, this monograph provides the following contributions:

- First encompassing book on multidimensional data flow covering different models of computation. In particular, both modeling, synthesis, and analysis are discussed in detail demonstrating the potential of the underlying concepts.
- The book bridges different technologies such as data flow modeling, polyhedral analysis, and hardware synthesis, which are normally only considered independently of each other in different manuscripts. Consequently, their combination possess significant difficulties, since even the terminology used in the different domains varies. By combining the above-mentioned technologies in one book and describing them in a consistent way, the book can leverage new potential in system design.
- Analysis in how far multidimensional data flow can better fit the designers' requirements compared to alternative description techniques, such as well-known one-dimensional data flow or communicating sequential processes.
- Description of how multidimensional data flow can coexist with classical one-dimensional models of computation.
- Explanation of a novel architecture for efficient and flexible high-speed communication in hardware that can be used in both manual and automatic system design and that offers various design alternatives trading achievable throughput against required hardware sizes.
- Detailed description of how to calculate required buffer sizes for implementation of static image processing applications. Various illustrations help to apply the method both in ESL tools and in manual system design.
- Compared to books on geometric memory analysis, a significant extension assures that this method can be applied for data reordering and image subsampling in hardware implementations.

- New concepts for embedded system design, such as trading communication buffer sizes against computational logic by different scheduling mechanisms.
- Various experimental results in order to demonstrate the capabilities of the described architectures and design methods. In particular, several example applications such as JPEG2000 encoding, Motion-JPEG decoding, binary morphological reconstruction, and multi-resolution filtering are discussed.

The content of this book has been created, edited and verified with highest possible care. Nevertheless, errors and mistakes of any kind cannot be excluded. This includes, but is not restricted to, missing information, wrong descriptions, erroneous results, possible algorithmic mistakes or citation flaws causing that algorithms may not work as expected.

Erlangen, Germany

Joachim Keinert
Jürgen Teich

Acknowledgments

This book is the result of a 5-year research activity that I could conduct in both the Fraunhofer Institute for Integrated Circuits IIS in Erlangen and at the Chair for Hardware-Software-Co-Design belonging to the University of Erlangen-Nuremberg. This constellation allowed me to combine the theory of system level design with the requirements for design of high-performance image processing applications. In particular, the experiences gained during the development of multiple embedded devices for image processing within the Fraunhofer research organization have been a valuable inspiration for the presented technologies. Therefore, I want to express my gratitude toward all those who supported me within this period of time.

I especially want to thank Prof. Jürgen Teich for supervising the underlying research activity and for his motivation to tackle the right mathematical problems, in particular data flow graph models of computation and polyhedral analysis. My superior at the Fraunhofer Institute for Integrated Circuits, Dr. Siegfried Fößel, also merits special thanks for the provided support and for his help in making this book possible. Dr. Christian Haubelt from the University of Erlangen-Nuremberg contributed to this book by means of multiple reviews and by his coordination of the SYSTEMCODESIGNER tool. This enabled its extension with a multidimensional design methodology in order to demonstrate the underlying concepts and techniques. In this context I could particularly profit from the work of Joachim Falk, designer of the SYSTEMOC library and of a huge tool set for manipulation of the resulting graph topologies. Similarly, the cooperation with Hritam Dutta and Dr. Frank Hannig has been an important prerequisite for combining data flow-based system design with polyhedral analysis. In addition, the various discussions with Prof. Shuvra Bhattacharyya, member of the University of Maryland, helped to better understand and evaluate the advantages of multidimensional system design. And of course I also want to thank Mr. Charles Glaser from Springer for his assistance in achieving this book.

Erlangen, Germany — Joachim Keinert
May 2010

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1
Introduction

1.1 Motivation and Current Practices

In the last few years, the progress in the design of embedded systems has been driven by the continuous development of new semiconductor devices offering more functionality than previous generations. Especially image processing applications could profit from this trend, because they are typically computationally intensive. Consequently, modern embedded devices permit the real-time execution of complex applications that some years ago would have been challenging even for huge workstations. Due to this fact, new application domains emerged ranging from digital acquisition, manipulation, and projection of cinematographic movies, over new medical diagnosis technologies, to hundreds of powerful consumer devices handling various multi-media content including images, sound, or video.

However, the constant increase of functionality provided per chip has led to a design productivity gap, which according to the International Technology Roadmap for Semiconductors [6] threatens to break the long-standing trend of progress in the semiconductor industry. Whereas Moore's law, which postulates the doubling of components per chip within 2 years, is considered to outlast at least the year 2020, the number of available transistors grows faster than the ability to meaningfully design them. This trend is even aggravated by the broad arrival of software programs or reconfigurable chips into modern embedded devices. According to [6], the demand for software (which also includes *field-programmable gate array (FPGA)* designs) is currently doubling every 10 months, while the productivity in this domain lies far behind and doubles only every 5 years. As a consequence, design costs are becoming the greatest threat to continuation of the semiconductor roadmap. Whereas *manufacturing non-recurring engineering (NRE) costs* are in the order of millions of dollars, they are routinely exceeded by the *design NRE costs*, which can reach tens of millions of dollars. Even worse, the difficulty to create functionally correct designs leads to many expensive silicon re-spins that could have been avoided by better design methodologies. Additionally, rapid technology changes shorten the product life cycles, which makes the return on investment more and more difficult. Consequently, time to market gets a critical issue for the semiconductor industry. Unfortunately, this fundamentally contradicts the fact that currently design and verification times are difficult to plan because they show a high uncertainty. Reference [6] thus concludes that the productivity or design technology gap represents an important crisis, which has to be addressed in the next 15 years: "Failure to effectively develop and deploy [. . .] new design methodologies will break the long-standing trend of progress in the semiconductor industry."

J. Keinert, J. Teich, *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*, Embedded Systems,
DOI 10.1007/978-1-4419-7182-1_1, © Springer Science+Business Media, LLC 2011

In order to be beneficial, these new design methodologies must support the developer in managing system complexity and verification. Whereas today each technology generation requires designers to consider more issues, new analysis methods and tools must be invented that permit to make critical design decisions on a higher level of abstraction and that allow exploration of the design in early stages. Unfortunately, today designers are still constrained to reason about systems at various levels of abstraction like block diagrams, state charts with little support from design automation tools. Consequently, new *electronic system level (ESL)* design tools are required that consider complete systems instead of individual modules. Furthermore, they have to provide techniques helping the designer to model complex systems on a high level of abstraction and to refine them until reaching the physical hardware implementation. This requires two fundamental aspects. The first addresses automatic analysis and optimization of both individual modules and complete systems in order to ease the burden of the developers such that they can focus on the critical design decisions. Furthermore, new verification techniques must be developed, since well-known verification techniques like RTL simulation are completely inadequate to cope with the increasing system complexity. As a consequence, due to time constraints, today's development processes typically settle for partial verification only. Unfortunately, this tremendously increases the risk for erroneous products, and thus expensive re-spins and recalls. In particular, inter-module communication is a constant source of troubles, because its verification only occurs at a very late stage, if at all.

However, despite this urgency, the transition from low-level development processes to ESL design tools in both industrial and academic projects stays very limited. This has two major reasons, namely costs and inadequateness in existing techniques and methodologies. Consequently, many research and commercial activities focus on behavioral synthesis tools like *GAUT* [73], *SPARK* [123], *CyberWorkBench [296], Forte Cynthesizer* [107], *ImpulseC* [151], *CatapultC [209], PARO* [130], *MMAlpha* [120], or *Synfora PICO Express*. All of them help to quickly generate hardware accelerators from untimed C or Matlab code and can thus be very useful for system design. Nevertheless, they face the difficulty that general C or Matlab code is complex to analyze in an automatic manner, because it has shown to be undecidable [132]. In other words, questions about program termination, for instance, cannot be decided by any imaginable algorithm. Consequently, the capability of the above-mentioned tools for system level analysis, such as automatic buffer size determination, is typically very restricted or even completely absent. Furthermore, behavioral compilers are typically limited to simple communication methods like FIFOs or signals. Image processing applications, on the other hand, employ complex communication patterns like *sliding windows* or *out-of-order communication* where data are read in a different order than written.

The same problem can be found for *SystemC*-based design flows and tools such as *Forte Cynthesizer* [107], *CoFluent Studio* [55], or *System-On-Chip Environment (SCE)* [90]. Whereas *transaction level models (TLM)* in SystemC gain acceptance as a tool-independent modeling technique for complex systems, the possibilities for analysis and verification other than by simulation are typically limited. Also *Simulink* [207], a widely applied tool for design of digital signal processing applications, employs a very general model of computation, which is difficult to analyze and optimize in an automatic manner. Furthermore, whereas high-level descriptions exchange complete images at once, a later hardware implementation requires signal communication, needing thus manual refinement.

Here, *data flow* models of computation are an interesting option for signal processing applications, because they provide well-defined execution semantics offering possibilities for analysis and optimization [9–11, 42, 71, 111, 112, 117, 156, 211–214, 228, 267, 272, 273,

301, 309]. To this end, the application functionality is split into a set of *processes*, also called *actors*, which exchange data over dedicated *communication channels*. Unfortunately, existing data flow-based system level design tools like *Ptolemy* [49], *PeaCE* [124], or *Grape-II* [187] only partially solve the problems identified above, because they are typically limited to one-dimensional models of computation like *synchronous data flow* [193] or *cyclo-static data flow* [45, 100, 240]. These have, however, difficulties in representing important properties of image processing applications, such as overlapping windows, out-of-order communication, parallel data access, or data parallelism. This leads to very complicated application models excluding many powerful analysis techniques such as polyhedral buffer size estimation.

In order to solve these difficulties, multidimensional models of computation are getting more and more into the focus of interest. *IMEM* [189] provides a data flow like design environment for *sliding window* image processing algorithms. It enables, in particular, the automatic determination of the required buffer sizes and the quick synthesis of efficient hardware buffer structures. *Gaspard2* [37], on the other hand, uses a more general model of computation and considers aspects such as UML modeling, presence of control flow, model transformations, and FPGA synthesis. The SystemCoDesigner tool [133, 170] offers the capability to combine both one-dimensional and multidimensional models of computations in order to describe complex applications. In particular it proposes the *windowed data flow* [164–166] model of computation based on which it permits automatic buffer size calculation for applications including out-of-order communication. Furthermore, high-performance communication primitives can be automatically synthesized in hardware. *Daedalus* [278] finally starts from static loop programs and automatically translates them into a data flow process network that can be mapped onto embedded multi-processor systems.

However, since these advances in research are quite recent, the use of multidimensional data flow in both academic and industrial projects is quite limited. Consequently, this book aims to give a corresponding overview on system level design, analysis, and synthesis of image processing applications using multidimensional data flow. In particular,

- It discusses how sliding window algorithms and out-of-order communication can be represented with multidimensional models of computation.
- It shows a method for representation of control flow in both one- and multidimensional models of computation.
- It illustrates the benefits of polyhedral analysis applied to multidimensional data flow, which—compared to simulation—permits the derivation of tighter buffer sizes in shorter time. Additionally, the impact of different implementation alternatives is investigated.
- It explains in detail how multidimensional data flow can be used to synthesize high-performance hardware communication primitives for both in-order and out-of-order communication. In particular, the book demonstrates how static compile time analysis helps to reduce the occurring run-time overhead while offering multiple design tradeoffs.

To this end, this monograph describes the different aspects of a design methodology relying on multidimensional data flow. While this enables the intuitive representation of array-based algorithms, it cannot cover all aspects of complex applications. Consequently, the book describes in addition how multidimensional data flow can be combined with classical one-dimensional models of computation such as synchronous data flow (SDF) or FunState-like models. For this purpose, the SystemCoDesigner ESL tool is chosen as an exemplary one-dimensional system level design environment and extended by a multidimensional model of computation particularly adapted for image processing applications. This leads to a seam-

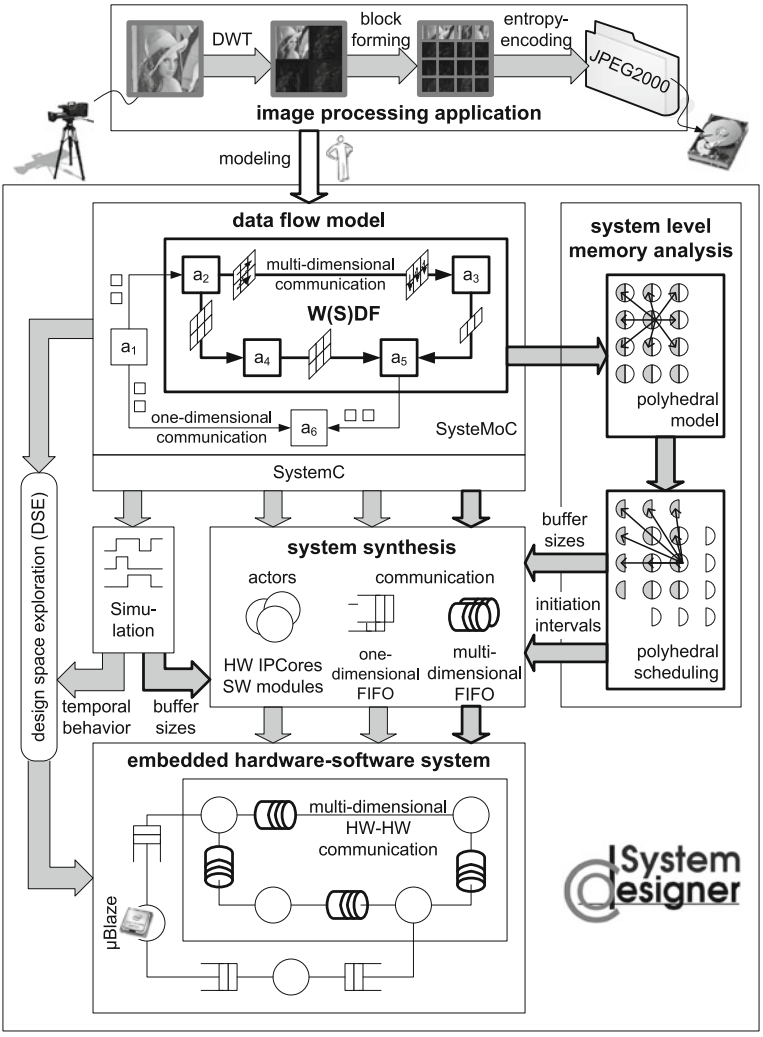


Fig. 1.1 Overview of a multidimensional design methodology for image processing applications. The *bold* parts encompass the major steps discussed in more detail in this book

less design flow for implementation of embedded image processing applications as summarized in the following section.

1.2 Multidimensional System Level Design Overview

The design of image processing systems consists of various steps described in more detail later on in this book. A corresponding overview about their relation and interaction is given

in Fig. 1.1. In more detail, this monograph follows a design methodology consisting of the following steps:

1. Specification of the image processing application by means of a data flow model of computation that encompasses both multidimensional and one-dimensional subsystems.
2. System verification by simulation or analytical methods.
3. System level analysis including automatic *design space exploration (DSE)* and memory size determination. The latter can be performed either by simulation or by polyhedral analysis.
4. Synthesis of the components required for building an embedded system. In particular, this includes generation of communication channels for high-speed communication in form of *one- and multidimensional FIFOs*. Furthermore, hardware and software modules for the involved actors have to be derived from the data flow specification.
5. Construction of the complete embedded system by assembling the different hardware and software components.

The bold parts in Fig. 1.1 are related to multidimensional data flow and will be discussed in detail within this book. Starting from a desired application, the first step consists in creation of a corresponding high-level model by the designer. This model specifies the functional behavior of the image processing application, such as a JPEG2000 encoder exemplified in Fig. 1.1. Such applications typically consist of filter operations like a wavelet transform, data reordering like code-block building, or data-dependent operations like entropy encoding. To this end, the book will discuss a multidimensional data flow model of computation, called *windowed data flow (WDF)*, as well as its static counterpart *windowed synchronous data flow (WSDF)*. In both cases, the actors, which provide the system functionality like a wavelet transform, exchange multidimensional images or arrays as exemplified in Fig. 1.1. However, in order to permit buffer-efficient implementations, these arrays are not produced and consumed as a whole unit. Instead, each source actor invocation only generates a smaller part, the so-called *effective token*, while the sink samples the image with possibly overlapping windows. By this approach, it is possible to represent well-known point, local, and global image processing algorithms like gamma transform, median filtering, or image segmentation. Integration of WDF and WSDF into SystemoC, a SystemC library for actor-oriented modeling of applications, enables the creation of executable specifications. Since SystemoC supports different models of computation like *synchronous data flow (SDF)*, *cyclo-static data flow (CSDF)*, or *Kahn process networks (KPN),* applications with varying potential for analysis and optimization can be represented in a uniform manner. By these means, WDF offers thus the following benefits:

- Intuitive representation of sliding window algorithms including virtual border extension and parallel data access
- Capability to represent global, local, and point algorithms and the resulting memory requirements
- Capability to represent task, data, and operation parallelism
- Possibility of expressing control flow
- Explicit specification of communication orders in order to support data reordering
- Tight interaction between one- and multidimensional data flow. This allows description of complex applications like JPEG2000 encoders order Motion-JPEG decoders
- Flexible delays
- Simplified analysis by restriction to rectangular window patterns without restraining the applicability of WDF

- Capability of data dependency analysis in order to support powerful polyhedral analysis
- Provision of powerful communication semantics simplifying module reuse. This is achieved by means of a communication primitive that allows to read or write several data elements in parallel and in possibly different orders
- Efficient verification by a high-level simulation of the functional specification and by formal analysis that guarantees bounded memory

After verification via simulation or analytical methods, the modeled application can be mapped to an embedded system consisting of processors, hardware accelerators, and the corresponding communication infrastructure. Although W(S)DF actors can principally be executed on a processor, this monograph focuses on situations where W(S)DF subsystems are implemented on an ASIC or FPGA. For the other actors, an automatic design space exploration decides which of them to execute on a dedicated hardware accelerator and which ones shall better run on a processor. In case there exist several IP cores performing the same functionality, but with different performance and resource requirements, the automatic design space exploration selects the best suited alternative. For this purpose, different configurations of the overall system are evaluated by evolutionary algorithms according to various criteria like necessary chip size and achievable latency and throughput. The values of the latter are obtained by high-level architectural simulation.

Once the optimum system configuration has been found, the next step consists in automatic system synthesis. This comprises generation of the hardware accelerators, instantiation of the micro-processors and the corresponding software, and communication synthesis. In this book, particular focus has been put on the automatic derivation of high-speed hardware primitives for multidimensional communication. Although they provide a FIFO-like interface for easy system integration, they significantly outperform classical FIFOs by

- permitting parallel access to several data elements on both the read and write side,
- allowing for one read and write operation per clock cycle while supporting very high clock frequencies,
- being able to exploit tradeoffs between achievable throughput and required hardware resources, and
- handling different read and write orders as, for instance, occurring in JPEG2000 tiling or block building.

The latter aspect eases actor reuse, because when implementing an actor it has not to be known in which order the predecessor generates or the successor requires the processed data. Instead, the communication channels reorder the data accordingly. In order to reduce the run-time overhead as much as possible, a static compile time analysis determines efficient circuits for fill level control and address generation. They have to guarantee (i) that the source never overwrites data still required by the sink, (ii) that the sink only reads valid data, (iii) and that the data are presented in the correct order to the sink. Compared to behavioral synthesis performed for one-dimensional actors, experiments show that the developed new communication primitive, called *multidimensional FIFO*, not only achieves higher throughput but also requires less resources.

An important step during communication synthesis is the determination of the required buffer sizes. For this purpose, a polyhedral model can be derived for the purely static application parts that can be covered by the windowed synchronous data flow (WSDF) model of computation. Such a polyhedral model represents the actor invocations and the resulting data dependencies in a mathematically compact manner in form of a point grid. Each point corresponds to an actor execution while dependency vectors indicate the flow of data. Such a

representation can thus be used for efficient determination of valid schedules by means of grid shifting, such that data elements are not read before being produced. This information can be used for calculation of the required buffer sizes. Alternatively, buffer analysis by simulation can be performed, which supports more general scheduling functions, and which can also be applied in the presence of control flow. The combination of these two techniques permits a complementary buffer analysis of WSDF graphs, offering several benefits as discussed in this monograph:

- Comparison of two different memory mapping schemes applicable to WDF edges
- Inclusion of both in-order and out-of-order communication where data are read in a different order than written
- Support of arbitrary scheduling functions in case of analysis by simulation. This allows to easily employ different scheduling strategies like self-timed parallel execution or "as soon as possible" (ASAP) scheduling
- Support of complex graph topologies including split and joins of data paths, down- and upsampling as well as feedback loops and multiple, unrelated sources
- Consideration of different scheduling alternatives for image up- and downsampling leading to tradeoffs between required communication memory and computational logic
- Generation of throughput-optimal schedules in case of polyhedral analysis
- Availability of a computationally efficient algorithm, which only requires solving small integer linear programs (ILPs) that only depend on the communication parameters of one single edge. This is important because ILPs belong to the class of NP-complete problems leading to exponential computation effort. In other words, by avoiding ILPs whose size increases with the number of actors contained in the data flow graph, it is possible to practically apply the buffer analysis method to huge application specifications
- Existence of two different solution methods for the integer linear programs encompassing both standard ILP solvers and exhaustive search. This two-folded approach permits to profit from the capacities of modern ILP solvers to quickly derive the searched solution while having a fall-back scenario in case the problem becomes intractable.

Once all the individual modules are successfully generated, they can be assembled to the overall embedded system. In case of the selected SYSTEMCODESIGNER ESL design tool, this is performed by usage of the *Xilinx Embedded Development Kit* (*EDK*) together with several commercial synthesis tools.

After this short summary of the overall design methodology followed in this book, the next chapter will now start to introduce the requirements on system level design environments for image processing applications. The identified needs will then be taken into account in the remainder of this book.

Chapter 2
Design of Image Processing Applications

In today's information society, processing of digital images becomes a key element for interaction with our environment and for transport of important messages. Consequently a huge effort has been undertaken in developing algorithms for image enhancement, transformation, interpretation, and compression. Whereas in the past computational capacities have shown to be the limiting factor, the recent progress in design of semiconductor devices allows implementation of powerful algorithms. Corresponding examples can be found in constantly increasing function ranges of cellular phones, in various types of object recognition, in complex acquisition of volumetric scans for medical imaging, or in digitization of production, transmission, and projection of cinematographic content.

The intention of this section is twofold. First, it wants to give a rough survey of the different kinds of algorithms used in complex image processing systems in order to explain the focus of the multidimensional design methodologies discussed in this book. Second, it aims to motivate for multidimensional electronic system level design flows by introducing several challenges that have to be coped with when implementing image processing algorithms. Both issues shall be covered by means of the JPEG2000 compression system [152, 198] because it represents a typical application with both complex algorithms and high computational effort. Furthermore, the experiences gained during manual implementation of a corresponding hardware encoder shall help to emphasize the benefits of an electronic system level approach using multidimensional data flow. Additionally to this motivating example, this book also studies several other applications like a Motion-JPEG decoder, morphological reconstruction, or a bilateral multi-resolution filter. This not only avoids that the discussed methodologies are too centric to one single scenario, but also permits to better emphasize the various advantages offered by the described design methodologies. However, in order to not overburden this motivating chapter, the corresponding applications will be discussed later on in the according chapters making use of them.

The remainder of this chapter is structured as follows. Section 2.1 introduces different fundamental categories of algorithms that occur in image processing applications and that have hence to be supported by useful system level design tools. In particular, it explains the notion of *static* and *dynamic*, as well as *point*, *local*, and *global* algorithms. In order to make them more concrete, Section 2.2 discusses the processing steps required for JPEG2000 image compression. In particular, it aims to clarify the different scenarios addressed in this monograph. Since JPEG2000 compression shows a huge computational complexity, its implementation on an embedded system requires a highly parallel architecture. Consequently, Section 2.3 presents some fundamental considerations on parallel execution of image processing applications that must be taken into account when developing new system level design methods.

J. Keinert, J. Teich, *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*, Embedded Systems,
DOI 10.1007/978-1-4419-7182-1_2, © Springer Science+Business Media, LLC 2011

Next, Section 2.4 enumerates the necessary implementation steps. Furthermore, it describes difficulties encountered during a manual implementation of a JPEG2000 hardware encoder that are related to the shortcomings of current design techniques. From these results, Section 2.5 derives requirements for a system level design methodology as it will be discussed in the following chapters of this monograph. Section 2.6 finally puts the identified requirements into relation with the remaining chapters for this monograph.

2.1 Classification of Image Processing Algorithms

Typical image processing applications use a huge amount of different algorithms. According to their behavior, they are often classified as being *static* or *dynamic*. The latter are also called *data dependent*. In other words, the decision which operation to execute depends on the currently processed (pixel) value. Entropy encoding is a typical example because the amount of produced output data strongly depend on the input pixel values. *Static* algorithms, on the other hand, transform the input image in a predefined manner. In other words, the operations to execute can be predicted beforehand and do not depend on the actual pixel values.

The distinction between these two algorithm classes is important because of their different characteristics. Dynamic algorithms, for instance, typically do not allow prediction of run-time or required buffer memory. In the best case, an upper limit can be derived, but mostly this task is highly complex. Hence, often simulation is the only possibility to derive important algorithm characteristics. This, however, significantly complicates optimizations like parallelization or automatic code modifications.

Static algorithms, on the other hand, are much more regular and can often be analyzed at compile time. Consequently, they enable powerful optimizations like buffer minimization or parallel execution. Furthermore, as static algorithms often execute on huge amount of data, it is of utmost importance to select the implementation alternative that fits best the application requirements. Whereas mobile devices, for instance, demand for energy-efficient implementations, interactive applications in medical imaging focus on small latency values. Consequently, system designers are responsible for not only implementing the correct functionality but also trading various design objectives like throughput, latency, energy consumption, and chip sizes.

In this context, it is important to distinguish between *global*, *local*, and *point* algorithms. The latter only require 1 input pixel in order to calculate a corresponding output pixel. Well-known examples are color inversion, gamma and color space transforms, or brightness adjustment. Local algorithms, on the other hand, need a small sub-region of the input image in order to generate the associated output pixel. Often, this kind of image manipulations can be implemented as *sliding window* operations, where a window traverses each input pixel. Median filtering, edge detection, and wavelet and discrete cosine transform are only some of the typical applications belonging to this category. Global algorithms finally cannot start operation until the complete input image is available. Image segmentation and line detection by the Hough transform are exemplary representatives for this class of image manipulations.

Whereas point algorithms are often trivial to implement and only require modest hardware resources, local algorithms are already more challenging because of increased memory requirements and a typically higher computational work load. Global algorithms finally are often less regular and more difficult to parallelize, which makes them particularly adapted for execution on general-purpose or embedded processors.

In order to clarify these concepts, the next section considers a concrete application in form of a JPEG2000 encoder. It requires interaction of different algorithms ranging from simple point algorithms over standard and enhanced sliding window algorithms up to highly complex dynamic application parts. The major intention of this analysis is the identification of different requirements for a system level design flow for image processing applications.

2.2 JPEG2000 Image Compression

Design of a complex application often starts by partitioning the overall system into several communicating modules, also called *blocks* or *processes*. This divide-and-conquer principle allows mastering of the complexity. Furthermore, it offers the possibility to implement and verify the modules independently from each other, such that several engineers can work in parallel.

Figure 2.1 shows a corresponding block diagram for a JPEG2000 encoder. JPEG2000 [152] is an algorithm for compression of still images, which is typically employed for applications with high exigencies to image quality. Furthermore, it can be advantageously used for compression of movies when fast access to individual images is required. Such scenarios can, for instance, be found in digital film production, content archiving, or medical applications. In comparison to the well-known JPEG standard [153], JPEG2000 offers several enhanced capabilities. In particular, it is possible to extract representations with lower resolutions and different image qualities from a compressed image without needing to decompress the overall image. Furthermore, both lossless and lossy compression are supported together with the capability to access only those image regions that are of particular interest.

In order to achieve these capabilities, the image has first to be transformed into the input format required by the JPEG2000 standard. In most cases, this does not require further processing. However, sometimes images are transmitted by the *progressive segmented frame (PSF)* technique [304]. In these cases, the odd and even image lines are transmitted as two consecutive fields, which have to be reordered before the JPEG2000 compression. Next, the image can be divided into several sub-images of equal size, the so-called *tiles*. As each of the following operations[1] can be applied to an individual tile without requiring the other tile data, this tiling operation increases the achievable parallelism during compression, reduces the required memory resources for the individual processing steps, and simplifies extraction of sub-images.

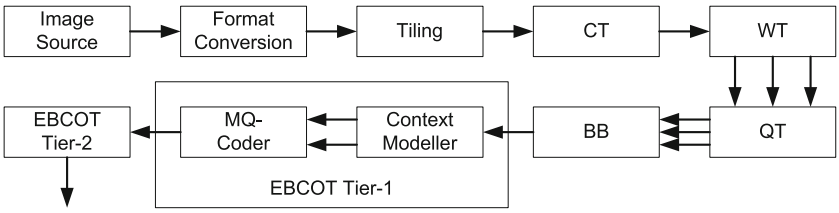


Fig. 2.1 Block diagram of a JPEG2000 encoder

[1] With some restrictions for the EBCOT Tier-2

Table 2.1 Filter types for generation of the wavelet subbands. *L* stands for low pass, *H* for high pass

	LL	LH	HL	HH
Horizontal filter	L	L	H	H
Vertical filter	L	H	L	H

After an optional *color transform (CT)*, the wavelet transform aims to reduce the spatial redundancy contained in the image. For this purpose, the image is successively decomposed into several subbands as shown in Fig. 2.2. In the first transformation step, the input image is decomposed into four subbands labeled as *LL0*, *LH0*, *HL0*, and *HH0*. Each subband is derived by filtering and downsampling the input image in both horizontal and vertical directions as summarized in Table 2.1. Further *decomposition levels* can be generated by successively filtering the LL subband as depicted in Fig. 2.2.[2]

For the filter operation, different wavelet kernels are available. In case of the JPEG2000 algorithm, typically either a lossless 5–3 or a lossy 9–7 kernel is used. In both cases, the filtering can be split into a horizontal and a vertical step. Each of them is performed by applying a *sliding window* algorithm. Figure 2.3 illustrates the principles of such an algorithm for the 5–3 kernel by means of a 1×5 tap-filter including vertical subsampling by a factor of 2. It traverses the input image in *raster-scan order*, hence from left to right and from top to bottom. For each filter position, a corresponding output pixel is calculated. The vertical downsampling is achieved by moving the window 1 pixel in horizontal direction and 2 pixels in vertical direction. As a result, the output image has only half the height of the input image. Since the sliding windows overlap, the input pixels are read several times. Because repetitive access to external memories is very expensive in terms of achievable throughput and energy consumption, local buffers in form of on-chip memory or caches can be used for improved system performance. Special consideration is required at the image borders, because here

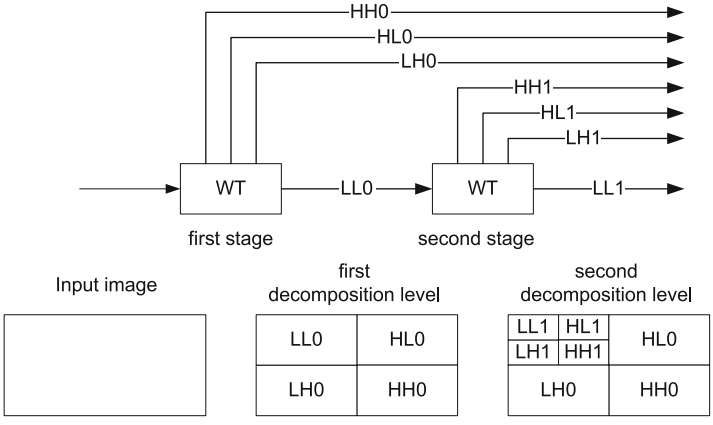


Fig. 2.2 Wavelet transform

[2] Part-1 of the JPEG2000 standard demands the usage of the so-called *Mallat decomposition* [203], in which only the LL subband is further refined into corresponding subbands.

parts of the window transcend the input image. This can be represented by virtually extending the image borders as shown in Fig. 2.3. In case of the wavelet transform, the values of these extended border pixels have to be derived by symmetric mirroring. Furthermore, due to occurring subsampling, the extended upper and lower borders might have different sizes. The same holds for the horizontal transform, except that the required local buffers are typically much smaller.

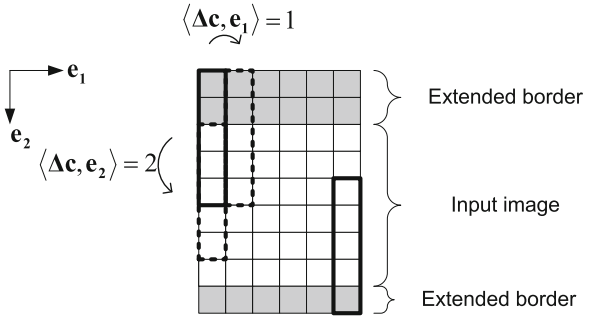


Fig. 2.3 Sliding window for vertical filtering with downsampling

For the wavelet transform, this sliding window algorithm can be implemented by means of two different strategies: (i) standard *convolution* and (ii) *lifting* based [81, 152]. The first approach uses two different filter kernels, one for the low-pass, the other for the high-pass filtering. The *lifting scheme*, on the other hand, exploits the particular properties of the filter coefficients in order to combine both the low-pass and the high-pass filtering to one sliding window algorithm, thus reducing the required computational effort. Furthermore, it permits a more efficient memory structure. Figure 2.4 illustrates the corresponding calculation scheme for a vertical 5–3 wavelet filter as used in the JPEG2000 algorithm [152]. In accordance to Fig. 2.3, the image height is supposed to amount 6 pixels. On the left-hand side, the corresponding vertical sliding window positions are depicted. The filter traverses the image in raster-scan order by moving first in direction $\mathbf{e_1}$ until reaching the end of the image. Then, it returns to the beginning of the image row and moves by two in direction $\mathbf{e_2}$. For each filter position, a low-pass (L) and a high-pass pixel (H) is generated. Each circle in Fig. 2.4 corresponds to several arithmetic operations that operate on the depicted input data in order to generate the required output data. The bold parts put the corresponding sliding window into relation with its arithmetic operations and the resulting output pixels.

Based on this representation, it can be seen that the wavelet filter is occupied only every second line. In other words, assuming that the image source generates 1 pixel per clock cycle, the filter will be idle half the time because the sliding window moves by 2 pixels in vertical direction. Hence, it can only operate every other image row. This observation can be exploited for hardware implementations that share the computation resources between all decomposition levels [200]. This is particularly useful for the 9–7 wavelet kernel, which contains several fixed-point or even floating point coefficients. Figure 2.5 shows the corresponding scenario belonging to Fig. 2.2. In order to exploit resource sharing, the different wavelet transforms have to be scheduled such that the second wavelet stage operates while the first one is idle, and the third stage executes while the two first ones are not busy and so on. In other words, all wavelet kernels depicted in Fig. 2.2 can use the same set of multipliers and adders thus reducing the required hardware resources.

After quantization (QT) of the subband pixels, the next step consists of entropy encoding. This step, however, works not on complete subbands, but on smaller blocks, the so-called *code-blocks*. Consequently a block-forming step has to be prepended, which is represented in Fig. 2.1 by the *block builder* (BB). It collects the arriving subband pixels and combines them into blocks whose size is typically 32×32 or 64×64. Depending on the implementation of the wavelet transform, the block builder is furthermore responsible for arbitration when several subband pixels arrive simultaneously.

Figure 2.6 illustrates the occurring production and consumption patterns for one single subband. The latter is generated by the wavelet transform in raster-scan order as indicated by Arabic numbers. The entropy encoder, however, requires the data block by block, thus leading to a read order as depicted by the arrows. This, however, implies that the data are not written and read in the same order (so-called *out-of-order communication*). Considering, for instance, pixels 3 and 18, the first is produced before the latter, but read much later. Hence, special communication primitives must support this operation.

After this block forming, each code-block is entropy encoded by the so-called *EBCOT Tier-1* algorithm. It consists of two parts, namely the *context modeler* and the *arithmetic encoder*. The context modeler traverses each code-block bit-plane by bit-plane and outputs for each bit a so-called *context*. The latter essentially describes the neighborhood values of the considered bit and permits thus to derive its most probable bit value. The context can be determined by a sliding window algorithm that shows very complex window movement and that requires further state information. The so-derived context together with the bit to encode is sent to the *MQ-coder*, which performs arithmetic coding. The context is used to derive the expected value of the processed bit. Depending on whether this expectation is met or not, more or less output bits have to be generated. Consequently, the MQ-coder is a heavy data-dependent operation.

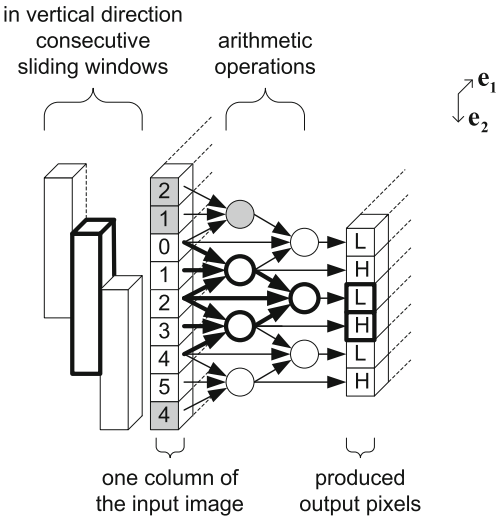


Fig. 2.4 Lifting scheme for a one-dimensional wavelet transform. *Squares* correspond to input and output pixels while *circles* represent arithmetic operations. Data dependencies are indicated by according edges. On the *left side*, the corresponding sliding windows are depicted. *Gray-hatched* pixels or operations are a consequence of the virtual border extension

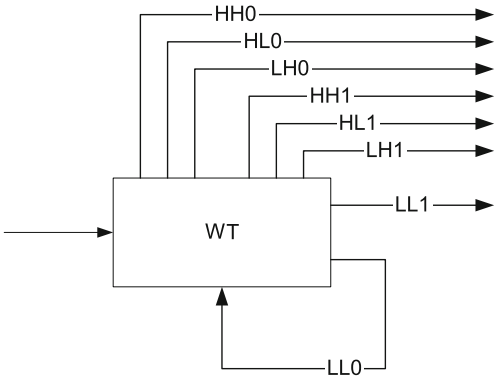


Fig. 2.5 Resource sharing between different wavelet decomposition levels

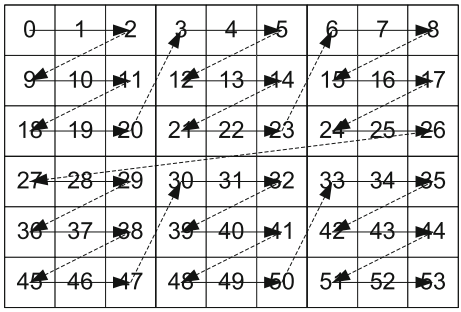


Fig. 2.6 Pixel production and consumption order for the JPEG2000 block builder (BB). Each rectangle represents a pixel generated by the wavelet transform. The corresponding write order is defined by numbers. The read order of the BB, on the other hand, is indicated by *dashed arrows*

The last step in JPEG2000 encoding is performed by the *EBCOT Tier-2*. The latter is responsible for rate allocation and codestream forming. Rate allocation is a global optimization problem that tries to achieve maximum image quality for a given compressed image size by selecting the most promising bits for inclusion into the final codestream. It includes complex calculations containing logarithmic functions and binary search. Once a corresponding solution has been found, the data have to be ordered in packets and corresponding headers have to be generated. This again requires complex algorithms like tag-tree [150] generation and Huffman-like compression. All of these algorithms are highly data dependent and partly work on complete images (global algorithms).

Based on the introduced JPEG2000 encoder the next section will shortly discuss several aspects of parallelism before Section 2.4 describes resulting implementation challenges.

2.3 Parallelism of Image Processing Applications

As real-time image processing system have to operate on huge amounts of data in little time, exploitation of parallelism is crucial in order to meet the corresponding timing requirements.

In this context, it is typically distinguished between (i) *task parallelism*, (ii) *data parallelism*, and (iii) *operation parallelism*.

Task parallelism means that all blocks of Fig. 2.1 can be executed in parallel, forming thus a *pipeline*. This can be established either by making each block working on a different image or a different image region. Whereas pipelining is straightforward for block diagrams without feedback loops, presence of the latter make this issue more complex.

Data parallelism means that one and the same operation is repeatedly applied to different data. Taking, for instance, the sliding window algorithm in Fig. 2.3, it can be seen that the same sliding window is applied at different image locations and hence to different input pixels. As long as a sliding window does not require values calculated by previous ones, they can be executed in parallel.

Operation parallelism finally corresponds to the fact that for the generation of a result value several independent calculations have to be performed. It is thus very similar to task parallelism, but addresses more fine-grained operations like individual additions or multiplications. A corresponding example is illustrated in Fig. 2.4. For the bold sliding window position, three sets of arithmetic operations depicted by bold circles have to be performed. Two of them can be performed in parallel because they are independent of each other. The third, however, requires their results as input data and has hence to be executed later on.

Unfortunately, design of parallel systems is more complex than sequential execution because the latter corresponds more to the reasoning of human beings. Consequently, system level design methods have to provide tools and methods that simplify creation of concurrent solutions.

The following section details these implementation steps in more detail.

2.4 System Implementation

The above-described JPEG2000 compression algorithm can be applied to both individual images as well as to huge frame sequences. Taking, for instance, the domain of cinema film production, digital acquisition requires real-time compression of huge images with up to 4096×2160 pixels at 30 frames per second.

In order to implement such a real-time JPEG2000 encoder, the abstract algorithm description given in Section 2.2 has to be successively refined to a hardware–software system. During this process, several problems have to be solved like

- selection of the best compression parameters,
- determination of the necessary calculation precision,
- calculation of the required buffer sizes,
- evaluation of implementation alternatives for each module,
- determination of the required hardware resources like general-purpose processors, *field-programmable gate arrays (FPGAs),* or dedicated chips,
- dimensioning of the used hardware resources by determination of the necessary processor speeds and FPGA sizes,
- RTL implementation of the individual algorithms,
- generation of reference data, followed by debugging and verification, and
- system optimization.

Unfortunately, each of the above steps is highly complex. Together with today's design practices, which operate on rather low levels of abstraction, system design risks to become

time-consuming, error-prone, and hence expensive. During manual implementation of such a JPEG2000 encoder, the following issues have found to be critical.

2.4.1 Design Gap Between Available Software Solution and Desired Hardware Implementation

This has been particular annoying during determination of the required calculation precision. Whereas software designs easily perform high-precision fixed-point or even floating point calculation, used bit widths have to be adjusted precisely for hardware solutions in order to achieve the required image quality with minimum number of resources. This, however, requires an encoder model that uses the same arithmetic operations as in hardware, but employs a higher level of abstraction than RTL in order to ease analysis and debugging. As such a model has not been available, a corresponding Matlab implementation had to be created and kept consistent with the desired hardware solution.

2.4.2 Lack of Architectural Verification

Selection of the best suited architecture is a very challenging task because often modifications in one module can influence the overall system. This is, for example, the case when deciding whether to employ resource sharing in the wavelet transform as described in Section 2.2. The latter not only reduces the required number of multipliers and simplifies the arbitration policy in the block builder but also increases the necessary storage memory and requires implementation of a complex scheduler. In particular, the latter is very challenging, because the different decomposition levels have to be calculated such that no buffer overflow occurs and that only few multiplexing operations are required in order to reduce hardware costs. Despite a careful design it has been missed during manual implementation that odd image sizes require special treatment that influences the overall throughput behavior. Consequently, the error has been detected very late in the RTL design phase, where it has been very difficult to understand its origins. Verification on a higher level of abstraction would have been much easier.

2.4.3 Missing Possibility to Explore Consequences of Implementation Alternatives

The selection of an implementation alternative for one module often influences the overall system performance. Besides the resource-sharing aspect for the wavelet transform, a similar scenario occurs for the EBCOT context modeler. Here, two implementation alternatives, namely pass-sequential and pass-parallel, exist [8]. Unfortunately, the pass-parallel variant not only is faster, but also requires more hardware resources. The resulting performance gain, however, does depend not only on the context modeler itself, but also on the following arithmetic encoder and possible data arbitration. This effect, however, cannot be predicted before hand and has to be measured after RTL implementation. Thus, selection of the best implementation alternative is very difficult.

2.4.4 Manual Design of Memory System

Implementation of the wavelet transform allows trading off achievable throughput against required hardware resources in form of on-chip and off-chip memory. Consequently, in order to select the best suited implementation alternative, each of them had to be analyzed concerning achievable throughput and required memory sizes. This included the elaboration of a corresponding memory mapping that permits sufficiently parallel data access required by the sliding window. Furthermore, the out-of-order communication occurring in the block builder led to a complex control flow in order to perform arbitration between the arriving subband coefficients and avoiding memory overflow. In both cases, significant time had to be spent in order to implement a correct solution.

2.4.5 Lack to Simulate the Overall System

Whereas a third-party software allowed execution of the JPEG2000 compression, it did not take into account the particularities of the hardware implementation like fixed-point calculation or simplified rate allocation. Consequently, this not only significantly complicated generation of reference data for verification, but made it impossible to predict occurring data quantities and achievable image quality at an early design state.

2.4.6 Inability to Precisely Predict Required Computational Effort for Both Hardware and Software

Together with the lack to simulate the overall system, determination of the required computation effort for both software and hardware implementation has been very critical. The reason is that the EBCOT performs data-dependent decisions whose execution time depends on the occurring data values. Since it is impossible to simulate huge amount of input images on RTL level, it has not been possible to predict the required number of EBCOT units and processor performance. Consequently, selection of the necessary FPGA sizes and processor speeds has been difficult.

2.5 Requirements for System Level Design of Image Processing Applications

After introduction of the JPEG2000 encoder and its refinement to a hardware–software system, this section aims to derive some requirements for system level design of such algorithms. As can be seen later on, these requirements can be transferred to other application examples, too. For that reason, this monograph is not only limited to the JPEG2000 algorithm, but will also consider a multi-resolution filter (see Section 7.1), a Motion-JPEG decoder (see Chapter 4), and morphological reconstruction (see Section 5.8.1). Nevertheless, at this point the JPEG2000 encoder shall be the major source of inspiration.

2.5.1 Representation of Global, Local, and Point Algorithms

This requirement is a direct conclusion of Section 2.1, which motivated that static image manipulations can be classified into global, local, and point algorithms. As they differ by the required calculation effort and required buffer size, this distinction must be visible in the system level design in order to enable efficient implementations. For local algorithms, sliding windows with different sizes, movement, and border processing have to be supported as discussed in Section 2.2.

2.5.2 Representation of Task, Data, and Operation Parallelism

This requirement is a direct conclusion of Section 2.3 in order to achieve sufficient system throughput. This is particularly true when processing huge image sizes, because for these applications, frequency scaling has reached its end.

2.5.3 Capability to Represent Control Flow in Multidimensional Algorithms

As shown in Section 2.2, image processing applications contain a huge amount of algorithms that operate on multidimensional arrays. In this context, sliding window algorithms represent a very important class. They are, however, not restricted to standard convolutions, but can also contain more complex calculation patterns like lifting schemes or significant control flow. Resource sharing during wavelet transform, for instance, requires coordination of different wavelet kernels with a complex scheduler.

2.5.4 Tight Interaction Between Static and Data-Dependent Algorithms

As shown in Section 2.2, image processing applications do not only operate on multidimensional data, but also include one-dimensional signal processing with or without data-dependent decisions. Corresponding examples are, for instance, arithmetic encoding or code-stream forming. In order to allow simulation of the overall system, which has been identified as important in Section 2.4, system level design methods must be able to handle both multi-dimensional and one-dimensional algorithms with and without control flow.

2.5.5 Support of Data Reordering

Besides sliding windows, data reordering is another important aspect in multidimensional applications. It occurs whenever pixels are not produced in the same order as being consumed. Corresponding examples are, for instance, frame conversion, image tiling, or code-block forming, which have all been described in Section 2.2.

2.5.6 Fast Generation of RTL Implementations for Quick Feedback During Architecture Design

As described in Section 2.4, implementation of many algorithms allows selecting between different design alternatives in order to trade throughput, latency, energy consumption, and hardware resources. However, the effects for the overall system are not easy to determine. Furthermore, manual creation of multiple design alternatives is prohibitive due to cost and timing constraints. Instead, automatic code transformations are required together with analysis capabilities in order to investigate their impact on the overall system.

2.5.7 High-Level Verification

This requirement is a direct consequence of Section 2.4. RTL simulation is clearly not a solution in order to verify architectural decisions like resource sharing or calculation precision. Instead, simulation and analysis on higher levels of abstraction are required for quick identification of errors.

2.5.8 High-Level Performance Evaluation

Although not being able to deliver exact results, high-level performance evaluation is required in order to make important design decisions. In case of a JPEG2000 hardware encoder, for instance, the EBCOT Tier-2 is often executed on a processor. However, selection of a suited model (number of cores, IO interfaces, . . .) is very difficult. This is because most processor architectures require special programming conventions in order to deliver optimal results. This kind of work is typically only done after selection of a processor architecture. Thus, the corresponding selection must base on high-level estimations.

2.5.9 Tool-Supported Design of Memory Systems

As memory bandwidth and size are a critical issue in almost every real-time system, corresponding tool support shall help the designer in achieving good and correct solutions in little time.

2.6 Multidimensional System Level Design

In accordance to the requirements identified in the previous section, this book aims to investigate in more detail corresponding multidimensional methodologies for system level design of image processing applications. Special focus is put on multidimensional point, local, and global algorithms without or with medium control flow and possible out-of-order communication. In particular, it shows how these kinds of algorithms can be modeled by multidimensional data flow in order to enable

- representation of task, data, and operation parallelism,

- high-level verification,
- system level analysis, and
- efficient high-level synthesis.

As this is a very huge and complex task, two well-defined sub-problems are considered, namely automatic scheduling and buffer analysis as well as efficient communication synthesis. These approaches are exemplarily integrated into a system level design tool called SYSTEMCODESIGNER, which enables tight interaction between multi- and one-dimensional algorithms, high-level performance evaluation, and automatic design space exploration.

More precisely, the following aspects are discussed by this book in more detail:

- It reviews different aspects of system level design and evaluates their applicability to image processing applications. In particular, it considers a huge amount of different modeling techniques and investigates their advantages and inconveniences when applied to image processing applications.
- It presents a data flow model of computation [164–166] that is particularly adapted to image processing applications by supporting sliding windows, virtual border extension, control flow, out-of-order communication, and high-level verification. Successful application to several examples shows the usefulness of multidimensional modeling.
- It discusses the combination of multidimensional system design with existing electronic system level design tools. To this end, the above-mentioned multidimensional model of computation is exemplarily integrated into the electronic system level design tool SYSTEMCODESIGNER [133, 170]. Usage of finite state machines permits to express both static and data-dependent algorithms. As a consequence, this leads to one of the first approaches that permit for tight interaction between one- and multidimensional application model parts.
- It describes and compares two different methods for automatic buffer size determination [163, 167, 171]. Whereas the first approach [167] bases on simulation and can thus be combined with arbitrary scheduling strategies or even control flow, the second [163, 171] is limited to static subsystems in order to profit from powerful polyhedral analysis. In particular, this leads to better solutions in shorter time. Both methods support out-of-order communication and the so-called *multirate* systems containing up- and downsamplers. Furthermore, two different memory mapping strategies are compared.
- It explains the necessary steps for automatic synthesis of high-speed communication primitives [168, 169] that support parallel reads and writes of several data elements within one clock cycle, in-order and out-of-order communication, and high clock frequencies. Furthermore, this book describes how different implementation alternatives can be generated automatically in order to trade required hardware resources against achievable throughput.

Chapter 3
Fundamentals and Related Work

After having presented the challenges and requirements for system level design of image processing applications, this chapter aims to discuss fundamentals on system level design and to give an overview on related work. Section 3.1 starts with the question how to specify the application behavior. In this context also some fundamental data flow models of computation are reviewed. Next, Section 3.2 gives an introduction to existing approaches in behavioral hardware synthesis. Communication and memory synthesis techniques are discussed separately in Section 3.4. Section 3.3 details some aspects about memory analysis and optimization. Section 3.5 reviews several system-level design approaches before Section 3.6 concludes this chapter with a conclusion.

3.1 Behavioral Specification

Specification of the application behavior is of utmost importance for efficient system design. It must be unambiguous, has to present all algorithm characteristics required for optimization and efficient implementation, and must not bother the designer with too many details while allowing him to guide the design flow in the desired manner [94]. Consequently, huge efforts in research have been undertaken in order to develop sound specification techniques. The corresponding findings are shortly reviewed in Section 3.1.1 before the following sections detail those approaches that are particularly relevant for image processing applications. As the latter belong to the category of data-dominant systems, special focus shall be paid to sequential languages (Section 3.1.2), one-dimensional (Section 3.1.3) and multidimensional data flow (Section 3.1.4).

3.1.1 Modeling Approaches

Sequential programs defined, e.g., in C, C++, Matlab, or Java belong to the most famous and widespread specification concepts nowadays. Introduction of procedures and functions as well as of the object-oriented programming paradigm has furnished powerful techniques to describe complex systems. Several loop constructs permit to specify iterative tasks, which repetitively apply the same operations to different input data. Consequently, they are responsible for huge parts of the overall execution time. Whereas the number of performed iterations

J. Keinert, J. Teich, *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*, Embedded Systems,
DOI 10.1007/978-1-4419-7182-1_3, © Springer Science+Business Media, LLC 2011

can depend on the input data in case of *while*-loops, *for*-loops[1] permit to predict the number of executions during compile time. As this renders static analysis and optimization possible, they are particularly important when mapping sequential programs to hardware accelerators.

Figure 3.1 illustrates the structure of such a nested *for*-loop. If all statements were contained within the inner loop body, the loop nest would be called *perfectly nested*. The *iteration variables* i_j are often grouped to an *iteration vector* $\mathbf{i} \in \mathbb{Z}^n$. Similarly, the loop boundaries can be expressed by vectors $\mathbf{i_B} = (i_{1B}, i_{2B}, i_{3B})$ and $\mathbf{i_E} = (i_{1E}, i_{2E}, i_{3E})$.

```
for(i1 = i1B; i ≤ i1E; i1++) {
  for(i2 = i2B; i ≤ i2E; i2++) {
    <statements>
    for(i3 = i3B; i ≤ i3E; i3++) {
      <statements>
    }
  }
}
```

Fig. 3.1 Structure of static (imperfectly nested) *for*-loop

Since, however, all statements are executed sequentially, contained application parallelism cannot be expressed explicitly and has to be extracted by complex analysis. Consequently, various other specification methods have been proposed [154]. Their semantics are described by a so-called *model of computation* (*MoC*), which defines communication, synchronization, and time representation [154, 192]. *Differential equations* of different orders permit to specify and simulate continuous time applications. In *discrete event systems,* different stateful entities (processes) communicate by exchange of signals. Change of a signal results in an event, which possesses a time stamp (time of occurrence) and a value. Occurrence of such an event can cause a state change of the affected entities as well as execution of some functionality. This concept is, for example, employed in *VHDL* [16, 146, 147] and *Verilog* [277]. *SystemVerilog* [34, 268] extends the latter by various concepts for enhanced verification. Unfortunately, their simulation is slow because they use a low level of abstraction and because all occurring events have to be sorted accordingly to their occurrence.

Synchronous/reactive models employ a similar concept, but the output responses are produced simultaneously with the input stimuli. The advantage of this assumption (called the synchrony hypothesis) is that such systems are easier to describe and analyze. Synchronous/reactive models are particularly adapted to describe systems that interact continuously with their environment. Corresponding languages are available in form of *Esterel* [35] or *Lustre* [127]. *Finite state machines* allow the specification of control-dominant algorithms. *Statecharts* [131] extend this concept by introduction of hierarchical states. This permits compact representation of complex control flow and concurrency. Broadcast communication enables a sender to emit messages without being blocked. Actions are performed during state transitions in zero time.

The definition of *timeless models* is independent of any notion of time. (Timeless) *Petri-Nets* [241] have been developed in order to model concurrent and asynchronous systems. Fast algebraic analysis methods are able to derive necessary conditions for reachability of a given system state (so-called *marking*) or existence of periodic schedules as well as sufficient conditions for boundedness of possible system states. Coverability trees define the set of

[1] Without *break* or *continue* statements.

system states that can be reached from a given initial state. Reachability graphs, on the other hand, can be used in order to verify which states can be attained from an arbitrary state, given a concrete initial marking. *Communicating sequential processes (CSP)* [136, 137] have been introduced in order to describe systems where concurrent entities communicate in a synchronized manner. In other words, a sender that wants to send some data stalls until the receiver is able to accept them. *Data flow* models of computation focus on data-dominant and computationally intensive systems that perform complex transformation and transport considerable amount of data. The application is represented by a directed graph whose nodes describe computations while edges correspond to communication. Depending on the requirement of the modeled application, different data flow semantics are available (see Sections 3.1.3 and 3.1.4). Consequently, Hsu et al. [138, 139] introduce the *data flow interchange format* (*DIF*), a common representation for different data flow models of computations. It serves as a standard language for specifying mixed-grained data flow models in order to simplify the data exchange between different tools. Furthermore, it provides a tool set for analysis and manipulation of data flow graphs.

The *unified modeling language (UML)* [108, 226] finally combines a set of models like sequence diagrams, a variant of Statecharts, class diagrams, or use cases. Whereas originally intended to standardize software development, recent research also considers system level modeling and hardware design [15, 31, 270]. However, semantic and consistency check is still an open issue when several models have to interact.

After this overview, the following sections will detail the specification approaches for data-dominated systems as they contain the image processing algorithms that are in the focus of this monograph.

3.1.2 Sequential Languages

Whereas sequential languages like C or Matlab are ideally adapted to mono-processor implementations, generation of complete systems requires the capability to represent task level parallelism. Furthermore, synthesis of hardware accelerators must be supported. In [94], several challenges are listed that have to be overcome in order to achieve these goals:

- Sequential languages do only define an execution order but do not contain any information about time. However, for hardware synthesis, the performed operations have to be executed at a certain clock cycle, and possibly in parallel. Hence, fine-grained scheduling is required.
- Sequential languages are by nature not able to represent parallelism.
- Whereas software typically operates on integer and float variables or even complex data structures, hardware requires flat structures on bit level.
- Manual hardware implementation employs various communication primitives like simple signals, registers, FIFOs, shift registers, or shared memories. Sequential languages, however, exclusively use shared memory communication.

Consequently, in order to simplify these problems, several C-variants have been proposed that are more adapted to hardware implementations [94]. *SA-C* [91, 290], for instance, introduces a special syntax for sliding windows and makes restrictions on allowed conditional statements and while-loops. *HardwareC* [183] introduces hardware-like structures and hierarchy. *BDL*, a C-variant processed by NEC's Cyber system [297], deviates greatly from ANSI-C by including processes with I/O ports, hardware-specific types and operations, explicit clock

cycles, and many synthesis-related compiler directives. Other approaches like *Handel-C* [61] and *SHIM* [93, 289] enrich the C language with the capability to express communicating sequential processes. Their principles are shortly summarized in the following section.

3.1.2.1 Communicating Sequential Processes

One of the most challenging aspects when designing parallel systems is synchronization of the different application threads. In the domain of software development, *mutexes* (mutual exclusion objects) and *semaphores* are well-known concepts in order to solve this task. However, their application is very complex and error prone due to occurring racing conditions and deadlocks. Design of hardware–software systems is even worse as special care has to be taken that no information in form of signals, interrupts, or value changes gets lost.

Communicating sequential processes (CSP) (see Section 3.1.1) can help in designing such systems because they use a synchronized communication scheme. To this end, *Handel-C* [61] enhances the C language with constructs permitting the description of such communicating processes. Hardware synthesis is simplified by supporting data types on bit level, shared or parallel function instantiation, creation of register banks, and RAM/ROM instances as well as semaphores for protected variable access. A hardware abstraction layer, called platform abstraction layer, aims to increase the portability of designs by easy access of peripheral components such as memories or video I/O.

SHIM [93, 95, 289] also employs the concept of synchronized communication, but excludes any *non-determinism*. Non-determinism means that a system is allowed behaving different for one and the same input stimulus. Although this behavior is required for some applications like networking or load balancing, focusing on deterministic systems is reasonable for many scenarios and greatly simplifies debugging and formal verification. In *SHIM*, deterministic application behavior is ensured by restricting to synchronized point-to-point communication.[2] Exchange of information can only take place, when both the source and the destination are ready. Apart from this communication, the different processes run completely asynchronous, i.e., their execution speed is not defined. Based on these communication schemes, typical scenarios like bounded FIFO communication or deterministic arbitration can be precisely defined while others like preemption or interrupts need to be approximated [95].

From this model, software synthesis for both uni- [96] and multiprocessor [97] implementations is possible. Static scheduling permits to improve attainable throughput. Reference [288] shows how to prove the absence of deadlocks.[3] The publication [95] finally considers hardware synthesis. In all cases inter-process communication plays an important role. Generation of software schedules, for instance, requires that the code is partitioned into functions communicating only at their exit. And hardware synthesis requires generation of combinational handshaking mechanisms for each occurring data exchange.

Specification of this communication, however, is one of the weak points of CSP as soon as complex data dependencies occur. This is because two processes can only exchange a value when both are ready to provide or accept it, respectively. Consequently, in case of out-of-order communication as described in Section 2.2, an additional translation unit has to reorder the data due to different write and read orders of the data source and sink. This operation,

[2] A slight relaxation is possible by copying written data to several destinations.

[3] Presence of deadlocks cannot be proved.

however, is hidden in some piece of C code. As a consequence, analysis is difficult, because huge images can quickly result in large state spaces that challenge any model checker. Consequently, automatic buffer size determination risks to become computationally expensive. Furthermore, as SHIM is limited to deterministic models, the user would be forced to decide exactly in which order to accept and to output the data in the translation unit. However, as shown later on in Chapters 6 and 8, out-of-order communication supports different schedules, which are best covered in a self-timed fashion.

Furthermore, CSP have difficulties to represent parallel data access. Taking, for instance, the sliding window introduced earlier in Fig. 2.3, each new window position requires 2 new pixels while the source generates only 1 pixel per invocation. In CSP, however, source and destination can only read and write the same data quantities. Consequently, another translation unit would be necessary. Additionally, the model does not support parallel communication on several channels, and FIFOs have to be described in a quite complex manner in form of communicating registers.

The underlying reasons for these difficulties can be found in the fact that CSP is not able to argue on data dependencies, but only on communication synchronization. Here data flow models enable more advanced representation and analysis techniques, which can be advantageously employed for design of image processing applications as shown in this monograph. Nevertheless, it might be interesting to investigate in how far CSP and data flow models can be combined.

3.1.2.2 SystemC

Besides the extension of the C language by communicating sequential processes, *SystemC* [230] is another approach for the design of complex systems. It bases on the C++ language by providing a set of classes for application modeling. Introduction of processes permits representation of coarse-grained parallelism while channels provide various communication mechanisms. Different levels of abstraction ranging from algorithmic models over *transaction-level models (TLM)* to *register transfer level (RTL)* help for stepwise system refinement. However, in contrast to *SHIM*, *SystemC* does not impose any model of computation. Whereas this results in highest expressiveness, it is also a curse for verification, automatic analysis, or parallelization.

3.1.3 One-Dimensional Data Flow

Whereas the previous approaches focused on the question, how to perform calculation, *data flow* models of computation emphasize the transport of data. For this purpose, the application is modeled as a graph $G = (A, E)$ consisting of vertices $a \in A$ interconnected by edges $e \in E \subseteq A \times A$. These edges represent communication, which takes place by transport of data items, also called *tokens*, from the source vertex src (e) to the sink snk (e) of the edge e. The vertices correspond to processes, also called *actors*. Different models of computation define rules for production and consumption of data while the functionality of the actors has to be defined somehow else. This can, for instance, be done by C, Java, or VHDL code or by usage of an implementation-independent language like CAL [98, 99].

The following sections revise important one-dimensional data flow models of computation and investigate their ability for system level design of image processing applications.

3.1.3.1 Synchronous Data Flow (SDF)

In *synchronous data flow (SDF)* [193], data consumption and production occur in constant rates defined by two functions

$$p : E \to \mathbb{N}, \tag{3.1}$$
$$c : E \to \mathbb{N}. \tag{3.2}$$

In other words, each actor *invocation*, also called *firing*, consumes $c(e_i)$ tokens from each input edge e_i and produces $p(e_o)$ tokens on each output edge e_o. This also means that an actor can only execute when enough input data are available. The special case of $p(e) = c(e) = 1 \, \forall e \in E$ is also denoted as *homogeneous synchronous data flow (HSDF)* graph. Although SDF models are a priori timeless, it is often assumed that both production and consumption are performed at the end of an invocation. Before graph execution starts, each edge can already store the so-called *initial tokens* whose number is given by

$$d : E \to \mathbb{N}_0.$$

The buffer capacity is assumed to be infinite. Furthermore, edges are not allowed changing the order of the arriving tokens. Hence, they can be thought as infinite FIFOs. The fill level of all FIFOs is defined as the *state* of the SDF graph.

Based on the production and consumption rates, a *balance equation* can be established [193]:

$$\Gamma_G \times \mathbf{r_G} = \mathbf{0}. \tag{3.3}$$

Its solution delivers the number of invocations that are required for each actor to return the graph into its initial state. Γ_G is the so-called *topology matrix* and can be calculated as

$$\Gamma_G = \begin{bmatrix} \Gamma_{1,1} & \cdots & \Gamma_{1,a} & \cdots & \Gamma_{1,|A|} \\ \vdots & & \vdots & & \vdots \\ \Gamma_{e,1} & \cdots & \Gamma_{e,a} & \cdots & \Gamma_{e,|A|} \\ \vdots & & \vdots & & \vdots \\ \Gamma_{|E|,1} & \cdots & \Gamma_{|E|,a} & \cdots & \Gamma_{|E|,|A|} \end{bmatrix},$$

with

$$\Gamma_{e,a} = \begin{cases} p(e) & \text{if } \mathrm{src}(e) = a \text{ and } \mathrm{snk}(e) \neq a \\ -c(e) & \text{if } \mathrm{snk}(e) = a \text{ and } \mathrm{src}(e) \neq a \\ p(e) - c(e) = 0 & \text{if } \mathrm{snk}(e) = \mathrm{src}(e) = a \\ 0 & \text{otherwise} \end{cases}.$$

It can be shown that for a connected SDF graph G, Eq. (3.3) has a solution if and only if $\mathrm{rank}(\Gamma_G) = |A| - 1$. In this case, G is called *consistent*, otherwise *inconsistent*, as its execution would cause unbounded token accumulation.

The minimal, strictly positive integer vector $\mathbf{r_G^*}$ that solves Eq. (3.3) is called *basic repetition vector*. It can be used for construction of the so-called *minimal periodic schedules*,

where each actor $a \in A$ is executed $\mathbf{r}_{\mathbf{G}}^{*}(a)$ times. Since such a schedule does not cause a net graph change of the buffered tokens on each edge, the overall application can be executed infinitely by periodic repetition of the minimal periodic schedule.

Unfortunately, even a consistent SDF graph might *deadlock*. In other words, none of the actors have enough tokens to fire. In order to detect such scenarios, it is necessary to construct an arbitrary schedule [160, 193] that invokes each actor $a \in A$ $\mathbf{r}_{\mathbf{G}}^{*}(a)$ times. If such a schedule does not exist, the SDF graph deadlocks.

Various other powerful analysis and synthesis methods have been developed in order to calculate achievable throughput [112], required buffer sizes [9–11, 111, 117, 267], or efficient schedules for software synthesis [42, 71, 211–214, 228, 272, 273, 301]. Furthermore, methods for hardware synthesis have been proposed [156, 309]. Unfortunately, SDF is not a good choice for system level modeling of image processing applications. Not only are they unable to express control flow and data-dependent decisions, image processing algorithms often do not follow the premise of constant production and consumption rates.

Figure 3.2 shows a corresponding simple example. The source in Fig. 3.2a is supposed to generate the input image pixel by pixel in raster-scan order from left to right and from top to bottom. The resulting image of size 5×6 is sampled by the filter actor, again in raster-scan order. Consequently, in general each new window position like that labeled by (3) requires a new input pixel. However, at the image borders, this rule is violated. Window position (1), for instance, requires $5+2=7$ input pixels from the source before calculation can start, because the pixels are generated in raster-scan order. On the other hand, the output pixel at location (2) can be calculated without needing any new input values, because all necessary pixels are already known from previous invocations.

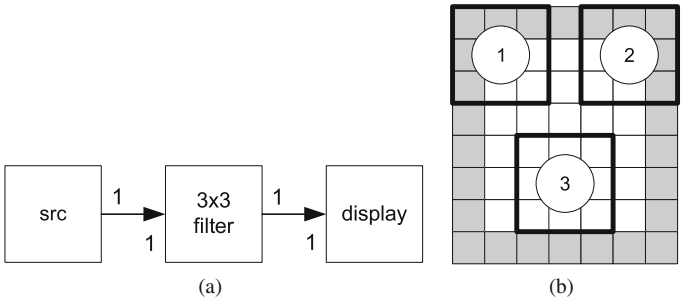


Fig. 3.2 Example sliding window algorithm. **a** SDF data flow graph of sliding window. **b** Filtering operation in the filter actor. *Gray shading* corresponds to the extended border (see Section 2.2)

This, however, means that the consumption rates are not constant. Consequently, this simple application can only be represented as SDF graph on the granularity of complete images. The corresponding consumption and production patterns are already depicted in Fig. 3.2a. For each produced input image, the filter generates a corresponding output image that is transmitted to the display device. However, such a model has severe impacts on the capacities for analysis and synthesis. First of all, since the tokens correspond to complete images, a huge memory is required on the individual edges. This is in contrast to typical hardware implementations, which require only 2 lines and 3 pixels as memory buffers for this kind of implementation [199]. Second, Fig. 3.2a is not able to express either available data parallelism (see Section 2.3) or requirements for parallel data access. In other words, it is not possible to see that several sliding windows can be calculated in parallel if required, and that calculation

of one window needs access to 9 input pixels in parallel in order to achieve high system throughput.

In order to alleviate these difficulties, CV-SDF [263] extends the SDF model by splitting images into slices representing lines or blocks for more buffer-efficient implementations. In order to support sliding window algorithms, it supports multiple reads of slices and frames. However, as the size of neither the frames nor the slices is specified, this reduces the possibilities for analysis. Furthermore, CV-SDF is limited to a one-dimensional representation of sliding window algorithms, and, in general, slices are coarser than pixels.

A similar construct can be found in *StreamIT* [115, 159, 206, 303], a programming language and a compilation infrastructure specifically engineered for modern streaming systems. The underlying model of computation extends SDF by non-destructive reads. In other words, an actor has the possibility of reading a value from an input FIFO without discarding it. Consequently, a token can be read multiple times. Furthermore, an actor is able to perform an initialization phase at the beginning of the graph execution. In order to ease scheduling, StreamIT graphs obey a restricted topology, in which each actor can only have one input and one output. Since realistic applications require more complex interconnections, special actors for feedback and split-join structures are introduced. Different policies like duplication or round-robin load balancing are supported.[4] However, although non-destructive reads simplify modeling of one-dimensional sliding windows, application to image processing applications stays complex due to processing of multidimensional arrays and occurring border processing.

Reference [210] uses a modeling approach containing two different views. The first one consists of an SDF graph while the second one specifies the precise dependencies between input and output pixels. Whereas this permits to detect contained data parallelism (see Section 2.3), it does not support the specification of out-of-order communication. Furthermore, Ref. [210] insists still on a communication granularity of complete images.

Synchronous piggybacked data flow (SPDF) [234–236] is another extension of SDF. It addresses the question how to propagate parameters through data flow graphs without side effects, redundant data copying, reducing parallelism, increasing buffer sizes, or complicating module reuse. However, it does not address modeling of sliding windows or out-of-order communication.

3.1.3.2 Cyclo-static Data Flow (CSDF)

Cyclo-static data flow (CSDF) [45, 100, 240] extends the previously described SDF by allowing periodically changing token consumption and production patterns. For this purpose, an actor is executed in a sequence of *phases* each producing and consuming a constant amount of tokens. The number of phases $L(a)$ can vary for each graph actor $a \in A$ and is given by a function

$$L : A \to \mathbb{N},\ a \mapsto L(a)\,.$$

As the quantity of tokens produced by $\mathrm{src}(e)$ depends on the current phase, Eq. (3.1) is replaced by

[4] The split actor belongs to the class of CSDF, described in Section 3.1.3.2.

$$p : E \times \{s \in \mathbb{N} : 0 \leq s < L\,(\mathrm{src}\,(e))\} \to \mathbb{N},\ (e, s) \mapsto p\,(e, s)\,.$$

Similarly, the number of tokens consumed by snk (e) in phase s is defined by $c\,(e, s)$

$$c : E \times \{s \in \mathbb{N} : 0 \leq s < L\,(\mathrm{snk}\,(e))\} \to \mathbb{N},\ (e, s) \mapsto c\,(e, s)\,.$$

$p\,(e, (k-1) \bmod L\,(\mathrm{src}\,(e)))$ corresponds to the number of tokens produced by the kth execution of src (e). The token consumption of the actor snk (e) can be calculated by $c\,(e, (k-1) \bmod L\,(\mathrm{snk}\,(e)))$.

The *state* of a CSDF graph corresponds to the fill level of the edge buffers and to the phase index of each graph actor. Based on this definition, a *balance equation* can be defined [45], whose solution gives the number of invocations for each actor such that no net change of the graph state occurs.

Due to these extensions, CSDF seems to be better adapted for modeling of image processing algorithms. Figure 3.3 shows a corresponding example illustrating the different scenarios occurring at the image borders.[5] All actors are assumed to operate in raster-scan order. Each execution of the filter generates 1 output pixel. Since the number of additional required input pixels per phase is not constant, the overall number of phases corresponds to the number of output pixels amounting 12 in this example. Since the source generates the pixels in raster-scan order, the first filter can only be applied after $4 + 2 = 6$ pixels are available. On the other hand, the four last filter positions do not require any additional data elements, because the value of the extended border pixels is either constant or can be derived from previously read pixels.

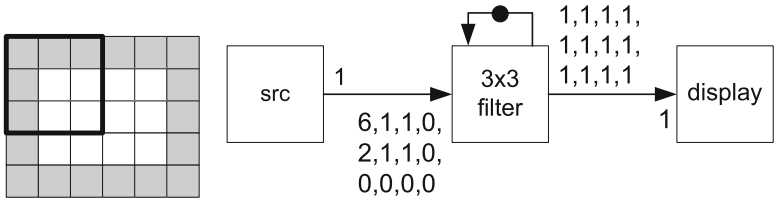


Fig. 3.3 CSDF model applied to a sliding window application. **a** Sliding window algorithm. **b** CSDF graph

Unfortunately, this modeling approach also shows several inadmissibilities that make system level design difficult:

- As the sliding window actor requires one phase per output pixel this can lead to very complex models when processing huge images.
- The filter actor in Fig. 3.1.3.2 requires hidden memory in order to buffer intermediate lines (the illustrated sliding window algorithm requires at least 2 lines and 3 pixels of storage).

[5] In [166], an alternative representation has been chosen in order to emphasize the behavior of a possible hardware implementation. As the latter is only able to read 1 pixel per clock cycle, the initial phase is split into several invocations. Furthermore, it has been assumed that the filter already reads the next input when processing the window at the right border.

- The phases of the filter actor do depend not only on the sliding window algorithm itself, but also on the production order of the source actor. In other words, if we replace the source by an alternative implementation that generates the image column by column instead of row by row, it is also necessary to rewrite the filter actor. This, however, reduces the possibility of module reuse.
- From Fig. 3.3b it is not obvious whether parallel data access is required.
- The CSDF model does not give detailed hints about the occurring data dependencies. In other words, it is not clear on which input pixel an output pixel depends. This, however, makes it impossible to use *polytope*-based analysis techniques (see Section 3.3.2 and Chapter 7), which have shown to be powerful tools for the design of image processing applications.
- As the 3×3 filter internally buffers data elements, it has to be equipped with a tight feedback loop in order to indicate that consecutive invocations are not independent. This, however, destroys all possibilities to exploit data parallelism.
- As already indicated by footnote 5 on page 31, the CSDF model is still quite implementation dependent. Taking, for instance, the last four phases in Fig. 3.3b, it can be seen that no input data are read. This fact can be exploited by hardware implementations in order to already read the next input image with the aim to reduce the execution time of the initial phase. However, this would need a completely different CSDF model.

An alternative interpretation of the CSDF model proposed in [85] only partly solves these problems. The basic idea consists in modeling each communication using two edges. The forward edge transports the proper tokens while the sink indicates via the backward edge how many tokens have been released from the buffer. Consequently, the sink phases only need to consume the additionally required data values. Based on this interpretation they are able to describe non-destructive reads, sliding windows, buffer sharing between different sinks, and partial production where tokens are generated during several actor invocations. On the other hand, these achievements have to be paid by increasing model complexity. Furthermore, the extended CSDF graph does not contain information about parallel data access, does not allow polytope-based analysis, and complicates module reuse. In addition, since CSDF is a one-dimensional model of computation, the address calculation in order to find the correct data elements in the buffer is left to the user.

3.1.3.3 Fractional Rate Data Flow (FRDF)

Fractional rate data flow (FRDF) [227] is another approach that aims to enhance the granularity of SDF graphs. To this end, FRDF allows production and consumption of *fractional tokens*. In other words, a composite data structure is only read or written partly.

Instead of Eqs. (3.1) and (3.2), token production and consumption are now defined as follows:

$$p : E \to \mathbb{N}^2 \; e \mapsto p(e) = \begin{pmatrix} p_{\text{num}}(e) \\ p_{\text{denom}}(e) \end{pmatrix} =: \frac{p_{\text{num}}(e)}{p_{\text{denom}}(e)},$$

$$c : E \to \mathbb{N}^2, \; e \mapsto c(e) = \begin{pmatrix} c_{\text{num}}(e) \\ c_{\text{denom}}(e) \end{pmatrix} =: \frac{c_{\text{num}}(e)}{c_{\text{denom}}(e)}.$$

$p_{\text{denom}}(e)$ denotes the number of (complete composite) tokens produced. However, this production can be performed in a fractional manner. $p_{\text{num}}(e)$ defines the fractional part of the composite token that is produced for each actor invocation. Note that the right-hand side

presents a short-hand notation in form of a fraction, which must not be canceled. Similarly $c_{\text{denom}}(e)$ defines the number of consumed composite tokens and $c_{\text{num}}(e)$ the accessed fractional part. However, whereas FRDF leads to efficient implementations, it does not consider out-of-order communication nor sliding windows.

3.1.3.4 Parameterized Data Flow

Parameterized data flow [39–41] is a framework that permits to define parametric models based on well-known static data flow models of computation such as SDF or CSDF. In case of SDF, the resulting model is called *parameterized synchronous data flow (PSDF)*. A parametrized data flow graph consists of hierarchical actors, also called *subsystems*, which are interconnected by edges. Each subsystem can be parametrized by *non-data flow* and *data flow parameters* whereas only the latter are allowed changing the consumption and production rates. The parameter values can be configured either by the owning subsystem itself or by a parent actor. For this purpose a subsystem consists of three different subgraphs. The *init graph* configures the data flow parameters of the subsystem that it is part of. The calculation of the corresponding values can only rely on parameters of the *init graph* because the latter does neither produce nor accept any data flow tokens. The *subinit graph* can only configure parameters that do not influence the external consumption and production rates of the subsystem that it is part of. However, in contrast to the init graph, the subinit graph can accept data flow input. The *body graph* finally performs the proper data processing of the application.

The execution order of these three subgraphs is strictly defined. For each data flow invocation of subsystem H, the data flow parameters of the latter have to be fixed. Furthermore, before execution of the subsystem H itself, the init graphs of all its child subsystems are fired. This means that the production and consumption rates of all child subsystems are fixed. Next the subinit graph of the subsystem H is executed, followed by its body graph, whereas in both cases a complete schedule period is run. As consumption and production rates might be variable, scheduling requires a mixture of quasi-static compile time analysis and dynamic run-time scheduling.

Based on the parameterized data flow framework, several variants have been proposed. In *blocked data flow (BLDF)* [179–181], huge data samples are propagated as parameters instead of classical FIFO transport. This helps avoiding costly copy operations because several actors can access the same parameters. Furthermore, array tokens can be split into sub-arrays, the so-called *copy sets,* which can be processed in parallel. Fine-grained data dependencies or fine-grained scheduling of sliding window algorithms for hardware implementation is not further considered.

The *dynamic graph topology specification* [179] is another extension of PSDF allowing graph actors and edges to be treated as dynamic parameters. By this means, it is possible to specify dynamic graph topologies instead of coding complex state machines. In order to avoid dynamic scheduling, all possible graph topologies are predicted at compile time.

3.1.3.5 Homogeneous Parameterized Data Flow (HPDF)

Homogeneous parameterized data flow (HPDF) [255–257] is another framework for generation of parametrized data flow specifications without the need to establish complex hierarchies. Instead, it is assumed that the source and the sink execute at the same rate. In other words, whereas a source actor is allowed to produce a variable number of data tokens, the sink

actor is constrained to read exactly the same number of tokens. In the final implementation, such a behavior can be obtained when actors with variable output rates write a header such that the sink knows how many tokens to read.

Whereas this seems quite restrictive, many image processing applications can be represented by this manner more intuitively than with the approach presented in the previous section. Nevertheless, it does not consider aspects like out-of-order communication, fine-grained data dependencies of sliding window algorithms, or parallel data access.

3.1.3.6 Data-Dependent Data Flow

Whereas the previous approaches either confined themselves to static applications or imposed restrictions on the permitted parameters, a huge amount of data flow models of computation address the specification of highly data-dependent algorithms.

Boolean-controlled data flow (BDF) [50], for instance, extends the SDF model of computation with two special actors called *switch* and *select*. The first one reads one input token and forward it to one of two possible outputs. The output to select is specified by a second Boolean input port. In the same way the *select* actor reads a token from one of two possible inputs and forwards it to a single output port. Already these simple extensions together with unbounded FIFOs are sufficient to generate a *Turing complete* model of computation. Simply spoken, this means that such a model can, in principle, perform any calculation that any other programmable computer is capable of.

On the other hand, this strongly diminishes the possibilities for analysis. For example, whereas BDF graphs permit to establish a balance equation, which depends on the ratio between *true* and *false* tokens on the Boolean inputs of each *switch* and *select* actor, the existence of a solution does not guarantee that the application can run with bounded memory as this would be the case for SDF graphs [50]. Furthermore, there exist graphs for which it is not possible to construct minimal periodic schedules (see Section 3.1.3.1), because they would depend on the sequence of occurring Boolean token values. Nevertheless, even in this case a schedule with bounded memory might exist. Unfortunately its determination might become complex or even impossible.

Integer-controlled data flow (IDF) [51] is an extension of BDF in that the *switch* and *select* actors can have more outputs or inputs, respectively. Whereas this helps to simplify the application models, it does not offer further expressiveness as BDF is already Turing complete.

Cyclo-dynamic data flow (CDDF) [298, 299] aims to enhance the analysis capabilities of dynamic data flow descriptions. Its major idea is to provide more context information about the problem to analysis tools than this is done by BDF. For this reason, an actor executes a sequence of phases whose length depends on a control token. The latter can be transported on any edge, but must be part of a finite set. Each phase can have its own consumption and production behavior, which can even depend on control tokens. Several restrictions take care that the scheduler can derive more context information as this would be the case for BDF graphs. For instance, a CDDF actor is not allowed to have a hidden internal state.

Kahn process networks (KPNs) [157] finally are a superset of all above data flow models as they only impose blocking reads (and writes) combined with communication over unbounded FIFOs. In other words, an actor is not allowed to poll the fill level of an input FIFO. Instead, it can only formulate a read request. If the FIFO does not have enough input data, the actor blocks until the demand can be fulfilled. KPNs play a fundamental row in system level design

as it can be proven that they execute deterministically. In other words, a KPN network delivers identical results for identical input stimuli independent of the order in which the different actors or processes are fired. As in parallel systems the exact invocation order can often not be predicted exactly, this property is in many cases fundamental for construction of correct systems. In contrast, *dynamic data flow (DDF)* can even show non-deterministic behavior.

3.1.3.7 FunState

FunState [264, 265] differs from the above approaches in that it is a common representation for different models of computation. This is possible by dividing FunState models into a data-oriented and a control-oriented part. The data-oriented part processes the tokens by means of functional units and FIFO or register communication. The execution of the functions is coordinated by finite state machines in the control-oriented part. For this purpose, each transition is labeled by a condition and an associated function. The condition formulates requirements on the number of available input tokens or token values. Depending on the nature of the finite state machine, different models of computation like SDF, CSDF, BDF, DDF as well as communicating finite state machines can be represented. Whereas this extends the class of applications which can be represented, it does not solve the deficiencies in modeling multidimensional applications.

3.1.3.8 Lessons Learned

Section 3.1.3 has reviewed a huge amount of one-dimensional data flow models of computation and has evaluated their applicability to system level design of hardware–software systems for image processing. To sum up, none of them has been able to cover all requirements formulated in Section 2.5. Major difficulties are in particular intuitive representation of sliding window algorithm including fine-grained data parallelism and border processing behavior as well as out-of-order communication. Cyclo-static data flow seems still to be the most promising solution. Nevertheless, major drawbacks like complex phase relations, difficult reusability, reduced generality, and missing information about parallel data access and about data dependencies followed by the impossibility of polytope-based analysis limit its usefulness.

Concerning the representation of control flow, BDF representatively introduced the balancing act between expressiveness and capabilities for analysis and optimization. The conclusion that follows consists in the need of methods that are able to cover important control flow scenarios such that they can still be analyzed. Unfortunately, BDF and IDF abstract from most context information, while CDDF is not very widespread. Only FunState offers a promising approach by its capacities to represent various models of computation. The framework of parameterized data flow finally shows a possibility of handling parametric applications.

3.1.4 Multidimensional Data Flow

As illustrated in Section 3.1.3, one-dimensional data flow models of computation are not able to represent image processing applications in an encompassing manner. The underlying problem can be found in the elimination of important information when transforming an algorithm working on a multidimensional array into a one-dimensional representation. Consequently, this section reviews existing work dealing with multidimensional data flow.

Unfortunately, such approaches are not very widespread. Instead, current research activities seem to be divided into two categories. On the one hand, powerful analysis and synthesis methods on *polytope*-based representations have been developed (see also Section 3.4). This kind of representation is typically obtained from loop descriptions. On the other hand, one-dimensional data flow-based system design is extensively investigated, which led to many useful results. However, the interaction between these two domains is still less considered and existing work about multidimensional modeling is only rarely cited. Consequently, the following sections aim to give an overview on existing approaches for multidimensional data flow modeling.

3.1.4.1 Multidimensional Synchronous Data Flow (MDSDF)

Multidimensional synchronous data flow (MDSDF) [64, 65] has been the first multidimensional data flow model of computation. Similar to SDF, an application is modeled as a graph $G = (A, E)$ consisting of vertices $a \in A$ interconnected by edges $e \in E \subseteq A \times A$. However, in contrast to SDF, each actor invocation consumes or produces an n-dimensional array consisting of several *data elements*:

$$p : E \to \mathbb{N}^n,\ e \mapsto \mathbf{p}(e),$$
$$c : E \to \mathbb{N}^n,\ e \mapsto \mathbf{c}(e).$$

The dot product $\langle \mathbf{p}(e), \mathbf{e_i} \rangle$ corresponds to the number of tokens produced in dimension i, $1 \leq i \leq n$. $\mathbf{e_i}$ is a Cartesian base vector. Similarly, $\langle \mathbf{c}(e), \mathbf{e_i} \rangle$ identifies the size of the read token in dimension i.

Figure 3.4 illustrates a simple example in form of a two-actor graph. The sink can only fire if all required input data elements are available. The order in which the tokens are produced or consumed is not specified in the model. Hence, the sink might execute in *row-major order* (from left to right and from top to bottom) while the source operates in *column-major order* (from top to bottom and from left to right). In order to ensure that all data elements produced by the source are also read by the sink, the number of invocations of each actor in each dimension i must be a multiple of the basic repetition vector $\mathbf{r_i^*}$ (see also Section 3.1.3.1). The latter can be calculated independently for each dimension i by means of a balance equation in the same manner as for SDF.

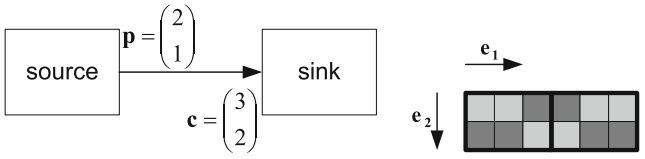


Fig. 3.4 MDSDF graph. **a** Graph. **b** Token production and consumption. Source tokens are depicted by two different *gray shadings* while *bold frames* correspond to sink tokens

Example 3.1 For the example illustrated in Fig. 3.4, the solution of the balance equations (one for each dimension) leads to

$$\mathbf{r_1^*} = (3, 2)^T,$$

$$\mathbf{r_2^*} = (2, 1)^T ,$$

meaning that the first actor (the source) has to execute three times in direction $\mathbf{e_1}$ and two times in $\mathbf{e_2}$. For the second actor, two invocations in direction $\mathbf{e_1}$ and one execution in $\mathbf{e_2}$ are sufficient.

Analogous to SDF, each edge can have initial data elements in form of multidimensional delay

$$d : E \quad \rightarrow \mathbb{N}_0^n,\ e \mapsto \mathbf{d}\,(e)\,.$$

Figure 3.5a illustrates their interpretation for two dimensions. $\langle \mathbf{d}, \mathbf{e_1} \rangle$ corresponds to the number of initial columns while $\langle \mathbf{d}, \mathbf{e_2} \rangle$ identifies the initial rows. In comparison with Fig. 3.4, this leads to a shift between the source and sink invocations. For more than two dimensions, $\langle \mathbf{d}, \mathbf{e_i} \rangle$ represents the number of initial hyperplanes as depicted exemplarily in Fig. 3.5b. Note that initial delays do not influence the result of the balance equation.

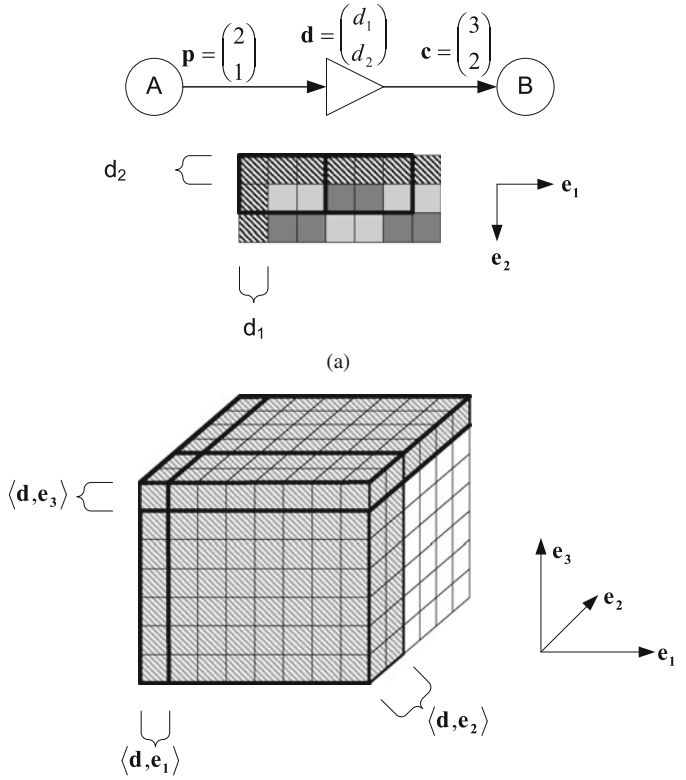


Fig. 3.5 MDSDF delay. **a** Two-dimensional delay. **b** Extension to three dimensions

Murthy and Lee [215] propose an extension that allows modeling of complex sampling patterns along arbitrary lattices. In [29, 30, 197], MDSDF is enhanced by annotation of array

indices for parallelization of sequential C programs. However, as MDSDF does not consider overlapping windows, its application to design of image processing applications is difficult if not impossible.

3.1.4.2 Communicating Regular Processes (CRPs)

Communicating regular processes (CRPs) [105] address specification of parallel and static multidimensional applications such that formal analysis and optimization are possible. To this end, applications are decomposed into processes communicating over multidimensional channels. The latter can be thought of an array of infinite size, in which each element can only be written once. Reading obeys similar semantics as for KPNs in that a process is not permitted to check whether a cell has already been written or not. Instead, the process blocks when reading an array element that has not already been produced. However, in contrast to KPNs, a channel might be read by multiple processes. This is possible because reading is non-destructive. In other words, there is no means of emptying the cell of a channel. From the theoretic point of view such a behavior can be obtained by assuming an infinite array where each cell has an associated valid bit. The latter is set to true during production of a given cell. From that point on it can be read as often as desired. As practical implementations, however, require bounded memory, static compile time analysis is necessary in order to figure out when a given data element is not accessed anymore and can be discarded from the channel.

In order to allow this kind of analysis, processes should consist of static loop nests and assignment statements on arrays whose subscripts must be affine forms in constants and surrounding loop counters. Based on these restrictions, polyhedral dependency analysis permits to construct static schedules such that the channel valid bits can be omitted and that finite memory is sufficient for implementation. A corresponding extension of the C language has been proposed that supports specification of CRPs. In essence, similar constructs as in SystemC have been used, however, without usage of object-oriented programming.

Although CRPs allow specification of complex static applications, the need to perform fine-grained dependency analysis before any execution of the model makes simulation computational expensive. Furthermore, a verification phase is required in order to ensure the single assignment property. Additionally, CRPs are not able to handle control flow or even data-dependent decisions due to the critical impact of static compile time analysis. And lastly, hardware implementations of CRPs require to statically schedule the complete system whereas hand-built solutions typically employ self-timed scheduling by means of FIFO communication.

3.1.4.3 Array-OL

Similar to CRPs, *Array-OL* [47] also addresses the specification of regular applications in form of process graphs communicating multidimensional data. However, in contrast to the previous approach, processes do not operate on arrays of infinite size. Instead, although one array dimension might be infinite, the processes themselves only operate on sub-arrays of given sizes and transform input arrays into output arrays. A hierarchical modeling approach allows to define fine-grained data dependencies on the granularity of individual data elements.

Originally developed by Thomson Marconi Sonar[6] [84], *Array-OL* serves as model of computation for the *Gaspard2* design flow described later on in Section 3.5.2.2. Furthermore, it has also been integrated into the Ptolemy framework [92]. In order to express full application parallelism, the execution order by which the data are read and written is not further specified except for the minimal partial order resulting from the specified data dependencies. In other words, it does not address the out-of-order communication described in Section 2.2, but only defines which output pixel depends on which input pixel.

For this purpose, applications are modeled as a hierarchical task graph consisting of (i) *elementary*, (ii) *hierarchical*, and (iii) *repetitive* tasks. *Elementary* tasks are defined by their input and output ports as well as an associated function that transforms input arrays on the input ports to output arrays on the output ports. *Hierarchical* tasks connect *Array-OL* subtasks to acyclic graphs. Each connection between two tasks corresponds to the exchange of a multidimensional array. A possible infinite dimension is interpreted as time while the other dimensions are considered to be toroidal. In other words, sliding window pixels that leave the array on the right border re-enter it on the left side. Whereas this complies with applications like the fast fourier transform, it complicates implementation of sliding window algorithms, which are typically non-toroidal.

Repetitive tasks as shown exemplarily in Fig. 3.6 are finally the key element for expressing data parallelism (see Section 2.3). In principle, they represent special hierarchical tasks in which a sub-task is repeatedly applied to the input arrays. Although there exist extensions for dependencies between different sub-task invocations [47], the latter are a priori considered as independent of each other. Hence, they can be executed parallel or sequentially in any order. The so-called *input tilers* specify for each sub-task invocation the required part of the input arrays. The produced results are combined by *output tilers* to the resulting array.

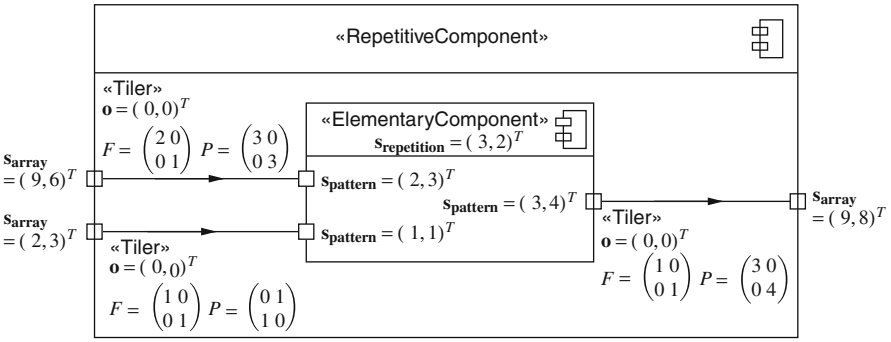


Fig. 3.6 Specification of repetitive tasks via tilers. Each port is annotated with the produced and consumed array size. The number of sub-task invocations is defined by vector $\mathbf{s_{repetition}}$. The tiler associated with each communication edge specifies the extraction of sub-arrays

Each tiler is defined by a paving and a fitting matrix as well as an origin vector $\mathbf{o}$ (see Fig. 3.6). The *paving matrix* P permits to calculate for each task invocation $\mathbf{i}$ the origin of the input or output pattern:

[6] now Thales

$$\forall \mathbf{i}, 0 \leq \mathbf{i} < \mathbf{s_{repetition}} : \mathbf{o_i} = (\mathbf{o} + P \times \mathbf{i}) \bmod \mathbf{s_{array}}.$$

$\mathbf{s_{repetition}}$ defines the number of repetitions for the sub-task. $\mathbf{s_{array}}$ is the size of the complete array; the modulo-operation takes into account that a toroidal coordinate system is assumed. In other words, patterns that leave the array on the right border re-enter it on the left side. Based on the origin $\mathbf{o_i}$, the data elements belonging to a certain tile can be calculated by the *fitting matrix* F using the following equation:

$$\forall \mathbf{j}, 0 \leq \mathbf{j} < \mathbf{s_{pattern}} : \mathbf{x_j} = (\mathbf{o_i} + F \times \mathbf{j}) \bmod \mathbf{s_{array}}.$$

$\mathbf{s_{pattern}}$ defines the pattern size and equals the required array size annotated at each port.

Usage of so-called *mode-automata* permits the modeling of control flow [185, 186]. For this purpose, a special task type, called *controlled task*, is introduced. As depicted in Fig. 3.7, it can select between different sub-tasks depending on a control port specifying the desired mode. Each sub-task must show the same interface; hence, the same input and output ports. If this is not the case, then a corresponding hierarchical wrapper must be generated that offers default values for the non-connected input ports and non-available output ports. The values for the control port can either be generated externally, for instance, by a user, or internally in the model. In the latter case, a *controller task* is introduced consisting of a finite state machine. Each state corresponds to a desired mode of the *controlled task*. State transitions can depend on external events or on input tokens generated by ordinary data tasks such that data-dependent decisions are possible.

The granularity on which the controlled task switches between alternative sub-tasks can be adjusted by intelligent creation of task hierarchies. By this means, it is possible to ensure that mode switches can only occur after complete processing of an image, a line, a block, or an individual data element. However, independently of the granularity, Array-OL requires a strict dependency relation between a produced output value and one or several input tokens. In other words, independently of the currently selected mode it can be decided which input data values are required in order to produce a given output value.

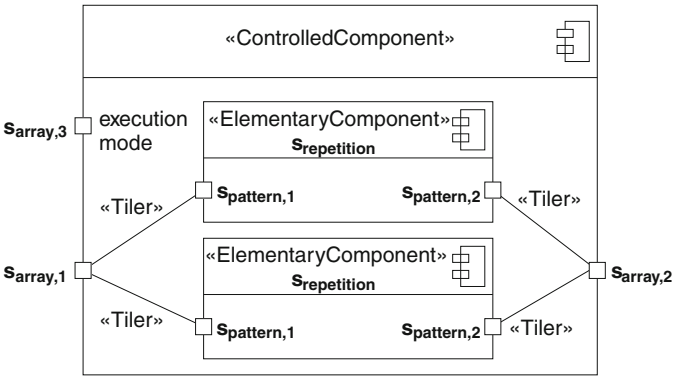


Fig. 3.7 Modeling of the different running modes in Array-OL

3.1.5 Conclusion

As already shown by the extension of Section 3.1, selection of the most suited specification method for system level design is a complex venture. Although widespread in software development, purely sequential approaches have severe difficulties to express system level parallelism. Consequently, in most cases they are extended by techniques that permit to describe communicating processes. Due to data exchange protected by handshaking protocols, *communicating sequential processes* (CSPs) offer advantages for design of heterogeneous systems. On the other hand, this also means that decoupling of processes is more complex. Furthermore, in case of image processing applications, they lack to express true data dependencies.

The latter are represented much more naturally in data flow models of computation. However, although a huge amount of different one-dimensional models of computation have been proposed and powerful analysis and optimization methods have shown their usefulness, application to image processing still poses some difficulties. Their major disadvantage is the incapability to express multidimensional data dependencies required for polyhedral analysis. This not only prevents application of many powerful analysis and synthesis methods, but also leads to specifications that are not able to represent important information like parallel data access. Furthermore, resulting models are complex and difficult to reuse.

Multidimensional models of computation, on the other hand, are far less spread than their one-dimensional counterparts. Many aspects like parametrization, control flow, and interaction with one-dimensional models of computation are rarely treated in literature. From the earlier mentioned approaches, *Array-OL* seems to offer the most benefits. Consequently the remainder of this book will rely on a similar model of computation, called *windowed data flow (WDF)*. It is more adapted to image processing applications by offering the following extensions.

3.1.5.1 Border Processing

Whereas *Array-OL* assumes toroidal arrays, such a behavior is rarely found in local image processing applications. Consequently, border processing is often not considered in examples using *Array-OL*. In contrast, *WDF* explicitly models virtual border extension in order to take the resulting particularities in scheduling into account.

3.1.5.2 Communication Order

In contrast to *Array-OL*, production and consumption order of data elements are an elementary part of the WDF specification such that scenarios like block building or tiling (see Section 2.2) can be accurately described.

3.1.5.3 Dependency Modeling vs. Data Flow Interpretation

Although *Array-OL* have many aspects in common with typical data flow models, it differs in that it models *data dependencies*, not *data flow* [47]. Whereas this differentiation is difficult to understand at the first moment, it can be clarified by means of Fig. 2.5. In fact, the control flow required for resource sharing of the wavelet transform does not change anything in the resulting data dependencies between the different wavelet subbands but just models temporal

constraints. Consequently, since *Array-OL* models data dependencies, resource sharing is difficult or even impossible to represent. A data flow actor, on the other hand, can easily depict such a scenario by including a finite state machine as, for instance, proposed by FunState (see Section 3.1.3.7).

Similar difficulties occur when due to data-dependent operations no real data dependencies can be established. Taking, for instance, a pattern matching operation that shall output the coordinates of all bottles found in an image. Whereas such a pattern matching can be performed by sliding window correlation, the amount of output coordinates depends on the image content. Furthermore, it is not possible to define during compile time the dependencies between sliding windows and coordinates.

Related to this aspect is the *Array-OL restriction* of identical interfaces for all modes of a task [185]. Consequently, each input port must be read (although the data might be discarded) and each output port must be written. Data flow specifications do not have these restrictions as clearly shown by the BDF switch and merge actors, thus simplifying application modeling.

Consequently, *WDF* employs data flow semantics instead of data dependency specification. Introduction of FunState-like finite state machines permits to detect the presence or absence of control flow. Furthermore, this approach also helps processing of cyclic graphs, which is restricted in *Array-OL*.

3.1.5.4 Tight Interaction Between One- and Multidimensional Data Flow

Due to the usage of data flow semantics, tight interaction between *WDF* and one-dimensional data flow models of computation is possible.

3.1.5.5 Restriction of Accepted Window Patterns

Whereas *Array-OL* permits arbitrarily complex sliding window forms, *WDF* uses rectangular geometries. Whereas this simplifies hardware generation, it does not reduce applicability of the model as shown later on in Section 5.8.1.3. This also avoids complex verification of the single assignment property as required for *Array-OL* and *CRP*.

3.1.5.6 Flexible Delays

In contrast to Array-OL, WDF delays can be used anywhere in the application specification.

3.2 Behavioral Hardware Synthesis

Independent of the methodology chosen for modeling, behavioral hardware synthesis is a key element for system level design of complex (image processing) applications. Consequently an enormous quantity of both academic and commercial solutions have been proposed addressing efficient hardware generation from high level of abstractions.

As this book does not primarily address behavioral synthesis, the current section is not intended to give an encompassing overview on existing tools or proposed approaches. Instead, it will investigate them from the system level design point of view. In particular, several fundamental approaches will be introduced and evaluated concerning their ability to handle, analyze, and optimize complex systems.

3.2.1 Overview

Although behavioral synthesis has been studied for several years now, efficient hardware generation is still a challenging problem [6]. Consequently, a huge amount of fundamentally different approaches have been proposed each offering its own benefits.

ASC [208] and *JHDL* [145] aim to describe RTL designs by C++ or Java classes. Whereas they provide many ways for user intervention, they stay on low levels of abstraction. In *Bluespec SystemVerilog,* systems are represented in form of cooperating finite state machines. They are modeled in form of rules describing under what condition state elements are updated. Whereas this results in more flexible and compact representations than VHDL or Verilog designs, the description is still rather low level as all state transitions are assumed to take exactly one clock cycle. Reference [56] enhances the possibilities for analysis and optimization by decomposing systems into finite state machines communicating over FIFOs. A state transition is assumed to take one clock cycle and communication queues are typically small. Several methods are proposed for automatic pipelining of applications even if they exceed simple feed-forward chains. Furthermore, different analysis and optimization methods are proposed including determination of minimum buffer sizes or pipeline optimization. However, most approaches stay rather low level and buffer analysis is computationally expensive.

Cascade [75] and *BINACHIP* [46] address the translation of software binaries into hardware accelerators. *AutoPilot* [1], formerly *xPilot* [70], uses the *LLVM* compiler [202] in order to derive an intermediate low-level representation using a virtual instruction set that can then be translated into hardware. In all cases, system level analysis and optimization are difficult due to the low levels of abstraction.

Template-based synthesis such as employed in *SHIM* (see Section 3.1.2.1) directly maps the resulting *control data flow graph (CDFG)* to hardware. Similar methods are employed by *CASH* [52, 53] in order to generate purely combinatorial logic. This means, however, that computation directly follows the sequential program without exploiting resource sharing, which can become expensive in case of large CDFGs.

Classical behavioral synthesis simply spoken generates a special-purpose processor that executes the input program. Allocation of varying hardware resources permits to trade computation time against required chip size. Program transformations like loop unrolling, constant propagation, or dead code elimination increase achievable parallelism and efficiency of the generated hardware implementation. Tools for classical behavioral synthesis are, for instance, *GAUT* [73], *SPARK* [122, 123], *SA-C* [216], *ROCCC* [121], *Forte Cynthesizer* [107], *ImpulseC* [151], *CatapultC* [209], or formerly *MATCH* [23].

Polyhedral synthesis finally targets the mapping of regular calculations in form of nested loops onto parallel processing units. Those processing units might be (weakly) programmable processors or special-purpose hardware accelerators. Their number determines the achievable throughput as well as required chip size. Polyhedral dependency analysis helps to derive strongly pipelined static schedules offering high throughput at the expense of supported program structures. Data-dependent while-loops, for instance, are typically not supported. *PARO* [130], *MMAlpha* [120], and *PICO Express* are corresponding tools providing this kind of high-level synthesis.

In the following sections, those approaches showing a relation to multidimensional system design shall be described in more detail.

3.2.2 SA-C

SA-C [91, 216, 250, 290] is a variant of the C language and has been developed in the context of the *Cameron* project [68, 128]. It focuses on efficient hardware generation for image processing applications described on higher level of abstraction compared to classical RTL coding. To this end, SA-C is limited to single assignment programs and introduces a special coding primitive for sliding window algorithms in order to describe image processing *kernels* like a median filter or a wavelet transform. The target architecture consists of a workstation equipped with an FPGA accelerator board executing the computational-intensive kernels. Communication takes place via frame buffers. In other words, the workstation stores the input images for the kernel to execute into some shared on-board memory. Once completed, the hardware accelerator transforms the input data into the output images, which are stored in the output memory.

In order to simplify hardware generation, an SA-C compiler has been developed that is able to synthesize a single image processing kernel into RTL code. For this purpose, the SA-C code is translated into an extended data flow representation. Its nodes not only represent simple operations like additions or multiplications but also more complex tasks like generation of sliding windows or control flow for while-loops. Compiler optimizations like loop fusion, common subexpression elimination, or loop unrolling help to attain high throughput while reducing the required hardware resources.

Application to different applications illustrates the achievable performance gained compared to optimized software execution. However, due to the restriction of single kernel synthesis, system level design is not addressed.

3.2.3 ROCCC

ROCCC (Riverside optimizing configurable computing compiler) [121] extends the previously described approach in that it starts with standard ANSI-C in form of nested loops. The addressed target architecture resembles that of Section 3.2.2 by assuming communication via shared external memories. In order to avoid I/O-bottlenecks, array accesses occurring in the computation algorithm are replaced with scalars allowing for parallel access. These scalars are fed by a so-called *smart buffer,* which caches recently accessed data in order to reduce the required memory traffic. Technically, the smart buffer consists of several concatenated registers and an address generator requesting the required pixels from the external memory. The state of the smart buffer is controlled by a finite state machine assuring correct data input and output. However, currently the smart buffer is not able to exploit multidimensional reuse. In other words, taking, for instance, a sliding window algorithm, data reuse is only exploited in horizontal, but not in vertical dimension. Reference [87] adopts this smart buffer in order to extend *ImpulseC*.

A particularity of *ROCCC* is its ability to handle while-loops within for-loops. Normally, the latter are executed such that in each clock cycle a new iteration is started. However, when a while-loop has to be executed, it stops the outer for-loops until its termination. For behavioral synthesis, the *SUIF* [275] compiler framework is used, which outputs an assembler like intermediate representation. These operations have then to be scheduled on the available hardware resources.

Although *ROCCC* introduces a significant extension of SA-C, it does neither address system level design nor out-of-order communication or buffer size determination.

3.2.4 DEFACTO

DEFACTO (***D**esign **E**nvironment **f**or **A**daptive **C**omputing **T**echnolo**g**y*) [88, 316] differs from the previous approaches in that it explicitly addresses system level design by communicating modules together with powerful analysis for data reuse occurring in sliding window algorithms and design space exploration. Hence, in order to select the best implementation alternative, different compiler transformations are applied whose impact is estimated by a behavioral synthesis tool. This permits to explore a large number of area and time tradeoffs in short time.

The *DEFACTO* compiler sets on top of the *SUIF* [275] compiler and addresses programs of communicating nested loops obeying the following restrictions [316]:

- No usage of pointers.
- All array subscript expressions are *affine,* which means that they are of the form $A \times \mathbf{i} + \mathbf{b}$, where A is a constant matrix, $\mathbf{b}$ a constant vector, and $\mathbf{i}$ the considered iteration vector (see also Section 3.1.1*).*
- The loop bounds must be constant leading to rectangular iteration spaces.
- The graph of loop nests is supposed to be acyclic [316].

In order to generate the desired implementation alternative, *DEFACTO* restructures the input C code with the aim to increase the parallelism by different loop transformations and code optimizations. An automatic design space exploration helps to select the performed optimization types and the applied parameters in such a way that the resulting implementation optimally exploits the available memory bandwidth while minimizing the required resources. Based on these methods, complex software systems can be accelerated by compiling parts of them for execution on an FPGA accelerator board while the remaining code is executed on a host PC [88]. Whereas in this case communication takes place via the PCI bus, it is also possible to synthesize loop nests that communicate directly in hardware [316, 317].

The following sections detail these aspects in some more detail by summarizing the proposed design flow including communication synthesis, memory organization, and data reuse exploitation.

3.2.4.1 Design Flow

Translation of the input C specification starts by a hardware–software partitioning step where it is decided which parts of the system shall execute in hardware. Furthermore, some basic compiler optimizations like common subexpression and dead code elimination help to improve the efficiency of the final implementation.

This step is followed by different high-level loop transformations that expose contained parallelism and possible data reuse. *Loop permutation,* for instance, switches the order by which the iteration variables are traversed. This allows reducing the so-called *reuse distance*. It indicates how much temporary storage is required if a data element shall only be fetched once from the external memory in order to increase the overall system throughput. *Tiling* also reduces the required storage quantity for intermediate data buffering by dividing the array to produce into several rectangular regions, which are processed sequentially. Unfortunately, this is typically at the expense of necessary external memory. Furthermore, it causes that data elements required for more than one output tile have to be read several times from the

external memory.[7] *Loop unrolling* finally restructures the loop nest such that several iterations are merged into one loop body in order to increase the contained parallelism.

After these high-level loop transformations, communication analysis identifies the data that are produced in one loop nest and consumed in one or more other loop nests. For each communicated array and each concerned loop nest, a compact representation called *reaching definition data access descriptor (RDDA)* is determined, which contains the following information:

- Whether the loop nest reads or writes the array
- Which statements are covered by the RDDA (an array can be read by multiple statements)
- Which data elements of the array are effectively accessed (a loop nest does not need to read or write all array elements)
- The traversal order of the array, hence which dimension is traversed first, and in which direction
- Which loop variable is dominant for each array dimension traversal

Although fine-grained data dependency analysis is impossible with this representation, it can be used for determination of the permitted communication granularity [317]. This bases on the underlying premise that communication need not to be performed on pixel level, but it is also possible to read and write large data blocks. In principle each *iteration variable* (see Section 3.1.1) corresponds to a possible granularity, whereas a granularity is only allowed if the resulting production and consumption occur in the same order. Consequently, fine-grained out-of-order communication is typically not possible. Instead, the communication has to take place in large blocks of data, which results in increased latency and required buffer sizes.

The communication itself can be implemented in two different fashions [317]. The first possibility consists in using an external memory in order to buffer the data that shall be transported. Then, however, only sequential execution of the source and the sink is possible. For on-chip memory, the data can be directly transmitted from the source to the sink via signal arrays and a simple handshaking protocol. In this case, both the sink and the source can execute in a pipelined fashion and each communication operation takes two clock cycles. However, this means that large register banks have to be used and that coarse-grained communication delivers better results than transport of individual pixels. Unfortunately, this does not correspond to the observations made for hand-built designs. Furthermore, a later publication [316] experienced complexity issues and considered thus only sequential execution of the different loop nests [316].

Next, memory access optimization in form of *variable splitting* and *scalar replacement* is performed in order to increase achievable system throughput. *Variable splitting* permits to distribute an array over several memory banks in order to avoid I/O bottlenecks when multiple members have to be accessed in parallel [26, 259]. In case that input data have to be read multiple times as for sliding window algorithms (see Section 2.2), *scalar replacement* exploits data reuse in order to avoid repetitive external memory accesses by temporarily storing array variables into register banks or chains [24, 25, 258].

The memory access and computation are decoupled by means of FIFOs in order to make the computation part independent of the memory timing [237]. To this end, the system is split

[7] Note that this optimization differs from the JPEG2000 tiling described in Section 2.2 in that the latter introduces an additional border processing in order to avoid the resulting multiple data accesses. As such a behavior, however, changes the produced output result, it cannot be used automatically by the *DEFACTO* compiler.

into three parts: (i) the memory subsystem; (ii) memory management; and (iii) computation (data path). The memory management connects the computational part with the memory subsystem by performing address calculation and arbitration between different accesses. The latter is performed by means of a finite state machine that might employ simple round-robin scheduling or that can profit from more detailed application knowledge. Furthermore, it is responsible for meeting the timing requirements of the used memory. The computational part, on the other hand, reads the data from simple FIFOs and writes the results also back to simple FIFOs.

The performance and chip size of the resulting design are analyzed by the automatic design space exploration, which runs a behavioral synthesis and checks whether all user constraints are met. By means of different heuristics that try to optimize chip area and processing speed, the parameters for the different loop optimizations can be adjusted [25, 238] and evaluated in a new run of the above-described processing steps. In this context, special care is taken that both computation and communication are balanced such that they do not slow down them reciprocally.

In [316], the design flow is applied to three examples in form of a machine vision kernel structured as three simple loop nests, an automatic target recognition kernel consisting of five loop nests, and a discrete cosine transform encompassing two loop nests.

3.2.4.2 Data Reuse

One of the major benefits of the *DEFACTO* compiler is its ability to exploit data reuse in an automatic manner [24, 25, 258]. This helps to avoid repetitive access to identical data elements in external memory and thus improving system throughput. As this is particularly beneficial for sliding window algorithms, it shall be discussed in more detail in the following section.

The reuse analysis can be applied to a single perfectly nested loop reading one or several arrays as shown in Fig. 3.8a. Each array α can be accessed with P_α array references obeying the following restrictions:

- All array indices are affine functions $A_\alpha \times \mathbf{i} + \mathbf{b_{\alpha,k}}$, $1 \leq k \leq P_\alpha$ of the iteration vector $\mathbf{i} = (i_1, i_2, \ldots, i_n)$ whereas i_1 belongs to the *outermost* loop level, i_n to the *innermost* one. The number of matrix rows defines the number of array dimensions. Note that the matrix A_α has to be the same for all references to a given array α.
- All loop boundaries are constant.
- Each column of A_α differs in at most one element from zero (the so-called *separability condition*).

The different accesses to an array α are organized in reuse chains as depicted in Fig. 3.8b. Each node corresponds to an array reference while arcs represent reused data flowing in edge direction, because identical data elements are accessed. Two different scenarios have to be distinguished. In *self-reuse,* the repetitive data access originates from one single array reference. In case of distinct array references, the terminology *group-reuse* is employed. In both cases, two loop iterations $\mathbf{i_1}$ and $\mathbf{i_2}$ access the same data element if and only if the following holds:

$$A \times \mathbf{i_1} + \mathbf{b_1} = A \times \mathbf{i_2} + \mathbf{b_2}\,,$$

$$A \times \left(\underbrace{\mathbf{i_2} - \mathbf{i_1}}_{\Delta \mathbf{i}} \right) = \mathbf{b_1} - \mathbf{b_2} \,,$$

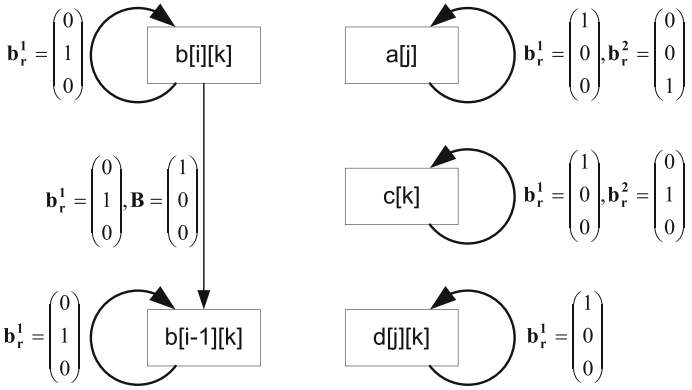
whereas $\mathbf{b_1} = \mathbf{b_2}$ when reuse occurs due to the same array reference (self-reuse). $\Delta\mathbf{i}$ is called a *reuse vector,* because it permits to derive the iteration $\mathbf{i_2}$ that reads the same data element than the considered iteration $\mathbf{i_1}$. The set of all possible reuse vectors can be calculated by $Ker\,(A) + \mathbf{B}$ where $\mathbf{B}$ is called *distance vector* and solves

$$A \times \mathbf{B} = \mathbf{b_1} - \mathbf{b_2} \,.$$

$Ker\,(A)$, the kernel of matrix A, is defined as

```
for(i=1; i <= i_max ; i++){
  for(j=1; j <= j_max ; j++){
    for(k=1; k <= k_max ; k++){
      ... = a[j];
      ... = b[i][k]+b[i−1][k];
      ... = c[k];
      ... = d[k];
    }
  }
}
```

(a)



(b)

Fig. 3.8 Reuse chain [89]: (**a**) Sample code and (**b**) Reuse chains for sample code

$$A \times \mathbf{x} = 0 \Leftrightarrow \mathbf{x} \in Ker\,(A)\,.$$

The base vectors $\mathfrak{B}_r = \{\mathbf{b_r^i} \mid 1 \leq i \leq \dim\,(Ker\,(A))\}$ of $Ker\,(A)$, also called *base reuse vectors*, depend on the structure of matrix A. A column of zeros corresponds to a so-called *free iteration variable,* which does not occur in any dimension of the considered array reference. As a consequence, this leads to self-reuse and corresponds to an elementary base reuse vector that differs only in one component from zero. Considering $c[k]$ in Fig. 3.8a, i and j are such free variables leading to base vectors of $\mathbf{b_r^1} = (1, 0, 0)^T$ and $\mathbf{b_r^2} = (0, 1, 0)^T$. Their linear combination determines the "distance" between accesses to identical data elements and are thus annotated as edge properties in the graph of reuse chains (see Fig. 3.8b). For each group-reuse additionally the distance vector $\mathbf{B}$ is added.

Defining $level\left(\mathbf{b_r^i}\right)$ as the outermost non-zero element of $\mathbf{b_r^i}$ $(\forall 1 \leq j < level\left(\mathbf{b_r^i}\right) : \langle \mathbf{b_r^i}, \mathbf{e_j} \rangle = 0)$, the base reuse vectors $\{\mathbf{b_r^i} \mid 1 \leq i \leq \dim\,(Ker\,(A))\}$ can be chosen such that

- for each loop level k, there exists at most one vector $\mathbf{b_r^i}$ with $level\left(\mathbf{b_r^i}\right) = k$;
- they contain at most two non-zero components (due to separability condition);
- they are lexicographically positive[8];
- that all components are integers with smallest possible absolute values.[9]

Based on these definitions it is now possible to avoid repetitive accesses to identical data elements situated in external memories by temporarily storing them into on-chip memory or even register banks. Principally this reuse exploitation can occur on different iteration levels. Taking, for instance, the reference $a[j]$ in Fig. 3.8, the latter accesses the same data element in each innermost iteration over k. Consequently, most of the memory accesses could be avoided by spending one single register as intermediate buffer. However, $a[j]$ also reads the same data elements for each iteration of the outermost loop. Consequently, even more memory fetches can be eliminated by providing a larger intermediate buffer of $j_{\max}$ elements. To this end a corresponding controller logic has to be generated that directs the array accesses to either the external or the chip-internal memory. Furthermore, the required intermediate buffer size has to be figured out.

In case where each row of A contains at most one element differing from zero (the so-called *single induction variables*), Refs. [89, 258] give the corresponding formulas for calculation of the required intermediate buffer sizes. Exploitation of *self-reuse* for a free iteration variable $\langle \mathbf{i}, \mathbf{e_l} \rangle$ for all iteration levels $k \geq l$ requires a buffer size of

$$m_{\text{self}} = \prod_{k=l}^{n} \begin{cases} 1 & \text{if } \exists \mathbf{b_r} \in \mathfrak{B}_r \text{ s.t. } \mathbf{b_r} = \mathbf{e_k} \\ \mathscr{I}\,(k) & \text{otherwise} \end{cases},$$

with $\mathscr{I}\,(k)$ identifying the number of loop iterations in level k.

For reuse chains containing group-reuse, the *reuse generator* has to be identified, representing the array reference reading the array elements first. Then the distance vector between the reuse generator and the most distant access of the chain in terms of occurring loop iterations has to be determined. Based on this result, a factor can be calculated by which m_{self}

[8] A vector $\mathbf{a} \in \mathbb{R}^n$ is called lexicographic positive if $\exists i : \forall 1 \leq j < i,\ \langle \mathbf{a}, \mathbf{e_j} \rangle \geq 0 \wedge \langle \mathbf{a}, \mathbf{e_i} \rangle > 0$.

[9] The latter can be achieved by dividing the vector with the greatest common divisor of all vector components.

has to be multiplied in order to propagate the stored intermediate values to the most distant reference. Further details can be found in [89, 258]. More insight into data propagation is given in [28].

In case where a row of A contains two elements differing from zero, Baradaran et al. [27] develop a corresponding solution for self-reuse. It can be extended to more than two non-zero matrix row items, but then not all reuse potential is exploited. Furthermore, group-reuse is not considered at all. Additionally the publication sketches a method for code generation of the controller logic that routes the array accesses to the correct memories, but not presented in detail.

For single induction variables, more details about code generation can be found in [28, 89]. The principal idea consists in identifying the iterations that read a reference first or write it last. In the first case, the corresponding data item is fetched from an external memory and put into a temporary on-chip storage. Writing operations only update the external memory if they are the last one touching a considered array member. Two different strategies, namely *loop peeling* and *predicated accesses*, are available. In the first case, some of the iterations are extracted into a separate loop nest while *predicated access* introduces a conditional statement for each array access. Ring buffers help to efficiently handle data reuse.

3.2.4.3 Evaluation and Relation with the Present Book

In comparison with other behavioral synthesis tools, *DEFACTO* stands out by its capability of design space exploration and reuse-analysis. Although the latter cannot be applied to all existing scenarios, it helps to significantly improve the achievable system performance in an automatic manner. This is particularly beneficial for standard sliding window applications. However, the available reuse alternatives together with different high-level transformations lead to a very huge number of possible implementation possibilities, which are difficult to explore automatically [24]. In particular, the fact that a solution can only be evaluated after running a behavioral synthesis is expensive.

Furthermore, *DEFACTO* reveals some of the challenges occurring in current behavioral synthesis techniques, namely generation of communicating modules. Not only do the employed concepts not match the principles used for many hand-built designs, but they also lead to significant complexity [316]. Consequently, the present book aims to put special attention on the inter module communication:

- To this end, it describes a hardware communication primitive that supports both in-order and out-of-order communication on different levels of granularity. In particular, it is not necessary to communicate large data blocks. Furthermore, one read and write request can be served in parallel per clock cycle while achieving high clock frequencies. This is accompanied by the capability to read and write several data values per request.
- Model-based analysis enable efficient buffer analysis in order to determine the size of the required communication memories. Corresponding operations are not performed in the *DEFACTO* design flow.
- The determined buffer sizes are directly taken into account by the developed communication primitive in order to obtain FIFO-like interfaces. This helps in construction of modular systems and is in contrast to the *DEFACTO* communication, which takes place by direct handshaking between the sink and the source [317].
- Finally, this book focuses on data flow specifications instead of sequential C code. This not only perfectly matches the typically used block diagrams (see Fig. 2.1) but also permits

a tight interaction with one-dimensional data flow models of computation, profiting from the various research activities performed in this domain.

3.2.5 Synfora PICO Express

PICO Express [205] differs from the previous high-level synthesis approaches in that it does not perform classical behavioral synthesis. Instead, it employs a so-called *architecture synthesis* by mapping a sequential C-program onto an embedded system. The latter consists of a custom *VLIW (very large instruction word)* processor and *a non-programmable accelerator (NPA)* subsystem. As illustrated in Fig. 3.9, this subsystem contains one or several modules that can be generated automatically from corresponding C code by *PICO Express*. The resulting target architecture is specified by a parameterizable template that predefines possible interconnect possibilities as well as the presence of accelerators. The chosen architecture for implementation is determined by automatic design space exploration. In order to be successful, the input program is not allowed to use pointers, dynamic memory allocation, recursion, goto-statements, or floating point operations. Several design constraints encompassing the necessary clock frequency, system throughput, or the desired target technology help to select the desired implementation alternative. Thereby, it is possible to exploit four different kinds of parallelism:

- *Instruction level parallelism* permits to execute several instructions of the loop body in parallel.
- *Inter-iteration parallelism* allows the simultaneous execution of several loop iterations in parallel.
- In *intra-task parallelism*, it is possible to execute several loop bodies in parallel.

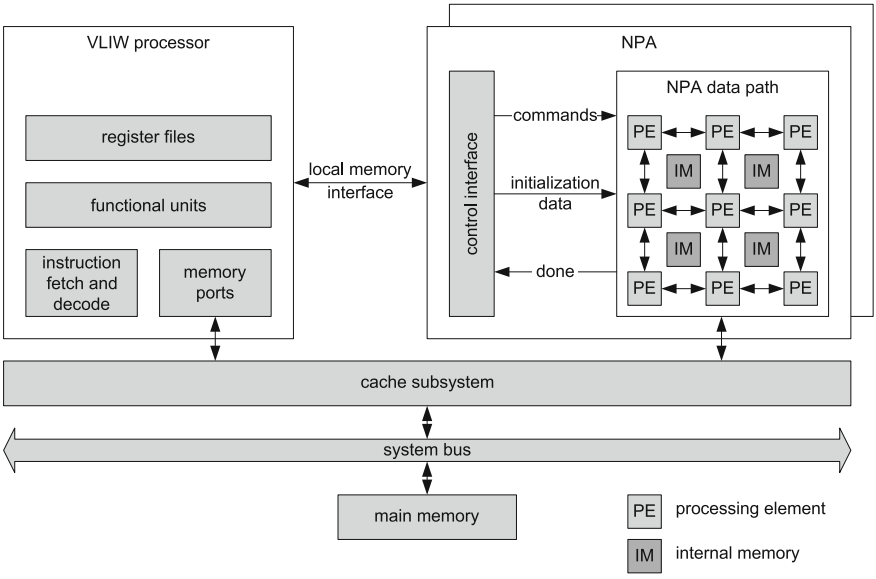


Fig. 3.9 PICO target architecture [162]

- *Inter-task parallelism* finally permits to execute several instances of a task in parallel, whereas a task consists of a sequence of nested loops. This kind of parallelism can be enabled or disabled by the user.

By these methods, Ref. [205] claims to generate results comparable to hand-built designs while achieving a productivity gain of factor 5–20. Several examples such as the image processing chain of a digital camera, a deblocking and deringing filter for block-based video decoders, or an H.264 encoder [76] for D1 video size with 30 fps illustrate the usefulness of the PICO Express tool.

The following sections will present its design flow and give a corresponding evaluation.

3.2.5.1 Design Flow

The first step of the design flow is the identification of the compute-intensive loop nests, also called *kernels*, which can be performed automatically or manually. The identified kernels have to be extended by corresponding input and output interfaces. This can be done by introducing *streaming function* calls as shown in Fig. 3.10. Configuration parameters like the size of the processed image are specified in form of variables, which can be accessed by control signals or by memory-mapped I/O in the final implementation. A simple new top-level design, which consists of several function calls as depicted in Fig. 3.11, represents the overall system functionality, whereas inter-module communication takes place by means of the previously described streaming functions. Furthermore, the software code that remains for execution on the VLIW processor is modified such that it communicates with the hardware accelerators in the desired manner.

Next, *PICO Express* determines and synthesizes for each *non-programmable accelerator (NPA)* the so-called *Pareto-optimal* implementation alternatives by mapping loop nests

```
void img_subblock() {
  for(int j - 0;j<ivsize ;j++){
    //Iterations over vertical direction
    for(int i=0; i<ihsize; i++){
      //Iterations over horizontal direction

      //Input Streaming function call
      //to read streaming data
      data_input_red = pico_stream_input_pixel_red();
      data_input_grn = pico_stream_input_pixel_grn();
      data_input_blu = pico_stream_input_pixel_blu();

      ... //computation kernel algorithmic body ...

      //computed data streamed out using
      //output streaming function calls
      pico_stream_output_red(dci_out[2]);
      pico_stream_output_grn(dci_out[1]);
      pico_stream_output_blu(dci_out[0]);
    }
  }
}
```

Fig. 3.10 Kernel pseudocode with streaming input and output loops [63]

```
void img_pipe() {
  Noise_correction();
  Demosiac_filter();
  Distortion_corrector();
  ...
  //Streaming functions work as FIFO's between blocks
  ...
  Image_scaler();
  Output_format();
  return;
}
```

Fig. 3.11 PICO top-level code [63]

onto corresponding *NPA templates*. These Pareto-optimal solutions have the property of not being *dominated* or outperformed in all their performance characteristics such as chip size or attainable throughput by alternative implementations. For this purpose, the *NPA template* defines a grid of customized *processing elements (PEs)*, *internal memories (IM)*, interfaces to external memory, and a memory-mapped host processor interface as depicted in Fig. 3.9. The host processor interface can be used for transferring commands or kernel parameters. A PE only communicates with its direct neighbors and contains a data path, which principally encompasses functional units and registers. These functional units as well as the associated interconnects are controlled by a loop-specific instruction sequencer taking care of the correct execution of the loop nest. A compact representation of the loop nest in form of a so-called *polytope* model together with the *OMEGA* software system [4] allows complex dependency analysis and permits to map the different loop iterations to processing elements and to schedule their execution [252]. Required data can be placed in either external or internal memory or in registers. Different mapping strategies and quantities of used PEs lead to alternative implementations.

Next, Pareto-optimal configurations of the custom instruction set processor are determined in dependency of the concrete input specification. This step also includes compilation of the remaining software with an adapted compiler taking the custom instructions into account. The so created NPAs and processors can now be assembled during automatic design space exploration to the overall system. Communication between individual NPAs can be performed by means of memory chips or streaming interfaces like FIFOs. Furthermore, NPAs offer external control and configuration signals together with a bus interface for control and monitoring. Special care is taken during automatic design space exploration that only modules with compatible communication schemes are interconnected. In order to achieve efficient implementations, *PICO Express* automatically balances the computation rates between different kernels. FIFO sizes can be determined via simulation.

Additionally, *PICO Express* is able to build a complete verification environment for the generated design. This includes generation of TLM modules for system simulation, RTL test benches that perform stress tests like stalling of the processing chain, as well as a C-RTL co-simulation. Furthermore, an extended simulation is available in order to find application bugs that do not become visible in software but that are disastrous in hardware-like bit width overflows, uninitialized variables, and out-of-bound accesses of arrays. By means of these techniques it is possible to detect wrong input specifications, its erroneous interpretation, or even errors of the employed tools.

3.2.5.2 Evaluation and Relation with the Present Book

In comparison with other high-level synthesis tools, PICO Express stands out by its capability to synthesize optimized systems on chip containing a processor and hardware accelerators. Furthermore, it uses polytope-based dependency analysis and architectural synthesis instead of classical behavioral synthesis. However, due to the usage of standard C code (obeying some restrictions), the offered system level analysis capabilities are limited. For instance, buffer sizes have to be determined during simulation, which can be problematic as shown later on in Section 7.2. Furthermore, inter-module communication is restricted to simple in-order data transport. Consequently, parallel data access or out-of-order communication as described in Section 2.2 is more complex to achieve and requires much more user invention. In [63], for instance, block building is performed by an external controller.

In contrast to this approach, the book in hand describes a data flow-based multi-dimensional system level design methodology:

- It discusses a multidimensional data flow model of computation for extended system level specification. Combination with well-known one-dimensional models of computation allows designing complex systems including data-dependent decisions or even non-determinism. Integration into a system level design tool called SYSTEMCODESIGNER permits to model, simulate, and optimize complex applications mapped on embedded systems including multiple processors and accelerators.
- It explains how to automatically synthesize a hardware communication primitive that permits fast in-order and out-of-order communication including parallel data access. Consequently, this is more powerful than standard FIFO communication.
- It discusses possibilities for efficient system level analysis in order to determine required buffer memories. In particular, two different methods based on simulation and polytope-based analysis are described and compared. This will show that simulation is much slower and might be less optimal than analytic methods. On the other hand, it can cover more general schedules leading to interesting tradeoffs between throughput and required buffer sizes.

3.2.6 MMAlpha

MMAlpha [62, 119] is an academic tool for generation of hardware accelerators using polyhedral analysis. However, in contrast to *PICO Express*, it uses a special specification language and does not handle complete SoC designs. The algorithm is described in form of linear recurrence equations, which are expressed by nested loops in single assignment form. In order to translate them into a hardware implementation, an execution time is assigned to each write operation of an array variable, which can either represent an internal variable or an output port. The execution time of each operation is assumed to be an affine function (see Section 3.2.4) of the loop variables such that all data dependencies are met.

Reference [62] illustrates how this concept can be employed for design of so-called one-dimensional *multirate* systems. They are characterized by the fact that some of the actors or processes execute less often than others due to the presence of up- and downsamplers. In other words, they execute at *different rates.* To this end, MMAlpha determines the execution rate of each module taking possible up- and downsamplers into account. Then, an *integer linear program (ILP)* can be established in order to determine the execution times of each individual module. Unfortunately, its solution might be quite computational intensive. Furthermore, the

considered multirate systems are limited to one-dimensional arrays instead of two- or more-dimensional arrays.

3.2.7 PARO

Similar to *MMAlpha*, *PARO* [130] also targets automatic hardware synthesis of individual loop nests. This is done by the application of a sequence of high-level transformations, followed by scheduling and RTL generation. For this purpose, the loop description is analyzed in order to build the polyhedral dependence graph containing all variables and operations. Based on this dependence graph, the operation scheduling and allocation are performed via *mixed integer linear programming*. It considers resource constraints, data dependencies, and throughput requirements. The obtained *operation schedule* gives the allocated resources as well as the execution times of the loop iterations and the contained operations.

This information can be used for automatic RTL implementation of the hardware accelerator. Figure 3.12 exemplarily shows the generated hardware architecture for a complex 3×3 sliding window algorithm. It consists of three parts, namely (i) the computation kernel, (ii) the internal delay buffers for efficient data reuse, and (iii) the controller. The computation kernel performs the arithmetic operations and employs heavy pipelining such that several pixels can be processed in parallel. It instantiates parameterizable components like adders or dividers which are available in an IP library. Similar to handcrafted designs, the internal delay buffers temporarily store the input pixels such that each of them is read only once in order to avoid I/O bottlenecks. Their size can be calculated from the static operation schedule and the

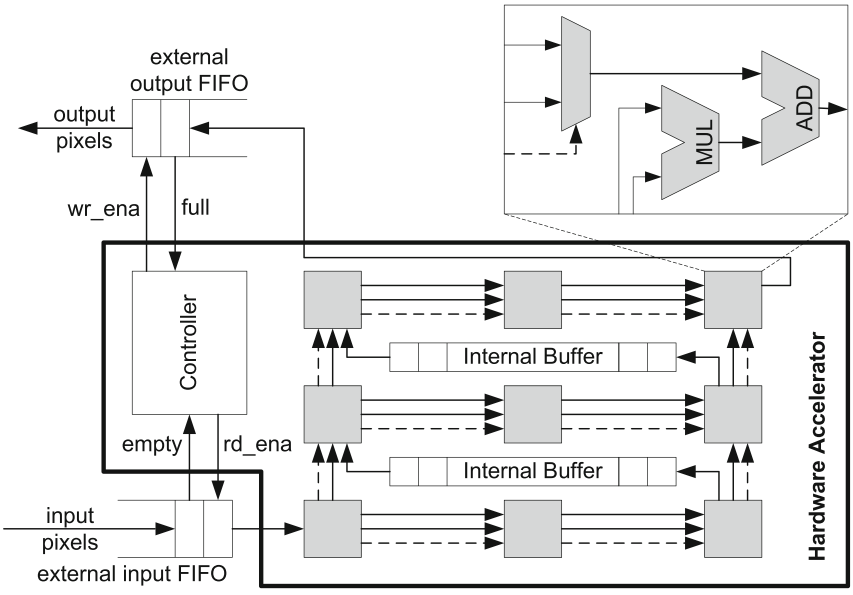


Fig. 3.12 Loop-accelerator hardware generated by the PARO compiler

underlying dependencies. The controller is responsible for keeping track of the image pixels being processed and orchestrates the correct computations and I/O.

Whereas this permits to quickly generate high-performance hardware accelerators, the design of complete systems is not considered further.

3.2.8 Conclusion

Although [6] states that efficient behavioral synthesis is not yet realized today, its breakthrough can be expected for the near future because traditional hardware design on register transfer level has more and more difficulties to cope with the constantly increasing complexity. A huge amount of different solutions having their own benefits are ready to conquer the markets. Polytope-based analysis, for instance, permits efficient parallelization of nested loops while classical behavioral synthesis is able to translate more general program structures. Consequently, a difficult question to solve is which methodology fits best the application needs. This becomes even harder when considering the enormous costs induced by one single license of a behavioral compiler. Fortunately, also on this battle field first low-cost solutions like *ImpulseC* are aggressively brought to market.

However, as far as can be seen today, it will stay an illusion to just download a code from the Internet and translate it into a working hardware implementation. Instead, the code has still to obey several restrictions and has also to be adapted for efficient synthesis results. Corresponding statements can be found for *PICO Express* [63] as well as *Forte Cynthesizer* [107]. Concerning the latter one, critical elements requiring special user attention are, for example, memory architectures, memory size optimizations, as well as address calculation. And despite ongoing research like support of pointers [253], it can be expected that these statements will still be valid for some time if not forever. Consequently, behavioral synthesis should rather be considered as a tool that helps a skilled designer to achieve his goal in shorter time than expecting the translation of arbitrary software into a piece of hardware.

In addition, it became obvious from the above discussion that analysis, optimization, and synthesis of complete systems are still challenging. Many of the previously mentioned tools focus on individual modules while others like *Forte Cynthesizer, PICO Express*, or *DEFACTO* offer first approaches covering complete systems. Nevertheless, there exists still potential for improvement. *DEFACTO*, for instance, uses communication structures not corresponding to manual implementations and it experienced complexity issues. *PICO Express* and *Forte Cynthesizer*, on the other hand, are limited to provision of simple communication functions implementing interfaces to signal or FIFO communication. Consequently, this complicates the design of image processing applications, which require both in-order and out-of-order communication as well as parallel data access and complex address calculation. Furthermore, the possibilities for system level analysis are typically limited to simulation. This is mainly due to the usage of unstructured *C* or *SystemC* modules, which are very hard to analyze automatically. Consequently, none of the discussed approaches offer, for instance, the capability to determine the amount of required buffer space in an analytical fashion.

This is exactly the point where multidimensional system level design as discussed in this book offers several benefits. Starting from the observation that efficient hardware synthesis requires input specifications with certain properties, and thus some code modifications, this monograph goes one step further by using structured *SystemC* modules. These *SystemC* modules consist of input and output ports together with a finite state machine whose transitions

have associated actions implementing some functionality. As a consequence, it is possible to determine to which data flow model of computation a module or actor belongs. This offers the capability of profiting from efficient system level analysis and synthesis methods available for the different data flow models of computation. The description of a multidimensional data flow model of computation particularly adapted for image processing then lays the foundation for expressing out-of-order communication and parallel data access. Both aspects can be efficiently implemented in hardware by means of a communication primitive called *multidimensional FIFO*. It provides a FIFO-like interface and thus adheres to the principle of system decomposition into communicating blocks. An efficient buffer analysis for static application parts finally illustrates the benefits of structured system level description.

However, before going into details, the following sections describe related work in memory and communication synthesis, memory analysis, and system level design.

3.3 Memory Analysis and Optimization

After having intensively discussed how to model image processing applications in order to represent important characteristics as well as how to synthesize them from a higher level of abstraction, this section addresses related work that considers determination of the required memory sizes. This is in fact a very important, but also very complex, task, independent of which specification method has been chosen. Data flow models of computation, for instance, assume in general infinite FIFO sizes, which of course is impossible to handle in real implementations. And sequential, software-oriented description techniques often store arrays whose size matches complete images. This is typically a good choice for standard software because modern systems are equipped with huge memories. However, it is prohibitive for embedded systems or hardware implementations, because in particular fast on-chip memory is a very scarce resource. Consequently, a variety of different techniques have been proposed that aim to determine the minimum amount of memory necessary to perform the desired task with the required throughput. In principle, they can be classified into three categories:

1. Memory analysis for one-dimensional data flow graphs
2. Scalar-based analysis
3. Array-based analysis

Because scalar-based methods do not scale well to processing of huge images, they are not described further, but the interested reader is referred to [140], which contains a brief overview. Instead, only the remaining categories will be shortly discussed in the following sections.

3.3.1 Memory Analysis for One-Dimensional Data Flow Graphs

In contrary to what might be expected, determination of the required buffer sizes on the communication edges is very challenging even for simple static data flow graphs. In order to illustrate some of the difficulties, Fig. 3.13 depicts an SDF graph (see Section 3.1.3.1) consisting of five actors and seven edges. If solemnly edge e_2 is considered and the rest of the graph is ignored, it is obvious that a buffer size of one token is sufficient for deadlock

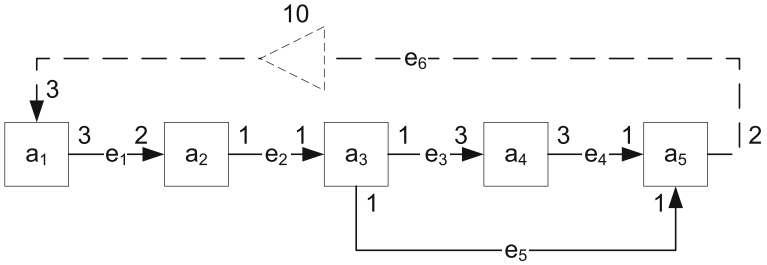


Fig. 3.13 Simple SDF graph that illustrates some of the challenges occurring during automatic determination of the required communication buffer sizes. Initial tokens are indicated by a *triangle*

free execution.[10] For edge e_1, things already become slightly more complex as in this case a buffer size of four tokens is necessary in order to avoid deadlock (actors a_1 and a_2 both fire once, then actor a_1 has to be executed again, leading to four tokens on edge e_1). Thus, the buffer needs to be bigger than the largest token size, which is due to prime consumption and production rates. Unfortunately, it is even not sufficient to consider each edge individually. Taking, for instance, actor a_3, the latter has to execute three times before actors a_4 and a_5 can eventually fire. However, this means that the buffer of edge e_5 has to amount at least three tokens although both its consumption and production rates equal one. And the situation gets even worse if parallel actor execution should be considered. Assuming that each actor requires the same execution time, the buffer of edge e_5 has to be increased to $3 + 2$ tokens because after the third invocation of actor a_3 it takes still two time units until actor a_5 finally consumes the first token.[11] In other words, the required buffer size depends not only on the graph topology but also on the used schedule. Unfortunately, already the determination of such a schedule is very complex, in particular when feedback loops like edge e_6 exist, because then a fully pipelined implementation might be impossible [247].

As a consequence of this complexity and of the variety of possible tradeoffs between buffer size and throughput, various solutions have been proposed in the literature. The work presented in [9–11], for instance, permits to determine tight upper bounds for the minimum buffer size of an arbitrary SDF graph. Reference [111] uses model checking techniques in order to deliver exact solutions. However, typically this will not lead to throughput-optimal solutions as shown by the previous discussion. Reference [267] proposes a method using model checking that permits to determine tradeoffs between attainable throughput and required buffer size. However, it is computationally expensive.

Reference [309] develops a fast estimate for the required edge buffers in SDF graphs and single-processor solutions. Bhattacharyya [42] derives the so-called *single appearance schedules* that minimize both code and edge buffers for single-processor implementations. In contrast, reference [182] abstains from these schedule types, but uses procedure calls, which leads to significantly reduced edge buffer sizes while still keeping code size and execution

[10] Note that in this case the execution semantics of the SDF graph slightly change because actor 2 has to check not only for sufficient input data, but also for enough free space on each output edge. From the theoretical point of view, however, this does not cause any difficulties since the occurring behavior can be modeled by introduction of a feedback edge between actors 3 and 2 [301].

[11] It is assumed that the buffer has to be allocated at the beginning of the producing invocation and can be released at the end of the consuming invocation.

speed acceptable. Reference [273] uses evolutionary algorithms in order to derive buffer-optimized schedules. Reference [228] considers data flow graphs transporting huge composite tokens. Buffer sharing together with the transport of pointers instead of the complete data structure enables efficient software generation in case of single-processor implementations. Reference [213] reduces the amount of required edge memory by sharing it between actor input and output ports, if possible. However, all of these approaches consider single-processor implementations and are difficult to apply for hardware designs.

References [116, 117, 224] use integer linear programming in order to derive minimum buffer sizes for schedules achieving the maximum possible throughput. The approach is limited to one-dimensional models of computation like SDF and HSDF. Furthermore, it risks to become computational intensive as shown in [301]. This chapter assumes linear schedules in order to simplify the buffer size calculation for SDF graphs. Reference [302] extends this method to CSDF graphs. Assuming a constant input stimulus, the technique can even be applied to dynamic data flow models of computation [32]. Publication [85] proposes a simple simulation in order to derive buffer sizes for CSDF graphs. Reference [66] even requires repeated simulation in order to determine buffer sizes that meet a given performance requirement. However, all of these approaches cannot be applied to multidimensional data flow models of computation as described in this monograph.

Finally, there also exist application-specific memory analysis such as [204] for *network on chips (NoC),* linear-chained high-speed networks performing data-dependent operations [260], or RTL synthesis [56]. The latter bases on iterative state-space enumeration and needs thus lots of computation. Furthermore, none of these approaches can be directly applied to buffer size calculation of multidimensional image processing applications.

Consequently, the next section summarizes methods that are particularly adapted for such scenarios.

3.3.2 Array-Based Analysis

Whereas in the previous section tokens have been considered as atomic, that is as non-decomposable, this approach is not feasible anymore for hardware synthesis of applications operating on huge multidimensional arrays. Considering, for instance, a simple sliding window algorithm with a window size of $p \times q$ pixels, it is well known that its hardware implementation requires an intermediate storage of $p - 1$ lines and q pixels independently of the image height [199]. Hence, in addition to the challenges identified in Section 3.3.1, it is furthermore necessary to determine which part of the token or array has to be effectively buffered.

Unfortunately, it has been shown that memory analysis becomes far too computational intensive when treating each pixel as a separate variable [21]. Instead, special analysis methods have been proposed that use more compact representations. They profit from the fact that both array dimensions and *live* variables can be described by *lattices* or *polytopes*. In this context, a variable is called *live* at time $t \in \mathbb{R}$ if its value has been produced at time $t_p \leq t$ and will be read at time $t_c \geq t$. Polytopes or lattices provide an efficient method for their description and are defined as follows:

Definition 3.2 A *polyhedron* is a set of vectors that can be described by intersection of half-spaces in $\mathfrak{R}^n$:

$$P = \left\{ \mathbf{x} \in \mathfrak{R}^n \mid A \times \mathbf{x} \leq \mathbf{b} \right\}.$$

$A \in \mathfrak{R}^{m,n}$ is a matrix with m rows and n columns and $\mathbf{b} \in \mathfrak{R}^m$ a vector. If $\mathfrak{R} = \mathbb{Z}$, P is called a $\mathbb{Z}$-polyhedron.

Definition 3.3 A bounded ($\mathbb{Z}$-)polyhedron with finite $\|P\|$ is called *($\mathbb{Z}$-)polytope*.

Definition 3.4 A *linearly bounded $\mathbb{Z}$-lattice*, shortly called *lattice*, is the image of an affine vector function over an iterator polytope:

$$L = \left\{ T \times \mathbf{i} + \mathbf{u} \mid A \times \mathbf{i} \leq \mathbf{b},\ \mathbf{i} \in \mathbb{Z}^n \right\},$$

with $A \in \mathbb{Z}^{m_A,n}$, $T \in \mathbb{Z}^{m_T,n}$, $\mathbf{i} \in \mathbb{Z}^n$, $\mathbf{b} \in \mathbb{Z}^{m_A}$, $\mathbf{u} \in \mathbb{Z}^{m_A}$.

Based on these definitions, various methods have been proposed that permit to determine the required memory space for an application processing multidimensional arrays. In most cases, they are applied to algorithms described in form of nested loops accessing one or multiple arrays and can be distinguished by different criteria:

- The assumed execution order of the loop nest can be fully or partly fixed or it may even not be specified at all.
- Loop execution is supposed to be sequential or parallel.
- Some approaches support sharing of memory space between different arrays while others do not.
- The determined buffer size might be accurate or only an estimation. Typically, in the latter case, the algorithm is much faster.
- Some approaches require a *perfectly nested loop* meaning that the code can contain exactly one innermost loop that has to contain all statements. Unfortunately, this makes analysis of communicating modules impossible.
- Finally, methods can be categorized in how far they need a computational-intensive solution of ILPs or whether simpler abstractions are used.

Zhao and Malik [314], for instance, develop a method for exact memory size estimation in case of array computation described by sequential code and perfectly nested loops. It is based on live variable analysis and integer point counting for the intersection or union of parametrized polytopes. References [22, 315] extend this work to sequential execution of communicating loop nests. Each array reference is described by a linearly bounded lattice (LBL). Next, these LBLs are decomposed into disjoint sets such that for each of them production and consumption times can be annotated. Based on this information the required buffer size can be derived.

References [79, 80] provide a mathematical framework for buffer analysis in order to investigate different memory mappings in the form of so-called *modular mappings*

$$\sigma : \mathbf{i} \longmapsto (M \times \mathbf{i}) \bmod \mathbf{m}.$$

$\mathbf{i}$ can be either an array index or a loop iteration generating the corresponding array element. This modular mapping assigns to each array element or iteration $\mathbf{i}$, a corresponding memory location $\sigma(\mathbf{i})$, and the number of overall required memory cells evaluates to $\prod_i \langle \mathbf{m}, \mathbf{e_i} \rangle$. In order to solve the question of good modular mappings, the authors present a corresponding theory that starts from a so-called *index difference set* $\mathscr{DS}$. The latter contains the difference of all conflicting array indices or iteration vectors that cannot be assigned to the same memory cell because they are simultaneously live:

$$\mathscr{DS} = \{\mathbf{i} - \mathbf{j} \mid \mathbf{i} \text{ and } \mathbf{j} \text{ cannot be assigned to the same memory cell}\}.$$

Based on these definitions, a criterion is developed in order to verify whether a modular mapping indeed does not assign the same memory location to two conflicting array elements. Furthermore, several heuristics are proposed in order to select a good modular mapping by choosing M and $\mathbf{m}$ accordingly. Due to the utilization of $\mathscr{DS}$, the presented framework is rather general and can be applied to both sequential and parallel implementations with or without memory sharing. Determination of $\mathscr{DS}$, on the other hand, is not in the focus of this chapter.

References [118, 194–196] employ special forms of modular mappings for parallel execution while [284] focuses on sequential programs. In [118], multidimensional arrays are linearized to one-dimensional ones by concatenating lines or columns. Based on this linearization, the largest distance m between two live data elements is determined. Since never more than m array elements will be live at a given time, each access to the linearized array $\tilde{a}[i]$ can be replaced by $\tilde{a}[i \mod m]$. Determination of m is done by solution of several ILPs. In addition, the authors present a heuristic for memory sharing between different arrays. Reference [284] employs an n-dimensional *bounding box* in the original n-dimensional space of array indices. In other words, each array access $a\left[i_1, i_2, \ldots, i_n\right] =: a[\mathbf{i}]$ is replaced by

$$a\left[i_1 \bmod m_1, i_2 \bmod m_2, \ldots, i_n \bmod m_n\right] =: a[\mathbf{i} \, mod \, \mathbf{m}],$$

where the values m_i are determined independently for each dimension by the maximum difference of the conflicting indices. Hence, all live data elements are surrounded by a rectangular multidimensional window representing the required memory space. References [194–196] use the same array access function, but the values m_i are not calculated independently of each other because this allows reduction of the required memory size. All these approaches have in common to purely base on integer linear programs. In contrast, this book will discuss a method exploiting geometric lattice representation in order to guarantee efficient scheduling and memory size calculation of multidimensional data flow graphs. In particular, special attention will be put on out-of-order communication and image up- and downsamplers.

Reference [293] proposes an ILP-based scheduling technique applicable to loop programs. For each operation contained in a loop nest, the authors determine the execution start time, the assignment to one of multiple processing units, and a so-called *period vector*. The latter specifies for each loop iteration level the time between two consecutive executions. By means of complex ILPs, these magnitudes can be determined such that required logic resources and memory can be minimized. Since this becomes computationally too expensive for non-trivial examples, they introduce several approximations and simplifications. Feautrier [105] proposes an ILP-based scheduling with bounded buffer size. In order to improve analysis speed, Feautrier employs *variable elimination* and *modular scheduling*. For this purpose, a channel schedule is introduced, which defines when a data element has been definitively written by the source. This channel function is assumed to be linear in the array index. By means of this channel schedule, it is possible to abstain from the internals of the different modules, which simplifies the ILPs to solve. In order to avoid unbounded memory size, a constraint is added such that the channels can be kept bounded. The channel bound has to be specified by the user.

Besides these ILP-dominated methods, several publications propose a more geometric interpretation in form of lattices and dependency vectors. Figure 3.14 shows a corresponding example in form of a nested loop and the associated lattice model. Each iteration is modeled

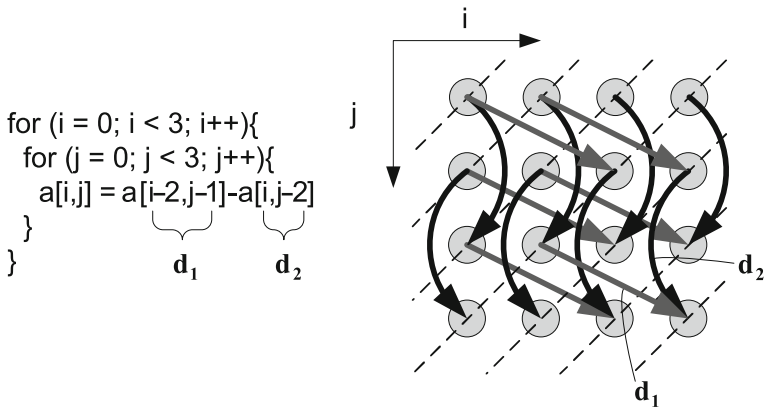


Fig. 3.14 Nested loop and corresponding lattice model. The *dashed lines* illustrate a schedule that executes all lattice points situated on the same diagonal in parallel

as a point in the lattice. The *dependency vectors* $\mathbf{d_i}$ indicate the data flow from a source iteration to a consuming sink iteration. Buffer size analysis is possible by considering all occurring dependency vectors. References [176–178], for instance, develop a technique that allows estimating the required memory sizes while the execution order is only partly fixed. Applied to Fig. 3.14 this means that the lattice might be traversed in *row-major* order or in *column-major* order, whereas the succession of further, non-illustrated loop levels has already been defined. Consequently, a designer can adjust the execution orders during stepwise refinement of the application in order to select good implementation alternatives. On the other hand, this unfixed execution orders reduce the accuracy of the memory size estimation. The latter is performed in two steps. First, a lower and an upper bound for the memory requirement is determined for each array. In case multiple arrays are present in the code, the memory estimation then continues by grouping simultaneously alive data dependencies. The group that requires the largest size requirement defines the memory requirement for the application. Since this approach can get computationally intensive, Hu et al. [143] present an extension which reduces the analysis time. Furthermore, the execution order is assumed to be fixed. Out-of-order communication or special properties of image up- and downsampling as well as throughput-optimized schedules are not further considered.

In order to apply the geometric memory estimation techniques to parallel execution of different loop nests, the latter have to be mapped into a common lattice by overlapping them in such a way that no invocation reads any data that will be produced in future. Corresponding techniques have been proposed by Danckaert et al. [78] and Verdoolaege et al. [291]. Reference [142] shows how this polytope embedding can be optimized for reduced memory requirements. Because its exact solution is computationally very expensive, a corresponding heuristic is developed. Since all source nodes are assumed to have fixed start times, graphs with multiple sources can result in overestimated memory requirements. Furthermore, all the above-mentioned research on geometric buffer analysis does not consider out-of-order communication nor throughput-optimal schedules and special properties of image up- and downsampling.

Reference [292] proposes a buffer analysis, which is part of the *Daedalus* design flow. It can be applied to the so-called *(parametric) static affine nested loop programs (SANLPs)* that consist of a sequence of nested loops whose loop boundaries are either constant or depend

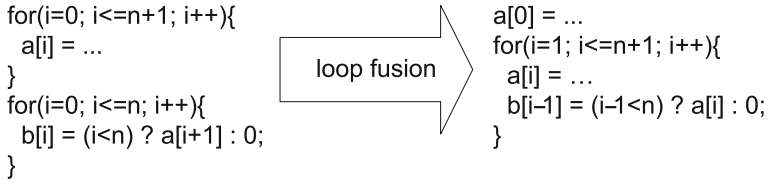


Fig. 3.15 Loop fusion example

only on enclosing loop iteration variables. Furthermore, all occurring array accesses must be affine forms[12] of the enclosing loop iteration variables. The first step of the proposed buffer analysis method consists in mapping all nested loops into a common iteration space, which in principle corresponds to loop fusion as depicted exemplarily in Fig. 3.15. This loop fusion is performed such that no statement reads data that will only be produced in future. This is rather simple when all loop nests have the same depth and iteration boundaries. More complex operations like scaling and loop reordering are required in the opposite case, a corresponding heuristic for different examples if part of Ref. [292]. After this loop fusion, the program under consideration consists of one single nested loop showing only so-called *self-dependencies*. The latter are called *uniform*, if the *dependency vectors* in form of the differences between the source and sink iteration vectors are constant for all iterations. Whereas in this case the buffer size can be calculated rather easily, general self-dependencies require much more complex calculations like determination of upper bounds via Bernstein polynomials or just simulation [292]. The method discussed in this book, on the other hand, particularly focuses on generation of throughput-optimized schedules, supporting out-of-order communication, feedback loops, and multirate systems.

Reference [276] also uses the geometric model illustrated in Fig. 3.14, but investigates parallel execution of different lattice points. Assuming, for instance, the example given in Fig. 3.14, no horizontal dependency vectors exist, thus allowing for parallel execution of a complete row. Furthermore, there also exist schedules that execute all lattice points situated on a diagonal in parallel. However, as shown in [276], selection of a schedule influences the required buffer size. Consequently, the authors develop a corresponding theory that permits to determine the best storage mapping for a given schedule, the best schedule for a given storage mapping, and a storage mapping that is valid for all one-dimensional affine schedules.

Whereas all these approaches start with a loop specification, Thörnberg et al. [282] apply to the *IMEM* design environment (see Section 3.5.2.1) and considers memory analysis for model-based design. Starting from a restricted SystemC model that is able to represent simple sliding window algorithms, they are able to directly extract the dependency vectors. In contrast to pure data flow semantics, they allow edges with multiple sinks. Next they embed each actor into a common iteration grid by solution of an integer linear program. Whereas this permits to derive optimum solutions taking data bit widths into account, ILPs risk to become computationally intensive for large applications. Out-of-order communication or multirate systems are not further considered due to the restricted expressiveness of the *IMEM* model of computation (see also Section 3.5.2.1). Additionally, none of the array-based analysis methods mentioned above tackles non-linear schedules as they might become interesting for out-of-order communication (see also Section 6.4).

[12] (see Section 3.2.4 for a mathematical definition)

3.3.3 Conclusion

Since manual buffer size determination is a laborious and error-prone undertaking, there exist numerous propositions in literature how to automatize this step. Nevertheless, as shown by the above discussions, its efficient solution is still a difficult question. This is partly due to the high complexity of the proposed solution approaches making several assumptions that render transfer to other applications challenging. For instance, except for the solution proposed in the context of IMEM, none of the approaches operate on multidimensional data flow graphs, which have been chosen as specification method for this monograph. As a consequence, out-of-order communication and image up- and downsampling are only sparsely or not at all considered. Additionally, application to hardware synthesis requires special considerations like throughput-optimal scheduling, which is often neglected in the geometric methods. And finally, computational complexity is still a critical issue. During experiments it could be shown that integer linear programs working well for one application suddenly become intractable for another.

Consequently, this book will discuss an efficient buffer analysis technique offering the following benefits:

- It can be applied to multidimensional data flow graphs.
- It performs throughput-optimized scheduling.
- It can be applied to complex graph topologies including feedback loops as well as split and join of data paths.
- It can handle out-of-order communication.
- It is able to consider scheduling alternatives for image up- and downsampling leading to tradeoffs between computational logic and communication buffers.
- It requires only to solve small integer linear programs in order to avoid explosion of the complexity for huge data flow graphs.
- It provides two different solution methods for the required integer linear programs. Whereas the first is mostly very fast, it can become computational intractable in several cases. In such situations, the second approach can be used, which is typically slower than the first, but which offers the great advantage of having bounded computation time.

3.4 Communication and Memory Synthesis

Typically, image processing applications work on huge amounts of data. Consequently, data exchange and buffering are a critical issue that has to be considered extensively in order to obtain efficient high-performance implementations. For this purpose, Section 3.1 investigated different methodologies for system level modeling such that all important algorithm properties like parallelism or communication orders are available to the analysis tools. Section 3.2 then considered behavioral synthesis methods and revealed deficiencies in system level analysis and communication flexibility. Consequently, the following sections aim to address communication and memory synthesis. In particular, they focus on four main aspects, namely

1. Memory mapping
2. Parallel data access
3. Data reuse including sliding windows
4. Out-of-order communication

3.4.1 Memory Mapping

Memory mapping considers the question how to distribute arrays or scalar variables to different physical memories. Reference [20] starts from a non-procedural loop program[13] and tries to figure out the best memory configuration. To this end, the authors establish an integer linear program that determines the number of required memory modules and their port configurations and bit widths such that the given number of parallel data accesses is possible while minimizing hardware costs. This does not take into account that FPGA implementations only allow limited configuration possibilities for embedded memory blocks. References [231, 232] develop an ILP formulation that maps $M \in \mathbb{N}$ data structures to $K \in \mathbb{N}$ memory instances belonging to $N \in \mathbb{N}$ memory types such that implementation costs are minimized. They are modeled as a weighted sum including memory latency, pin delay, and pin I/O costs. For the latency calculation it is assumed that the access frequency of a data structure is proportional to its size. Reference [155] focuses on the question how to build a logical memory from memory primitives available in a library. Reference [274] proposes a corresponding solution that reduces energy consumption when the attainable clock frequencies do not represent the bottleneck. However, none of these publications considers interaction between memory mapping and address generation nor communication synthesis in streaming applications.

Reference [257] sketches how homogeneous parameterized data flow (HPDF) descriptions (see Section 3.1.3.5) can be (manually) mapped to hardware. In particular, it investigates different tradeoffs between the number of memory banks and the achievable system speed. This is achieved by sharing memories between different communication edges and combining several pixels into one memory word such that the memory bandwidth is exploited optimally. Reference [254] adds a technique in order to take advantage of data parallelism in data flow graphs. To this end, a greedy algorithm selects the most promising actors for duplication. Added switch actors are responsible for correctly routing the data. This book, on the other hand, employs a multidimensional data flow representation. This permits to consider parallel data access of a single individual actor and to investigate resulting tradeoffs between required hardware resources and achievable throughput. Furthermore, it presents a methodology for automatic synthesis of different memory mappings that is able to take out-of-order communication into account.

Reference [106] finally describes how to analyze memory hierarchies with data flow graphs by introduction of special copy actors that transfer data from slow external memory to fast local memory. This primarily bases on the extended CSDF interpretation [85] as discussed in Section 3.1.3.2.

3.4.2 Parallel Data Access

Since typical memories only offer one read and/or write access per clock cycle, parallel access to several data elements requires design of special memory architectures. Reference [114], for instance, describes the mapping of complete arrays to different memories, such that a minimum overall execution time results. Parallel access to one individual array is not possible as the latter one is assumed as monolithic. Corresponding techniques are proposed in [239] by

[13] This kind of loop programs indicate data dependencies instead of being supposed to execute all iterations sequentially.

considering different access patterns occurring in graphical display systems. Reference [295] is restricted to parallel block accesses whose position and size can vary during run-time. Reference [14] considers parallel access to data elements separated by a constant, but arbitrary offset, also called stride. However, none of those publications considers communication in streaming applications. Furthermore, modulo-operations make hardware implementation expensive if they are not powers of 2.

Reference [184] proposes a two-layered memory architecture that provides parallel access to blocks of size $a \times b,\ a, b \in \mathbb{N}$ with arbitrary origin. The first layer consists of an ordinary memory called *linearly addressable memory* while the second layer is addressed by block coordinates and exploits data reuse when accessing overlapping blocks. Based on some simple statistical assumptions the authors show that this memory hierarchy outperforms a one-layer architecture. The applicability of the described technique is limited to two-dimensional applications. Furthermore, it does not consider streaming applications and requires block sizes being powers of 2 in order to be efficient.

3.4.3 Data Reuse

In case an application repeatedly accesses the same data elements, exploitation of data reuse by intermediate buffering in on-chip memory is an efficient means in order to reduce external memory bandwidth. Besides the techniques implemented in the DEFACTO compiler (see Section 3.2.4), Weinhardt and Luk [300] present a simpler approach that is able to exploit reuse in the innermost loop level by introduction of corresponding shift registers that delay the data until it is required again. Reference [7] extends the reuse analysis to the so-called indirectly indexed arrays whose index expressions contain other array variables. Furthermore, transformations for locality improvement are proposed that reduce the amount of required *scratch-pad memory* for temporary buffering. References [141, 144] apply to software synthesis and analyze loop nests in order to determine which parts of an array shall be placed into a *scratch-pad memory (SPM)*. The main idea is to analyze for each loop dimension how many data elements are reused from previous iterations and how many data elements have to be fetched from the global memory. This permits to calculate the number of expensive external memory accesses that can be saved by putting the reused data into the SPM. Furthermore, it is possible to determine how large the latter has to be. Selection of different array references and loop depths results in a Pareto-front of required SPM size and reduced memory accesses.

Besides these rather general approaches, there exist several publications about efficient reuse for standard sliding window algorithms. Reference [199], for instance, presents several architectures ranging from no internal buffering over mixed approaches to full reuse exploitation. Publications [310, 311] introduce the so-called *block buffer method*, which buffers a small rectangular extract of the overall buffer and processes all contained sliding windows. Then it moves to the next block. Depending on the chosen block size, a smaller or larger part of the image pixels have to be read several times from the external memory, thus providing a method to trade throughput against on-chip memory. Border processing or multirate algorithms using up- and downsampling are not further considered. Furthermore, the papers only consider synthesis of individual sliding window modules but not complete image processing systems.

3.4.4 Out-of-Order Communication

As identified in Section 2.2, out-of-order communication is a frequent operation in image processing applications. Whereas most approaches are limited to in-order communication via

signals, FIFO, or buses, this paragraph discusses related work that investigates out-of-order data transfer.

Section 3.2.4 already discussed communication synthesis in the *DEFACTO* behavioral compiler. However, since a communication granularity is only allowed when resulting in in-order communication, this typically results in huge data transfers increasing latency and required buffer sizes. Figure 3.16, for instance, shows the tiling operation occurring in JPEG2000 compression as discussed in Section 2.2. However, since the iteration variables i_1 and i_2 are interchanged on the source and sink side, all three iteration variables are not permitted as communication granularity and only complete images can be communicated. In other words, the source has to terminate a complete image before the sink can start operation. Consequently, in this book a more sophisticated scheduler will be discussed. It allows the sink to start much earlier such that the production of the last line of the first tile ($i_2 = 0$) accompanies its consumption. In other words, both the latency and the required buffer size get significantly smaller.

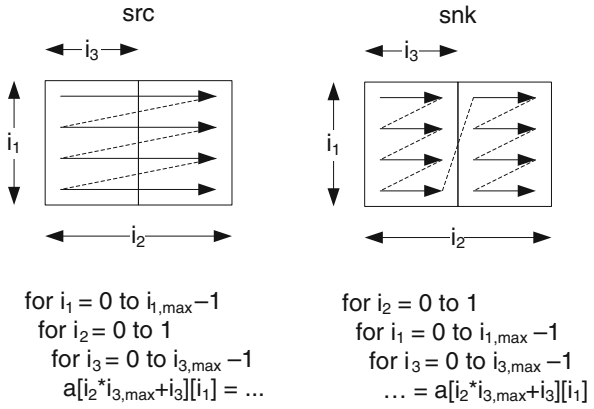


Fig. 3.16 Tiling operation as discussed in Section 2.2. The nested loop pseudocode describes the read and write order depicted by corresponding *arrows*

Reference [287] compares different address generation techniques for out-of-order communication in software channels. Reference [320] proposes a corresponding hardware implementation using a context-addressable memory (CAM). While this is very flexible, it is relatively expensive in terms of required hardware resources and achievable clock frequencies. Furthermore parallel read and write operations are not considered. Reference [286] discusses how out-of-order communication can be detected in nested loop specifications and how the required CAM size can be determined when assuming that each value is read only once. Reference [318] presents an alternative implementation for out-of-order communication using normal RAM together with a valid bit instead of CAM modules. Due to required polling of the valid bit, read and write operations are slow taking three and two clock cycles, respectively. Furthermore, determination of the required memory size is still an open issue.

Reference [69], on the other hand, keeps FIFO communication, but modifies the production order by essentially translating the loop iterations via mapping tables. The authors show that for the matrix multiplication, this leads to reduced latency while achieving the same throughput. This has to be paid with required loop unrolling.

3.4.5 Conclusion

Due to huge amounts of data, communication and memory synthesis play an important role in efficient system level design of image processing applications. As discussed, most related work only considers individual aspects such as memory mapping, parallel data access, or data reuse. In particular, communication control required to avoid destruction of data that is still required or reading of invalid values is rarely taken into account.

In contrast, this book describes a communication primitive for fast and efficient out-of-order communication. Due to the usage of a FIFO-like interface and integration into a system level design method, it significantly helps in designing complex systems by jointly addressing memory mapping, address generation, parallel data access, and out-of-order communication. In particular, it permits to simultaneously perform one read and write operation per clock cycle while supporting parallel data access and high achievable clock frequencies. Automatic synthesis allows trading communication throughput against required hardware resources such that an optimum implementation can be selected.

3.5 System Level Design

As discussed in Chapter 1, system complexity risks to get a major danger for future technical progress. Consequently new design methods are required that permit faster design of powerful hardware–software systems. Section 3.1 already reviewed possible specification techniques, followed by an analysis of different behavioral synthesis tools. Nevertheless, description, analysis, and optimization of complete systems as well as communication generation for image processing applications still remain challenging due to several limitations of available tools and methodologies. Consequently, this section aims to present several approaches that use a system level view in order to perform, for instance, memory analysis, efficient communication synthesis, or system performance evaluation. In particular, three different aspects shall be shortly discussed, namely (i) embedded multi-processor software design, (ii) model-based simulation and design, and (iii) system level synthesis and exploration. A complementary survey can also be found in [86].

3.5.1 Embedded Multi-processor Software Design

Efficient design of complex embedded systems requires both the generation of hardware accelerators and software running on multiple embedded processors. Since creation of energy saving and multi-threaded software applications is very challenging, different tools have been proposed aiming to simplify this task.

3.5.1.1 Omphale

In addition to the data flow-based approaches presented in Section 3.1.3.1, references [43, 44] introduce the *Omphale* design flow for *Multi-processor systems on chip (MPSoCs)*. The application is described by means of a task graph that consists of loop nests with static loop boundaries as depicted in Fig. 3.17a. The bold text corresponds to the *loop body.* Communication takes place by accessing shared arrays.

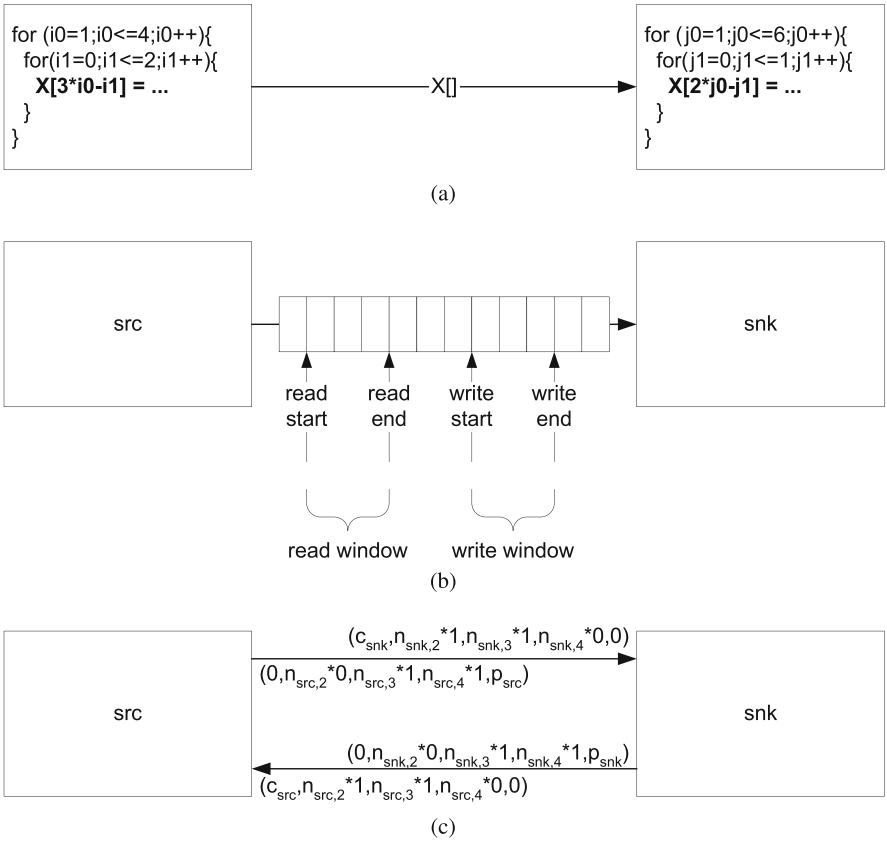


Fig. 3.17 Communication synthesis in Omphale. **a** Task graph, communicating via shared arrays. The *loop body* is printed in *bold*. **b** Mapping to a ring buffer. **c** CSDF model for analysis. Initial tokens are not shown

After mapping such a task graph to multiple processors, communication is implemented by means of circular buffers that permit out-of-order communication. To this end, a corresponding *read* and *write window* is introduced as illustrated in Fig. 3.17b. The source can randomly access all data elements situated in the write window, while the sink is only permitted to access the data elements within the read window. The read and write windows are never allowed to overlap.

In order to control the position and the size of the read and write windows, the tasks have to request and release data elements from the communication channel. By default, each invocation of the task's loop body starts with acquiring one data element on each input and output edge. This means that the end pointers of the read and write windows are moved by one to the right in Fig. 3.17b. In case this would lead to overlapping read and write windows, the task is blocked. Since the communication buffer is organized in form of a ring buffer, a corresponding wraparound is performed when exceeding the end of the buffer. Next, the task reads an arbitrary data element within the read window from each input edge and writes the result values to one data element within the write window. Finally, the task releases one data

element from each input and output buffer. In other words, the corresponding start pointers are moved by one to the right, taking a possible buffer wraparound into account. After these operations, the next invocation of the loop body is possible.

In order to ensure that a task execution only accesses data elements within the read and write windows, these have to be sufficiently large. In order to control their size, a task can deviate from the above-described default behavior: During the first execution, also called *lead-in* phase, a task acquires more than one data element. For an input edge, their number must be sufficiently large such that for all future executions of the loop nest, which only acquire one data element, the accessed data elements do not exceed the end pointer of the read window. The same holds for output edges and the write window. Next, the task enters the so-called *lead-out* phase, where it acquires one data element, but do not release any of them. This phase can consist of several loop body executions. Their number has to be chosen such that for all future invocations, which release one data element, only data elements within the read or write windows are accessed. Then the task enters the default behavior acquiring and releasing one data element per execution. In other words, the size of the read and write windows remains constant while it increases during the lead-in and lead-out phases.

From the above description it gets obvious that the lead-in phase causes the anticipated request of data elements, because more data elements are requested than read or written. Similarly, the lead-out phase delays the release of data elements. Consequently, in order to be consistent both effects have to be undone at the end of the task execution. In other words, when all data elements from the array are acquired, the task enters an inverse lead-in phase during which only data elements are released, but not acquired. After the last execution of the loop body, all remaining data elements are released, before a new task execution cycle can start.

The so-defined execution semantics can be translated into a CSDF graph (see Section 3.1.3.2) as depicted in Fig. 3.17c. During each execution, a CSDF actor reads one data element from the input buffers and writes one data element to the output buffers. Note that from the CSDF model is not visible which data element is read or written exactly. Instead the CSDF edges model the acquisition and release of data elements from the ring buffer. To this end, each communication edge is replaced by two CSDF edges. In the first phase performing the lead-in, each actor a acquires c_a data elements from the input edges. For the source actor of the original task graph edge, this corresponds to the movement of the write window end pointer, while for the sink of the task graph edge, it is associated with the movement of the read window end pointer. Next, each actor a enters the lead-out phase, where it executes $n_{a,2}$ times acquiring one data element, but not releasing anything. Then the actor a fires $n_{a,3}$ times with the default behavior by acquiring and releasing one data element. After $n_{a,4}$ executions without acquiring new data elements, the last phase finally releases all remaining data elements. Note that the phase model can get more complex in case of multiple edges.

The size of the ring buffers can be represented by means of initial tokens not depicted in Fig. 3.17c. A corresponding buffer analysis algorithm for CSDF graphs is available in [302] (see also Section 3.3.1).

Compared to this approach, this book describes a methodology that offers more flexibility in that the actors do not have to consume and produce one token per invocation. Instead, parallel access to several data elements and exact scheduling without lead-in and lead-out phases is possible. This eliminates the risk of a possible deadlock and can improve latency, throughput, and buffer requirements. Furthermore, *Omphale* focus on multi-processor architectures while this book describes a hardware communication primitive allowing for parallel data access. Additionally, the buffer analysis presented in this monograph can directly operate on the

multidimensional representation without needing a conversion to CSDF. As a consequence, more implementation tradeoffs are accessible.

3.5.1.2 ATOMIUM

ATOMIUM [57, 58, 60, 140] stands for "**A** **T**oolbox for **O**ptimizing **M**emory **I**/O **U**sing geometrical **M**odels" and represents a complex tool for buffer size analysis and memory hierarchy optimization of C-programs mapped to MPSoCs in terms of power, performance, and area. For this purpose, it transforms an input specification corresponding to a C-subset called *CleanC* [149] into an optimized output code. Global loop transformations such as loop interchange, reversal, fusion, and shifting or tiling help to reduce memory requirements or cache misses by improving data access locality. Furthermore, code and data parallelism can be revealed for improving performance. Data reuse analysis avoids expensive repetitive access to data elements situated in slow external memories in order to reduce power consumption and improve performance. Frequently accessed data are placed into small, but fast on-chip memory, while the external memories only hold the remaining data. Bandwidth considerations help to determine the number of required memory modules and ports. Buffer size analysis and memory sharing between different arrays allow reducing the actually required storage capacity. A tool called *SPRINT* [67] helps to split the optimized code into several tasks that can be mapped to different processors. Three different communication channels are available, namely (i) simple FIFOs, (ii) block FIFOs, and (iii) shared memory, whereas the latter does not offer any synchronization and is only employed on user request. Block FIFOs communicate on the granularity of arrays and allow random data access.

Compared to this approach, this book puts its focus on synthesis of hardware primitives for fast out-of-order communication with parallel data access. Furthermore, the memory analysis is extended by throughput-optimized scheduling of out-of-order communication and by tradeoff analysis in multirate applications. Finally, system specification uses multidimensional data flow instead of a C-subset.

3.5.2 Model-Based Simulation and Design

Whereas the approaches in the previous section principally started from sequential C code and focused on advanced software synthesis methods, there also exists an important number of model-based design tools. In this case, the application is typically represented by a more or less formal model of computation ranging from continuous time simulations with differential equations and discrete event systems over finite state machines to data flow models of computation.

Simulink [207] is probably one of the best known model-based design environments. Originally intended for continuous time simulations, it has been extended to discrete time applications. In both cases the application is modeled in form of blocks communicating via continuous time signals. In case of discrete applications, each block samples the input signal with a given rate and outputs piece-wise constant signals. Consequently it might happen that output values are read multiple times or not at all. In order to ease application design, Simulink provides a block diagram interface and various predefined blocks together with the support of various data types. Automatic code generation for both DSPs and general-purpose processors as well as its ability to model FPGA applications greatly helps in design of complex embedded systems. On the other hand, the absence of a strict model of computation

makes automatic system analysis (apart from simulation) very challenging. Furthermore, although Simulink channels are able to carry multidimensional arrays, derivation of an efficient hardware implementation still requires lots of user intervention in form of serialization, data buffering, and memory size determination. Reference [129] proposes an alternative software synthesis flow for Simulink designs. Elimination of needless copy operations and buffer sharing together with memory-optimized schedules leads to a significant reduction of required memory. Hardware generation, on the other hand, is not in the focus of this chapter. *National Instruments LabVIEW* FPGA [218] offers similar concepts to Simulink, but operates on lower levels of abstraction.

Ptolemy [49] focuses on efficient modeling and simulation of embedded systems by supporting interaction of different models of computation including CSP, continuous time applications, discrete event semantics, synchronous-reactive systems, or data flow (see Section 3.1). This permits a designer to represent each subsystem of an application in the most natural model of computation, also called *domain*. For this purpose hierarchical graphs can be constructed in order to embed one model of computation into another one. In order to avoid redundant coding, an actor or module can be reused in different domains. An actor language called *CAL* [98, 99] has been proposed for high-level design of actors that are controlled by a finite state machine. While there exist first code generators for both C and VHDL, corresponding support for multidimensional models of computation or buffer analysis is not available. *PeaCE* [124, 125, 175] sets on top of Ptolemy and provides a seamless codesign flow from functional simulation to system synthesis. Moreover, *PeaCE* supports the generation of an FPGA prototype from SDF graphs. Support of flexible finite state machines permits the specification of control flow like switching between different applications running on the same embedded system. Data processing can be expressed by piggy backed synchronous data flow (see Section 3.1.3.1) or fractional rate data flow (see Section 3.1.3.3). However, no support for multidimensional models of computation and thus out-of-order communication and parallel data access is available. The same holds for *Grape-II* [187], a system level development environment for specifying, compiling, debugging, simulating, and emulating digital signal processing applications. There, applications are described as CSDF graphs (see Section 3.1.3.2), and actors have to be coded in C or VHDL. This code is directly used for system implementation, which supports commercial DSP processors or FPGAs. Tools for resource estimation, partitioning, routing, and scheduling ease design of complex systems. *Polis* [18] finally uses a finite state machine-like representation for hardware–software codesign. It incorporates a formal verification methodology and supports hardware synthesis, application-specific real-time operating system synthesis, and software synthesis in form of C code. However, multidimensional signal processing is not considered further.

After the presentation of these rather general model-based simulation and design tools, the following two sections will detail image processing centric approaches as well as research focusing on multidimensional signal processing.

3.5.2.1 Image Processing Centric Approaches

Image processing applications are typically represented in form of processing chains consisting of several blocks performing pixel manipulations. Since this is perfectly suited for parallel implementation, several approaches exist that aim to map this kind of applications to FPGAs or multiple processors.

References [33, 77], for instance, provide a C++ front-end for design of local image processing algorithms for FPGAs. For this purpose, a library has been created

that includes a general local algorithm core, frequent point and global algorithms, as well as components for arithmetic combination of images. While this template-based approach simplifies the system level design task, it limits the calculations that can be performed. Furthermore, only two-dimensional applications can be represented and out-of-order communication is not considered. Reference [217] describes the *Champion* project. It addresses the mapping of image processing applications described in *Khoros*, a graphical programming environment from Khoral Research, onto an FPGA board. In particular the authors consider automatic partitioning of presynthesized blocks to reconfigurable FPGAs. Generation of these modules can be simplified by means of SA-C [128] (see also Section 3.2.2). On the other hand, buffer analysis or out-of-order communication as well as modeling by data flow is not considered. Furthermore, *Khoros* seems to be discontinued.

References [13, 161, 219, 221] present the *model-integrated computing (MIC)* methodology. It focuses on the formal representation, composition, analysis, and manipulation of models during the design process. Corresponding domain-specific modeling languages can be defined by means of meta-models. This allows describing both the application in consideration and the corresponding target architecture as well as possible mapping possibilities. In previous work, this paradigm has been applied to synthesis of multi-processor image processing applications [210, 220]. The application is defined in form of an SDF graph, which is refined by a second layer in order to express data dependencies on pixel granularity (see also Section 3.1.3.1). This information permits to automatically split a computationally intensive operation into several processes running on different processors. However, only graphs with exactly one source and one sink are considered. Furthermore, hardware synthesis is not part of this research.

IMEM finally is a tool for the design of image processing applications consisting of sliding window algorithms only. It provides a *SystemC* library that is able to describe sliding windows with up to three dimensions [280]. These windows do not need to be rectangular but can have more complex forms. Based on these models, IMEM is able to generate highly optimized buffer structures and address generators [190] for Xilinx FPGAs. Simultaneous read and write operations over the same port permit two sliding window algorithms to be mapped to the same BRAM [229]. The solution of an ILP or a corresponding heuristic steers this mapping such that the overall required number of BRAMs is minimized [188, 225, 281]. Usage of chip-external RAM permits the creation of huge three-dimensional sliding windows. Polyhedral buffer analysis helps to determine the required buffer size for applications with more complex topologies than simple chains [279, 282] (see also page 66). Combination with a behavioral synthesis tool enables quick generation of the filter kernels, which can be connected to the optimized storage structures offering parallel data access [189]. Generation of placement constraints can reduce the required dynamic energy consumption [191]. However, in contrast to the approach discussed in this book, *IMEM* is limited to simple sliding windows and considers neither out-of-order communication nor multirate systems with up- and downsamplers. Furthermore, border processing is only implicitly specified making it less general than the model of computation proposed in this monograph. Additionally, *IMEM* is not included into a data flow environment making integration of data-dependent algorithms like entropy encoding more difficult. This is also true because the generated buffer structures use a feed-forward structure where the sink always has to be ready for processing the delivered data. The communication primitive described in this book, on the other hand, uses FIFO semantics making it possible for the data sink to stop operation if necessary.

3.5.2.2 System Level Design Tools for Multidimensional Signal Processing

Whereas the previous approaches in principle provide more or less flexible primitives for direct mapping of image processing applications to FPGAs, there also exist more general solutions. They start from a multidimensional application specification, which is then analyzed and translated into a hardware–software implementation. This section exemplarily aims to introduce two corresponding design environments, namely the *Gaspard2* and *Daedalus* frameworks.

Gaspard2 [83] is an UML-based system level design tool for high-performance applications described in *Array-OL* (see Section 3.1.4.3). As a consequence, it is particularly adapted for regular iterative applications like image filtering or matrix multiplication.

The system specification consists of three parts, namely (i) application modeling, (ii) architecture modeling and (iii) mapping of tasks to hardware resources. The corresponding modeling semantics are defined by the so-called *MARTE* profile, which defines a *meta-model* describing the properties that an application model must obey to. In particular, it enables the description of parallel computations in the application software part, the parallel structure of its hardware architecture part, and the association of both parts. Furthermore, Gaspard2-specific extensions are provided in order to support for instance IP-core-based hardware synthesis.

Several transformations have been developed in order to automatically transform the input specification into different target languages, such as Lustre, Signal, OpenMP C, SystemC, or VHDL. The mapping specification is such that repetitive tasks can be distributed to multiple resources of a regular architecture [48, 242]. Implementation typically requires to manually provide the code for the elementary tasks (see Section 3.1.4.3) and the target technology [110].

Several publications [36–38] consider synthesis of hardware accelerators in FPGAs. The corresponding refinement is only possible if the Array-OL specification does not contain control in form of mode-automata (see Section 3.1.4.3). In case of fine-grained UML specifications, it is possible to generate RTL code without needing manually designed IP cores. Automatic design space exploration determines the achievable parallelism that still fits into the FPGA. For this purpose, different model types are provided. The *black box view* does not consider any details of the FPGA and is only used when a system with more than one FPGA or other heterogeneous components shall be represented. The *quantitative view* enumerates the available resources like BRAMs and DSPs and helps in finding the optimum parallelization of the considered algorithm. The *physical view* finally also takes routing resources into account. It is used for optimized mapping by taking the regular data dependencies of the Array-OL specification into account. As a result, a constraint file is generated, which can be used by commercial place-and-route tools.

Figure 3.18 illustrates the results of communication synthesis for a 3×3 sliding window on a 4×4 input image. The input tiler reads a complete image and divides it into several sliding windows by help of combinatorial logic. Then, all possible sliding windows are calculated in parallel and combined to the overall image. If this high degree of parallelism is not possible due to resource constraints, the different sliding windows are serialized as shown in Fig. 3.18b. This, however, means that the following modules reading the result image have to wait until the complete array has been produced [36]. As a consequence, both alternatives are expensive. Furthermore, no border processing is taken into account. Additionally, *Gaspard2* does neither consider cooperation with one-dimensional data flow models of computation nor buffer analysis.

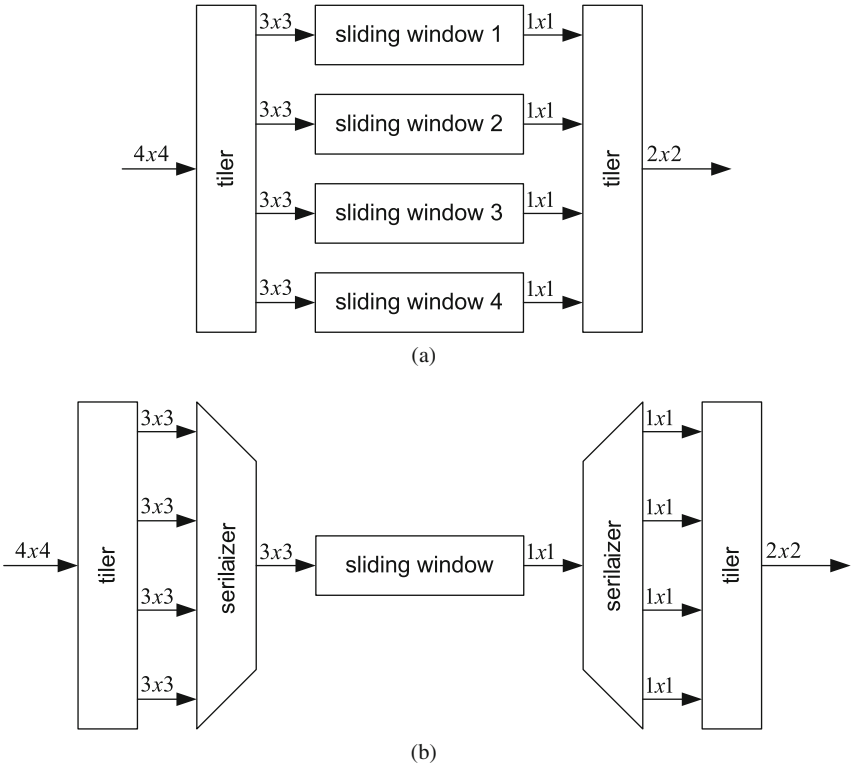


Fig. 3.18 Communication synthesis in Gaspard2 [36]. **a** Parallel execution of repetitive tasks. **b** Sequential execution of repetitive tasks

Daedalus is a framework for modeling, exploration, and synthesis of applications processing multidimensional arrays [278]. Applications are modeled by means of a top-level C or Matlab program consisting of static affine nested loops [174, 249, 262]. If not otherwise specified by the user, each called function is mapped to a process as exemplarily depicted in Fig. 3.19. Each process has an associated iteration domain, which can be derived from the enclosing loop boundaries. In Fig. 3.19, this is represented by a set of dots each corresponding to an invocation of the process. As can be seen, the iteration domain need not to be rectangular. Each invocation of a process reads one or more data items as specified in the loop program and generates corresponding output values. An ILP-based dependency analysis determines for all input data the producing source invocation. Based on this information, the process can be translated into a CSDF-like data flow actor [173], having several input and output ports. An internal finite state machine takes care that each process invocation reads the input data from the correct source processes and writes the results to the correct output. In Fig. 3.19, for instance, the invocations of process P3 can be partitioned into sets reading from different source processes leading to the depicted finite state machine. Furthermore, an actor has one or more associated functions processing the data. Additionally, the dependency analysis can be used to classify each communication channel into two different categories [285], namely (i) in-order and (ii) out-of-order communication. Furthermore, it is distinguished whether data

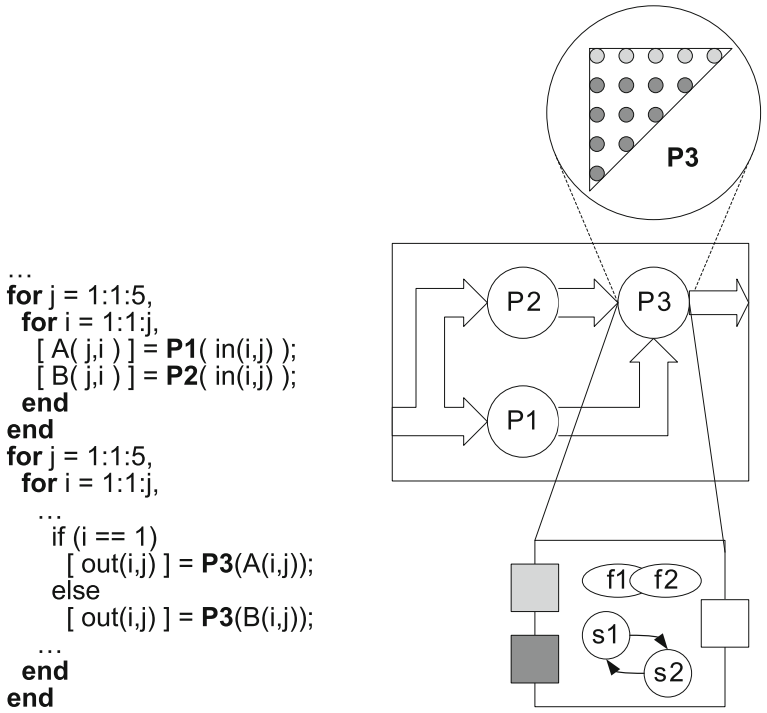


Fig. 3.19 Translation of a static affine nested loop program (SANLP) into a Kahn process network (KPN). **a** SANLP. **b** Corresponding KPN. If not specified otherwise, each process of the KPN graph represents a function found in the nested loop program. The resulting number of executions per process can be derived from the iteration domain represented by a set of circles in the above illustration. Each circle corresponds to a function call and thus to an invocation of the corresponding process. In order to synthesize the processes, these invocations are partitioned into sets showing the same input and output behavior as illustrated by different shades in the above illustration. A finite state machine controls which set is currently active in order to assure the desired communication behavior and in order to call the correct process functions f_i

items are read multiple times or not. As out-of-order communication or multiple reads require expensive communication channels, the latter are avoided when possible by exploiting data reuse.

The so-generated data flow graph can now be mapped onto a corresponding hardware–software platform. In case of hardware acceleration, *Daedalus* can automatically generate the corresponding communication infrastructure [319] whereas the functions have to be provided in form of IP cores. They can be either generated manually or by using PICO Express [126] (see also Section 3.2.5). The FIFO sizes are determined by automatic buffer analysis performed on the original loop program [292] as described in Section 3.3.2 on page 62. Furthermore, the data flow graph can be mapped to a multi-processor system consisting of several *Xilinx MicroBlaze* processors. Different communication channels can be instantiated encompassing cross-bar, shared-bus, and point-to-point communication [222, 223]. A design space exploration tool called Sesame [101] helps to select the best mapping alternatives.

For this purpose, the Kahn processes have to be instrumented in order to generate the so-called events. They correspond to communication requests or required computation and are processed by the target architecture. By this means, it is possible to obtain estimates for system performance, component utilization, occurring contention and critical path analysis. For this purpose, it is necessary to annotate the occurring events with information like consumed energy or required execution time. These values can be obtained by running an instruction set simulator either during exploration or before hand [243]. Moreover, different loop transformations such as loop unrolling permit to derive different process networks with different performance characteristics from the same static affine nested loop program [261].

Compared to Daedalus, the remainder of this monograph differs in several aspects. First of all, Daedalus uses static affine nested loop programs as input specification, whereas this book opts for a multidimensional model of computation. It includes the capability to express data-dependent decisions, and it can tightly interact with one-dimensional models of computation in order to cover application parts not working on arrays. In particular Section 4.2.2 demonstrates how the resulting information can be advantageously used during automatic design space exploration. In contrast, Deadalus requires to hide this kind of algorithm behind an opaque function that is not visible for system analysis. Moreover, since the multidimensional data flow model of computation directly includes the sampling with overlapping windows, their handling gets easier. For the Daedalus framework, on the other hand, the input code has to be designed in a specific way in order to obtain efficient memory structures [292].

Besides the different approaches employed during buffer analysis as already discussed in Section 62, communication synthesis is another source of distinction. Whereas this book describes a hardware communication primitive for efficient out-of-order communication, Daedalus tries to transform the input specification such that it can be handled with normal FIFO communication. If this is not possible, expensive context-addressable memory is used. Alternatively, also a static memory allocation is available, which is, however, still slower than the multidimensional FIFO discussed in this monograph (see also Section 3.4.4). Furthermore, Daedalus does not consider the tradeoff between communication throughput and required hardware resources.

3.5.3 System Level Mapping and Exploration

In addition to *Daedalus* described in Section 3.5.2.2, there exists a huge number of tools that also consider mapping of applications to complex platforms including point-to-point and bus communication, multiple processors, and hardware accelerators. In contrast to Daedalus, however, they do not employ a multidimensional model of communication and are thus described in a separate section.

CoFluent Studio [55] allows modeling complex applications by executable block diagrams. Annotation of timing information helps to perform temporal simulation in order to estimate achievable throughput in the final implementation. This application specification can then be mapped to an architecture consisting of hardware accelerators, micro-processors, and nodes for data transport like shared memory. Simulations return different statistics like response times or processor utilization. However, as *CoFluent Studio* does not use any underlying model of computation, system analysis different than simulation is difficult if not impossible. Furthermore, timing annotation is complex because due to data-dependent operations, corresponding worst-case execution times have to be derived manually. Additionally,

CoFluent Studio does not provide any synthesis path and the design space exploration has to be performed manually. *CoWare Platform Architect* [74] focuses on the design of processor-centric, software-intensive product platforms. SystemC TLM simulations enable high-level verification of the developed system. However, similar to *CoFluent Studio,* system analysis apart from simulation is difficult.

The *system-on-chip environment (SCE)* [90] provides tools and libraries in order to refine an application described in SystemC on a high level of abstraction successively into a final hardware–software implementation. *Synopsys System Studio* [269] can be used to model, simulate, and synthesize signal processing applications using data flow semantics including single rate, multirate, and dynamic data flow together with extended finite state machines. Support of fixed-point arithmetic helps to perform the transition from floating point specification to the final implementation on a DSP or FPGA. Multi-processor support can significantly improve simulation speed. However, multidimensional models of computation are not supported.

Koski [158] is also dedicated to SoC design and supports automatic design space exploration and synthesis. The input specification is given as *Kahn process network* modeled in *UML*. The Kahn processes are refined using Statecharts. The target architecture consists of the application software, the platform-dependent and platform-independent software, and synthesizable communication and processing resources. Moreover, special functions for application distribution are included performing inter-process communication for multi-processor systems. During design space exploration, Koski uses simulation for performance evaluation. In [172], Kianzad and Bhattacharyya propose a framework called *CHARMED* (Co-synthesis of HARdware-software Multi-mode EmbeddeD systems) for the automatic design space exploration of periodic multi-mode embedded systems. The input specification is given by several task graphs where each task graph is associated with one of M modes. Moreover, a period for each task graph is given. Associated with the vertices and edges in each task graph, there are attributes like memory requirement and worst-case execution time.

A quite different approach is proposed in form of the *Metropolis* framework [19, 82]. It is an environment for *platform-based* design of complex systems. These are composed of several predefined components like processors, buses, or available hardware accelerators. Support of stepwise refinement using different levels of abstraction aims to facilitate design reuse and to provide analysis and verification capabilities by usage of formal semantics. For this purpose, a corresponding design language called *Metropolis meta-model* is proposed, which does not only cover functional behavior and architecture description, but also constraint that guide later synthesis or verification. Execution of the application on a given target architecture results in a sequence of events that are processed by the so-called *quantity managers*. They annotate each occurred event by different quantities such as required energy, execution time, or invocation time. This information can be used both for scheduling and performance evaluation. Currently, Metropolis offers an interface not only for a SystemC-based simulator but also for a model-checker, behavioral synthesis, and design space exploration.

SPARCS [233] finally targets synthesis and partitioning for adaptive reconfigurable computing systems consisting of multiple FPGAs and memory modules. It accepts design specifications at the behavior level in form of task graphs, where each task is specified in VHDL. Based on this information, the *SPARCS* system automates the process of mapping (i) task computations to the FPGA resources, (ii) abstract memories to physical memories, and (iii) data flow to the interconnection fabric. This is accomplished while attempting to meet constraints on the clock-speed and on the number of available clock cycles.

However, none of the above-mentioned approaches considers multidimensional modeling and the resulting challenges like out-of-order communication or memory analysis.

3.6 Conclusion

This chapter has given an encompassing overview on system level design of image processing applications. In accordance with the main focus of this book, four major aspects have been investigated in more detail, namely (i) application modeling; (ii) behavioral synthesis of computation, memory structures, and communication; (iii) system level analysis for automatic buffer size determination; and (iv) existing frameworks for system level design.

To sum up, it can be said that behavioral synthesis tools like *PICO Express*, *Forte Cynthesizer*, or *DEFACTO* promise to simplify design of individual modules. While starting from general *C* or *Matlab* code offers the advantage of being able to handle a great variety of existing code, the absence of clear models of computation makes analysis beyond module frontiers very complex if not impossible. The same problems can be found for system level design tools setting on top of general languages like *SystemC* or *UML*.

Consequently, Section 3.1 has reviewed a huge variety of specification methods. In particular, they have been examined according to their capacity to represent important properties of image processing algorithms, such as overlapping windows, parallel data access, out-of-order communication, control flow, and data dependencies. Whereas in principle data flow models offer the most advantages in representing data-dominant applications, image processing algorithms operate on multidimensional arrays making their representation in one-dimensional models difficult or incomplete. In order to overcome these drawbacks, multidimensional specifications are interesting alternatives, because they offer a certain affinity to loop specifications. Among others, this opens the potential for array-based buffer analysis whose usefulness has been shown by various publications.

Astonishingly, multidimensional data flow lives still quite in the shadows. The amount of available models of computation is very restricted and many of them cannot cover image processing applications in their entirety, including overlapping windows, virtual border extension, control flow, and out-of-order communication. Fortunately, the situation seems to change, since the research community pays more and more attention toward multidimensional electronic system design level design. *Gaspard2,* for instance, uses multidimensional modeling and demonstrates how this can be combined with UML representation and presence of control flow. Available model transformations and an FPGA synthesis path target system optimization and implementation. *IMEM*, on the other hand, uses a more restricted model of computation excluding, for instance, out-of-order communication, but offers efficient communication synthesis and buffer analysis. *Daedalus* finally starts from static loop programs and automatically translates them into a data flow process network that can be mapped onto embedded multi-processor systems.

In other words, multidimensional system level design seems to be a promising methodology in order to solve some of the challenges occurring nowadays in design of embedded systems. Consequently, the remainder of this book aims to discuss in more detail a corresponding system level design flow including both multidimensional and one-dimensional application modeling. A finite state machine permits to express control flow, if necessary. As a consequence, it can be applied to applications containing both static and data-dependent algorithms which work on multidimensional arrays or on one-dimensional streams of data. Moreover, it offers the possibility for powerful polyhedral analysis in order to determine efficient schedules or buffer sizes. Whereas various publications exist in this domain, their application is very complex, because typically multiple mathematical conditions have to be fulfilled. Consequently, this book describes and compares two different analysis methods. Whereas simulation comes out to be more flexible, it delivers sub-optimal results in general and is

rather slow. Consequently, an alternative polyhedral method will be discussed which is able to handle complex multidimensional data flow graphs including feedback loops, out-of-order communication, multirate applications, and throughput-optimized schedules. In addition, the information contained in the multidimensional representation can be advantageously used for high-speed communication synthesis exceeding the capacities of known related approaches. Integration into a system level design tool called SYSTEMCODESIGNER enables high-level performance evaluation and automatic design space exploration. By these means, design of complex image processing applications can be simplified significantly.

Chapter 4
Electronic System Level Design of Image Processing Applications with SystemCoDesigner

As already discussed in Chapter 2 of this book, today's image processing systems are getting more and more complex. Consequently, new methodologies are required for implementation of hardware–software systems in order to raise the level of abstraction above the currently used RTL or C coding techniques. Section 2.5 has identified several requirements on such a new design flow like, for instance, efficient representation of point, local, or global algorithms, the capacity to express different kinds of parallelism as well as control flow, and the support for out-of-order communication, high-level verification, simulation on different levels of abstraction, or high-level performance evaluation.

In this context the major focus of this book has been put onto modeling, analysis, and hardware synthesis of multidimensional algorithms. Since, however, complex applications also contain algorithm parts which are better described by one-dimensional data flow, the discussed approaches for multidimensional modeling, analysis, and synthesis have to be compatible with corresponding one-dimensional ESL design tools and techniques. To this end, this chapter exemplarily summarizes the *electronic system level (ESL)* design tool SystemCoDesigner [134, 170]. It addresses modeling, simulation, optimization, and synthesis of embedded hardware–software systems and will later be used to demonstrate how multidimensional design techniques can coexist with one-dimensional ESL design methodologies.

In more detail, Section 4.1 gives an overview on the corresponding design flow. Section 4.2 presents its application to a Motion-JPEG decoder in order to demonstrate how SystemCoDesigner is able to solve the requirements identified in Section 2.5. In this context, the Motion-JPEG decoder has been preferred against the JPEG2000 encoder described in Section 2.2 because it shows the same fundamental challenges, but is less complex to implement. Furthermore, it does not require sliding window algorithms, which, as discussed in Section 3.1.3, are difficult to represent by one-dimensional models of computation assumed in this section. Section 4.3 finally evaluates the obtained results. Based on the performed observations, it furthermore explains the need for multidimensional modeling, analysis, and synthesis as introduced in the following chapters of this monograph.

4.1 Design Flow

The overall design flow of the SystemCoDesigner is based on (i) actor-oriented modeling in SystemC; (ii) hardware generation for some or all actors using behavioral synthesis; (iii) determination of their performance parameters such as required hardware resources, throughput, and latency; (iv) design space exploration for finding the best candidate architectures;

J. Keinert, J. Teich, *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*, Embedded Systems,
DOI 10.1007/978-1-4419-7182-1_4, © Springer Science+Business Media, LLC 2011

and (v) automatic platform synthesis. Figure 4.1 depicts the cooperation of these steps as implemented in SystemCoDesigner.

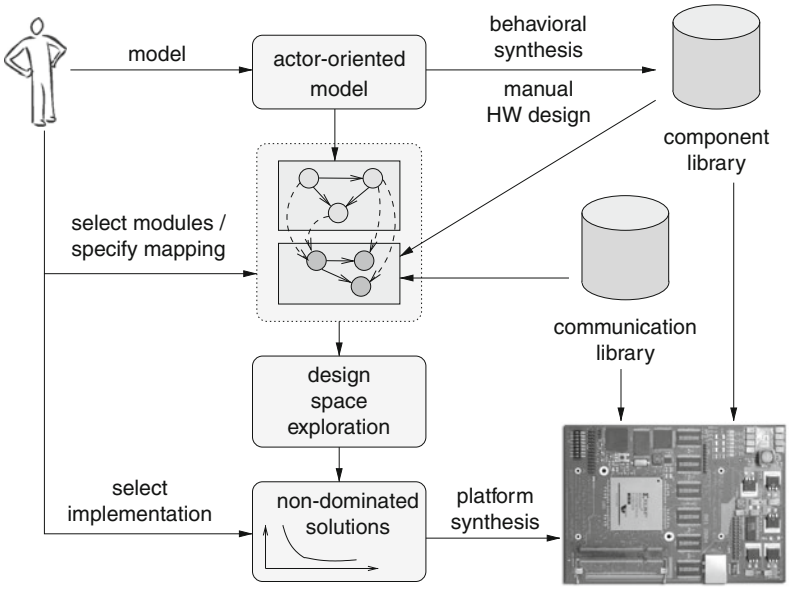


Fig. 4.1 ESL design flow using SystemCoDesigner: The application is given by an actor-oriented model described in SystemC. Behavioral synthesis or manual RTL code generation is used to create *architecture templates*. Design space exploration using *multi-objective evolutionary algorithms* automatically searches for the best architecture candidates. The entire SoC is implemented automatically for FPGA-based platforms

4.1.1 Actor-Oriented Model

The first step in the proposed ESL design flow is to describe the application by an *actor-oriented data flow model*. For this purpose, the functionality of the application is split into several *processes*, or *actors,* which communicate over dedicated communication *channels* transporting streams of *tokens*. Each actor reads some input data and transforms them into a corresponding output stream, which can be processed by a successive actor. In this book, two different channel types are considered, namely the so-called SysteMoC-*FIFO* and a new *multidimensional FIFO*. The multidimensional FIFO is able to handle sliding windows, out-of-order communication, and parallel access to several data elements and will be further discussed in Chapter 5. The SysteMoC-FIFO on the other hand offers the classical one-dimensional FIFO functionality extended by a (restricted) random access possibility.

Figure 4.2 shows the actor-oriented model of a Motion-JPEG decoder [153], consisting of processes for codestream parsing, entropy-decoding (*Huffman Decoder*, *Inverse ZRL*, *DC Decoder*), pixel decorrelation (*Inverse Quant*, *Inverse ZigZag*, *IDCT*, *YCbCr Decoder*), and data reordering (*Frame-Shuffler*).

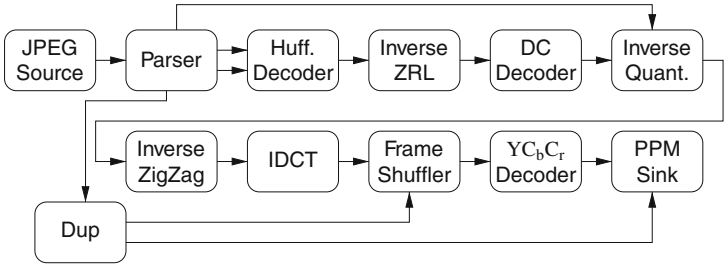


Fig. 4.2 Actor-oriented model of a Motion-JPEG decoder, transforming a Motion-JPEG data stream into uncompressed images. Each block corresponds to a task performing a particular operation. Communication is illustrated by edges

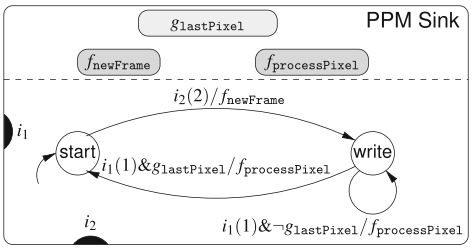
4.1.2 Actor Specification

The actor specification itself uses a special subset of SystemC, defined by the SYSTEMOC library [102]. The latter provides constructs to describe actors with well-defined communication behavior and functionality. For this purpose, each actor contains a finite state machine, also called *communication finite state machine*. Each transition between two states is annotated with a predicate called *activation pattern.* It evaluates whether enough input data and output space are available on the input and output channels. Furthermore, data-dependent *guards* permit to check for given token values or state variables of the actor. If all conditions for an outgoing transition of the current state evaluate to true, the transition is taken by executing a given function, called *action.*

Figure 4.3 illustrates a corresponding example actor, which belongs to the Motion-JPEG decoder shown in Fig. 4.2. Each transition of the communication finite state machine is annotated by an activation pattern and an action to execute. $i_2\,(2)$, for instance, requests the existence of at least two tokens on the channel connected to input *port* i_2. $g_{\texttt{lastPixel}}$ is a data-dependent guard in form of a function and verifies the value of the actor-internal variable `missingPixels`. This variable is set by the action `newFrame`, which is depicted in Fig. 4.3, and which is called when performing the corresponding communication finite state machine transitions. Access to the channel values is possible via the actor *ports* as exemplified in lines (2) and (3). The specified offset in brackets supports a restricted out-of-order communication. `i2[0]`, for instance, reads the next token of the connected input channel while `i2[1]` accesses the successor token. For well-defined behavior, the offset must never exceed the number of tokens requested by the activation pattern of the communication finite state machine.

4.1.3 Actor and Communication Synthesis

Each actor described in SYSTEMOC can then be transformed into both hardware and software modules. Whereas the latter is achieved by simple code transformations, the hardware modules are built by means of *Cynthesizer* [107], a commercial behavioral synthesis tool. This allows SYSTEMCODESIGNER to quickly extract important performance parameters such as throughput and required hardware resources in form of flip-flops, lookup tables, and




```
void newFrame(){                    1
 size_t Height = i2[1];             2
 size_t Width = i2[0];              3
 missingPixels =                    4
  Width * Height;                   5
}                                   6
```

Fig. 4.3 Depiction of the *PPM Sink* actor from Fig. 4.2 along with the source code of the action $f_{\texttt{newFrame}}$. The transitions of the state machine are annotated with *activation pattern/action* pairs

multipliers. These values can be used to evaluate different solutions found during automatic design space exploration. Alternatively, the user can manually create VHDL code for selected actors or use special high-level synthesis tools like *PARO* (see Section 3.2.7) in order to generate optimized implementations.

Similarly, hardware implementations for the different communication channels have to be provided. For one-dimensional data exchange via SystemoC-FIFOs, they can be simply taken from a communication library. Multidimensional communication on the other hand requires sophisticated analysis techniques and allows exploiting tradeoffs between required hardware resources and achievable throughput. Further details will be discussed in Chapter 8.

4.1.4 Automatic Design Space Exploration

As a result of the above synthesis steps, several implementations are available for both the actors and the communication channels, which can be characterized in terms of throughput, latency, or chip size. An automatic *design space exploration (DSE)* is then able to select those components that lead to an optimal or near-optimal overall system. The fewer the components are allocated, the less the hardware resources are required. In general, however, this comes along with a reduction in throughput, since the instantiated processors have to execute more and more functionality. As a consequence, this leads to tradeoffs between execution speed and implementation costs. The same is typically true for the choice between huge-sized and fast IP Cores, or slow, but less resource intensive ones.

Multi-objective optimization, together with symbolic optimization techniques [135, 251] are used to find a set of *nondominated* solutions. Each of these has the property to be not dominated by other known solutions in all objectives. For example, the design space exploration might return two solutions showing a typical tradeoff (e.g., throughput vs. flip-flop count). One solution has good performance and a huge effort in area. Another solution may use less area and achieves only lower performance. This means that both solutions do not dominate each other, hence the user may want to select the one serving his needs best, either the faster one or the cheaper one.

In order to permit an automatic design space exploration, the designer has to create a so-called *architecture template* that specifies all possible hardware modules and processor cores as well as their interconnection. Corresponding tool support helps to accelerate this process. The task of the automatic design space exploration then consists of selecting a subset of the hardware resources and mapping both the actors and the communication channels on those resources such that the resulting solution is Pareto-optimal.

Figure 4.4 shows an extract of an architecture template that belongs to the Motion-JPEG decoder depicted in Fig. 4.2. The information which actor and communication channel can be mapped to which hardware resource is carried by the so-called *mapping edges*. For each possible implementation alternative of an actor or a channel, a mapping edge is introduced connecting the SystemoC model with the corresponding hardware resource. In Fig. 4.4, for instance, the *Huffman Decoder* can be executed on a microprocessor or a dedicated IP Core. In order to enable automatic performance evaluation, each mapping edge is annotated with the action execution times of a SystemoC actor or FIFO when mapped to the hardware resource identified by the mapping edge. The action-accurate delay annotation improves the throughput and latency estimations when the execution times for the different actions, and hence actor invocations, are not identical. Furthermore, it permits different execution times for identical actions when mapped to different hardware resources. The hardware sizes on the other hand are directly associated to the resources of the architecture model.

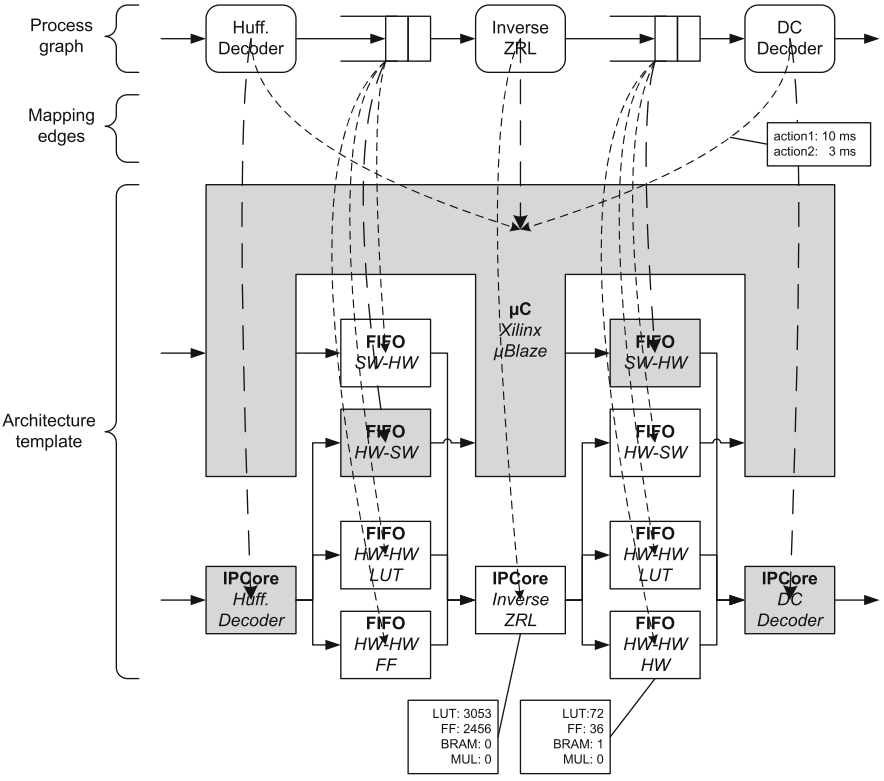


Fig. 4.4 Extract of the architecture template used for the Motion-JPEG decoder. Due to illustration purposes, SW–SW communication channels are not depicted. Furthermore performance annotations are only printed for selected resources and mapping edges. *Bold mapping edges* and *gray-shaded hardware resources* correspond to one possible implementation alternative

Based on this information, a possible implementation of the application specified by the SystemoC graph can be evaluated in terms of chip size, throughput, and latency. The latter two are obtained by simulation of the application using system level performance models,

which take the binding to hardware modules and processor cores into account. This is established by the *virtual processing components* (VPC) framework [266], which simulates a particular architecture modeled in SystemC in order to derive average case estimations for latency and throughput. As a consequence, SystemCoDesigner is able to select optimized solutions, from which the designer can choose the most promising ones.

4.1.5 System Building

After this decision has been taken, the last step of SystemCoDesigner design flow is the automatic generation of the corresponding FPGA-based SoC implementation in order to enable rapid prototyping. Due to the employed actor-oriented application modeling, complex optimized SoCs can be assembled by interconnecting the hardware IP Cores representing actors and potential processor cores with special communication modules provided in a corresponding library. Furthermore, the program code for each microprocessor is generated. Finally, the entire SoC platform is compiled into an FPGA bit stream using, for example, the *Xilinx Embedded Development Kit (EDK)* [307] tool chain for Xilinx FPGAs.

4.1.6 Extensions

In addition to this basic design flow, several extensions and optimizations are available. Falk et al. [103], for instance, propose a method that targets performance improvements of software implementations obtained from SystemCoDesigner application models. To this end, static actors can be combined to so-called clusters that can be scheduled statically. Reimann et al.'s study [246] is situated in the context of reliable systems and investigates how to place so-called *voter* modules, which are able to detect system faults. This makes SystemCoDesigner also applicable to future systems where due to smallest semiconductor structures chips cannot be supposed anymore to be fully reliable.

4.2 Case Study for the Motion-JPEG Decoder

After having described the overall design flow of SystemCoDesigner, this section presents and discusses the quality of results obtained for a Motion-JPEG Decoder described in SysteMoC. At this stage, only one-dimensional communication in form of the SysteMoC FIFOs has been used. A Xilinx Virtex II FPGA (XC2V6000) has been selected as target platform for the implementations running at a clock frequency of 50 MHz. The objectives taken into account during design space exploration have been (i) throughput, (ii) latency, (iii) number of required flip-flops, (iv) lookup tables, and (v) block RAMs resp. multipliers.[1] All hardware accelerators have been obtained via behavioral synthesis with Forte Cynthesizer.

Table 4.1 illustrates the development effort spent for realization of the complete Motion-JPEG decoder and its different hardware–software implementations. The first item

[1] In Xilinx Virtex-II FPGAs, hardware multipliers and BRAMs partially share communication resources and can hence not be allocated independently. Consequently, they are combined to one exploration objective.

represents the activity of breaking the JPEG standard [153] down into a SystemoC graph of actors. Module implementation encompasses the encoding of the actor's communication finite state machine as well as the corresponding actions and guards. During the integration phase, the actors have been connected to the complete system. Thanks to the actor-oriented methodology, this step could be accomplished very quickly, since the interface specification is relatively simple due to the applied FIFO communication semantics. Then, debugging has been required in order to detect errors in both the specification and the module implementations. Finally, coding constructs not supported by the behavioral synthesis tool have been identified and replaced by equivalent instructions. Additionally, although not mandatory for obtaining a working system, the performance of selected actors has been improved by means of synthesis constraints. The YCbCr conversion, for instance, could be accelerated by means of loop unrolling.[2] Thanks to the integration of Forte Cynthesizer, the hardware accelerators for the different actors could be obtained directly from the SystemoC specification. Hence, whereas traditional system design mostly requires the creation of both a so-called golden model and the corresponding RTL implementation, behavioral synthesis helps the designer to avoid this redundant effort. Furthermore, since SystemoC offers a higher level of abstraction compared to RTL, the designer can progress more quickly. Taking the number of lines of code as a measure for complexity, a RTL design would have been 8–10 times more costly than the SystemoC specification. The latter can be translated into a huge number of different hardware–software systems, which would not have been possible when directly recurring to RTL implementation.

4.2.1 Comparison Between VPC Estimates and Real Implementation

Table 4.2 shows the properties of some solutions found by design space exploration. The architecture template includes one MicroBlaze processor, one hardware accelerator per SystemoC actor, and different communication channels. Consequently, this enables hardware-only, software-only, and mixed hardware–software alternatives. Latency and throughput are VPC simulation estimates for JPEG streams consisting of four QCIF images (176×144 pixels). Table 4.3 provides the performance values of the corresponding hardware implementations in order to give an idea about the achievable accuracy of the VPC estimations. Whereas it is not possible to give an upper bound of the occurring discrepancies, because the objectives used during design space exploration are nonmonotonic in the

Table 4.1 Motion-JPEG development effort

Development activity	Person days
Specification and interface definition	4
Module implementation	16
Integration	3
Debugging	11
Code adaption for synthesis	4

[2] Its impacts are directly taken into account by the automatic design space exploration, because both the resource requirements and the execution times are determined after synthesis of the actors.

mathematical sense, the values typically observed are situated in the ranges resulting from Tables 4.2 and 4.3.

Table 4.2 VPC simulation results

# SW actors	Latency (ms)	Throughput (fps)	LUTs	FFs	BRAMs/MULs
0	12.61	81.1	44,878	15,078	72
1	25.06	40.3	41,585	12,393	96
8	4,465	0.22	17,381	8,148	63
All	8,076	0.13	2,213	1,395	29

Table 4.3 Hardware synthesis results

# SW actors	Latency (ms)	Throughput (fps)	LUTs	FFs	BRAMs/MULs
0	15.63	65.0	40,467	14,508	47
1	23.49	43.0	35,033	11,622	72
8	6,275	0.16	15,064	7,540	63
All	10,030	0.10	1,893	1,086	29

The differences in the required hardware sizes occurring between the predicted values during automatic design space exploration and those measured in hardware can be explained by postsynthesis optimizations. The difference in the block RAMs, for instance, could be traced back to the fact that the SystemoC specification uses 32-bit communication channels, although they are not always entirely required. This fact offers a possibility for the Xilinx tools to trim some BRAMs from the FIFO primitives. Furthermore, the size of the MicroBlaze processor changes depending on how many FSL links are effectively connected for hardware–software communication.

The discrepancy between the VPC estimations for latency and throughput and those measured for the hardware-only solution could be traced back to the time spent in complex guards occurring, for instance, in the Huffman Decoder. The underlying reason is that VPC uses an event-based simulation where the evaluation of guards is performed in zero time by the simulation kernel, which is not true for the final hardware implementation.

The discrepancy between the VPC estimations for latency and throughput and those measured for hardware–software or software-only solutions has two major reasons, namely cache influence and occurring schedule overhead. Both aspects are further evaluated in the following two sections.

4.2.1.1 Evaluation of the Schedule Overhead

In order to simulate a SystemoC application, *VPC* uses the *SystemC* event mechanism. In other words, as soon as a transition of the communication finite state machine becomes enabled, a corresponding event is set. A scheduler tracks all notified events, which allows him to run the simulation by repeatedly selecting an enabled actor for execution. However, this means that from the VPC's point of view, selection of an actor that can be executed is performed in zero time, because the corresponding activities are hidden in the simulation kernel.

The real implementation on the MicroBlaze processor, however, performs a simple round-robin scheduler. In other words, all actors bound to a common MicroBlaze are checked in a

round-robin fashion whether one of their transitions can be taken or not. As the time for evaluation of the corresponding activation patterns (including guard functions) is not negligible, a corresponding schedule overhead results whenever an actor is polled, i.e., checked for execution, while no transition is possible. This leads to a time consumption that cannot be taken into account by the VPC framework, as the latter uses an event-based scheduling strategy, thus leading to a discrepancy between the execution times predicted by the VPC and those measured in the final software implementation.

Table 4.4 evaluates the schedule overhead for two different Motion-JPEG streams and a software-only implementation. The values have been obtained by means of a special hardware profiler that monitors the program counter of the MicroBlaze processor in order to determine when the latter is occupied by looking for the next action to execute. As can be seen, scheduling of the different actions takes a significant amount of time, thus explaining the discrepancies between the VPC estimates and the measurements on the hardware implementation.

Table 4.4 Schedule overhead measurements

M-JPEG stream	Run-time(s)	Overhead
QCIF 176 × 144	42.93	11.17 s (26.0%)
Lena 256 × 256	151.04	35.17 s (23.2%)

4.2.1.2 Evaluation of the Influence of the Cache

Table 4.5 illustrates the influence of a cache by means of a software-only solution. It shows the overall execution time for processing four QCIF frames.[3] As can be seen, the simple enabling of both an instruction and a data cache with 16 KB each results in a performance gain of factor 3.4. This is due to the fact that both instruction code and program data are stored on an external memory with relatively large access times. However, since the cache behavior depends on the program execution history, a simple reordering of the program code can lead to significant changes in the action execution times, which cannot be taken into account by the VPC framework.

Table 4.5 Influence of the MicroBlaze cache for software-only solutions

	Processing time for four images(s)
Without cache	146.3
With cache	42.9

Consequently, in order to obtain more precise simulation results an instruction set simulation taking caches and guard execution times into account would be necessary. Similarly, a time-consuming hardware synthesis would be necessary in order to reduce the discrepancy between the VPC hardware estimations and the exact implementation results. This, however, is prohibitive during automatic design space exploration since the number of solutions that

[3] Note that this value divided by 4 does not result in the latency for one image. This is due to the round-robin scheduler, which allows that processing of an image might start before the previous one has been finished.

can be investigated, and hence the possibility to find all optimal implementations, strongly decrease. Instead, SYSTEMCODESIGNER uses the fast VPC simulations in order to find a manageable set of good implementations that can then be further investigated by more precise simulation techniques. By providing an automatic synthesis path, SYSTEMCODESIGNER significantly simplifies this opportunity, and thus helps to evaluate a relatively large number of design points. This compensates for the fact that the VPC estimates might not be dominance preserving.

4.2.2 Influence of the Input Motion-JPEG Stream

Selection of an optimum hardware architecture for the Motion-JPEG decoder depicted in Fig. 4.2 depends, among other criteria, on the question whether it fulfills a given throughput constraint. This, however, is a difficult task due to the data dependency of some of the involved actors, like the Huffman decoder. In other words, the achievable system throughput does not only depend on the chosen hardware implementation but also depend on parameters such as image size and image quality, which is related to the compression ratio or file size.

The previous results have shown how the VPC framework can be used to quickly estimate the performance of a given implementation, based on accurate action execution times. The latter are derived by application profiling based on a particular Motion-JPEG stream, which is a laborious process. However, in contrast to other work, VPC not only simulates a fixed and constant execution time for each actor invocation but also permits the association of an individual delay to each action. The latter should be far less dependent on the processed JPEG file than the actor invocations themselves. That means, performance data obtained from one JPEG stream can be reused for other streams as well, which significantly simplifies the evaluation of a given implementation, and which helps to efficiently estimate the expected execution times for given input streams.

Table 4.6 Comparison between simulated and measured execution times for four images and different JPEG streams

Input JPEG stream	VPC estimation(s)	SW implementation (without schedule overhead's)	Rel. error (%)
QCIF (176 × 144) (5.5 KB)	31.54	31.76	0.7
QCIF (176 × 144) (36.2 KB)	57.34	55.29	3.7
Lena (256 × 256) (55.6 KB)	116.37	115.87	0.4

Table 4.6 shows the achieved results for a software-only implementation when processing different JPEG streams, whereas only the first one has been used to derive the actor execution times. The first two JPEG streams only differ by the encoded image quality and thus deliver different workloads for the data-dependent actors like the *parser* and the *Huffman decoder*. The third JPEG stream uses a completely different image. In both cases, there is a pretty good match between the predicted and the measured values, thus allowing for highly accurate system evaluations. Thanks to the action-accurate exploration where the data dependency can be expressed in the actor's communication finite state machine. Consequently, it can also be taken into account during exploration, while the action execution times themselves ideally do not depend on the processed input image.

4.3 Conclusions

This chapter has presented and evaluated a methodology to automatically optimize and generate system on chips (SoCs) from an abstract actor-oriented model written in SystemC. Consequently, complex systems like the presented Motion-JPEG decoder can be implemented more quickly compared to traditional RTL design flows. Integration of behavioral synthesis avoids redundant implementation on different levels of abstraction. High-level performance evaluation and automatic design space exploration permit to early evaluate different implementation alternatives and select the most promising ones. This not only includes the influence of the target architecture, but also the impact of data-dependent operations like entropy decoding. Automatic system synthesis avoids manual creation of various implementation alternatives in order to verify whether they fulfill all user requirements in terms of throughput, latency, and chip sizes. This is particularly important, as high-level performance evaluation showed to be able to direct the designer toward a good implementation. However, it cannot take every detail like postsynthesis optimizations, cache influence, and scheduling overhead into account in order to not sacrifice a wide-ranging evaluation of different implementation alternatives.

On the other hand the case study also demonstrated the challenge of designing high-speed applications operating on huge image sizes. Whereas this chapter mostly focused on QCIF images of 176×144 pixels, modern applications like medical image processing and video compression for high definition images or even digital cinema movies work on frames with up to 4096×4096 pixels. This, however, makes both system analysis and synthesis much more complex, because a simple simulation of the Motion-JPEG decoder for four QCIF images already takes approximately 30 s. Furthermore, the automatically generated hardware currently only achieves approximately 60 QCIF frames per second using a clock frequency of 50 MHz.

These challenges, however, cannot be solved by just modifying the synthesis or simulation part. Instead they have to be taken into account already during the modeling phase, because usage of one-dimensional models of computation leads to restrictions that are difficult to deal with in the later design flow. This can, for instance, easily be seen by means of the shuffle actor, which is responsible to forward the Y, the C_b, and the C_r value of a pixel in the correct order and in parallel to the YC_bC_r decoder. The current version of the design has been written such that these values are placed sequentially in the input FIFO of the shuffle actor. Consequently, three FIFO accesses are required in order to generate one YC_bC_r value, making the hardware implementation relatively slow. Furthermore, it leads to a relatively complex address calculation and communication finite state machine, which is difficult to analyze and to optimize in an automatic manner. In particular, it is very difficult if not impossible to automatically derive an implementation variant that processes 2 pixels in parallel. The situation gets even worse, when considering sliding window algorithms. As those cannot be represented directly in a one-dimensional model of computation (see also Section 3.1), the designer is responsible for handling the data reuse occurring due to multiple reads of the same data value. This, however, either leads to destroyed parallelism, because all data values are situated in the same input FIFO, or the designer has to build a complex FIFO structure imitating the hardware implementation of sliding window algorithms. Unfortunately, the latter representation is very difficult to analyze in an automatic manner, thus making optimizations like loop-unrolling quasi-impossible.

Consequently, the remainder of this book discusses the benefits of a multidimensional design methodology that can extend classical ESL techniques as implemented in the SystemCoDesigner tool. In particular, Chapter 5 describes an intuitive method for

representing sliding window algorithms and out-of-order communication. This not only simplifies application design but also permits to derive a polyhedral model that can be used for automatic buffer analysis. Compared to buffer analysis via simulation, this improves the analysis speed as well as the quality of the obtained result. Furthermore, a high-speed communication primitive can be derived from the multidimensional model of computation, which can handle both in-order and out-of-order communication, sequential and parallel data access, and very high clock frequencies together with pipelined operation. Automatic synthesis capabilities along with the possibility to exploit tradeoffs between attainable throughput and required hardware resources permit to quickly evaluate different system implementations. It thus forms an important element for generation of high-performance image processing designs on a higher level of abstraction.

Chapter 5
Windowed Data Flow (WDF)

Modern image processing applications not only induce huge computational load but also are characterized by increasing complexity. As exemplarily shown in Section 2.2, they typically consist of a mixture of static and data-dependent algorithms and operate on both one-dimensional and multidimensional streams of data. Efficient implementation is only possible by exploiting different kinds of parallelism, namely task, data, and operation-level parallelism (see Section 2.5). Out-of-order communication and sliding windows with parallel data access require complex communication synthesis.

This, however, imposes various requirements on the specification methods (see also Section 2.5):

- Capability to represent global, local, and point algorithms and the resulting buffer requirements.
- Capability to represent task, data, and operation parallelism.
- Possibility of expressing control flow.
- Tight interaction between static and data-dependent algorithms.
- Support of data reordering.

Section 3.1 has summarized a huge amount of specification techniques concerning their capability to fulfill the above requirements. Purely sequential specifications, for instance, are rather seldom employed because they lead to complex analysis and communication synthesis as discussed in Section 3.2.4 in the context of the *DEFACTO* behavioral compiler. Communicating sequential processes (CSPs) are particularly useful when components interact in form of handshaking. However, complex data dependencies and parallel data access for sliding windows and out-of-order communication are difficult to represent (see Section 3.1.2.1). One-dimensional data flow specifications suffer from huge memory requirements (see Section 3.1.3) or complex models with many phases, reduced module reuse, hidden memory, and the difficulty to express data dependencies, parallelism, and parallel data access (see Section 3.1.3.2). Essentially, this is due to the mapping of a multidimensional problem to a one-dimensional model of computation. However, as shown in Section 3.1.4, also existing multidimensional models of computation are not able to fulfill all above requirements: *MDSDF* cannot handle sliding windows (see Section 3.1.4.1), *CRPs* need expensive analysis and do not allow to include control flow or data-dependent decisions (see Section 3.1.4.2), and *Array-OL* finally uses toroidal border processing instead of virtual border extension and does not consider tight interaction between one-dimensional and multidimensional application parts.

J. Keinert, J. Teich, *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*, Embedded Systems,
DOI 10.1007/978-1-4419-7182-1_5, © Springer Science+Business Media, LLC 2011

Consequently, this chapter aims to introduce two alternative models of computation particularly adapted image processing application, namely the *windowed data flow (WDF)* and its static counterpart *windowed synchronous data flow (WSDF)*. Both essentially model the sampling of multidimensional arrays with possibly overlapping windows as described in Fig. 2.3 belonging to Section 2.2. Virtual border extension considers the particularities occurring at the image borders. The introduction of multidimensional delay elements allows representing feedback loops while fine-grained dependency analysis permits efficient hardware implementations. Compared to approaches discussed in Section 3.1, this leads to the following benefits (see also Section 3.1.5 for a detailed discussion):

- Intuitive representation of sliding window algorithms including their parallel data access. In particular, data elements can be read several times, which is difficult to represent in well-known models of computation like SDF (Section 3.1.3.1), CSDF (Section 3.1.3.2), and MDSDF (Section 3.1.4.1).
- Capability to represent global, local, and point algorithms and the resulting memory requirements.
- Capability to represent task, data, and operation parallelism.
- Possibility of expressing control flow.
- Explicit specification of communication orders in order to support data reordering.
- Inclusion of virtual border extension.
- Tight interaction between one-dimensional and multidimensional data flows.
- Flexible delays.
- Restriction to rectangular window patterns in order to simplify analysis without lessening the applicability of WDF.
- Capability of data dependency analysis in order to support powerful polyhedral analysis (see also Chapter 7).

The remainder of this chapter is as follows. Section 5.1 introduces the basic semantics for multidimensional communication between two modules. Section 5.2 provides some mathematical conditions for balanced and valid communication. Section 5.3 describes how data reordering can be captured, followed by a discussion on communication control in Section 5.4. Section 5.5 is dedicated to static applications that obey several restrictions defining the *windowed synchronous data flow (WSDF)* model of computation. Such specifications can be checked for bounded memory execution as discussed in Section 5.6. Section 5.7 targets integration of WDF into the system level design tool SystemCoDesigner (see Chapter 4) before Section 5.8 presents two case studies in form of a morphological reconstruction and a lifting-based wavelet transform in order to demonstrate the usefulness of the chosen approach. Section 5.9 then discusses the limitations of the current modeling technique before Section 5.10 concludes the chapter.

5.1 Sliding Window Communication

5.1.1 WDF Graph and Token Production

Similar to any other data flow model of computation, applications are modeled in WDF by means of a data flow graph G, which operates on an infinite stream of data. Figure 5.1 depicts a corresponding graphical representation for a simple WDF graph with two actors a_1 and a_2 interconnected by one single edge. Triangles correspond to so-called *initial tokens,* while

diamonds define *border processing*. Both aspects will be described later on in this section. O^{write} and O^{read} define the communication order and are discussed in Section 5.3.

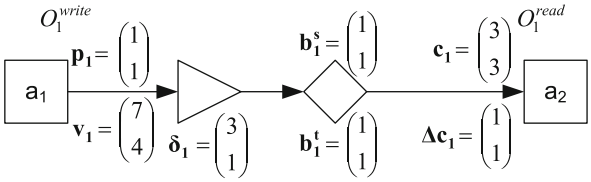


Fig. 5.1 WDF graph with two actors and a single edge for illustration of the introduced notation

Mathematically, a WDF data flow graph G can be defined as a tuple[1]

$$G = \left(A, E, \mathbf{p}, \mathbf{v}, \mathbf{c}, \Delta\mathbf{c}, \boldsymbol{\delta}, \mathbf{b^s}, \mathbf{b^t}, O^{\text{write}}, O^{\text{read}}\right),$$

where $\mathbf{p}$, $\mathbf{v}$, $\mathbf{c}$, $\Delta\mathbf{c}$, $\boldsymbol{\delta}$, $\mathbf{b^s}$, $\mathbf{b^t}$, O^{write}, O^{read} are the graph parameters, while A and E define its topology. In order to ease notation, the parameters are typically not listed explicitly in the tuple leading to the following short-hand notation:

$$G = (A, E)\,.$$

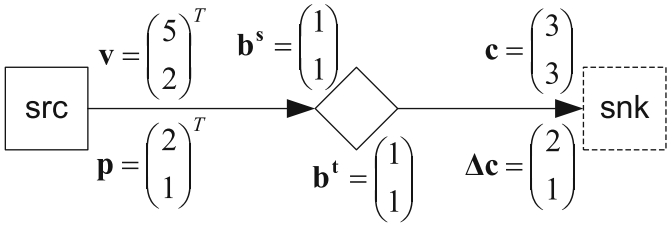
A is the set of vertices representing *processes* or *actors*. $E \subseteq A \times A$ defines the set of *edges* interconnecting the actors via corresponding *ports*. Each edge $e \in E$ transports an infinite stream of multidimensional arrays having $n_e \in \mathbb{N}$ dimensions. However, in order to avoid huge buffer size requirements and to support fine-grained modeling, these arrays are not produced as a whole, but in smaller parts, the so-called *effective tokens*. They consist of several *data elements* whose number is specified by the function $\mathbf{p} : E \ni e \mapsto \mathbf{p}(e) \in \mathbb{N}^{n_e}$. Assuming, for instance, $n_e = 2$, the dot product $\langle \mathbf{p}(e), \mathbf{e_1} \rangle$ defines the number of columns for the effective token produced by the actor $\text{src}(e)$, whereas $\langle \mathbf{p}(e), \mathbf{e_2} \rangle$ corresponds to the number of rows. $\mathbf{e_1}$ and $\mathbf{e_2}$ are Cartesian base vectors. The so-generated data elements are combined to *virtual tokens* whose size is defined by the function $\mathbf{v} : E \ni e \mapsto \mathbf{v}(e) \in \mathbb{N}^{n_e}$. These virtual tokens correspond, for instance, to images or sub-images and define the set of pixels on which the sink performs its sliding window operation.

This WDF token production is exemplified in Fig. 5.2. Each write operation of the source actor generates an effective token consisting of two columns and one row. These effective tokens are combined to virtual tokens consisting of 5×2 data elements. The number of produced virtual tokens can be calculated by means of a local balance equation that will be derived in Section 5.2. Essentially, it takes care that each produced data element is also consumed. In the scenario depicted in Fig. 5.2, the size of the effective token is not a multiple of the virtual token size such that at least two virtual tokens have to be produced.

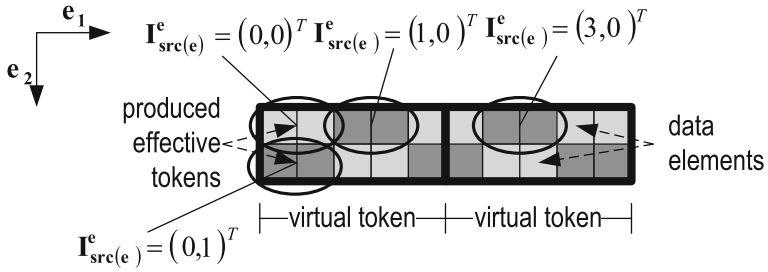
For identification purposes, each write operation of actor $\text{src}(e) \in A$ on edge $e \in E$ can be labeled by means of an n_e-dimensional *iteration vector* $\mathbf{I^e_{src(e)}} \in \mathbb{N}_0^{n_e}$, where $\langle \mathbf{I^e_{src(e)}}, \mathbf{e_i} \rangle$

[1] References [165, 166] additionally use the concept of virtual token unions of size $\mathbf{u}$. However, since they have been superseded by the communication order described in Section 5.3, they are omitted in this monograph ($\mathbf{u} = \mathbf{1}$).

defines how many previous write operations have already occurred in dimension i before the currently considered one. Thus, write operation $\mathbf{I}^{\mathbf{e}}_{\mathbf{src(e)}}$ produces the $\left(\langle \mathbf{I}^{\mathbf{e}}_{\mathbf{src(e)}}, \mathbf{e_i} \rangle + 1\right)$th effective token on the output edge $e \in E$ in dimension i. Figure 5.2 illustrates the corresponding principle by annotating some of the effective tokens with the vector $\mathbf{I}^{\mathbf{e}}_{\mathbf{src(e)}}$ of the corresponding write operation.



(a)



(b)

Fig. 5.2 Example for WDF token production showing the composition of a virtual token **v** by effective tokens **p**. **a** WDF graph fragment. **b** Combination of effective tokens to virtual tokens

5.1.2 Virtual Border Extension

The so-constructed virtual tokens can now be sampled by the sink actor with possibly overlapping windows. However, since most sliding window algorithms require virtual border extension, a corresponding concept has also to be introduced in WDF. Therefore, each virtual token can be extended by a virtual border whose size is defined by two functions $\mathbf{b^s}$ and $\mathbf{b^t}$:

$$\mathbf{b^{s,t}} : E \ni e \mapsto \mathbf{b^{s,t}}(e) \in \mathbb{Z}^{n_e}$$
$$\langle \mathbf{v}(e), \mathbf{e_i} \rangle + \langle \mathbf{b^s}(e), \mathbf{e_i} \rangle + \langle \mathbf{b^t}(e), \mathbf{e_i} \rangle > 0.$$

Their interpretation for positive values is shown in Fig. 5.3, which corresponds to the graph depicted in Fig. 5.2a. Negative values lead to suppression of data elements. As can be seen, each virtual token is extended by a border whose size at the "upper left corner" is defined by $\mathbf{b^s}$ (s stands for start), whereas $\mathbf{b^t}$ (t stands for termination) is associated to the "lower right corner." In other words, for $n_e = 2$, $\langle \mathbf{b^s}, \mathbf{e_1} \rangle$ and $\langle \mathbf{b^t}, \mathbf{e_1} \rangle$ define the number of

columns to add at the beginning, respectively, end of the virtual token. $\langle \mathbf{b^s}, \mathbf{e_2} \rangle$ and $\langle \mathbf{b^t}, \mathbf{e_2} \rangle$ correspond to the number of rows to add. For more than two dimensions, the interpretation is straightforward. For each virtual token, there exist two border hyperplanes, which are orthogonal to $\mathbf{e_i}$. Consequently, $\langle \mathbf{b^s}, \mathbf{e_i} \rangle$ defines the number of hyperplanes to add at the border with the smaller coordinates in direction $\mathbf{e_i}$ while $\langle \mathbf{b^t}, \mathbf{e_i} \rangle$ defines the same for the border with the higher coordinates. The so-enlarged virtual token is referred to as *extended virtual token*.

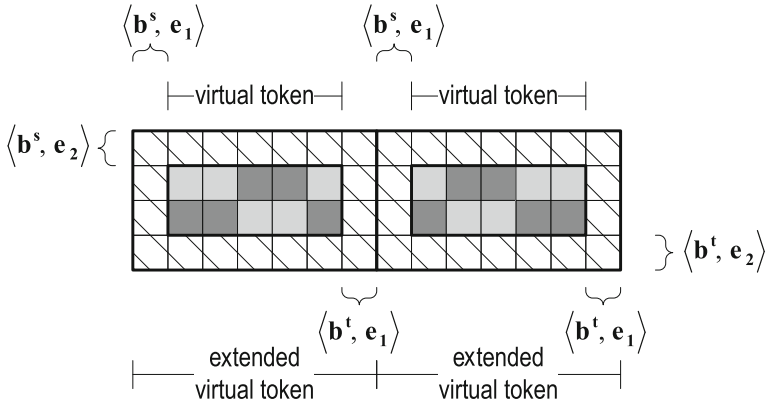


Fig. 5.3 Illustration of the virtual border extension in the WDF model of computation

An important fact is that the data elements belonging to the extended border are not produced by the source actor. Instead, they can be assumed to have a predefined value depending on the border processing algorithm. The latter is not an explicit part of the WDF model of computation. Instead, it supports any border processing algorithm that does not violate the token consumption rules defined in the following paragraph.

5.1.3 Token Consumption

The so-extended virtual tokens can now be sampled by the edge sink with possibly overlapping *windows*. The latter are typical for many local image processing applications and are defined by the window size as well as the window movement. For each read operation, the sink actor snk (e) accesses such a sliding window whose size is defined by the function $\mathbf{c} : E \ni e \mapsto \mathbf{c}\,(e) \in \mathbb{N}^{n_e}$. However, such a read operation can only be performed if enough input data elements are available. Inside a virtual token, the *window movement* is defined by the *sampling vector*[2] $\Delta\mathbf{c}\,\mathbf{(e)}$, where $\langle \Delta\mathbf{c}\,\mathbf{(e)}, \mathbf{e_i} \rangle$ defines the window shift in dimension i. This is, however, only valid if the window does not leave the borders of the extended virtual token. Otherwise, sampling of the next virtual token starts at its "upper left corner."

[2] In accordance with [215], [165, 166] proposed to use a diagonal sampling matrix in order to describe the window movement, because this permits to represent non-rectangular sampling patterns. However, since this will not be used in this book , the matrix is replaced by a vector in order to ease notation.

Figure 5.4 illustrates the WDF token consumption by listing all possible sliding window positions for the edge depicted in Fig. 5.4a. Each of the sliding window positions corresponds to a sink read operation, which can be identified by an *iteration vector* $\mathbf{I}^{\mathbf{e}}_{\mathbf{snk(e)}} \in \mathbb{N}_0^{n_e}$, where $\langle \mathbf{I}^{\mathbf{e}}_{\mathbf{snk(e)}}, \mathbf{e_i} \rangle$ defines the number of read operations occurred in dimension i before the currently considered one. In other words, the read operation $\mathbf{I}^{\mathbf{e}}_{\mathbf{snk(e)}}$ reads in each dimension i the $\left(\langle \mathbf{I}^{\mathbf{e}}_{\mathbf{snk(e)}}, \mathbf{e_i} \rangle + 1 \right)$th sliding window. For Fig. 5.4, the window moves by $\langle \Delta\mathbf{c}, \mathbf{e_1} \rangle = 2$ in horizontal direction and by $\langle \Delta\mathbf{c}, \mathbf{e_2} \rangle = 1$ in vertical direction. However, this movement only occurs in the inner of each extended virtual token. After reaching its end, the window jumps to the beginning of the neighbor-extended virtual token. Although Fig. 5.4 is restricted to $n_e = 2$ dimensions for illustration purposes, extension to $n_e > 2$ is straightforward. Note that the order in which the sink traverses the multidimensional array transported on the WDF edge is not specified yet. In other words, the temporal order of the scenarios depicted in Fig. 5.4 might, for instance, be defined as

$$(0,0)^T, (1,0)^T, (2,0)^T, (3,0)^T, (0,1)^T, (1,1)^T, (2,1)^T, (3,1)^T.$$

However,

$$(0,0)^T, (0,1)^T, (1,0)^T, (1,1)^T, (2,0)^T, (2,1)^T, (3,0)^T, (3,1)^T$$

would be equally possible. Specification of this execution order is subject of Section 5.3.

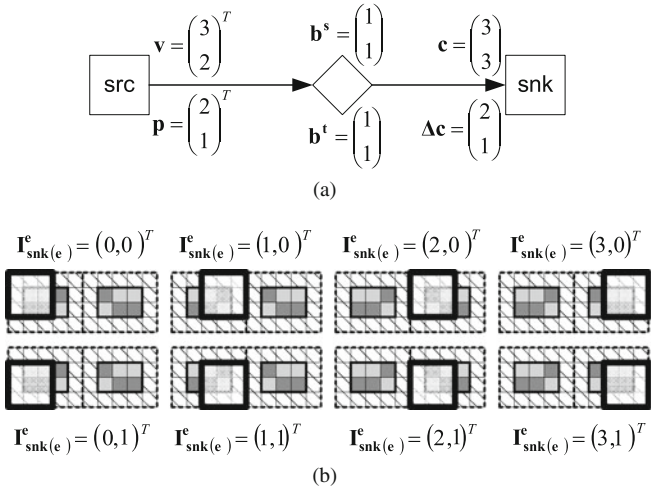


Fig. 5.4 Illustration of token consumption. **a** WDF graph corresponding to Fig. 5.2a, but using $\mathbf{v} = (3,2)^T$ for illustration purposes. **b** Existing sliding windows

5.1.4 Determination of Extended Border Values

As already mentioned previously, extended border pixels are not produced by the corresponding source actor. Consequently, their value has to be defined by a border processing algorithm, which is not directly part of the WDF communication specification. Instead, any algorithm is permitted that can determine the value of each border data element without requiring data elements situated outside the current sliding window. The corresponding algorithm can then be implemented directly in the sink actor without influencing its external behavior. Constant border extension, for instance, where each border pixel is set to a constant value, perfectly meets the previous requirement. However, it is also possible to assume more complex border processing algorithms like symmetric border extension.

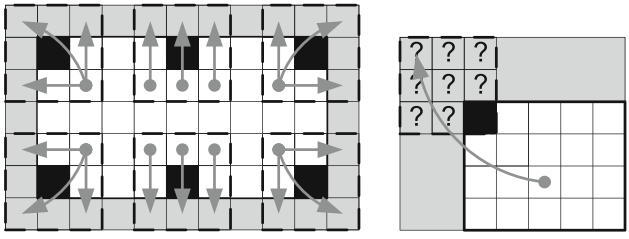
Figure 5.5a shows a corresponding example for a simple 3×3 window with symmetric border extension. The extended border pixels are shaded gray. Their value can be determined by pixel mirroring, as depicted by means of corresponding arrows. In this case, each sliding window covers all pixels required for derivation of the border values. However, this is not true anymore for the scenario shown in Fig. 5.5b. In this case determination of the border values requires knowledge about the data elements situated outside the current sliding window, which is not possible in WDF. Thus, in order to support this scenario, the sliding window has to be enlarged to a size of 5×5 data elements.

5.1.5 WDF Delay Elements

WDF delay elements allow to define which and how many data elements are already available when starting graph execution. This is essentially required for applications with feedback loops as discussed in Section 5.8.1. To this end, each edge has an associated vector $\boldsymbol{\delta}$, which defines the number of initial hyperplanes and whose interpretation is similar to MDSDF (see also Section 3.1.4):

$$\boldsymbol{\delta} : E \ni e \mapsto \boldsymbol{\delta}(e) \in \mathbb{N}_0^{n_e}.$$

$\langle \boldsymbol{\delta}(e), \mathbf{e_i} \rangle$ defines the number of initial hyperplanes orthogonal to $\mathbf{e_i}$. Figure 5.6 shows the corresponding interpretation for $n_e = 2$ in combination with virtual border extension. As can be seen, the multidimensional delay $\boldsymbol{\delta}(e) = (3, 1)^T$ defines the number of data elements that already exist before the source actor starts with any invocation. Note, however, that the source



(a) (b)

Fig. 5.5 Symmetric Border Extension in WDF. *Arrows* depict which pixels are copied into the extended border. **a** Derivation of border value. **b** Illegal border extension in WDF

actor is not influenced at all by the presence of these initial data elements.[3] Consequently, the data elements produced by src (e) are shifted as depicted by the bold frame in Fig. 5.6. This also means that data elements are available for the second virtual token before the source starts its proper production.

In this context, it is important to remind that WDF graphs operate on an infinite stream of multidimensional arrays. Thus, after production of the first virtual token depicted in Fig. 5.6, a second one is started, which is supposed to be added in dimension $\mathbf{e}_{\mathbf{n_e}}$. Consequently, initial tokens cause that data elements produced during the first virtual token are placed into the second one as exemplified in Fig. 5.6. On the other hand, other data elements might not be read at all by the corresponding sink actor and can just be discarded.

5.2 Local WDF Balance Equation

After having introduced the WDF communication semantics, this section develops corresponding conditions for *valid application models*. This means in particular that all data elements produced on an edge $e \in E$ are either consumed by the corresponding sink or suppressed by negative border extension when assuming $\boldsymbol{\delta} = 0$.

Corollary 5.1 *Given a sink actor* snk (e) *of edge e as depicted in Fig. 5.7. Then, the number of read operations on edge e for processing one single input virtual token and for dimension i is given by*

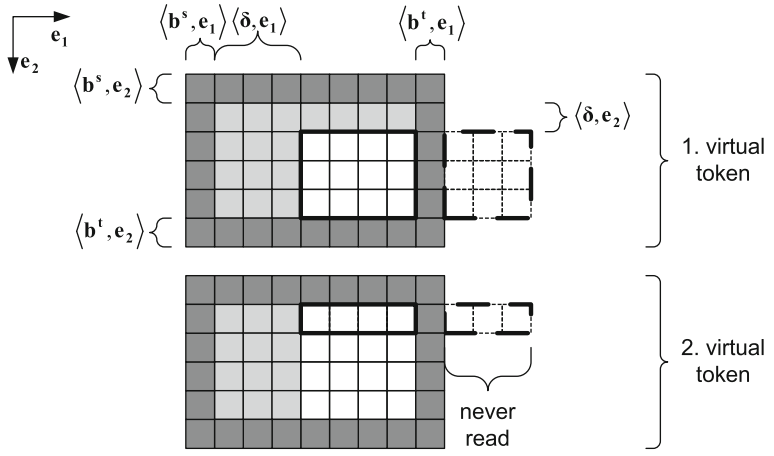


Fig. 5.6 Interpretation of WDF delays (*light gray*) ($\boldsymbol{\delta} = (3, 1)^T$) in the presence of border processing (*dark gray*) ($\mathbf{b^s} = \mathbf{b^t} = (1, 1)^T$). The *bold frames* indicate the data elements produced by the source actor for the first virtual token with $\mathbf{v} = (7, 4)^T$. The size of the effective token is irrelevant and could, for instance, amount $\mathbf{p} = (1, 1)^T$

[3] Assuming edge buffers of infinite size.

$$\langle \mathbf{r}_{vt}(e), \mathbf{e_i} \rangle = \frac{\langle \mathbf{v_e}, \mathbf{e_i} \rangle + \langle \mathbf{b_e^s} + \mathbf{b_e^t}, \mathbf{e_i} \rangle - \langle \mathbf{c_e}, \mathbf{e_i} \rangle}{\langle \Delta \mathbf{c_e}, \mathbf{e_i} \rangle} + 1. \tag{5.1}$$

Proof By definition in Section 5.1, WDF delays do not influence the number of sink and source invocations. Thus, vector $\boldsymbol{\delta}$ can be ignored. The initial read operation in dimension i needs $\langle \mathbf{c_e}, \mathbf{e_i} \rangle$ new data elements. Each consecutive firing only needs $\langle \Delta \mathbf{c_e}, \mathbf{e_i} \rangle$ new data elements, because windows can overlap. As a consequence, supposing one virtual token, the number of consumed data elements can be calculated to

$$= \frac{\underbrace{\langle \mathbf{c_e}, \mathbf{e_i} \rangle + \langle \Delta \mathbf{c_e}, \mathbf{e_i} \rangle \times (\langle \mathbf{r}_{vt}(e), \mathbf{e_i} \rangle - 1)}_{\text{consumed data elements in dimension } i}}{\underbrace{\langle \mathbf{v_e}, \mathbf{e_i} \rangle + \langle \mathbf{b^s} + \mathbf{b^t}, \mathbf{e_i} \rangle}_{\text{size of virtual token in dimension } i\text{, including extended borders}}}.$$

As described in Section 5.1, sampling with possibly overlapping windows only occurs within one single virtual token. If the sliding window reaches the end of a virtual token, it "jumps" to the next one. The following definition takes care that the last window indeed touches the outermost data element of an extended virtual token.

Definition 5.2 A WDF edge $e \in E$ is called *valid*, if and only if

$$\forall 1 \leq i \leq n : \langle \mathbf{r}_{vt}(e), \mathbf{e_i} \rangle \in \mathbb{N}. \tag{5.2}$$

Definition 5.3 A *valid WDF graph* does not contain any invalid edges.

Example 5.4 Considering the WDF graph given in Fig. 5.4a, $\mathbf{r_{vt}}(e)$ can be calculated as

$$\mathbf{r}_{vt}(e) = \begin{pmatrix} \frac{3+1+1-3}{2} + 1 \\ \frac{2+1+1-3}{1} + 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \end{pmatrix}.$$

Hence, the sink has to perform two read operations in each dimension in order to cover the complete virtual token. This corresponds exactly to what is shown in Fig. 5.4b. Moreover, the edge is valid as the right and lower window boundaries touch the corresponding edges of the extended virtual token.

Whereas a valid WDF edge ensures that the sink reads complete virtual tokens, the next corollary defines a condition assuring that in case of $\boldsymbol{\delta}(e) = \mathbf{0}$, all data elements produced by the source are either read by the sink or suppressed by a corresponding border processing. In this case, the edge is called *locally balanced*. This is an important property for execution with bounded memory as discussed later on in Section 5.6.

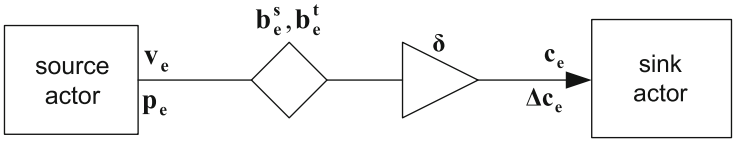


Fig. 5.7 Single WDF edge

Corollary 5.5 Local WDF Balance Equation
In order to be locally balanced, the source has to perform

$$k_i^e \times \frac{\operatorname{scm}\left(\langle \mathbf{p}, \mathbf{e_i} \rangle, \langle \mathbf{v}, \mathbf{e_i} \rangle\right)}{\langle \mathbf{p}, \mathbf{e_i} \rangle} = k_i^e \times \frac{\langle \mathbf{v}, \mathbf{e_i} \rangle}{\gcd\left(\langle \mathbf{p}, \mathbf{e_i} \rangle, \langle \mathbf{v}, \mathbf{e_i} \rangle\right)}$$

write operations in each dimension i, with $k_i^e \in \mathbb{N}$ being an arbitrary positive integer. $\operatorname{scm}(a, b)$ *defines the smallest common multiple of a and b,* $\gcd(a, b)$ *the greatest common divisor. This leads to read operations of the sink.*

$$\underbrace{k_i^e \times \frac{\operatorname{scm}\left(\langle \mathbf{p}, \mathbf{e_i} \rangle, \langle \mathbf{v}, \mathbf{e_i} \rangle\right)}{\langle \mathbf{v}, \mathbf{e_i} \rangle}}_{\substack{\text{number of produced} \\ \text{virtual tokens}}} \times \langle \mathbf{r_{vt}}(e), \mathbf{e_i} \rangle = \underbrace{k_i^e \times \frac{\langle \mathbf{p}, \mathbf{e_i} \rangle}{\gcd\left(\langle \mathbf{p}, \mathbf{e_i} \rangle, \langle \mathbf{v}, \mathbf{e_i} \rangle\right)}}_{\substack{\text{number of produced} \\ \text{virtual tokens}}} \times \langle \mathbf{r}_{\mathrm{vt}}(e), \mathbf{e_i} \rangle$$

Proof The source must produce complete virtual tokens. Thus, the number of data elements produced by the source must be a multiple of the smallest common multiple of $\langle \mathbf{p}, \mathbf{e_i} \rangle$ and $\langle \mathbf{v}, \mathbf{e_i} \rangle$. The sink has to perform $\langle \mathbf{r_{vt}}(e), \mathbf{e_i} \rangle$ read operations per virtual token.

Definition 5.6 Local Schedule Period
The local schedule period contains all read and write operations resulting from the local WDF balance equation.

Definition 5.7 Token Space
The *token space* of an edge $e \in E$ contains all data elements produced during execution of the WDF edge.

Thus, in order to determine the number of required invocations, the designer must provide the values k_i^e for each edge $e \in E$ and for each dimension i. Technically, this information is included in the specification of the communication order as discussed in the following section, because it contains the overall number of read or write operations in each dimension.

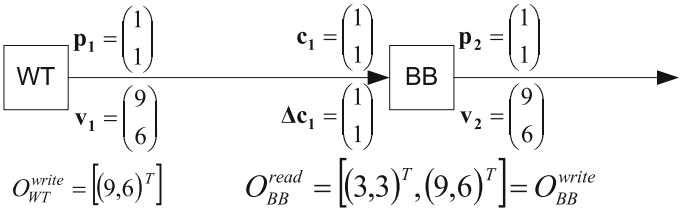
5.3 Communication Order

The previous two sections have introduced a formal model for sliding window communication. Furthermore, a condition for verification of the communication correctness has been derived. However, the communication order is still undefined. As the latter is a very important aspect in system level design of image processing applications (see Section 2.2), this section provides a corresponding formalism. It permits specification of different read and write orders such that image tiling or code-block building can be accurately described.

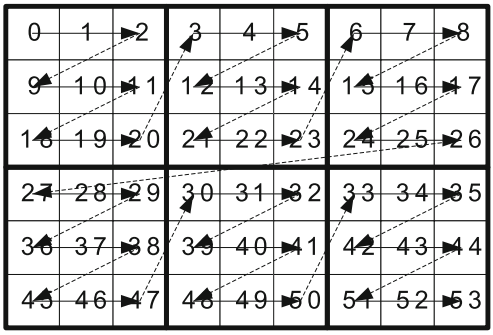
Principally, the communication order can be represented by two sequences of iteration vectors $\left[\mathbf{I}^{\mathbf{e}}_{\mathrm{src},\mathbf{1}}, \mathbf{I}^{\mathbf{e}}_{\mathrm{src},\mathbf{2}}, \ldots\right]$ and $\left[\mathbf{I}^{\mathbf{e}}_{\mathrm{snk},\mathbf{1}}, \mathbf{I}^{\mathbf{e}}_{\mathrm{snk},\mathbf{2}}, \ldots\right]$ defining for each edge the temporal succession of both the read and the write positions. However, this would permit to specify very irregular communication orders, which are difficult to analyze and synthesize, reducing thus the efficiency of the obtained system level design implementation. Consequently, this section introduces the so-called *hierarchical line-based (HLB)* communication order scheme, which

is sufficiently general to represent occurring out-of-order communication such as image tiling and block forming while still permitting efficient buffer analysis and communication synthesis. In case an algorithm does not follow the corresponding communication scheme, it can still be covered using WDF by reducing the modeling granularity. A corresponding example will be shown in Section 5.8.1.3.

In order to make the problem formulation more concrete, Fig. 5.8 resumes the application of code-block forming required for JPEG2000 image compression (see Section 2.2). The *wavelet transform (WT)* generates a partial image of wavelet coefficients, which have to be reordered by the *block builder (BB)* into blocks of 3×3 pixels.[4] These blocks can then be processed by one or more entropy encoders (EB). As illustrated by Arabic numbers in Fig. 5.8, the wavelet transform writes the coefficients line by line, which is also called *raster-scan order*. In contrast, the BB consumes the pixels block by block as depicted by corresponding arrows. This leads to so-called *out-of-order communication*, because the read and write order differs.



(a)



(b)

Fig. 5.8 Code-block forming in JPEG2000. *Arabic numbers* describe the production order of the wavelet coefficients while the *arrows* depict the read order of the BB. **a** WDF graph. **b** WT and BB invocation order

Figure 5.8a depicts the corresponding WDF graph. Each invocation of the wavelet transform generates one output pixel ($\mathbf{p_1} = (1, 1)^T$), which is assembled to an image of 9×6 pixels. The BB actor on the other hand reads 1 pixel per invocation using a non-overlapping window of size $\mathbf{c_1} = (1, 1)^T$.

[4] This block size has only be chosen for illustration purposes. In reality, blocks are much larger and their size has to be a power of 2.

Whereas this model precisely defines the input data required by a given read operation as well as by which write operation they are generated, it does not deliver any information about the occurring invocation order (see also Section 5.1). Since this, however, has a tremendous impact on the required edge buffer size (see Chapter 6), each input and output port is annotated with a so-called *sequence of firing blocks* defining the communication order.

Definition 5.8 A *sequence of firing blocks* $O = \left[\mathbf{B_1}, \ldots, \mathbf{B_q}\right]$ consists of a list of vectors obeying the following property:

$$\forall 1 < k \leq q,\ \forall 1 \leq j \leq n_e : \frac{\langle \mathbf{B_k}, \mathbf{e_j} \rangle}{\langle \mathbf{B_{k-1}}, \mathbf{e_j} \rangle} \in \mathbb{N}. \tag{5.3}$$

The number of sequence members $|O| = q$ is called *firing levels*.

This sequence of firing blocks defines a block hierarchy of read or write operations, which are executed in raster-scan order. Taking, for instance, the BB actor shown in Fig. 5.8 this would lead to

$$O_{\mathrm{BB}}^{\mathrm{read}} = \left[(3,3)^T, (9,6)^T\right].$$

In other words, the BB actor is defined to execute 3×3 read operations line by line. Using the notation of Fig. 5.8, this leads to the read operations 0,1,2,9,10,11,18,19, and 20. The so-formed firing block is concatenated in such way that it leads to $\langle \mathbf{B_2^{BB}}, \mathbf{e_1} \rangle = 9$ read operations in horizontal direction and $\langle \mathbf{B_2^{BB}}, \mathbf{e_2} \rangle = 6$ read operations in vertical direction. Consequently, the number of individual firing blocks in horizontal direction can be calculated as $\frac{\langle \mathbf{B_2^{BB}}, \mathbf{e_1} \rangle}{\langle \mathbf{B_1^{BB}}, \mathbf{e_1} \rangle} = \frac{9}{3} = 3$. The corresponding value in vertical direction amounts $\frac{\langle \mathbf{B_2^{BB}}, \mathbf{e_2} \rangle}{\langle \mathbf{B_1^{BB}}, \mathbf{e_2} \rangle} = \frac{6}{3} = 2$. These individual firing blocks are once again traversed in raster-scan order from left to right and from top to bottom. In the inner of each firing block, the BB actor also uses raster-scan order leading to the read order shown in Fig. 5.8b. The interpretation for different values of n_e and $|O|$ is straightforward. Condition (5.3) takes care that only complete firing blocks occur.[5] Note that by means of this notation, also k_i^e of Section 5.2 is implicitly given as follows:

$$B_{q_{\mathrm{src}}}^{\mathrm{src}} = k_i^e \times \frac{\mathrm{scm}\left(\langle \mathbf{p}, \mathbf{e_i} \rangle, \langle \mathbf{v}, \mathbf{e_i} \rangle\right)}{\langle \mathbf{p}, \mathbf{e_i} \rangle},$$

$$B_{q_{\mathrm{snk}}}^{\mathrm{snk}} = k_i^e \times \frac{\mathrm{scm}\left(\langle \mathbf{p}, \mathbf{e_i} \rangle, \langle \mathbf{v}, \mathbf{e_i} \rangle\right)}{\langle \mathbf{v}, \mathbf{e_i} \rangle} \times \langle \mathbf{r}_{\mathrm{vt}}(e), \mathbf{e_i} \rangle.$$

$B_{q_{\mathrm{src}}}^{\mathrm{src}}$ is the largest firing block describing the write operation on edge $e \in E$. Similarly, $B_{q_{\mathrm{snk}}}^{\mathrm{snk}}$ is the largest firing block for the read operation on edge $e \in E$. From these equations, it can be seen that the largest firing blocks for the read and write order have to obey the following relation:

[5] It could be imagined to relax this condition in order to support also incomplete code-blocks, which can occur in JPEG2000 compression for certain relations between code-block and image size. As this, however, complicates both analysis and synthesis, it is not further detailed.

$$\frac{B_{q_{\text{src}}}^{\text{src}}}{B_{q_{\text{snk}}}^{\text{snk}}} = \frac{\langle \mathbf{v}, \mathbf{e_i} \rangle}{\langle \mathbf{p}, \mathbf{e_i} \rangle \times \langle \mathbf{r}_{\text{vt}}(e), \mathbf{e_i} \rangle}.$$

5.4 Communication Control

After having explained the communication semantics for an individual edge, this section describes the communication control performed within an actor. In other words, it establishes a relation between the individual actor invocations or firings and the occurring read and write operations. This is performed in two steps. Section 5.4.1 shows how WDF communication can be mapped to FIFO semantics. Based on these results, a communication finite state machine can be constructed that defines on which input and output ports communication takes place for a given actor invocation. Corresponding details can be found in Section 5.4.2.

5.4.1 Multidimensional FIFO

One-dimensional data flow models of computation such as SDF (see Section 3.1.3.1) or CSDF (see Section 3.1.3.2) typically employ FIFO communication as depicted in Fig. 5.9. Such a FIFO can transport a one-dimensional stream of data. Corresponding control information like *full* and *wr_count* informs the source how much space is still available for writing new data. Similarly, *empty* and *rd_count* indicates the sink whether new data are available. New data can be written by the source issuing a *write request* while the sink can retrieve the next data via a *read request*.

In contrast to this FIFO data exchange, WDF employs different communication semantics that allow sampling of multidimensional arrays with sliding windows using different production and consumption orders. Although this seems to be completely different from the above-described FIFO semantics, a WDF edge can be considered as an extended, multidimensional FIFO if the consumption and production orders are assumed to be FIFO parameters.

Figure 5.10 shows two corresponding examples for sliding window and out-of-order communication. If we assume that in both cases the corresponding multidimensional FIFO knows all WDF edge parameters depicted in Fig. 5.1 together with consumption and production orders $O_{\text{src}}^{\text{write}}$ and $O_{\text{snk}}^{\text{read}}$, then it can always determine whether the window required next by the sink is already available or not. Assuming, for instance, the scenario given in Fig. 5.10a, it can be easily seen that the first sliding window for the sink actor is ready as soon as the source has produced the second pixel of the second line. Moreover, if the source is supposed to have already produced all gray-shaded pixels, sophisticated

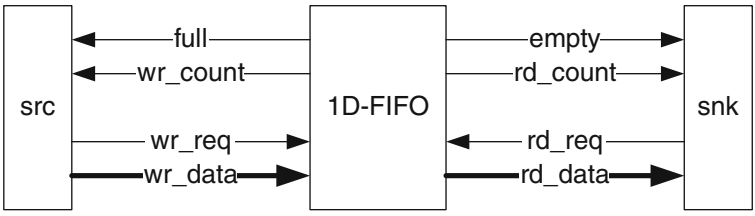


Fig. 5.9 Principle of one-dimensional FIFO communication

calculation presented later on in Chapter 8 is able to derive that the sink can definitively perform 10 read operations without any data element to be missing. This means not only that corresponding *empty* and *rd_count* information can be provided but also that the sink can start operation before a complete image is available.

As a consequence, it is possible to employ fine-grained and memory efficient scheduling. In other words, it is not necessary to provide a buffer space big enough to store a complete image. Instead, only parts of the image generated by the source must be buffered, which permits an efficient implementation. The exact amount of required memory depends on the implemented image processing algorithm and the applied memory organization and will be further discussed in Chapters 6 and 7. For instance, the algorithm depicted in Fig. 5.10a requires a minimum buffer of 2 lines and 3 pixels.

Once the buffer size is fixed, the multidimensional FIFO can also determine whether an effective token from the source can be accepted or not. This decision has to take into account which data elements are still required and which effective token will be produced next. In Fig. 5.10a, for instance, the source has already filled three buffer lines. If we suppose that this corresponds to the maximum buffer size, the source cannot write any effective tokens anymore and has to wait until a buffer element can be freed again. This situation can be indicated to the source by means of a corresponding *full* flag. Sophisticated analysis techniques even

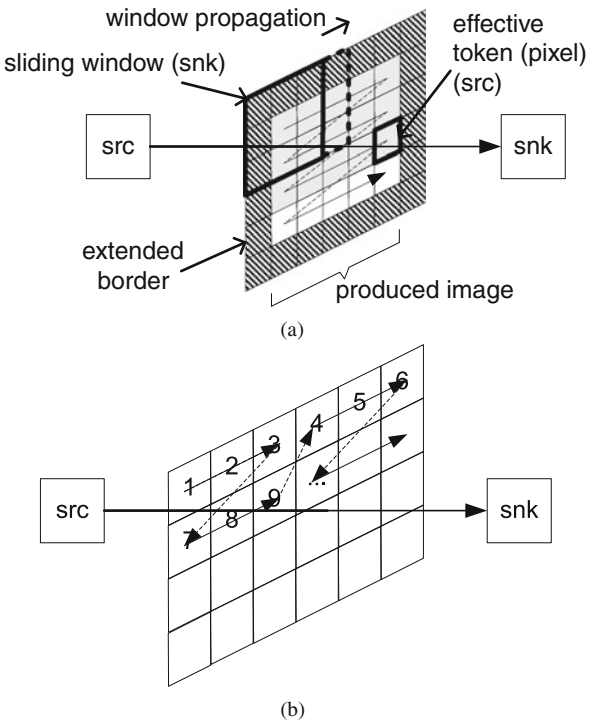


Fig. 5.10 Principle of the multidimensional FIFO. **a** Sliding window. **b** Out-of-order communication. The production order is defined by *Arabic numbers*, while the consumption order is indicated by corresponding *arrows*

allow calculation how often the source can be definitively executed without overwriting still required data. Further details on this aspect can be found in Chapter 8.

In other words, a WDF edge can be considered as a special FIFO called *multidimensional FIFO*. This name has been chosen because it alludes to the fact that the multidimensional FIFO provides a FIFO-like interface including full and empty signals as well as read and write requests. Note, however, that the multidimensional FIFO only follows the first-in first-out semantics in that equivalent data elements belonging to different local schedule periods do not overtake each other. However, in the inner of such a schedule period, the read and write order might differ. Furthermore, sliding windows permit to read data elements several times, and due to virtual border extension a sink is even allowed to read data elements that have not been produced by the corresponding source.

5.4.2 Communication Finite State Machine for Multidimensional Actors

The previous section has shown that multidimensional WDF communication can be interpreted as extended FIFO semantics. In particular, read and write control is similar to one-dimensional FIFO communication. Consequently, this permits to combine multidimensional communication with control flow in order to represent powerful application scenarios such as sharing of the wavelet transform kernel in JPEG2000 (see Sections 2.2 and 5.8.2) or data-dependent decisions. Furthermore, it enables tight interaction between one-dimensional and multidimensional models of computation.

For this purpose, communication control is performed by enriching each WDF actor with a *communication finite state machine* (see Section 4.1). This perfectly fits into the SYSTEMCODESIGNER design flow described in Chapter 4. Furthermore, an actor can have both one-dimensional and multidimensional ports connecting to one-dimensional or multidimensional FIFOs, respectively. The communication finite state machine consists of one or more *states*, connected by *transitions*. With each transition, a *condition* and an *action* are associated. The condition consists of predicates over available tokens and token values that are accessible through the input ports of the WDF actor. Furthermore, they may include requirements on the free spaces in the channels connected to the output ports, as well as conditions on expected data values in form of so-called *guards*. If a condition evaluates to true, the corresponding transition can be taken and the associated action is executed.

Figure 5.11 shows an example of a multidimensional actor including a communication state machine together with one-dimensional and multidimensional ports. It forwards the incoming data from the multidimensional port i_1 to either the multidimensional port o_1 or o_2, depending on the control value present on the one-dimensional port i_c. As i_1 is a multidimensional port, i_1 (1) means that the action associated with the transition requires access to one sliding window of the multidimensional FIFO connected to port i_1. The corresponding window position is implicitly given by the scanning order of the multidimensional FIFO and is hence not specified in the state machine. Therefore, i_1 (1) is true, if all data elements required for the next sliding window operation are available on port i_1. Similarly, o_1 (1) is true if the buffer associated with the multidimensional channel can accept the next effective token. Its position is once again implicitly given by the scanning order of the multidimensional FIFO. The size of both the effective token and the sliding window is defined by the corresponding WDF FIFO parameters.

The data dependency of the actor is expressed in the state machine by taking into account the value of the control token: i_c (1) && (i_c [0] == 0) evaluates to true if at least one token is available on the one-dimensional channel associated with port i_c and if the value of the first token is zero. Furthermore, the selection of the next transition can depend on the current sliding window position within the current local schedule period as defined in Definition 5.6. To this end, each multidimensional input and output port provides a method *pos* () that returns the current window or token position.[6] $i_1.pos\,() = (0,0)^T$, for instance, means that the current sliding window is situated in the upper left corner of the current schedule period or image. For the given example actor in Fig. 5.11, this construct is used in order to ensure that state changes can only occur at the beginning of a schedule period (which shall equal a single image in this example). If all conditions for a transition are true, then the corresponding actor functionality, here $copy2o_1$ or $copy2o_2$, is executed in order to move to the next state. They consume the requested data from the input FIFO connected to the input port i_1 and write it to the FIFO attached to the corresponding output port.

5.5 Windowed Synchronous Data Flow (WSDF)

The above-described model of computation offers the advantage to cover both one-dimensional and multidimensional algorithms with and without control flow. Although this presents a great benefit compared to related work presented in Sections 3.1.3 and 3.1.4, this has to be paid by difficult analysis and verification techniques. In fact, since the communication state machine might contain arbitrarily complex guards, automatic analysis is extremely challenging if not impossible. Unfortunately, this contradicts the user's needs identified in Section 2.5, such as system level analysis for determination of required buffer sizes as well as formal validation of the data flow specification.

Unfortunately, data-dependent algorithms elude this kind of analysis as shown by Buck [50] (see also Section 3.1.3.6). However, most image processing systems contain important

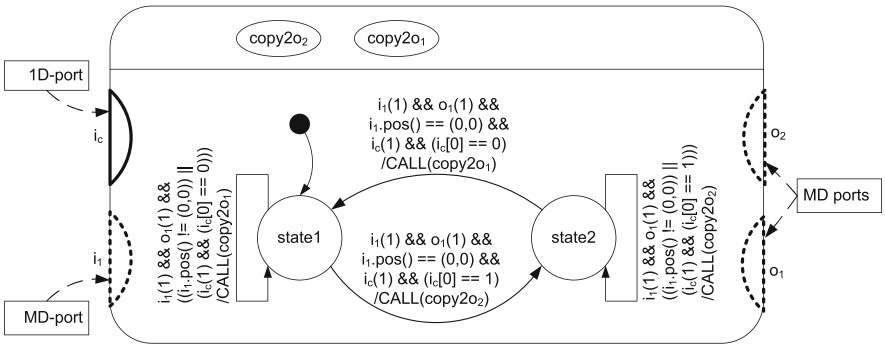


Fig. 5.11 WDF actor including a communication finite state machine together with one-dimensional and multidimensional ports

[6] More precisely spoken, *pos* () returns the value of the hierarchical iteration vector defined later on in Section 6.2. It describes the current window or token position by taking the communication order into account.

parts that are purely static and that can thus be analyzed and verified by much more powerful methods. Section 5.6, for instance, presents a method for quick verification whether a multidimensional data flow model is consistent and whether it allows execution within bounded memory. And Chapter 7 introduces a new approach for fast scheduling and buffer analysis in case of complex multirate algorithms and out-of-order communication. Their successful application, however, requires the data flow model to obey the following restrictions:

1. The arrays transported on all edges $e \in E$ of the data flow graph have the same dimensionality. In other words, $n_e = n$.
2. Each actor invocation induces communication on all input and output ports.
3. All input and output ports are connected to multidimensional FIFOs and show the same communication order.

The first restriction is not of any importance and just simplifies notation because an array with $n_e < n$ dimensions can always be embedded into n dimensions. Restriction 2 causes all data flow actors to be static. Restriction 3 finally leads to affine data dependencies, which is important for buffer analysis.

Figure 5.12 illustrates the problem of different communication orders for one actor. The latter transforms an image of 3×10 pixels into one of size 5×6. Although both images show thus the same number of pixels, data dependencies are very complex. In fact, output pixel $(x, y)^T$ depends on input pixel

$$\left((y \times 5 + x) \bmod 3, \left\lfloor \frac{y \times 5 + x}{3} \right\rfloor \right)^T .$$

However, this function is non-linear. Unfortunately, none of the buffer-analysis techniques presented in Section 3.3 supports such kinds of algorithms because they make the problem tremendously more complex. Furthermore, for practical applications, such situations can be easily avoided by different communication orders of different actors.

Figure 5.13a, for instance, shows a transpose operation occurring when calculating the two-dimensional inverse discrete cosine transform. It performs an image rotation and leads thus to different read and write orders for the transpose actor. Fortunately, this can be easily overcome by replacing the transpose actor with a simple copy actor as depicted in Fig. 5.13b when changing the read order of the sink actor. In contrast to Fig. 5.13a, this kind of model is supported by the buffer analysis presented in Chapter 7.

In other words, the above restrictions not only cover a wide range of applications but also permit efficient system level analysis. For precise distinction between the general windowed data flow and its static variant, the latter is called *windowed synchronous data flow (WSDF)*. This is in accordance with the terminology used for *synchronous data flow* defining a static data flow model of computation. Due to missing data-dependent operations, it enables enhanced automatic system level analysis. Thus, for efficient system level design, it is important to know which parts of the system follow WSDF semantics. Note, however, that the designer is not obliged to declare each actor whether it obeys the rules of WDF or WSDF. Instead, an actor can be automatically classified belonging to the WSDF model of computation, if the following holds:

1. All communication orders for all ports of an actor must be identical. In other words, the communication order can also be regarded as an actor property, which is called *invocation order*.

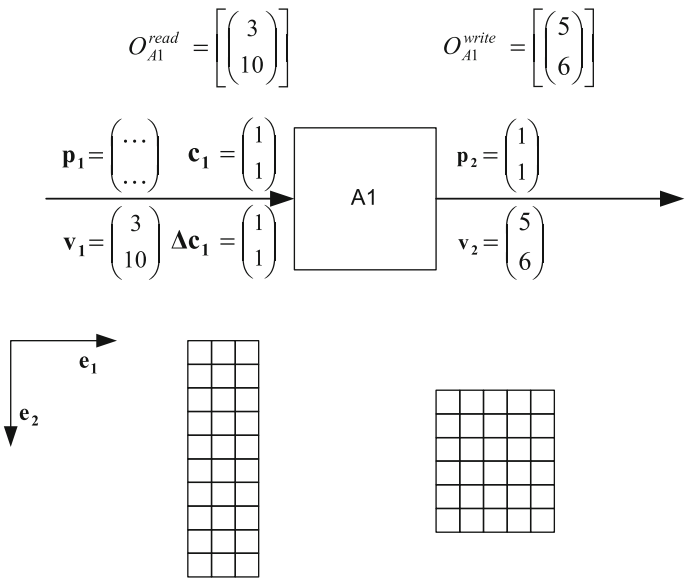


Fig. 5.12 Problem of different communication orders

2. The communication state machine can be reduced to a single state and each transition induces communication on every port. Further details about this aspect as well as possible restrictions can be found in [312].

Once such a classification has been performed, it enables enhanced system level analysis. As a first example, the next section will introduce the WSDF balance equation. It helps to verify whether the corresponding data flow graph can be executed within finite memory. As this is of course a prerequisite for any implementation, it can be considered as an important possibility for system level verification. Later on, Chapter 7 will present a further example of powerful system level analysis in form of automatic buffer size determination.

5.6 WSDF Balance Equation

Implementation of a WSDF graph is only possible if it can be executed within finite memory. In other words, it must be possible to construct a schedule in form of a sequence of actor firings $S = [a_1, a_2, \ldots]$ that does not cause unbounded token accumulation. Unfortunately, not all graphs obey this condition. Consequently, this section develops a corresponding necessary and sufficient condition for memory-bounded execution of a WSDF graph.

Definition 5.9 *WSDF-Schedule*
A *schedule* for a WSDF-Graph G is a sequence of actor firings

$$S = [a_{j_1}, a_{j_2}, \ldots, a_{j_k}, \ldots].$$

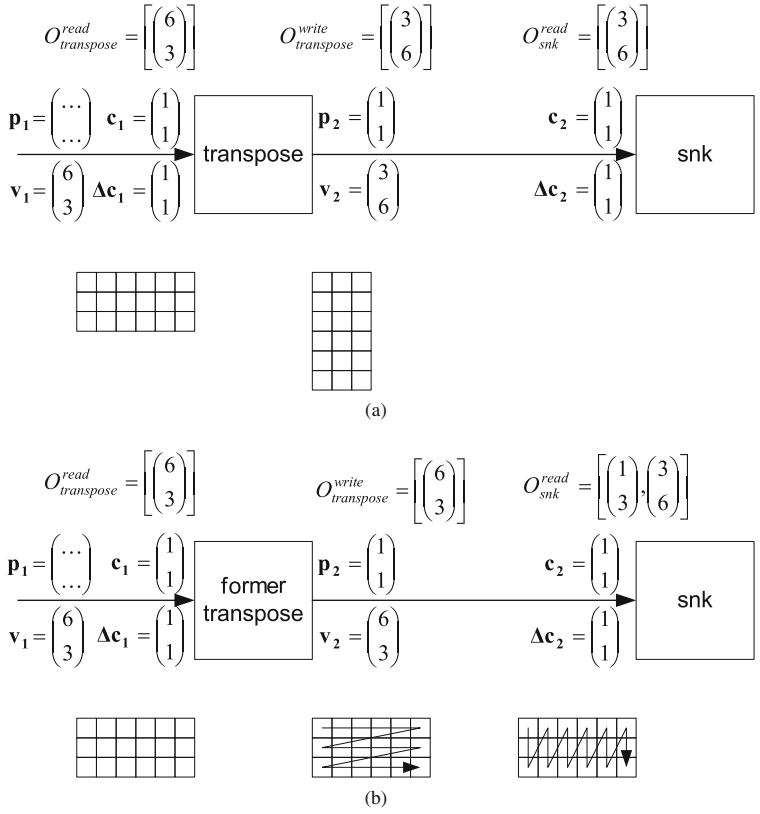


Fig. 5.13 Different modeling approaches for a transpose operation. **a** Rotate image. **b** Different communication orders for different actors

The sequence $[j_1, j_2, \ldots, j_k, \ldots]$ defines which actor to execute in the kth step. The current read and write positions are indirectly given by the invocation orders as defined in Section 5.3. The schedule can be of infinite or finite length. A finite schedule is repeated periodically, where each repetition is called a *schedule period*.

As infinite sequence lengths are not practical for real implementations, finite schedules play a dominant role in data flow-based design. However, this is only possible when finding a finite schedule sequence S that, starting from a given graph state, can be repeated infinitely and that does not cause a net change of the data elements stored in each edge buffer.[7]

In order to guarantee that the schedule sequence can be repeated infinitely often, it must return the WSDF graph into its initial state. More precisely, each effective token and sliding window to produce or read next must be at an equivalent position to the situation before starting the schedule sequence. This means that on each edge $e \in E$ and for each

[7] If such a schedule sequence cannot be found, the number of data elements stored in the edge buffers has to change permanently. As they are, however, positive, the only possibility is a—possibly slow—convergence toward infinity.

dimension i, $1 \leq i \leq n$, the source and sink must have produced and consumed the same number of virtual tokens.[8] Taking, for instance, the example given in Fig. 5.4b, the source actor has to produce eight virtual tokens in horizontal and two virtual tokens in vertical direction, whereas the sink has to consume the same number. Then the next schedule period can apply exactly the same sequence of actor firings than the previous one without increasing the required buffer size. This leads to the following definition:

Definition 5.10 Periodic WSDF Schedule

A *periodic WSDF schedule* is a finite schedule that invokes each actor of the graph G at least once and that returns the graph into its initial state. In other words, it must not cause a net change in the number of stored data elements on each edge $e \in E$, and both the source and the sink actor have to return to the equivalent of their initial firing positions.

In other words, such a periodic schedule can be repeated infinitely as long as it is valid.

Definition 5.11 Valid WSDF Schedule

A *valid WSDF schedule* is a periodic WSDF schedule that does not cause a graph deadlock. A deadlock occurs, if an actor planned for execution by schedule S is not fireable because not all required input data are available.

Definition 5.12 Consistency of WSDF graph

A WSDF graph is called *consistent*, if and only if it has at least one valid WSDF schedule.

However, unfortunately there exist WSDF graphs for which it is not possible to derive such periodic schedules. In other words, their implementation requires infinite memory, which is of course not practical. Consequently, the following sections derive a necessary and sufficient condition for a periodic schedule, the so-called *WSDF balance equation*. Note, however, that a solution to the latter is not sufficient for the existence of a valid periodic schedule. This has to be checked apart using, for instance, the methods proposed in [160, 193] (see also Section 3.1.3.1).

5.6.1 Derivation of the WSDF Balance Equation

Section 5.2 has already introduced a corresponding condition for valid communication on a single WSDF edge. However, this is not sufficient when considering a complete graph. Since an actor might have several inputs and outputs, it has to be taken care that communication is valid on all edges $e \in E$ of the WSDF graph. This is performed by taking care that all actors read complete virtual tokens on each input edge. Additionally, for each edge the number of produced and consumed virtual tokens must be identical for each dimension $1 \leq i \leq n$.

In order to take care that an actor $a \in A$ reads complete virtual tokens on each input, the *minimal actor period* $\langle \mathbf{L}(a), \mathbf{e_i} \rangle$ represents the minimal number of invocations in dimension i

$$\forall 1 \leq i \leq n : L_i(a) := \langle \mathbf{L}(a), \mathbf{e_i} \rangle = \mathrm{scm}_{e \in E,\, \mathrm{snk}(e)=a} \left(\langle \mathbf{r}_{\mathrm{vt}}(e), \mathbf{e_i} \rangle \right). \tag{5.4}$$

scm stands for the smallest common multiple, where $\mathrm{scm}(\emptyset) = 1$. Consequently, for an actor $a \in A$ that is not sink of any edge, this yields to

[8] WSDF currently requires that virtual tokens are produced and consumed completely. In other words it is not allowed to process a virtual token only half.

$$\forall 1 \leq i \leq n : L_i(a) = 1.$$

Example 5.13 Figure 5.14 shows an example graph consisting of three actors. Actor a_1 and actor a_3 are pure source actors. As a consequence, we get $\mathbf{L}(v_1) = \mathbf{L}(v_3) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. Actor a_2 is the sink of two edges e_1 and e_2 with the following repetition count for one single virtual token:

$$\mathbf{r}_{\mathrm{vt}}(e_1) = \begin{pmatrix} \frac{6-3}{1} + 1 \\ \frac{4-3}{1} + 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \end{pmatrix}, \ \mathbf{r}_{\mathbf{vt}}(e_2) = \begin{pmatrix} \frac{5-3}{1} + 1 \\ \frac{5-3}{1} + 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \end{pmatrix}.$$

By means of Eq. (5.4), we can derive for the minimal actor period

$$\mathbf{L}(v_2) = \begin{pmatrix} \mathrm{scm}(4, 3) \\ \mathrm{scm}(2, 3) \end{pmatrix} = \begin{pmatrix} 12 \\ 6 \end{pmatrix}.$$

Thus, after 12 invocations in horizontal direction and 6 in vertical direction, complete virtual tokens have been consumed on both input edges.

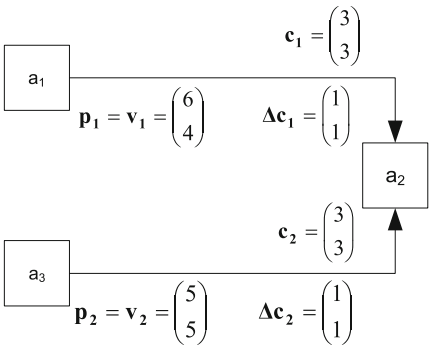


Fig. 5.14 WSDF example graph for explication of the actor period

After having guaranteed that each actor only reads complete virtual tokens, the following theorem can also be used to ensure that for each edge, the number of produced and consumed data elements is equal. Otherwise, the graph would lead to either a deadlock or unbounded data element accumulation. Such a graph is hence not *balanced*, because it cannot be implemented in real systems.

Theorem 5.14 *WSDF Graph Balance Equation*
Given a valid WSDF-Graph $G = (A, E, \mathbf{p}, \mathbf{v}, \mathbf{c}, \Delta\mathbf{c}, \delta, \mathbf{b}^{\mathbf{s}}, \mathbf{b}^{\mathbf{t}})$, *such that* $\forall e \in E \wedge \forall 1 \leq i \leq n : \langle \mathbf{r}_{vt}(e), \mathbf{e_i} \rangle \in \mathbb{N}$. *Then, the number of actor invocations in dimension* i *in order to return the WSDF graph to a state equivalent to the initial one can be calculated by*

$$\mathbf{r_i} = M_i \times \mathbf{q_i}, \ \textit{with} \ M_i = \begin{bmatrix} L_i(a_1) & 0 & \cdots & 0 \\ 0 & L_i(a_2) & & \\ \vdots & & \ddots & \\ 0 & & & L_i(a_{|A|}) \end{bmatrix}. \tag{5.5}$$

Both $\mathbf{r_i}$ *and* $\mathbf{q_i}$ *are strictly positive integer vectors.*
$\mathbf{r_i}\left(a_j\right) := \langle \mathbf{r_i}, \mathbf{e_j} \rangle$ *is the number of invocations of actor* a_j *in dimension* i.
$\mathbf{q_i}\left(a_j\right) := \langle \mathbf{q_i}, \mathbf{e_j} \rangle$ *defines, how often the minimal period of actor* a_j *in dimension* i *is executed and can be calculated by*

$$\underbrace{\begin{bmatrix} \Gamma_{1,1,i} & \cdots & \Gamma_{1,a,i} & \cdots & \Gamma_{1,|A|,i} \\ \vdots & & \vdots & & \vdots \\ \Gamma_{e,1,i} & \cdots & \Gamma_{e,a,i} & \cdots & \Gamma_{e,|A|,i} \\ \vdots & & \vdots & & \vdots \\ \Gamma_{|E|,1,i} & \cdots & \Gamma_{|E|,a,i} & \cdots & \Gamma_{|E|,|A|,i} \end{bmatrix}}_{\Gamma_i} \times \mathbf{q_i} = \mathbf{0}, \tag{5.6}$$

with

$$\Gamma_{e,a,i} = \begin{cases} L_i\left(a\right) \times \langle \mathbf{p_e}, \mathbf{e_i} \rangle & \text{if } a = \operatorname{src}\left(e\right) \\ 0 & \text{otherwise} \end{cases} - \begin{cases} \frac{L_i(a)}{\langle \mathbf{r}_{vt}(e), \mathbf{e_i} \rangle} \times \langle \mathbf{v_e}, \mathbf{e_i} \rangle & \text{if } a = \operatorname{snk}\left(e\right) \\ 0 & \text{otherwise} \end{cases}. \tag{5.7}$$

Proof Equation (5.5) ensures that for each actor $a \in A$, the number of invocations is a multiple of its minimal period. Otherwise, some windows would have different positions relative to the corresponding virtual token. Since sliding window algorithms, however, show cyclo-static behavior in that the number of newly required input pixels depends on this position (see Section 3.1.3.1), the resulting graph state would not be equivalent to the initial one.

Equation (5.6) ensures for each edge $e \in E$ that the number of produced and consumed data elements is identical. To this end, Eq. (5.7) calculates the number of produced or consumed data elements per minimal actor period. Two different cases are distinguished:

1. $a = \operatorname{snk}\left(e\right)$:
   The execution of one minimal period of actor a corresponds to $L_i\left(a\right)$ actor invocations. Since the consumption of one virtual token needs $\langle \mathbf{r_{vt}}\left(e\right), \mathbf{e_i} \rangle$ actor firings, $\frac{L_i(a)}{\langle \mathbf{r_{vt}}(e), \mathbf{e_i} \rangle} \in \mathbb{N}$ virtual tokens are consumed. For each virtual token,

   $$\langle \mathbf{c_e}, \mathbf{e_i} \rangle + (\langle \mathbf{r_{vt}}\left(e\right), \mathbf{e_i} \rangle - 1) \times \langle \Delta c_e \times \mathbf{e_i}, \mathbf{e_i} \rangle = \langle \mathbf{v}, \mathbf{e_i} \rangle + \langle \mathbf{b^s} + \mathbf{b^t}, \mathbf{e_i} \rangle$$

   data elements are consumed (see Corollary 5.1). However, only $\langle \mathbf{v_e}, \mathbf{e_i} \rangle$ of them have been produced by the source actor, the others are introduced by the virtual border extension.
2. $a = \operatorname{src}\left(e\right)$:
   The execution of one minimal period of actor a corresponds to $L_i\left(a\right)$ actor invocations. For each source actor invocation, $\langle \mathbf{p_e}, \mathbf{e_i} \rangle$ data elements are produced.

If Eq. (5.6) holds, then the number of produced and consumed tokens is identical. As we execute each actor an integer multiple times of its minimal actor period, we are sure that only complete virtual tokens are consumed. Consequently, the same is valid for production and the WSDF graph returns into a state equivalent to the initial one.

As the topology matrix in Eq. (5.6) has exactly the same properties as for SDF graphs, it can be derived from [193] that for a connected graph there exists a strictly positive integer vector $\mathbf{q_i}$, if and only if

$$\operatorname{rank}(\Gamma_i') = |A| - 1.$$

Consequently, also $\mathbf{r_i}$ is integer and strictly positive. For each dimension i, the smallest possible integer vector $\mathbf{q_i}$ defines the minimal repetition vector $\mathbf{r_i^{min}}$ whose components correspond to the minimal number of actor invocations that are necessary to return the WSDF graph into a state equivalent to the initial one. In other words, each actor $a_j \in A$ has to fire $\langle k_i \times \mathbf{r_i^{min}}, \mathbf{e_j} \rangle$ times in dimension i, with $k_i \in \mathbb{N}$. Otherwise, not all produced data elements are consumed, which can lead to unbounded memory requirements. Invocation orders violating this constraint can thus be rejected by the system level design tool. In other words, the balance equation can be used for high-level verification of a given WSDF specification.

5.6.2 Application to an Example Graph

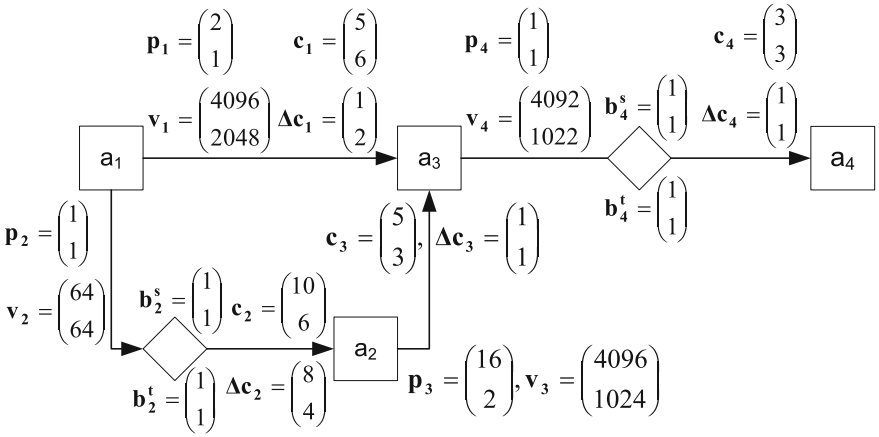


Fig. 5.15 Example WSDF graph for illustration of the balance equation

Figure 5.15 shows an example WSDF graph. The graph behavior is borrowed from image processing, however generalized in some aspects in order to show all elements of the WSDF model. Actor a_1 produces two output images. The first one has a size of 4096 × 2048 pixels. The size of the second image shall be half the width and is cut into sub-images of 64 × 64 pixels each. Actor a_2 takes these blocks, performs a complex downsampling in vertical direction and an upsampling in horizontal direction, and combines the resulting blocks to an image with 4096 × 1024 pixels. Actor a_3 consumes the image of actor a_1 and downsamples it in vertical direction. Furthermore, it consumes one image of actor a_2. Both the images from actor a_1 and actor a_2 are processed with a sliding window of size 5 × 6 and 5 × 3, respectively. The result is passed to actor a_4, which performs a sliding window operation with virtual border extension.

5.6.2.1 Actor Periods

First of all, we have to calculate the minimal periods of each actor. The corresponding results are shown in Tables 5.1 and 5.2.

Table 5.1 Repetition counts for consumption of a single virtual token, based on the graph shown in Fig. 5.15

Edge	(a_1, a_3)	(a_1, a_2)	(a_2, a_3)	(a_3, a_4)
$\mathbf{r}_{\mathrm{vt}}(e)$	$\begin{pmatrix}4092\\1022\end{pmatrix}$	$\begin{pmatrix}8\\16\end{pmatrix}$	$\begin{pmatrix}4092\\1022\end{pmatrix}$	$\begin{pmatrix}4092\\1022\end{pmatrix}$

Table 5.2 Minimal actor periods for the example shown in Fig. 5.15

Actor	a_1	a_2	a_3	a_4
$\mathbf{L}(v_j)$	$\begin{pmatrix}1\\1\end{pmatrix}$	$\begin{pmatrix}8\\16\end{pmatrix}$	$\begin{pmatrix}4092\\1022\end{pmatrix}$	$\begin{pmatrix}4092\\1022\end{pmatrix}$

The topology matrices for both dimensions allow the calculation of the vectors $\mathbf{q_i}$:

$$\underbrace{\begin{bmatrix}2 & 0 & -4096 & 0\\ 1 & -64 & 0 & 0\\ 0 & 128 & -4096 & 0\\ 0 & 0 & 4092 & -4092\end{bmatrix}}_{\Gamma_1} \times \begin{pmatrix}q_{1,1}\\q_{1,2}\\q_{1,3}\\q_{1,4}\end{pmatrix} = \mathbf{0}$$

$$\underbrace{\begin{bmatrix}1 & 0 & -2048 & 0\\ 1 & -64 & 0 & 0\\ 0 & 32 & -1024 & 0\\ 0 & 0 & 1022 & -1022\end{bmatrix}}_{\Gamma_2} \times \begin{pmatrix}q_{2,1}\\q_{2,2}\\q_{2,3}\\q_{2,4}\end{pmatrix} = \mathbf{0}.$$

It is easy to verify that

$$\mathrm{rank}\,(\Gamma_1) = \mathrm{rank}\,(\Gamma_2) = 3 = |A| - 1.$$

The minimal number of actor invocations in multiples of actor periods can be calculated to

$$\mathbf{q_1} = \begin{pmatrix}q_{1,1}\\q_{1,2}\\q_{1,3}\\q_{1,4}\end{pmatrix} = \begin{pmatrix}2048\\32\\1\\1\end{pmatrix},\ \mathbf{q_2} = \begin{pmatrix}q_{2,1}\\q_{2,2}\\q_{2,3}\\q_{2,4}\end{pmatrix} = \begin{pmatrix}2048\\32\\1\\1\end{pmatrix}.$$

As a consequence, the minimal repetition vectors amount

$$\mathbf{r_1^{min}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 4092 & 0 \\ 0 & 0 & 0 & 4092 \end{bmatrix} \times \mathbf{q_1} = \begin{pmatrix} 2048 \\ 256 \\ 4092 \\ 4092 \end{pmatrix}$$

$$\mathbf{r_2^{min}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 16 & 0 & 0 \\ 0 & 0 & 1022 & 0 \\ 0 & 0 & 0 & 1022 \end{bmatrix} \times \mathbf{q_2} = \begin{pmatrix} 2048 \\ 512 \\ 1022 \\ 1022 \end{pmatrix}.$$

$$O_{a_1} = \left[\begin{pmatrix} 2048 \\ 2048 \end{pmatrix}\right],\ O_{a_2} = \left[\begin{pmatrix} 6 \\ 16 \end{pmatrix}, \begin{pmatrix} 256 \\ 512 \end{pmatrix}\right],\ O_{a_3} = O_{a_4} = \left[\begin{pmatrix} 4092 \\ 1022 \end{pmatrix}\right]$$

would thus be valid invocation orders. On the other hand

$$O_{a_1} = \left[\begin{pmatrix} 2048 \\ 2048 \end{pmatrix}\right],\ O_{a_2} = \left[\begin{pmatrix} 6 \\ 16 \end{pmatrix}, \begin{pmatrix} 256 \\ 512 \end{pmatrix}, \begin{pmatrix} 512 \\ 1024 \end{pmatrix}\right]$$
$$O_{a_3} = O_{a_4} = \left[\begin{pmatrix} 4092 \\ 1022 \end{pmatrix}\right]$$

is not permitted, since actor a_2 would require more input data elements than produced by actor a_1. Consequently, the graph would deadlock. Similarly,

$$O_{a_1} = \left[\begin{pmatrix} 2048 \\ 2048 \end{pmatrix}\right],\ O_{a_2} = \left[\begin{pmatrix} 6 \\ 16 \end{pmatrix}, \begin{pmatrix} 256 \\ 512 \end{pmatrix}\right],\ O_{a_3} = O_{a_4} = \left[\begin{pmatrix} 2046 \\ 511 \end{pmatrix}\right]$$

is not allowed, since in this case, actor a_4 would not consume all data elements, leading thus to unbounded token accumulation on edge e_4.

5.7 Integration into SystemCoDesigner

The model of computation described so far precisely defines the communication behavior of data flow actors independently of any programing language. Nevertheless, such an application model can only develop its full usefulness, if it can also be executed for analysis and debugging purposes. In other words, a design environment needs to be provided that enables simple specification, execution, debugging, and analysis of multidimensional data flow graphs. To this end, different frameworks have been proposed in literature, such as the Java-oriented *Ptolemy* (see Section 3.5.2), *Metropolis* (see Section 3.5.3), the *DIF* package (see Section 3.1.1), or the *SystemC*-based SysteMoC library. The latter is part of the SystemCoDesigner described in Chapter 4 and has been exemplarily extended in order to describe a possible implementation strategy of the previously introduced WDF model of computation. As a result, it is possible to implement complex WDF graphs on top of the *SystemC* library including the support of one-dimensional and multidimensional communication.

Figure 5.16 illustrates the way how to specify WDF actors using the library elements of SysteMoC. The exemplified actor performs an image rotation of 180°. It is derived from the SysteMoC class `smoc_actor` (line 1) and contains two two-dimensional ports (line 3). Input ports perform by default constant border extension (see Section 5.1) with a value of 0. However, alternative border processing methods like symmetric border extension

are also available and can be activated by corresponding template arguments (not shown in Fig. 5.16). Furthermore, parameters can be propagated to the border processing algorithms by corresponding constructor parameters. The actions called during state transitions are defined by simple C++ functions as exemplified in lines (8)–(11). The current read or write position can be obtained by the `iteration` method of the ports. The data belonging to a sliding window or an effective output token are accessed by squared brackets whose coordinates are interpreted relative to the current sliding window or effective token position. The communication parameters like sliding window size **c** or the communication order are declared in the constructor of the actor (lines 18–24). `ul_vector_init` provides a quick method to generate parameter vectors. The communication parameters are propagated to the connected multidimensional FIFO assuring correct WDF communication. Lines (26)–(37) finally declare the different transitions of the state machine. In this case, the predicates verify the current write position via `getIteration`[9] and the availability of input data and output space. For the illustrated actor, the communication state machine consists only of one single state labeled `start`. Thus, each transition returns to exactly this state (lines 31 and 37). `in(0,1)` declares a so-called non-consuming read. In other words, the sliding window is read, but not moved to the next position. `in(1)`, on the other hand, causes both reading and moving to the next position.

The so-defined actors can be interconnected to complex WDF graph topologies. In order to speed up simulation as much as possible, the instantiated multidimensional FIFOs use a very simple memory organization. The buffer is realized as a rectangular array, which has the same extent than the token space (see Definition 5.7) except for dimension $\mathbf{e}_{\mathbf{n_e}}$ where it might be smaller or larger. Memory is only freed in complete hyperplanes orthogonal to $\mathbf{e}_{\mathbf{n_e}}$. Whereas this increases the memory required for simulation, it makes memory management less complex and thus faster. For the final hardware implementation, however, a more efficient memory management can be used as explained in Chapter 8.

5.8 Application Examples

During the explanation of windowed data flow, already various simple examples like sliding windows, block building, image rotation, or the transpose operation have been introduced. This section addresses more complex applications in order to illustrate the benefits of the model of computation. In particular, it shows that the restriction to rectangular windows and hierarchical line-based communication orders does not impact the applicability to less regular algorithms. Moreover, it emphasizes the benefit of covering both static and control-intensive algorithms.

5.8.1 Binary Morphological Reconstruction

The binary morphological reconstruction [294] is an algorithm that extracts objects from monochrome images. It has been selected as case study because it is easily understandable,

[9] Due to implementation details, the current read or write position within an action is returned via the `iteration` method. On the other hand, in the definition of the communication state machine, the function `getIteration` has to be used.

```
1  class m_wdf_rot180_ex: public smoc_actor {
2  public:  /* Ports */
3   smoc_md_port_in<int,2> in; smoc_md_port_out<int,2> out;
4  private:
5    /* Internal states */
6    const unsigned size_x;  const unsigned size_y;
7    /* Action */
8    void copy_pixel() {
9      unsigned int x = out.iteration(0,0);
10     unsigned int y = out.iteration(0,1);
11     out[0][0] = in[size_x-x-1][size_y-y-1]; }
12   /* States of communication state machine */
13   smoc_firing_state start;
14 public:
15   m_wdf_rot180_ex( sc_module_name name,
16                    unsigned size_x, unsigned size_y)
17     : smoc_actor(name, start),
18       in(ul_vector_init[1][1], //comm. order
19          ul_vector_init[size_x][size_y], //c
20          ul_vector_init[size_x][size_y], //delta_c
21          sl_vector_init[0][0], //bs
22          sl_vector_init[0][0]), //bt
23      out(ul_vector_init[1][1], //p
24          ul_vector_init[size_x][size_y]), //comm. order
25      size_x(size_x),size_y(size_y) {
26      start = /* First transition */
27              (in(0,1) && out(1))
28              >> ((out.getIteration(0,0) != size_x-1) ||
29                  (out.getIteration(0,1) != size_y-1))
30              >> CALL(m_WDF_rot180_ex::copy_pixel)
31              >> start
32            /* Second transition */
33            | (in(1) && out(1))
34              >> ((out.getIteration(0,0) == size_x-1) &&
35                  (out.getIteration(0,1) == size_y-1))
36              >> CALL(m_WDF_rot180_ex::copy_pixel)
37              >> start;
38      ...
```

Fig. 5.16 Specification of a multidimensional actor in SystemoC

however very challenging concerning its modeling requirements, because it contains static and dynamic points, local, and global algorithms. Additionally, it uses non-rectangular sliding windows.

5.8.1.1 Definition of the Binary Morphological Reconstruction

The binary morphological reconstruction is fed with two equally sized $w \times h$ monochrome input images, the *mask image* $I_m : D \to \{0, 1\}$ and the *marker* or *seed image* $I_s : D \to \{0, 1\}$, where $D = \{1, \ldots, w\} \times \{1, \ldots, h\}$ is the so-called *discrete image domain,* a vector set of all pixel coordinates. Both images can contain one or several *connected components* formed by white pixels, i.e., the value 1. As a result, the binary morphological reconstruction returns the mask image where all components are eliminated that have not been marked by a white pixel in the seed image.

Figure 5.17 gives a corresponding example, where the mask I_m contains three connected components, from which only those marked by the seed I_s are copied to the resulting image. The latter can be calculated by different methods [294], namely (i) iterative dilatation, (ii) iterative dilatation with two passes, and (iii) FIFO based.

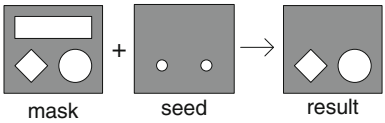


Fig. 5.17 Exemplary binary morphological reconstruction

5.8.1.2 Calculation by Iterative One-Dilatation

The one-dilatation is a local algorithm that can be implemented by means of a 3×3 sliding window. Let $I : D \to \{0, 1\}$ be the input image with $D = \{1, \ldots, w\} \times \{1, \ldots, h\}$, and let $N_G(\mathbf{p}) = \{\mathbf{y} \in D \mid \|\mathbf{y} - \mathbf{p}\|_\infty \leq 1\}$ be the set of pixels belonging to the sliding window at position $\mathbf{p} \in D$, where $\|\mathbf{x}\|_\infty = \max_{1 \leq i \leq n} \{\langle \mathbf{x}, \mathbf{e_i} \rangle\}$ is the maximum norm. Then, the resulting image K of the one-dilatation is defined as follows:

Algorithm 1 Binary reconstruction by iterative dilatation

```
I = I_s
Repeat until stability{
  dil: K(p) = max{I(q) | q ∈ N_G(p)}
  min: I(p) = min(K(p), I_m(p))
}
```

$$\forall \mathbf{p} \in D : K(\mathbf{p}) = \max\{I(\mathbf{q}) \mid \mathbf{q} \in N_G(\mathbf{p})\}.$$

In other words, all components of the input image are extended by 1 pixel in all directions.

The resulting algorithm for the binary morphological reconstruction [294] is given in Algorithm 1. It extends all components of the seed image I_s by a one-dilatation and cuts all pixels that overlap the mask I_m. This is performed until no pixel modification occurs anymore. In other words, it contains a data-dependent control flow, which cannot be represented by purely static models of computation.

Figure 5.18 shows the WDF graph for the morphological reconstruction by iterative dilatation. The triangles correspond to initial tokens, dashed lines to one-dimensional communication. Due to reasons of simplicity, actors for data duplication are not drawn. Furthermore, communication parameters are omitted.

The graph reads two images, the mask and the seed. The latter is processed by a one-dilatation, followed by the calculation of the pointwise minimum with the mask image (see Algorithm 1). The *difference* and the *maximum* actors determine, whether those processing steps have caused any change in the seed image by calculating the maximum occurring difference. If a change has been detected, the *output switch* and the *seed selector* forward the result of the *minimum* actor to the *dilatation* actor for a new iteration. The corresponding mask is duplicated by the *mask selector*. If the *difference* actor cannot detect any difference

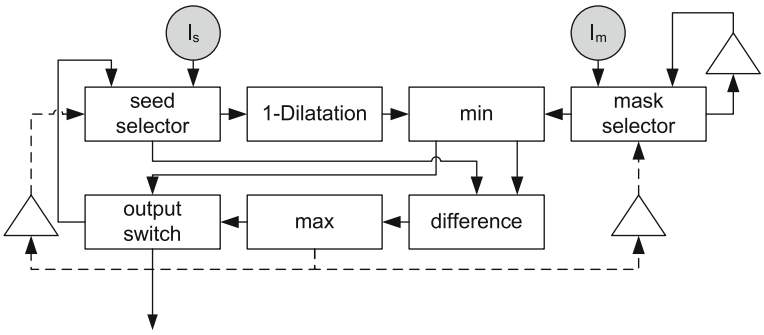


Fig. 5.18 Simplified data flow graph for binary reconstruction by iterative dilatation

to the previous seed image, then the *output switch* outputs the result and the *mask* and s*eed selectors* read two new images.

The *mask* and *seed selectors* as well as the *output switch* are data-dependent actors, because the decision which image to read respectively to which port to write depends on the control values determined by the *maximum actor*. This is necessary in order to model the "repeat until" loop contained in Algorithm 1. The control value is a one-dimensional token, whereas all other communicated data are part of a multidimensional array representing the image.

The triangles specify one-dimensional or multidimensional initial tokens. The one-dimensional control token tells the *seed and mask selectors* to start by reading an input image delivered by the sources. The multidimensional initial data elements are used to delay the input mask for the next iteration. This is necessary, because the mask selector has to buffer the mask image I_m generated by the source for a possible next iteration with the same mask. In order to be accessible to a possible buffer analysis algorithm, the required memory is not hidden into the actor, but buffering is performed by storing it in the edge buffer marked with the triangle. However, whether this duplicated mask is indeed required can only be decided after the *minimum* actor has finished. In other words, the *mask selector* has to buffer the mask on the edge without knowing whether it is really required. As a consequence, each time it reads a new mask generated by the source, it also has to discard the last buffered mask image because it will not be used anymore. Hence, for the first mask image that is read by the mask selector, also an image to discard must be provided. This can be realized by an initial image.

5.8.1.3 Iterative Dilatation with Two Passes

After having restricted to rectangular windows in the previous section, it will be shown next, how the concepts of multidimensional communication can be extended to less regular applications, thus forming a powerful base for many image processing applications. For this purpose, let us consider the binary morphological reconstruction based on iterative dilatation with forward and backward pass [294]. It has the advantage to be much quicker than the algorithm described in Section 5.8.1.2. This is because the modifications performed by a dilatation step are immediately taken into account by writing it directly into the seed image I_s. In contrast, the algorithm described in Section 5.8.1.2 only updates the seed image after having completely processed the mask image.

Forward Pass

Similar to the one-dilatation, the forward pass extends the seed image and cuts pixels that extend the mask. However, the modified pixel values are immediately taken into account for the next sliding window position. Let I_{2i+0} be the input image in the ith iteration of the forward pass and the initial image be the seed image, i.e., $I_0 = I_s$. Then the resulting image I_{2i+1} of the forward pass is defined as follows:

$$\forall \mathbf{p} \in D: \quad I_{2i+1}(\mathbf{p}) = \tag{5.8}$$

$$\min(\, I_m(\mathbf{p}) \,, \tag{5.9}$$

$$\max(\, \{I_{2i+0}(\mathbf{p})\} \cup \tag{5.10}$$

$$\{I_{2i+1}(\mathbf{q}) \mid \mathbf{q} \in N_G^+(\mathbf{p}) \setminus \{\mathbf{p}\}\})). \tag{5.11}$$

$N_G^+(\mathbf{p})$ stands for a non-rectangular neighborhood of 5 pixels around a position $\mathbf{p}$ and is illustrated in Fig. 5.19a. Gray-shaded window pixels represent already calculated values, where $\mathbf{p}$ refers to the currently processed pixel as indicated by the previous equation.

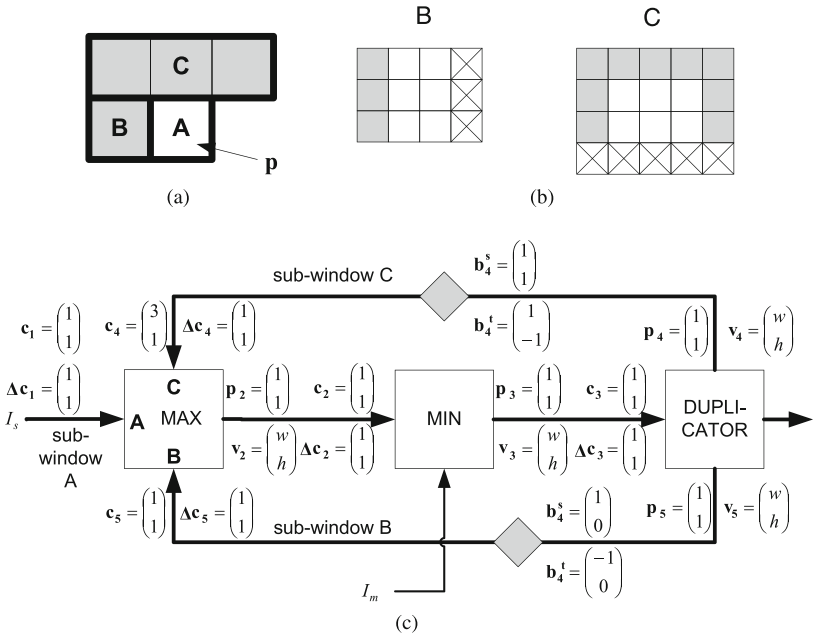


Fig. 5.19 **a** Irregular window and **c** its realization in WDF. **b** shows the border processing assuming an input image size of 3×3 pixels. $\mathbf{p}$ refers to the currently processed pixel (see Eq. (5.11)). Crossed data elements are suppressed

This non-rectangular window traverses the seed image in raster-scan order. In order to fit into the WDF modeling approach, the neighborhood has to be split into three different sub-windows. Window A samples the original seed image, whereas windows B and C work on the already modified pixel values due to the immediate access to updated pixels in Eq. (5.11).

This fact is modeled by two feedback loops in Fig. 5.19c. They clearly show that the sub-windows B and C work on the already updated data. The two feedback loops only differ by the applied border processing. This is shown in Fig. 5.19b, assuming an image size of 3×3 pixels. Gray-shaded pixels belong to the extended border, crossed data elements are suppressed. Whereas for sub-window B, the image is extended on the left border and reduced on the right one, sub-window C modifies all four borders. Note that the extended border is not generated by the source actor (see Section 5.1).

The data flow representation in Fig. 5.19c already allows to detect possible throughput problems due to the tight feedback loop when the sum of the execution times of the three actors is too large [247]. In fact, since sub-windows B and C start on the extended border, the *dilatation* actor can execute exactly once. Then, however, the corresponding result has to be processed by the *minimum* actor and the *duplicator* before the next data element for sub-window B is available. Consequently, the invocation frequency of the *dilatation* actor, and hence the throughput, depends on the execution times of the *minimum* and the *duplication* actor.

Backward Pass

In order to deliver correct results, the forward pass has to be followed by a second run with a seed and mask image rotated about 180°. This so-called backward pass calculates the images $I_{2i+0}(\mathbf{p})$ (see Eq. (5.11)). Figure 5.20 shows the corresponding extract of the data flow graph performing the rotation. For reasons of simplicity, it is assumed that I_s is already rotated and focus on processing of I_m. Since the rotation needs the last pixel of the input image in order to generate the first output pixel, it represents a global image processing algorithm. These algorithms are very expensive in terms of buffer requirements. Since they read and write a complete image as a whole, both edges e_1 and e_2 need to buffer a complete image.

The minimum actor, however, is only a point algorithm. Hence, it has no benefit at all from being able to access the complete image as a whole. In order to solve this problem, a non-consuming read of input ports can be used. In other words, the actor *rot180* is able to read several times the same input image. Each time it produces only one output pixel, which is immediately written to edge e_2. In order to generate the correct pixel output order, *rot180* internally keeps track of the next pixel position to read as depicted in Fig. 5.16. Once the complete input image has been processed, it informs the multidimensional channel to discard the current image.

By means of this modeling approach, the required memory on edge e_2 reduces from a complete image to a single pixel, because the *minimum* actor does not need to wait until the complete image has been rotated. Hence, the overall rotation only requires the buffering of the input image, which leads to an overall buffer reduction of factor 2.

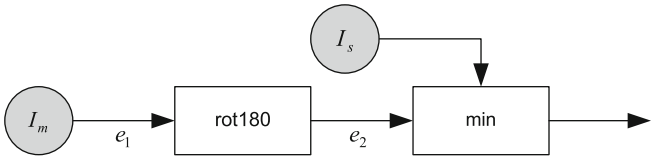


Fig. 5.20 Data flow graph showing the rotation about 180°

Algorithm 2 FIFO-based morphological reconstruction

```
(00) init: ∀p ∈ D{
(01)   if (I_s(p) = 1 and I_m(p) = 1)
(02)     if (∃q ∈ N_G(p), I_s(q) = 0)
(03)       fifo_add(p)
(04) }
(05) prop: while (fifo not empty){
(06)   p ←fifo_first()
(07)   ∀q ∈ N_G(p){
(08)     if (I_s(q) = 0 and I_m(q) = 1)
(09)       I_s(q) = 1, fifo_add(q)
(10)   }
(11) }
```

5.8.1.4 FIFO-Based Morphological Reconstruction

The algorithms discussed in Sections 5.8.1.2 and 5.8.1.3 use a more or less regular sampling of the images in raster-scan order. However, there exists also global algorithms that require random access to the different image pixels. This is typical for strongly data-dependent algorithms like image segmentation. The FIFO-based morphological reconstruction, proposed in [294], is another example.

Algorithm 2 shows the corresponding pseudocode. It consists of two phases, the initialization phase and the propagation phase. *fifo_add* adds the coordinates to a one-dimensional FIFO, *fifo_first* retrieves the coordinates from there.

As can be seen from Algorithm 2, the order by which the pixels of the mask and the seed image are accessed is data dependent and hence cannot be predicted. In particular, lines (06) and (07) result in "jumping windows" whose size is defined by N_G. However, such a behavior has a tremendous impact on data reuse and scheduling such that many advantages like high-speed communication synthesis described in Chapter 8 would be lost.

Consequently, instead of extending the WDF model of computation accordingly, it is preferable to abstract from the actual pixel access order by modeling this algorithm as a global one that needs access to a complete image. This perfectly matches with the final implementation because due to irregular data access, the seed and mask images have to be stored in huge random access memories. By declaring the FIFO-based binary morphological reconstruction as a global algorithm, this property is perfectly visible in the abstract model.

Figure 5.21 shows the corresponding data flow graph. In a first step, the *reconstruction* actor reads the seed image pixel by pixel and copies it to the output port by performing non-producing writes. In other words, the pixels are added to the output image without releasing it for access by the sink. Next, the initialization phase of Algorithm 2 is traversed. Whereas the mask image is directly accessed from the corresponding input port using non-consuming reads, the seed image can be retrieved from the output buffer, since the contained data have not already been forwarded to the sink. In other words, the multidimensional output FIFO acts as a temporary storage. The same happens during the propagation phase. Only after termination of the propagation phase, the output image is dispatched such that the sink can read it.

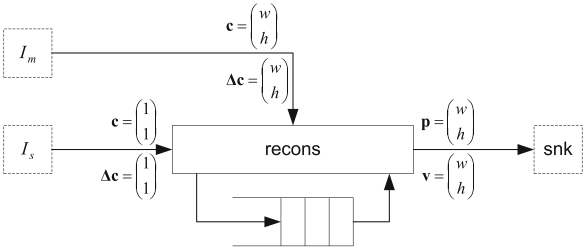


Fig. 5.21 WDF graph for FIFO-based reconstruction. w and h stand for the image width and height, respectively

5.8.2 Lifting-Based Wavelet Kernel

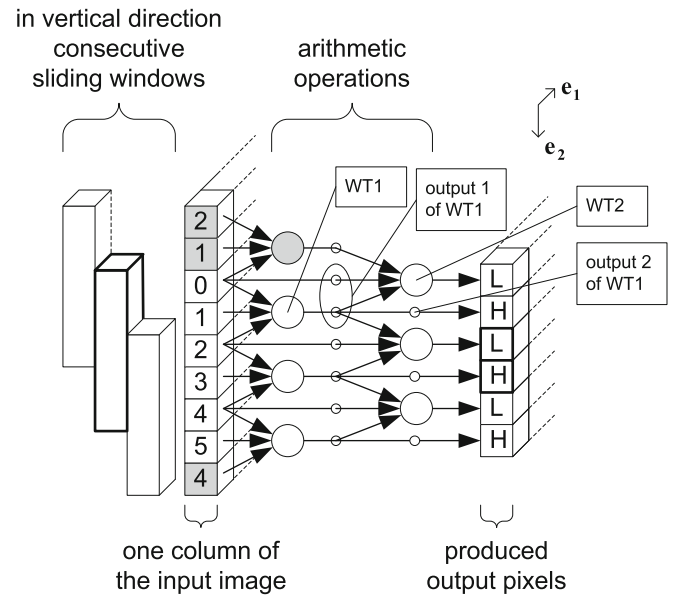
As shown in the previous section, WDF can be used to model complex applications containing point, local, and global algorithms with various degrees of control flow and even irregular sliding windows. In order to clarify the concepts, this section targets another complex image processing application, namely lifting-based two-dimensional wavelet transform with resource sharing.

Figure 5.22 resumes the principles of the lifting scheme from Section 2.2. It depicts a one-dimensional (vertical) wavelet transform assuming a 5–3 kernel as proposed by the JPEG2000 standard [152]. As discussed in Section 2.2, the input image is sampled with a filter that consists of 5×1 pixels and that moves by 2 pixels. For each filter position, two output pixels, a low-pass and a high-pass coefficient, are generated. Corresponding border extension takes care that the operation is fully reversible.

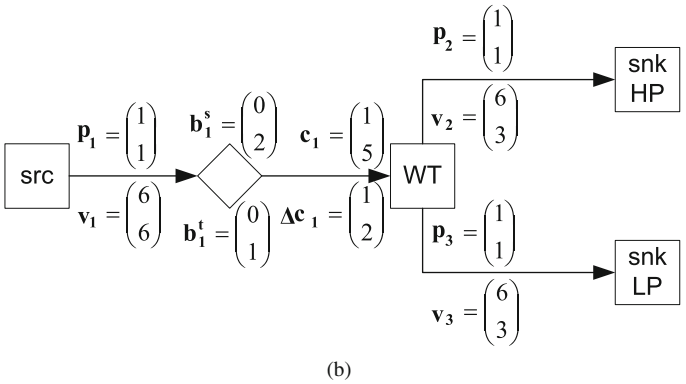
Such a filter can be easily modeled in WSDF as shown in Fig. 5.22b. However, it requires buffering of 4 lines and 1 pixel, whereas an optimal implementation only needs 3 lines and 1 pixel. For the bold filter position, for instance, lines 2 and 3 and the intermediate result labeled by WT1 have to be stored into a corresponding memory in order to be able to generate both the low-pass and the high-pass output.

Fortunately, the same behavior can be represented in WSDF by splitting the filter into two filter stages as depicted in Fig. 5.23. The first stage calculates the high-pass result on output 2. Furthermore, it forwards the same value to filter stage 2, together with the input value. Based on this information, filter stage 2 can calculate the low-pass result. Implementation of WT1 requires a storage buffer of 2 lines and 1 pixel, whereas WT2 only needs 1 line and 2 pixels due to parallel data production. In other words, the lifting-based modeling significantly reduces the necessary memory capacities. The configuration of the required border extension can be derived from Fig. 5.22a. From these building blocks, the two-dimensional wavelet transform can be obtained by concatenation of a one-dimensional horizontal and vertical processing step.

Since each of the vertical processing steps requires intermediate buffering, the JPEG2000 standard [152] envisages so-called *tiling* (see also Section 2.2). This means that the input image is divided into partial images, which are processed completely independently. This reduces the line width and the corresponding memory requirements for temporal buffering. From the definition of the WSDF communication given in Section 5.1, it can be easily seen that this corresponds exactly to the division of the input image into several virtual tokens, because both sampling with sliding windows and virtual border extension are performed independently for each virtual token. Figure 5.24 shows the corresponding extension of the



(a)



(b)

Fig. 5.22 **a** Lifting scheme for a one-dimensional vertical wavelet transform. Gray-shaded data elements belong to the extended border. **b** Corresponding WSDF graph

WSDF graph depicted in Fig. 5.23. This time, the source generates an image of 12 × 12 pixels in raster-scan order. The *tiler* reads this image by a window of size 1 × 1, but uses a different consumption order and divides it into 2 × 2 tiles. Note that the consumption and production orders are identical for all ports of an actor, such that they are annotated as actor properties. These tiles are now processed by the wavelet transform in a sequential manner, thus reducing the effective line width and so the required buffer size. Note, however, that the tiling itself represents an out-of-order communication, which is expensive in terms of memory (see also Chapter 6). Nevertheless, tiling might be interesting when taking into account that it can be implemented using an external memory while the wavelet transform requires parallel

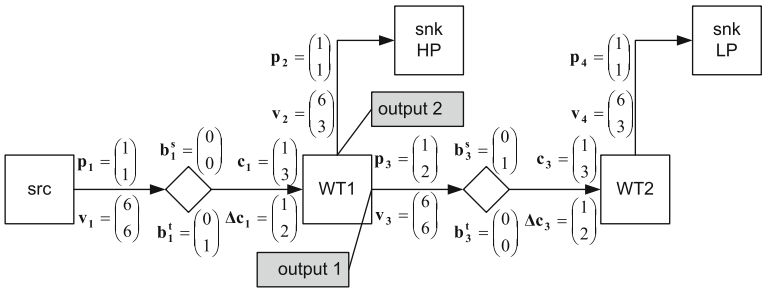


Fig. 5.23 Lifting-based one-dimensional vertical wavelet transform in WSDF

data access, which can be better handled by on-chip storage. Thus, tiling provides a tradeoff between off-chip and on-chip memory requirements.

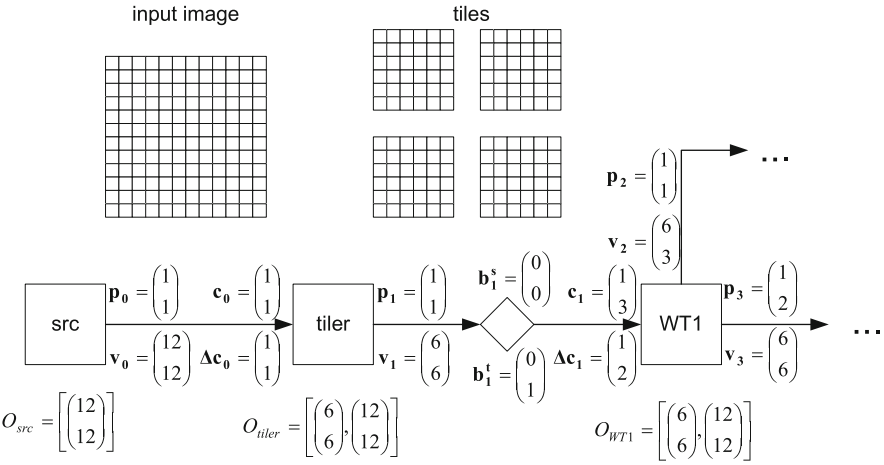


Fig. 5.24 Extension of the lifting-based wavelet transform by tiling

The explanations given so far have assumed only one decomposition level. In other words, the input image is only processed by one wavelet filter. However, as already indicated in Section 2.2, the resulting LL subband typically is once again processed by a new wavelet stage in order to improve the spatial decorrelation of the image pixels. Figure 5.25 again depicts the corresponding scenario for two decomposition levels. As can be seen, each decomposition delivers a resulting LL subband with only half the height (and width) of the input. In other words, let h be the height of the input image. Then the following equation derives an upper bound for the number of lines belonging to all LL subbands:

$$h \times \left(\frac{1}{2} + \frac{1}{4} + \ldots\right)$$
$$= h \times \sum_{j=1}^{\infty} \left(\frac{1}{2}\right)^{j}$$

$$= h \times \lim_{j \to \infty} \left(\frac{1 - \left(\frac{1}{2}\right)^{j+1}}{1 - \frac{1}{2}} - 1 \right)$$
$$= h.$$

Thus, the line numbers of all occurring LL subbands are less or equal than the input image height h. Note that the formula assumed that all subbands show an even height. If this is not the case, resource sharing is still possible, but requires more complex scheduling (see also Section 2.4). From Fig. 5.23, it can be seen that both WT1 and WT2 are only occupied every second line due to the vertical downsampling. Consequently, during these idle lines, the other wavelet decompositions can be calculated, reducing thus the required hardware resources. Figure 5.26 depicts the corresponding WDF graph for two decomposition levels. For reasons of simplicity, it omits the horizontal transform, but assumes that the vertical transform is directly fed into the next wavelet stage. The input image enters the WT1 module on edge e_1 and is decomposed into a high-pass part on edge e_2 and a low-pass part on edge e_3. The latter is further processed by WT2 and fed back to WT1 via edge e_4. This edge stores the arriving pixels until WT1 is ready to process the corresponding subband. As soon as this is true, the data were read from edge e_4 and decomposed into a high-pass part on edge e_7 and a low-pass part on edge e_5. WT2 finally generates the final LL subband.

In contrast to the previous cases, this time WT1 and WT2 contain a communication state machine controlling from which port to read and to which port to write. Different alternatives are imaginable, ranging from multiplexing on pixel level up to more coarse-grained scheduling. In case of the manual designed JPEG2000 encoder (see Section 2.2), it has been preferred to only interleave lines in order to reduce the number of required multiplexers. Such a scheduler can also be realized in WDF by querying the positions of the read windows and write tokens (see Section 5.4.2). Such a model would have allowed to verify the architecture on a higher level of abstraction compared to RTL as done in the manual implementation. By this means, it would have been easier possible to detect an architectural error in that odd image heights require special handling, because in this case the LL subband height has to be

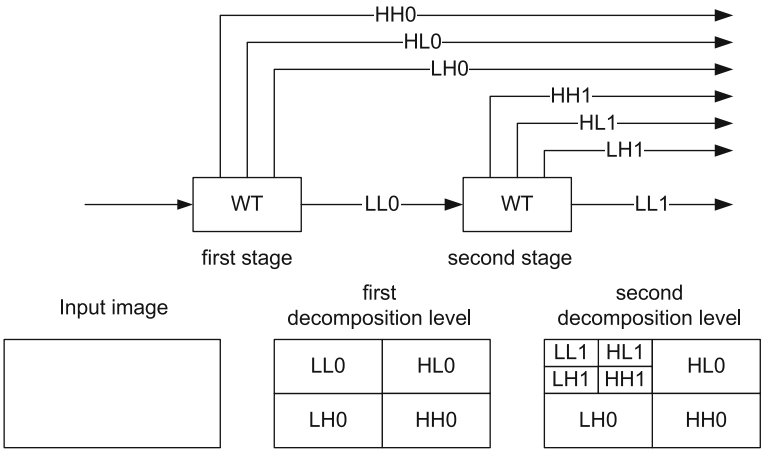


Fig. 5.25 Spatial decorrelation with two decomposition levels

rounded up. Furthermore, it would have allowed to investigate different schedule alternatives improving thus the final implementation and shortening the design time.

5.9 Limitations and Future Work

Although WDF is restricted to sampling with rectangular sliding windows, Section 5.8 has explained the capability of the WDF model of computation to represent complex applications. This includes in particular enhanced buffering techniques used for the lifting-based wavelet transform, non-rectangular sliding windows that directly modify the input image, and out-of-order communication. Consequently, WDF offers important capabilities for data flow-based system level design of image processing applications. Nevertheless, conducted research also revealed potential for improvements that is worth to be considered in future work. By these means it would be possible to extend the applicability of the modeling methodology and to assure highly efficient system implementation. Consequently, the following paragraphs shortly discuss some limitations and potential for future work.

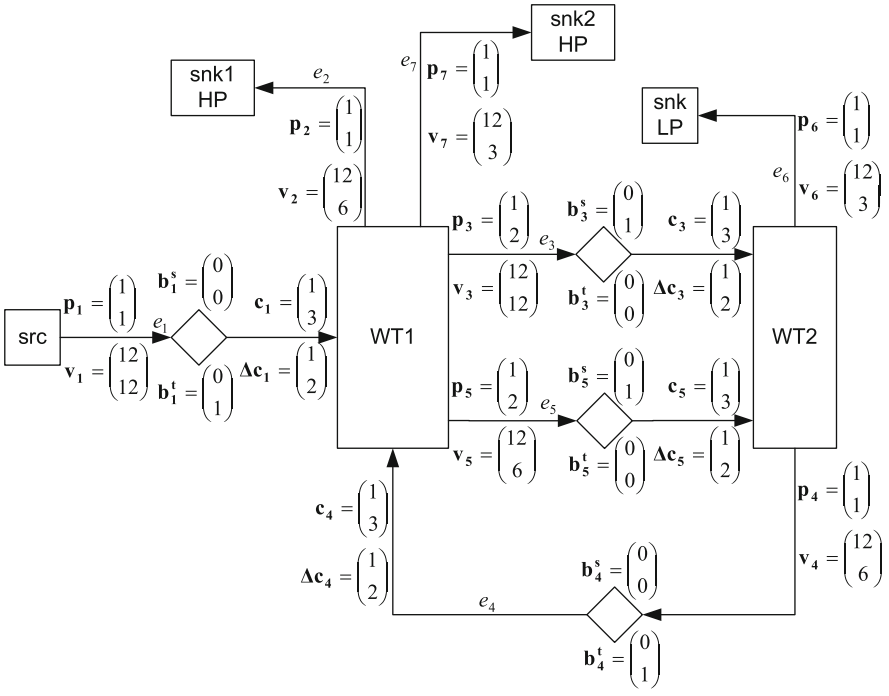


Fig. 5.26 Lifting-based wavelet resource sharing. Due to illustration purposes, only the vertical transform is depicted. WT1 and WT2 represent the lifting stages as introduced in Fig. 5.23. The input pixels enter the first lifting stage via edge e_1 and are forwarded to the second stage by means of edge e_3. The feedback edge e_4 permits to reuse the computational resources for the second decomposition level. In this case, the data are forwarded to the second wavelet stage WT2 via edge e_5. The high-pass and low-pass result coefficients are processed by the corresponding sink actors

The most important aspect to solve can be found in supporting parametric multidimensional applications. They are characterized by the fact that certain parameters such as the image extensions are not known at compile time, but can only be determined during run-time. Corresponding examples can, for instance, be found in a JPEG2000 decoder that is requested to support different image sizes. Consequently, the concrete dimensions to use cannot be set during compile time, but have to be derived from the JPEG2000 codestream header. WDF on the other hand currently does not support these scenarios since it assumes all parameters to be static. One possibility to solve these difficulties consists, for instance, in combining WDF with the parameterized data flow framework discussed in Section 3.1.3.4. This and other alternatives are, however, out of scope for this monograph.

In addition to these parametric applications, further potential for improvement can also be found in allowing for more complex out-of-order communication than what have been presented in Section 5.3. It introduced the so-called *firing blocks*, based on which the following chapters will elaborate methods for efficient buffer analysis and high-speed communication synthesis. In order to reduce their complexity, Eq. (5.3) requests that all firing blocks of a given firing level have the same size. Unfortunately, there exist applications that do not fulfill these requirements, as, for instance, the JPEG2000 compression of an image whose size equals 2048×1080 pixels. If the latter shall be divided into code-blocks of size 64×64, it is obvious that the last row of code-blocks only contains $1080 \bmod 64 = 56$ rows. Since currently unequal firing blocks are not permitted, this situation must be handled by extending the input image with a corresponding border. However, it might be more efficient to relax the condition formulated in Eq. (5.3). Since this significantly complicates both memory analysis and communication synthesis, it is not further considered in this book. The same holds for sliding window algorithms that use different window movements $\Delta\mathbf{c}$ depending on the current image position as found in a H.264 deblocking filter [248]. Though the corresponding behavior could be imitated by a corresponding finite state machine it might be interesting whether direct support of different window movements leads to better system efficiency.

5.10 Conclusion

As discussed in Section 3.1, a huge amount of specification techniques have been proposed in literature, ranging from sequential languages including loop constructs to various data flow models of computation. In this context, multidimensional data flow seems to be a promising candidate for data-intensive applications operating on arrays, since it combines several benefits such as block-based specification, applicability of graph algorithms for system analysis, and fine-grained data dependency analysis for efficient implementation. It is thus surprising that the existence and application of multidimensional data flow in both industrial products and academic research is currently rather minor. This is partly due to the fact that offered solutions miss important capabilities, such as representation of overlapping windows (MDSDF), control flow (CRP), out-of-order communication (IMEM), virtual border extension, or tight interaction with one-dimensional models of computation, equally required for description of complete systems.

The windowed data flow (WDF) of computation as discussed in this chapter offers all these capabilities, which makes it particularly adapted for point, local, and global image processing algorithms. It provides new communication semantics for modeling of sliding window algorithms including initial data elements and virtual border extension. Inclusion of read and write orders permits to represent data reordering occurring in many image pro-

cessing applications. The so-resulting communication channels can be interpreted as special FIFOs, also called *multidimensional FIFOs*. This not only helps to model data-dependent decisions but also enables tight interaction between one-dimensional and multidimensional data flow models. Static parts of the overall system can be identified by analysis of the communication state machine. Such parts permit advanced system analysis. The balance equation, for instance, helps to detect specification errors leading to unbounded token accumulation.

All these aspects can be advantageously used for system level design as illustrated by two case studies. Application to various implementation alternatives of the binary morphological reconstruction not only illustrated the need for control flow but also explained how irregular sliding windows and instantaneous modification of the input image can be covered with WDF. Also non-standard sliding window algorithms like the lifting-based wavelet transform can be easily described. In this context, WDF not only is able to exploit optimized buffer schemes but also covers tiling and even resource sharing allowing for efficient high-level simulation and verification.

Chapter 6
Memory Mapping Functions for Efficient Implementation of WDF Edges

Selection of a good memory architecture is a crucial step during embedded system design, since it heavily influences the required chip size, achievable throughput, and occurring power dissipation. This is particularly true in case of image processing applications because of the huge amounts of data that have to be processed.

Reference [109], for instance, reports that clock distribution and memory belong to the most energy-consuming parts of a typical ASIC. In case of a digital audio broadcast (DAB) chip, memory contributes to 66% of the global power dissipation [140]. Similar values are reported for an H.264/AVC core [201], where SRAM has been responsible for 70% of the energy requirements. Usage of external RAM is even more expensive due to the necessary energy-consuming driver circuits. Consequently, the memory subsystem typically accounts for 50–80% of the power consumption [59, 291, 305]. One possibility to limit these power requirements consists in trying to reduce the amount of required buffer size, since individual memory accesses become more expensive with increasing capacities [17, 59, 60, 109, 245]. Essentially, this can be explained by the leakage power of each individual memory cell as well as by the row decoding [109].

Storage size reduction, however, is beneficial not only due to power considerations but also due to economical aspects, because memory represents an important cost factor. Reference [201], for instance, designs a fully scalable MPEG-2 and H.264/AVC video decoder for QCIF images. Although the image sizes are rather small (176×144), and despite an additional external SD-RAM, SRAM occupies approximately a quarter of the overall chip, which is a significant portion.

Finally, memory requirements heavily influence achievable performance. Taking, for instance, real-time compression of videos for digital cinema production, a single image requires a storage capacity of 303.75 Mbit.[1] However, this by far exceeds the chip-internal memory available in modern FPGAs. A Xilinx *XC4LX100*, for instance, offers approximately 4 Mbit memory, whereas a Xilinx *XC5VLX330T* is equipped with approx. 11 Mbit of available memory. Thus, such a buffer capacity can only be provided in form of external memory, whose access is much slower compared to on-chip memory. This is particularly true as the latter permits parallel data accesses, which is not true anymore for external RAM.

Thus, usage of small buffer sizes is of triple interest. First, it helps avoiding external memory, which increases achievable throughput and simplifies the layout of the printed circuit board due to the reduced number of required chips. Second, it decreases necessary chip sizes

[1] $4096 \times 2160 \times 3 \times 12\ bit$, assuming three components and 12 bit precision.

J. Keinert, J. Teich, *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*, Embedded Systems,
DOI 10.1007/978-1-4419-7182-1_6, © Springer Science+Business Media, LLC 2011

and thus costs. And finally, it helps to reduce energy consumption, because larger memories lead to more expensive individual accesses.

Consequently, minimization of the required memory is an important step in the design of image processing applications. Its results, however, do depend not only on the application itself but also on the chosen implementation strategy, in particular the memory mapping function. In fact, for one and the same algorithm, the amount of required memory varies depending on which memory model is chosen for the implementation. Thus, memory minimization includes two aspects, namely selection of a suitable memory mapping function and determination of the resulting memory requirements for a given image processing application. Whereas the latter aspect is deferred to Chapter 7, this chapter searches for a buffer model that helps for efficient implementation of a W(S)DF communication edge. To this end, this chapter addresses the following aspects:

- Comparison of two different memory mapping schemes applicable to WDF edges
- Description of an analysis method for buffer size determination by simulation Due to its linear complexity in terms of produced tokens, it can be applied to large-sized images
- Investigation of out-of-order communication and non-linear schedules. The latter help to employ arbitrary scheduling strategies like self-timed parallel execution or ASAP (as soon as possible) scheduling

The remainder of this chapter is as follows. Section 6.1 starts to explain the exact problem to solve. Next, Section 6.2 presents some mathematical preliminaries that are required for the discussion of two different memory models as done in Section 6.3. Section 6.4 finally presents the results obtained for different application scenarios. Note that the employed buffer analysis method is not detailed in this chapter, but has been moved to Appendix A, because it is not required for further understanding.

6.1 Problem Formulation

As motivated above, minimization of the required memory is a crucial step in design of efficient image processing systems. Figure 6.1, for instance, shows an extract of a JPEG2000 encoder as discussed in Sections 2.2 and 5.8.2. It includes the image source, a tiler in order to split the input image into four smaller sub-images, and the two-dimensional wavelet transform. For reasons of simplicity, the latter has been combined to one module instead of splitting it into several lifting steps (see also Section 5.8.2). Each of these processing steps requires a given amount of memory depending on both the algorithm and the implementation strategy. Most of these buffer requirements result from temporary storage of data elements due to sliding windows, data reordering, and data synchronization. The latter occurs if an actor has several inputs and the corresponding data elements arrive at different times. Since WDF explicitly models point, local, and global image processing algorithms by means of sliding windows with different sizes, all these expensive buffers make part of the WDF communication edge buffers. In other words, each WDF edge has to provide a sufficient buffer space such that the sink actor can access all required data elements. The actors themselves typically only contain smaller amounts of memory like individual registers for temporary storage. As they can be determined during software compilation, behavioral synthesis, or manual RTL coding, the following explanations are limited to the communication edge buffers.

The task to solve in this chapter hence consists in storing the data elements transported on a WDF edge in such a way that it leads to small buffer requirements while keeping the

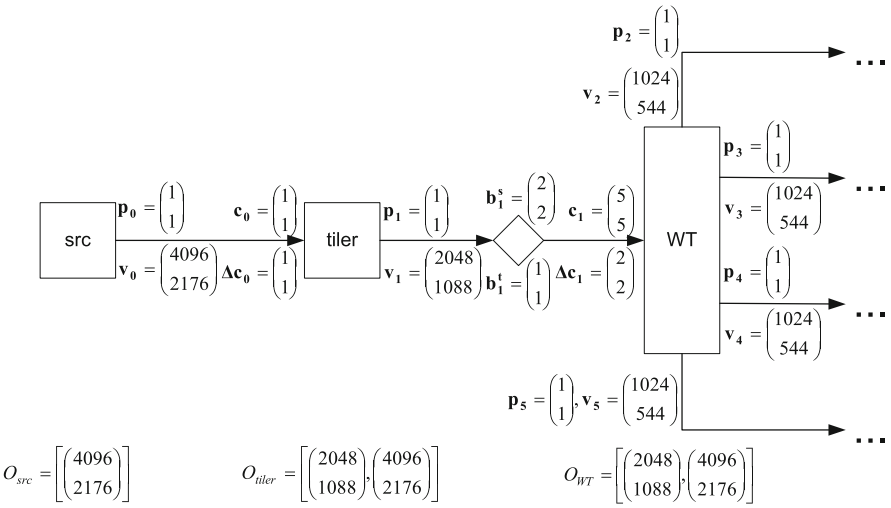


Fig. 6.1 Extract of a JPEG2000 encoder

overhead for address management acceptable. This placement of data elements into memory cells has to be performed such that no data elements are overwritten that will be required later on.

Figure 6.2 illustrates the corresponding problem by means of a typical sliding window algorithm (Fig. 6.2a) and a data reordering operation (Fig. 6.2b). In both cases, the gray-shaded data elements are those that have to be buffered at least on the WDF communication edge for the depicted sliding window and effective token positions, because otherwise they would miss later on. Assume, for instance, that the source in Fig. 6.2b has just written pixel 21, which is immediately read by the sink. Then, the edge buffer has to store just not only this pixel, but also the so-called *live* pixels 16, 17, and 18 as they have already been produced by the source and will be required later on.

Definition 6.1 A data element is called *live* at time t when it has been produced at time $t_p \leq t$ and will be read at time $t_c \geq t$.

Remark 6.2 Note that the data elements situated on the extended border do not influence the required memory size because they are not produced by the source actor. Instead their value is assumed to either have a predefined value or that it can be derived from the remaining sliding window data elements (see Section 5.1) Consequently, extended border pixels are not stored in the edge buffers.

Principally, the required storage capacity for each edge can be derived from the maximum amount of live data elements for all values of t during which the WDF graph is executed. However, this would require ideally packed storage buffers. In other words, the written pixels must be placed such into the memory that no "holes" or unused memory cells occur. Whereas for special cases like simple sliding window algorithms, this is pretty easy, a general solution requires dynamic memory management during run-time. Reference [320] proposes a corresponding solution in form of a content-addressable memory (CAM). However, implementation in hardware is very expensive. Instead, static mapping functions are typically used

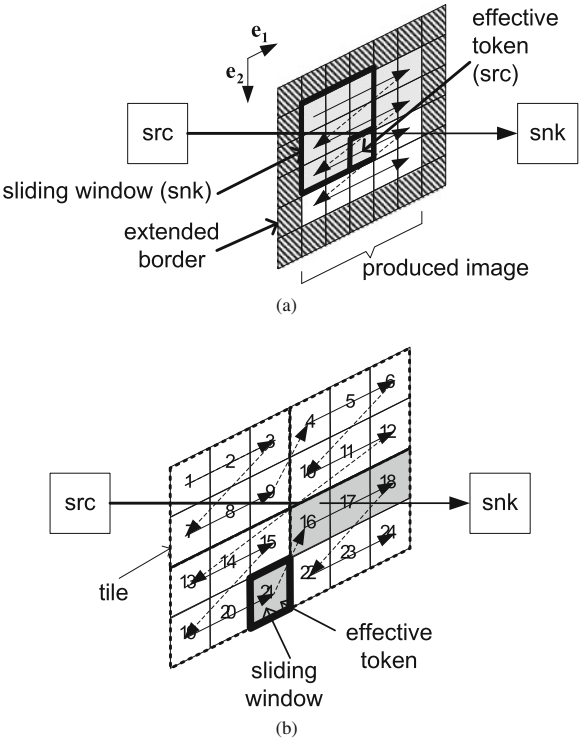


Fig. 6.2 Live data elements. Arabic numbers correspond to the production order while *arrows* indicate the read order. *Gray-shaded rectangles* represent the live pixels that have to be buffered at least in memory for the shown window positions. **a** Sliding window. **b** Out-of-order communication

$$\sigma : G_e \rightarrow \mathbb{N}^t, t \in \mathbb{N},$$

where G_e defines the set of all data elements processed on edge $e \in E$. These mapping functions are typically restricted to linear expressions and modulo-operations for simple implementation and assign to each data element a corresponding memory cell in a (hyper-) rectangular grid. This has to be performed in such a way that two live data elements are never placed to the same storage location. However, since σ is a static and mostly a piece-wise linear function, optimum packing is typically not achieved. In other words, the required memory size might be larger than the ideal one, whereas the overhead depends on the specific form of σ.

Consequently, the following sections will investigate two different memory mapping functions in order to compare them in terms of required buffer sizes when applied to WDF communication edges. To this end, a corresponding simulation environment has been created that is able to take arbitrary schedules into account and that is thus more flexible than analytic methods as discussed in Section 3.3.2.

However, before going into details, the following section discusses a read and write order enumeration scheme that includes the communication order and that is an important prerequisite for the rest of the chapter.

6.2 Hierarchical Iteration Vectors

As discussed in Section 5.2, WDF imposes some restrictions on valid communication such that it is possible to derive for each dimension i the number of read and write operations occurring within one (local) schedule period (see Definition 5.6). Consequently, each of them can be identified by an iteration vector $\mathbf{I}_{\text{snk}} \in \mathbb{N}^{n_e}$, respectively, $\mathbf{I}_{\text{src}} \in \mathbb{N}^{n_e}$. The set of read or write operations occurring in the jth (local) schedule period is given by

$$F(j) = \times_{i=1}^{n_e-1} \{0, \ldots, \langle \mathbf{I}_{\max}, \mathbf{e_i} \rangle\} \times \left\{ (j-1) \times \left(\langle \mathbf{I}_{\max}, \mathbf{e_{n_e}} \rangle + 1 \right), \ldots, j \times \left(\langle \mathbf{I}_{\max}, \mathbf{e_{n_e}} \rangle + 1 \right) - 1 \right\},$$

where $\mathbf{I}_{\max} + \mathbf{1}$ defines the number of read or write operations in each dimension as calculated by the local balance equation (see Corollary 5.5). Unfortunately, this enumeration does not contain any information about the actually occurring invocation order.

Figure 6.3 shows the corresponding problem for dividing an image into 2×2 tiles. Each rectangle corresponds to a read operation of the first (local) schedule period whereas the read order is indicated by corresponding arrows. As can be easily seen, there is no simple relation between the components of the iteration vectors $\mathbf{I}$ and the communication order.

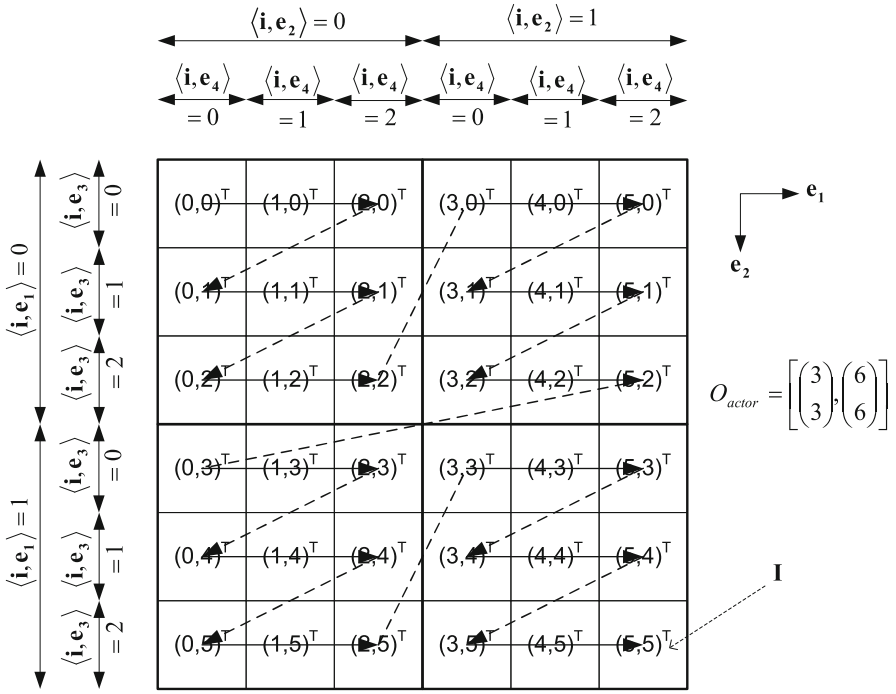


Fig. 6.3 Iteration vectors and their relation with the communication order

As this, however, complicates the following analysis, this section introduces the so-called *hierarchical iteration vectors* $\mathbf{i}$, which take the firing block hierarchy into account (see Defini-

tion 5.8). In order to distinguish them from the non-hierarchical, or *flat* ones, they are written in lowercase letters.

Definition 6.3 Given a WDF edge e with n_e-dimensional tokens and a sequence of firing blocks $O = [\mathbf{B_1}, \ldots, \mathbf{B_q}]$ with q firing levels (see Definition 5.8). Then the function $\theta_e : \mathbb{N}^{n_e} \rightarrow \mathbb{N}^{n_e \times q}$ maps the (flat) iteration vectors $\mathbf{I} \in \mathbb{N}^{n_e}$ to the so-called *hierarchical iteration vectors* $\mathbf{i} \in \mathbb{N}^{n_e \times q}$ with

$$\forall 1 \leq j \leq n_e,\ 0 \leq k < q : \langle \mathbf{i}, \mathbf{e_{n_e \times (q-k)-j+1}} \rangle = \left\lfloor \frac{\langle \mathbf{I}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_{k+1}}, \mathbf{e_j} \rangle}{\langle \mathbf{B_k}, \mathbf{e_j} \rangle} \right\rfloor \tag{6.1}$$

and with $\mathbf{B_0} = \mathbf{1}$.[2]

These hierarchical iteration vectors define to which firing block a read or write operation belongs to. Figure 6.3 shows their interpretation for the tiling example. $\langle \mathbf{i}, \mathbf{e_1} \rangle$ and $\langle \mathbf{i}, \mathbf{e_2} \rangle$ define to which tile the current read operation belongs to while $\langle \mathbf{i}, \mathbf{e_3} \rangle$ and $\langle \mathbf{i}, \mathbf{e_4} \rangle$ correspond to the coordinates relative to the tile borders.

Based on these definitions, the communication order can be expressed by the help of the lexicographic order.

Definition 6.4 Given a vector space $\mathscr{T} \subseteq \mathbb{R}^n$. Then $\prec$ is called *lexicographic order* on $\mathscr{T}$ when the following holds:

$$\mathbf{u_1}, \mathbf{u_2} \in \mathscr{T},\ \mathbf{u_1} \prec \mathbf{u_2} \Leftrightarrow \exists i, 1 \leq i \leq q : \\ \langle \mathbf{u_1}, \mathbf{e_i} \rangle < \langle \mathbf{u_2}, \mathbf{e_i} \rangle \wedge \forall 1 \leq j < i : \langle \mathbf{u_1}, \mathbf{e_j} \rangle = \langle \mathbf{u_2}, \mathbf{e_j} \rangle.$$

If $\mathbf{u_1} \prec \mathbf{u_2}$, $\mathbf{u_1}$ is called *lexicographic smaller* than $\mathbf{u_2}$.

Example 6.5 $(4, 2, 10)^T \prec (4, 3, 1)^T$.

Corollary 6.6 *Given two read or write operations* $\mathbf{i_1}$ *and* $\mathbf{i_2}$*. Then,* $\mathbf{i_1}$ *is executed before* $\mathbf{i_2}$ *if and only if* $\mathbf{i_1} \prec \mathbf{i_2}$.

Proof By definition.

Example 6.7 Consider the read operations $\mathbf{I_1} = (2, 2)^T$ and $\mathbf{I_2} = (3, 0)^T$ in Fig. 6.3. Application of Definition 6.3 leads to $\mathbf{i_1} = (0, 0, 2, 2)^T$ and $\mathbf{i_2} = (0, 1, 0, 0)$. Since $\mathbf{i_1} \prec \mathbf{i_2}$, $\mathbf{i_1}$ is executed before $\mathbf{i_2}$, which perfectly fits to the order depicted by the arrows.

With these definitions, it is now possible to introduce two different memory models describing for each WDF edge the mapping of data elements to memory cells.

6.3 Memory Models

As discussed in Section 6.1, the required memory size for each WDF communication edge does depend not only on the edge parameters and the employed schedule, but also on the memory mapping function $\sigma : G_e \rightarrow \mathbb{N}^{t_e}$, $t_e \in \mathbb{N}$. The

[2] The relation $\theta : \mathbf{I} \mapsto \mathbf{i}$ can be considered as a so-called *multidimensional schedule*.

latter associates with each data element $\mathbf{g_e} \in G_e$ produced on edge $e \in E$ a corresponding memory cell $\sigma(\mathbf{g_e})$, whereas $\sigma(\mathbf{g_e}) \in \mathbb{N}^{t_e}$ represents coordinates in a (hyper-)rectangular grid of memory cells. G_e is the set of all data elements produced on edge $e \in E$, also called *token space* in the following. These data elements can be identified by n_e-dimensional vectors $\mathbf{g_e} \in G_e \subseteq \mathbb{N}^{n_e}$. The relation between such a data element $\mathbf{g_e}$ and its producing write operation is given by

$$P_e : G_e \ni \mathbf{g_e} \mapsto \mathbf{I}_{\text{src}} \in \mathbb{N}^{n_e}, \tag{6.2}$$

with

$$\forall 1 \leq i \leq n_e : \langle \mathbf{I}_{\text{src}}, \mathbf{e_i} \rangle = \left\lfloor \frac{\langle \mathbf{g_e}, \mathbf{e_i} \rangle - \langle \boldsymbol{\delta}(e), \mathbf{e_i} \rangle}{\langle \mathbf{p}(e), \mathbf{e_i} \rangle} \right\rfloor$$

and $\mathbf{p}$ defining the effective token size and $\boldsymbol{\delta}$ the vector of initial data elements (see Section 5.1).

This section introduces two different mapping functions that shall be further investigated for usage on WDF edges. The first one corresponds to a rectangular memory model as used in [194] and the second one performs a linearization in production order. In both cases, this section assumes that all data elements of one communication edge shall be placed into one single memory module, and hence into one single address space. This assumption helps to simplify the following explanations. Later on in this book (see Sections 8.2.3 and 7.6.2), this condition will be relaxed in order to enable high-throughput hardware communication primitives and more efficient memory mappings. However, the basic principles remain the same and the techniques presented in this chapter can still be used by applying them individually for each memory module.

6.3.1 The Rectangular Memory Model

The rectangular memory model is established by the mapping function of the form $\sigma_e(\mathbf{g_e}) = \mathbf{g_e} \bmod \mathbf{m_e}$. It assigns to each data element represented by a data element identifier $\mathbf{g_e}$ a memory address $\sigma_e(\mathbf{g_e})$ such that live data elements are mapped to different memory cells. The buffer can be thought of as a cuboid consisting of $\prod_{i=1}^{n_e} \langle \mathbf{m_e}, \mathbf{e_i} \rangle$ data elements, with n_e being the number of token dimensions on edge $e \in E$.

The value of $\mathbf{m_e}$ has to be chosen such that two live data elements are not mapped to the same memory cell. Given the set of live data elements $\mathscr{L}(t) \subseteq G_e$ at time t, this can be achieved by the so-called *successive moduli technique* [79], calculating one component of $\mathbf{m_e}(t)$ after the other. Figure 6.4 shows a typical distribution of the live data elements $\mathscr{L}(t)$. Suppose that $\langle \mathbf{m_e}(t), \mathbf{e_2} \rangle$ is determined before $\langle \mathbf{m_e}(t), \mathbf{e_1} \rangle$. $\langle \mathbf{m_e}(t), \mathbf{e_2} \rangle$ has to be chosen in such a way that it is possible to assign a different address $\sigma_e(\mathbf{g_e}) = \mathbf{g_e} \bmod \mathbf{m_e}(t)$ to each live data element $\mathbf{g_e} \in \mathscr{L}(t)$ for at least one value of $\langle \mathbf{m_e}(\mathbf{t}), \mathbf{e_1} \rangle$. This can be achieved by the following calculation:

$$\langle \mathbf{m_e}(t), \mathbf{e_2} \rangle = 1 + \max \{ \langle \Delta \mathbf{l}, \mathbf{e_2} \rangle \mid \langle \Delta \mathbf{l}, \mathbf{e_1} \rangle = 0 \}, \tag{6.3}$$

where $\Delta \mathbf{l} = \mathbf{l_2} - \mathbf{l_1}$ and $\mathbf{l_1}, \mathbf{l_2} \in \mathscr{L}(t)$. This formula ensures that a different address is assigned to two live data elements that are impossible to distinguish by their component $\mathbf{e_1}$. For the example given in Fig. 6.4, this leads to $\langle \mathbf{m_e}(t), \mathbf{e_2} \rangle = 2$. For calculation of

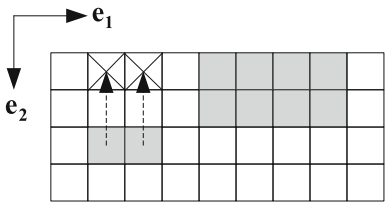


Fig. 6.4 Successive moduli technique. The live data elements are shaded by a *gray color*. The *crossed squares* represent the new virtual distribution of the live data elements resulting from the modulo-operation in dimension $\mathbf{e_2}$

$\langle \mathbf{m_e}(t), \mathbf{e_1} \rangle$, it has to be taken into account that due to the modulo-operation in dimension $\mathbf{e_2}$, only two different lines can be effectively distinguished. This leads to a new live data element distribution (see Fig. 6.4). In order to take this effect into account, $\langle \mathbf{m_e}(t), \mathbf{e_1} \rangle$ is calculated independently of dimension $\mathbf{e_2}$, such that the live data element distribution in this direction has no influence:

$$\langle \mathbf{m_e}(t), \mathbf{e_1} \rangle = 1 + \max\{\langle \Delta \mathbf{l}, \mathbf{e_1} \rangle\}. \tag{6.4}$$

For Fig. 6.4, this leads to $\langle \mathbf{m_e}(t), \mathbf{e_1} \rangle = 7$. In case coordinate $\mathbf{e_1}$ is processed before coordinate $\mathbf{e_2}$ by exchanging the two vectors in the above formulas, the corresponding result amounts to $\mathbf{m_e}(\mathbf{t}) = (4, 3)^T$. Thus, whereas this solution requires $4 \times 3 = 12$ memory cells, the previous alternative led to $2 \times 7 = 14$ buffer locations. In other words, in order to find the best solution, permutation of the different coordinate vectors is necessary when applying the successive moduli technique.

The overall memory size for a WDF edge is given by the component-wise maximum of all possible $\mathbf{m_e}(t)$:

$$\mathbf{m_e} = \max_t (\mathbf{m_e}(t)). \tag{6.5}$$

In other words, for each time t all live data elements have to be determined in order to calculate the vector $\mathbf{m_e}(t)$ using Eqs. (6.3) and (6.4). Since this risks to get computationally intensive, special techniques have to be employed for solving this problem. Because their exact details are not necessary for understanding the remaining chapters, those are not further discussed, but can be found in Appendix A.

Nevertheless, it can be seen that the rectangular memory model results in simple mapping functions. On the other hand, it delivers non-optimum buffer sizes for algorithms with overlapping windows. Using, for example, the scenario given in Fig. 6.2a, the buffer would encompass three complete lines instead of 2 lines and 3 pixels.

6.3.2 The Linearized Buffer Model

In order to solve the above-mentioned problem, a second mapping function using linearization in production order shall be investigated. This means that two consecutively produced data elements are assigned to two consecutive memory addresses. The corresponding mapping function is given by

$$\sigma_e\left(\mathbf{g_e}\right) = \Theta_e\left(\mathbf{g_e^h}\right) \bmod B_e,\ B_e \in \mathbb{N}, \tag{6.6}$$

with

$$\Theta_e\left(\mathbf{g_e^h}\right) = \sum_{i=1}^{n_e \times (q_e^{\text{src}}+1)} \left[\left(\langle \mathbf{g_e^h}, \mathbf{e_i} \rangle - \min_{\mathbf{g_e^h} \in G_e^h} \left(\mathbf{g_e^h}\right) \right) \prod_{j=i+1}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) \right], \tag{6.7}$$

$$\mathbf{g_e^h} = M\left(\mathbf{g_e}\right) = \begin{pmatrix} \theta_e\left(P_e\left(\mathbf{g_e}\right)\right) \\ \left(\langle \mathbf{g_e}, \mathbf{e_n} \rangle - \langle \boldsymbol{\delta_e}, \mathbf{e_n} \rangle\right) \bmod \langle \mathbf{p_e}, \mathbf{e_n} \rangle \\ \vdots \\ \left(\langle \mathbf{g_e}, \mathbf{e_1} \rangle - \langle \boldsymbol{\delta_e}, \mathbf{e_1} \rangle\right) \bmod \langle \mathbf{p_e}, \mathbf{e_1} \rangle \end{pmatrix} \in \mathbb{Z}^{n_e \times (q_e^{\text{src}}+1)},$$

$$\Delta \mathbf{g_e^h} = \max_{\mathbf{g_e^h} \in G_e^h} \left(\mathbf{g_e^h}\right) - \min_{\mathbf{g_e^h} \in G_e^h} \left(\mathbf{g_e^h}\right).$$

$\mathbf{g_e^h}$ is called *hierarchical data element identifier*, because it identifies a data element not by its coordinates $\mathbf{g_e}$ in the token space, but by the producing write operation. The latter is obtained via the function P_e defined in Eq. (6.2) and translated into a hierarchical iteration vector via the function θ_e explained in Definition 6.3. q_e^{src} corresponds to the number of firing levels defining the write order (see Definition 6.3). $\left(\langle \mathbf{g_e}, \mathbf{e_i} \rangle - \langle \boldsymbol{\delta_e}, \mathbf{e_i} \rangle\right) \bmod \langle \mathbf{p_e}, \mathbf{e_i} \rangle$ identifies the position of the data element in the inner of the produced effective token. Subtraction of the initial data elements is required in order to take the shift between the data elements and producing write operations into account (see also Section 5.1).

With these definitions, Eq. (6.7) simply performs linearization in production order. In other words, two consecutively produced data elements are assigned to two consecutive memory addresses. G_e^h defines the set of all occurring hierarchical data element identifiers and $\Delta \mathbf{g_e^h}$ is the difference of the component-wise maximum and minimum.[3]

Example 6.8 Consider Fig. 6.3 and assume that each rectangle corresponds to a write operation producing exactly one data element ($\mathbf{p} = (1, 1)^T$). Then, the hierarchical data element identifiers are given by

$$\mathbf{g_e^h} = \left(\left\lfloor \frac{\langle \mathbf{g_e}, \mathbf{e_2} \rangle}{3} \right\rfloor, \left\lfloor \frac{\langle \mathbf{g_e}, \mathbf{e_1} \rangle}{3} \right\rfloor, \langle \mathbf{g_e}, \mathbf{e_2} \rangle \bmod 3, \langle \mathbf{g_e}, \mathbf{e_1} \rangle \bmod 3, 0, 0 \right)^T.$$

The first two coordinates tell to which produced image tile the data element $\mathbf{g_e}$ belongs to. The next two coordinates give the position of the data element in the inner of the image tile. The last two coordinates are always zero, because one effective token consists of exactly one data element. Applying Eq. (6.7), for instance, to the two consecutively produced data elements $\mathbf{g_{e,1}} = (2, 2)^T$ and $\mathbf{g_{e,2}} = (3, 0)^T$ leads to $\mathbf{g^h_{e,1}} = (0, 0, 2, 2, 0, 0)^T$ and $\mathbf{g^h_{e,2}} = (0, 1, 0, 0, 0, 0)^T$. Since $\Delta \mathbf{g_e^h} = (\infty, 1, 2, 2, 0, 0)^T$ ($\langle \Delta \mathbf{g}, \mathbf{e_1} \rangle = \infty$ because WDF graphs operate on infinite streams of data, see also Section 6.2), application of Eq. (6.7) leads to

[3] Note: $\langle \Delta \mathbf{g}, \mathbf{e_1} \rangle = \infty$.

$$\Theta_e\left(\mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}}\right) = 0+0+2\times 3+2\times 1+0+0 = 8,$$
$$\Theta_e\left(\mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}}\right) = 0+1\times 3\times 3+0+0+0+0 = 9,$$

hence two consecutive addresses for two consecutively executed write operations.

The following corollary formally proves this property of linearization in production order.

Corollary 6.9 *Given two data elements* $\mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}}$ *and* $\mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{2}}$*, with* $\mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}} \prec \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{2}}$*, then it holds*

$$\Theta_e\left(\mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}}\right) < \Theta_e\left(\mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{2}}\right).$$

Proof

$$\begin{aligned}
&\Theta_e\left(\mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{2}}\right) - \Theta_e\left(\mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}}\right)\\
= \quad & \textstyle\sum_{i=1}^{n_e\times(q_e^{\mathrm{src}}+1)}\left[\langle \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{2}} - \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}}, \mathbf{e_i}\rangle \times \prod_{j=i+1}^{n_e\times(q_e^{\mathrm{src}}+1)}\left(\langle \Delta\mathbf{g}^{\mathbf{h}}_{\mathbf{e}}, \mathbf{e_j}\rangle + 1\right)\right]\\
= \quad & \textstyle\sum_{i=1}^{l-1}\left[\langle \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{2}} - \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}}, \mathbf{e_i}\rangle \times \prod_{j=i+1}^{n_e\times(q_e^{\mathrm{src}}+1)}\left(\langle \Delta\mathbf{g}^{\mathbf{h}}_{\mathbf{e}}, \mathbf{e_j}\rangle + 1\right)\right] +\\
& \textstyle\sum_{i=l}^{n_e\times(q_e^{\mathrm{src}}+1)}\left[\langle \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{2}} - \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}}, \mathbf{e_i}\rangle \times \prod_{j=i+1}^{n_e\times(q_e^{\mathrm{src}}+1)}\left(\langle \Delta\mathbf{g}^{\mathbf{h}}_{\mathbf{e}}, \mathbf{e_j}\rangle + 1\right)\right],
\end{aligned}$$

where l has been chosen to

$$l = \min\left\{i \mid \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{2}} > \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}}\right\}.$$

Such an l exists, because $\mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}} \prec \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{2}}$. This property also ensures that $\langle \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{2}} - \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}}, \mathbf{e_j}\rangle = 0$ $\forall 1 \le j < l$. Consequently this leads to

$$\begin{aligned}
&\Theta_e\left(\mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{2}}\right) - \Theta_e\left(\mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}}\right)\\
= \ & \textstyle\sum_{i=l}^{n_e\times(q_e^{\mathrm{src}}+1)}\left[\langle \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{2}} - \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}}, \mathbf{e_i}\rangle \times \prod_{j=i+1}^{n_e\times(q_e^{\mathrm{src}}+1)}\left(\langle \Delta\mathbf{g}^{\mathbf{h}}_{\mathbf{e}}, \mathbf{e_j}\rangle + 1\right)\right]\\
\ge \ & \textstyle\sum_{i=l+1}^{n_e\times(q_e^{\mathrm{src}}+1)}\left[\langle \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{2}} - \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}}, \mathbf{e_i}\rangle \times \prod_{j=i+1}^{n_e\times(q_e^{\mathrm{src}}+1)}\left(\langle \Delta\mathbf{g}^{\mathbf{h}}_{\mathbf{e}}, \mathbf{e_j}\rangle + 1\right)\right] +\\
& \textstyle\prod_{j=l+1}^{n_e\times(q_e^{\mathrm{src}}+1)}\left(\langle \Delta\mathbf{g}^{\mathbf{h}}_{\mathbf{e}}, \mathbf{e_j}\rangle + 1\right).
\end{aligned}$$

Since

$$\left\langle \min_{\mathbf{g}^{\mathbf{h}}_{\mathbf{e}}\in G^h_e}\left(\mathbf{g}^{\mathbf{h}}_{\mathbf{e}}\right), \mathbf{e_i}\right\rangle \le \langle \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{2}}, \mathbf{e_i}\rangle \le \left\langle \max_{\mathbf{g}^{\mathbf{h}}_{\mathbf{e}}\in G^h_e}\left(\mathbf{g}^{\mathbf{h}}_{\mathbf{e}}\right), \mathbf{e_i}\right\rangle,$$

$$\left\langle \min_{\mathbf{g}^{\mathbf{h}}_{\mathbf{e}}\in G^h_e}\left(\mathbf{g}^{\mathbf{h}}_{\mathbf{e}}\right), \mathbf{e_i}\right\rangle \le \langle \mathbf{g}^{\mathbf{h}}_{\mathbf{e},\mathbf{1}}, \mathbf{e_i}\rangle \le \left\langle \max_{\mathbf{g}^{\mathbf{h}}_{\mathbf{e}}\in G^h_e}\left(\mathbf{g}^{\mathbf{h}}_{\mathbf{e}}\right), \mathbf{e_i}\right\rangle,$$

and

$$\Delta \mathbf{g_e^h} = \max_{\mathbf{g_e^h} \in G_e^h} \left(\mathbf{g_e^h} \right) - \min_{\mathbf{g_e^h} \in G_e^h} \left(\mathbf{g_e^h} \right),$$

it follows that

$$-\langle \Delta \mathbf{g_e^h}, \mathbf{e_i} \rangle \leq \langle \min_{\mathbf{g_e^h} \in G_e^h} \left(\mathbf{g_e^h} \right) - \mathbf{g_{e,1}^h}, \mathbf{e_i} \rangle \leq \langle \mathbf{g_{e,2}^h} - \mathbf{g_{e,1}^h}, \mathbf{e_i} \rangle.$$

Since $\langle \Delta \mathbf{g_e^h}, \mathbf{e_i} \rangle \geq 0$, it can be concluded that

$$\begin{aligned}
& \Theta_e \left(\mathbf{g_{e,2}^h} \right) - \Theta_e \left(\mathbf{g_{e,1}^h} \right) \\
\geq\ & \textstyle\prod_{j=l+1}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) - \\
& \textstyle\sum_{i=l+1}^{n_e \times (q_e^{\text{src}}+1)} \left[\langle \Delta \mathbf{g_e^h}, \mathbf{e_i} \rangle \times \prod_{j=i+1}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) \right] \\
=\ & \textstyle\prod_{j=l+1}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) - \\
& \textstyle\sum_{i=l+1}^{n_e \times (q_e^{\text{src}}+1)} \left[\left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_i} \rangle + 1 \right) \times \prod_{j=i+1}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) \right] - \\
& \textstyle\sum_{i=l+1}^{n_e \times (q_e^{\text{src}}+1)} \left[\prod_{j=i+1}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) \right] \\
=\ & \textstyle\prod_{j=l+1}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) - \\
& \textstyle\sum_{i=l+1}^{n_e \times (q_e^{\text{src}}+1)} \left[\prod_{j=i}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) - \prod_{j=i+1}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) \right] \\
=\ & \textstyle\prod_{j=l+1}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) - \\
& \textstyle\sum_{i=l+1}^{n_e \times (q_e^{\text{src}}+1)} \prod_{j=i}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) + \\
& \textstyle\sum_{i=l+1}^{n_e \times (q_e^{\text{src}}+1)} \prod_{j=i+1}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) \\
=\ & - \textstyle\sum_{i=l+2}^{n_e \times (q_e^{\text{src}}+1)} \prod_{j=i}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) + \\
& \textstyle\sum_{i=l+1}^{n_e \times (q_e^{\text{src}}+1)} \prod_{j=i+1}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) \\
=\ & - \textstyle\sum_{i=l+1}^{n_e \times (q_e^{\text{src}}+1)-1} \prod_{j=i+1}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) + \\
& \textstyle\sum_{i=l+1}^{n_e \times (q_e^{\text{src}}+1)} \prod_{j=i+1}^{n_e \times (q_e^{\text{src}}+1)} \left(\langle \Delta \mathbf{g_e^h}, \mathbf{e_j} \rangle + 1 \right) \\
=\ & 1 \\
>\ & 0\ .
\end{aligned}$$

However, as WDF graphs operate on infinite streams of data, the above corollary also indicates that $\Theta_e \left(\mathbf{g_e^h} \right)$ is not bounded. Hence, it cannot be used directly in order to select a memory cell of the edge buffer. Instead, Eq. (6.6) maps different linearized addresses $\Theta_e \left(\mathbf{g_e^h} \right)$ to the same memory cell $\sigma_e \left(\mathbf{g_e} \right)$. This of course is only possible, if no conflicts occur. In other words, the buffer size B_e must be chosen in such a way that two live data elements are not mapped to the same memory cell. For a given set of live data elements $\mathscr{L}(t) \subseteq G_e$, this can be achieved by searching those live data elements that lead to the smallest and largest value of Θ_e:

$$B_e(t) = \max_{\mathbf{l} \in \mathscr{L}(t)} \Theta_e(\mathbf{l}) - \min_{\mathbf{l} \in \mathscr{L}(t)} \Theta_e(\mathbf{l}) + 1. \tag{6.8}$$

The overall buffer size can then be calculated to

$$B_e = \max_t B_e(t) .$$

6.4 Simulation Results

Based on the above memory mapping functions, the buffer size for a given WDF edge can be derived by means of simulation. For this purpose, two actors of a single WDF edge are executed using two different example scheduling strategies. The first performs sequential ASAP scheduling and prioritizes the sink actor. In other words, it repeatedly checks whether enough data elements are available to execute the sink actor. Only if this is not the case, the source is fired, otherwise only the sink actor is executed.

Such a schedule offers the advantage of resulting in the minimum possible buffer size. Unfortunately, it typically leads to bad throughput, since the actors are not executed concurrently, but only exclusively. Consequently, a second scheduling alternative is investigated that simulates parallel, self-timed execution of the WDF edge. In other words, supposing that enough data elements are available for execution of the sink actor, first the source and then the sink are fired. In both cases, the simulation framework monitors the live data elements in order to determine the maximum occurring buffer requirements as discussed in Sections 6.3.1 and 6.3.2. Since not required for further understanding, the detailed technique for this monitoring is not described in this chapter, but can be found in Appendix A.

Table 6.1 shows the corresponding simulation results when applied to different scenarios:

1. Scenario (1) corresponds to edge e_1 of Fig. 6.1 and performs a two-dimensional wavelet transform with a 5×5 sliding window.
2. Scenario (2) equals scenario (1), but uses a sliding window size of $\mathbf{c} = (3, 3)^T$ and $\Delta\mathbf{c} = (1, 1)^T$ as well as a border extension defined by $\mathbf{b^s} = \mathbf{b^t} = (1, 1)^T$.
3. Scenario (3) equals scenario (2), but uses one row of initial tokens ($\mathbf{d} = (0, 1)^T$).
4. Scenario (4) equals scenario (3), but the initial tokens cover two horizontal tiles ($\mathbf{d} = (0, 1088)^T$).
5. Scenario (5) models a simple vertical downsampler with the following parameters:

$$\mathbf{p} = (1, 1)^T , \ \mathbf{v} = (2048, 1088)^T , \ \mathbf{c} = \Delta\mathbf{c} = (1, 2)^T , \ \mathbf{b^s} = \mathbf{b^t} = (0, 0)^T ,$$
$$\mathbf{O}_{\text{src}} = \left[\begin{pmatrix} 2048 \\ 1088 \end{pmatrix} \right], \ \mathbf{O}_{\text{snk}} = \left[\begin{pmatrix} 2048 \\ 544 \end{pmatrix} \right].$$

   As shown later on in Section 8.5.5, this simple downsampler can be used as a building block for more sophisticated algorithms leading to better image quality.
6. Scenario (6) equals scenario (5), but accesses the window pixels sequentially:

$$\mathbf{p} = (1, 1)^T , \ \mathbf{v} = (2048, 1088)^T , \ \mathbf{c} = \Delta\mathbf{c} = (1, 1)^T , \ \mathbf{b^s} = \mathbf{b^t} = (0, 0)^T ,$$
$$\mathbf{O}_{\text{src}} = \left[\begin{pmatrix} 2048 \\ 1088 \end{pmatrix} \right], \ \mathbf{O}_{\text{snk}} = \left[\begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 2048 \\ 1088 \end{pmatrix} \right].$$

Table 6.1 Simulation results comparing the rectangular and the linearized memory model

		Rectangular buffer		Linearized	
Scenario	Schedule	**m**	#Cells	B_1=#Cells	Improvement (%)
(1)	ASAP	$(2048, 5)^T$	10, 240	8197	20
	par	$(4096, 5)^T$	20, 480	8198	59.9
(2)	ASAP	$(2048, 3)^T$	6144	4099	33.3
	par	$(4096, 3)^T$	12, 288	4100	66.6
(3)	ASAP	$(4096, 3)^T$	12, 288	2, 232, 323	−18,066.7
	par	$(4096, 3)^T$	12, 288	2, 232, 324	−18,066.7
(4)	ASAP	$(4096, 1088)^T$	4, 456, 448	4, 456, 448	0
	par	$(4096, 1090)^T$	4, 464, 640	4, 458, 498	0.1
(5)	ASAP	$(2048, 2)^T$	4096	2049	50
	par	$(2048, 2)^T$	4096	2050	50
(6)	ASAP	$(2048, 1)^T$	2048	2048	0
	par	$(2048, 2)^T$	4096	4096	0
(7)	ASAP	$(4, 2)^T$	8	8	0
	par	$(12, 2)^T$	24	12	50
(8)	ASAP	$(2048, 1088)^T$	2, 228, 224	4, 452, 352	−99.8
	par	$(4096, 1088)^T$	4, 456, 448	4, 452, 354	0.1
(9)	ASAP	$(2048, 1088)^T$	2, 228, 224	2, 228, 224	0
	par	$(4096, 1088)^T$	4, 456, 448	4, 452, 354	0.1
(10)	ASAP	$(1984, 64)^T$	126, 976	129, 024	−1.6
	par	$(2048, 123)^T$	251, 904	249, 986	0.8
(11)	ASAP	$(2048, 31)^T$	63, 488	63, 488	0
	par	$(2048, 62)^T$	126, 976	124, 994	1.6

This situation occurs in case of behavioral communication synthesis when aiming to reduce the number of required memory channels as described in Section 8.2.5.

7. Scenario (7) imitates an MDSDF graph (see Section 3.1.4.1) with $\mathbf{p} = (2, 2)^T$ and $\mathbf{c} = \Delta\mathbf{c} = (3, 2)^T$.
8. Scenario (8) corresponds to edge e_0 of Fig. 6.1 and performs image tiling.
9. Scenario (9) performs the inverse tiling operation. In other words, it combines 2×2 sequentially produced images into one huge output image written in raster-scan order.
10. Scenario (10) performs the block-building operation discussed in Fig. 2.6 on page 15. It divides an image of 2048×1088 pixels into blocks of size 64×64. This corresponds to the tiling operation of scenario (8), but uses much smaller tile sizes.
11. Scenario (11) finally equals scenario (10), but uses a block size of 32×32 pixels.

From these results, several conclusions can be drawn. Consider first scenarios (1)–(5), which represent sliding window algorithms with or without overlapping windows, and for which production and consumption occur in the same order. Apart from scenario (3), it can be stated that these cases are better covered by the linearized buffer model for both self-timed parallel and sequential ASAP execution. This is because the rectangular buffer model always has to store complete tile lines, whereas this is not true for the linearized buffer model. Consider, for instance, scenario (2), 2 lines and 3 pixels are sufficient in case of ASAP scheduling, leading to an overall buffer size of $2 \times 2048 + 3 = 4099$ data elements. This is the minimum buffer size required for a 3×3 sliding window and used in many different hand-built implementations [244]. For the self-timed parallel scheduling, the linearized buffer model

requires one additional memory cell in comparison with the sequential ASAP schedule. This is because the source already writes the next pixel value while the sink is still accessing the current window.

The rectangular memory model, on the other hand, can only handle complete image rows, leading to an overall buffer requirement of 3 image lines. The situation gets even worse when considering self-timed parallel execution. The reason is illustrated in Fig. 6.5, showing the tiles produced by the source actor. Due to parallel actor invocation it happens that the sink is still reading the upper right tile of the first image whereas the source already produces the first pixel of the lower left tile. As a consequence, the maximum horizontal distance between two live data elements counts not only one, but two image tiles. Consequently, a single buffer line consists of 4096 instead of 2048 data elements leading to a huge waste of buffer space. The linearized buffer model, on the other hand, does not show these problems, because due to linearization in production order, the difference between the linearized addresses $\Theta(\mathbf{g_2}) - \Theta(\mathbf{g_1})$ amounts just 1. Hence, parallel actor execution does not have the tremendous effect on the buffer size as for the rectangular buffer model.

Scenario (1) leads to similar conclusions. Whereas the rectangular buffer model requires storage of 5 complete tile lines, 4 lines and 5 pixels are sufficient for the linearized buffer model, leading to an overall storage size requirement of $4 \times 2048 + 5 = 8197$ data elements. Note, however, that due to $\Delta\mathbf{c} = (2, 2)^T$, this does not correspond to the optimum anymore. The underlying reason can be found in the assumption that all data elements of a WDF edge have to be mapped to the same address space. Consequently, in Sections 7.6.2 and 8.2.2, corresponding extensions will allow reducing the buffer requirements to 3 lines and 10 pixels. Nevertheless, even without these improvements, the linearized memory model still outperforms the rectangular one.

This is not true anymore in case of scenario (3). Here, the underlying reason can be found in the presence of initial tokens that do not follow the production order of the source, leading to enormous waste of memory. This is graphically depicted in Fig. 6.6. Due to the row of initial data elements, the write operations are shifted as discussed in Section 5.1. Consequently, the linearized buffer model has to assume that not only the initial data elements, but also those depicted by broken lines are live, leading to an enormous waste of memory. As this, however, seems to be a rather artificial scenario, it will be ignored in the rest of this monograph. In scenario (4), on the other hand, two complete horizontal tiles are available in form of initial

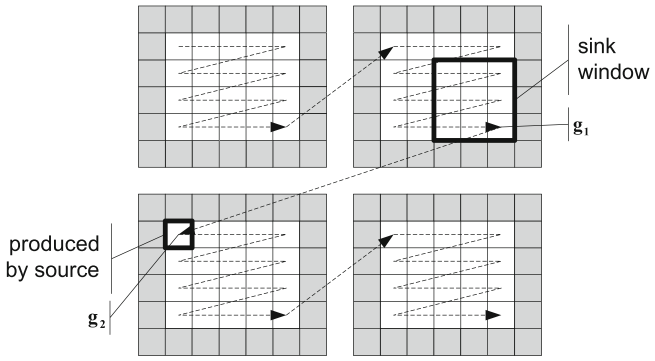


Fig. 6.5 Worst-case data element distribution for scenario (2). The *dashed arrows* represent the production order. *Gray-shaded* data elements belong to the virtually extended border

data elements. As a consequence, they follow the production order and avoid extensive waste of memory.

Scenarios (6)–(11) demonstrate the dramatic impact of out-of-order communication on the necessary buffer sizes. Although most of them employ a window size of $\mathbf{c} = (1, 1)^T$, the resulting memory requirements are much larger than for the sliding window algorithms. Figure 6.7, for instance, illustrates the worst-case occurring for the inverse tiler described in scenario (9) and sequential ASAP scheduling. In this case, the source is only executed when the sink is blocked due to missing data elements. This occurs, for instance, when the sink is waiting for data element $\mathbf{g_2}$. However, this data element is only produced after all the other gray-hatched data elements. As a consequence, for the linearized buffer model, a buffer is required that is able to store one image tile, hence $2048 \times 1088 = 2,228,224$ data elements. The rectangular buffer model requires the same amount of memory cells when sequential ASAP scheduling is performed. This is because a pixel belonging to the left image tile and the corresponding pixel of the right image tile are never live simultaneously. In other words, the gray-hatched line of the right image tile and the non-hatched line of the left tile can share the same buffer line, leading to an overall buffer size of exactly one image tile.

At this point, it is interesting to consider the tiling operation (Scenario (8)), which is just the opposite to the above-discussed operation. However, in this case the rectangular memory model outperforms the linearized one by almost a factor of 2. Figure 6.8 illustrates the corresponding worst-case distribution of live data elements, which occurs when the sink is waiting for data element 34. The production order is indicated by Arabic numbers, which also correspond to the write address $\Theta_e(\mathbf{g_e})$ of the produced data elements $\mathbf{g_e}$, whereas the read order is depicted by dashed arrows. Consequently, while waiting for data element 34, also data element 5 is live and has to be stored in the edge buffer. However, according to

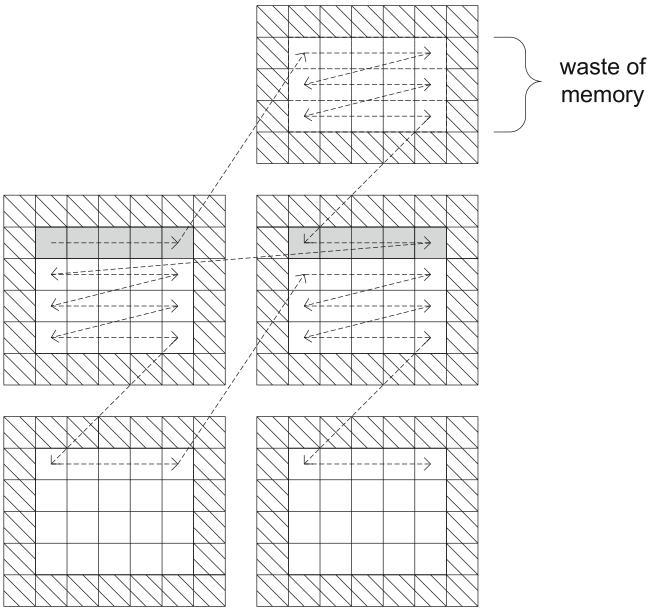


Fig. 6.6 Initial data elements not following the production order. *Gray shading* identifies the initial data elements. Extended border pixels are marked by corresponding *stripes*

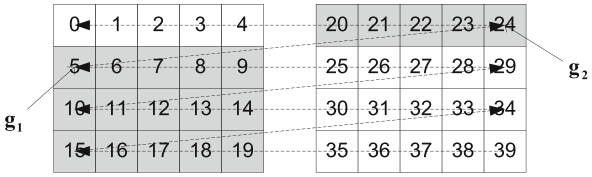


Fig. 6.7 Worst-case distribution of live data elements for the inverse tiling operation and sequential ASAP scheduling. *Arabic numbers* depict the production order while the *dashed arrows* represent the read order

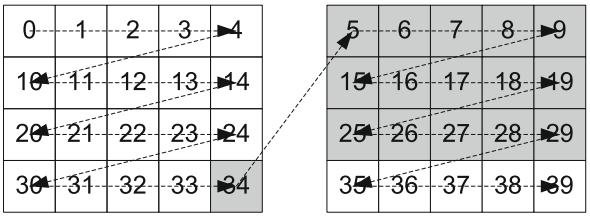


Fig. 6.8 Worst-case distribution of live data elements for the tiling operation and sequential ASAP scheduling. *Arabic numbers* depict the production order while the *dashed arrows* represent the read order

Eq. (6.8), this means that the overall buffer size needs to encompass $34 - 5 + 1 = 30$ data elements, which corresponds to almost two tiles. On the other hand, the rectangular memory model can be implemented using $\mathbf{m} = (5, 4)^T$, assuming the image sizes given in Fig. 6.8, and requiring thus only one tile. However, the occurring waste of memory for the linearized buffer model strongly depends on the tile size. This can be seen by considering scenarios (10) and (11). Both perform image tiling (or block building), but use much smaller tile (or block) sizes. In this case, the benefits of the rectangular memory model become smaller or vanish even completely. Moreover, the linearized buffer model even outperforms the rectangular one, when considering self-timed parallel execution.

In this context, it is furthermore worth to mention that self-timed parallel execution of out-of-order communication is significantly more expensive than sequential ASAP scheduling, because it can require up to twice as much as memory (see scenario (6)). This can be exemplarily explained by means of Fig. 6.7 belonging to scenario (8). Assuming that the source finally generates data element 24, the sink has to process not only the right image tile, but also most of the left one. Hence, while the sink reads, for instance, data elements, 5–9 and 25–29, the source can produce the data elements 25–34, corresponding to two tile rows. In other words, while the sink effectively removes one tile row in form of data elements 5–9 (data elements 25–29 have to be considered as live, because data element 19 is still required), the source generates two tile rows. As a consequence, the edge buffer must be approximately as large as two image tiles, which can be confirmed by Table 6.1. The latter also shows that the linearized buffer model is slightly better than the rectangular one.

6.5 Conclusion

This chapter has compared two different memory mappings concerning their impact on the required buffer size. In contrast to what might be expected, this question has shown to be anything than easy. For this reason, different scheduling strategies and application scenarios have been considered by means of a corresponding simulation framework for a single WDF edge. The achieved results could deliver several important insights into the problem of memory mapping. First of all, out-of-order communication has come out to be much more expensive than typical sliding window algorithms. Moreover, the selected scheduling strategy shows a tremendous impact on the required buffer size in case of out-of-order communication while this is not true for in-order communication.

Furthermore, the chapter has demonstrated the difficulty of selecting a good memory mapping function. For self-timed parallel execution, the linearized buffer model outperforms the rectangular mapping function more or less significantly. Thus, in case of high-performance image processing applications, the linearized buffer model seems to be preferable, in particular as it can be efficiently synthesized in hardware as demonstrated later on in Chapter 8. For low-performance applications, the situation is more difficult. Whereas the rectangular buffer suffers from shortcomings in case of overlapping sliding windows, it delivers good results in case of out-of-order communication. Nevertheless, it stays problematic that the granularity by which the buffer size can be adjusted is rather coarse. Whereas a linearized buffer can be increased or decreased by individual data elements, this is not possible for the rectangular buffer model. Furthermore, it has to be added that the buffer size determination for applications requiring tokens with more than two dimensions is straightforward for the linearized buffer model, while the rectangular mapping function causes more difficulties (see Appendix A.1). Because of these reasons, the remainder of this monograph will focus on high-performance applications using the linearized buffer model. In particular, the next chapter will investigate how the buffer size of complete WSDF graphs can be determined. To this end, more sophisticated scheduling strategies are required than when considering individual edges as done in this chapter. This is necessary in order to cope with complex graph topologies containing feedback loops and actors with multiple inputs and outputs.

Chapter 7
Buffer Analysis for Complete Application Graphs

As already shown in Chapter 6, buffer size determination is an important step in system level design of image processing applications because it helps to improve throughput by avoiding external memories. Furthermore, it is possible to reduce power dissipation and chip sizes and thus costs. Consequently, a buffer analysis technique based on simulation has been presented in the previous chapter that can be applied to arbitrary scheduling strategies. By this means, two different memory mappings, expressed in different *memory models*, have been compared. Whereas the first one uses a rectangular array structure, the second performs linearization in production order. As a result, it could be shown that both strategies have their advantages and drawbacks and that high-speed applications requiring parallel processing are best covered by the linearized buffer model.

This chapter considers automatic buffer analysis for complete WDF graphs as introduced in Chapter 5. In other words, for a given application specification, it aims to determine the memory size required for the different WDF communication edges. To this end, first a simulation approach will be evaluated by extending the SystemCoDesigner simulation engine (see Chapter 4) with a corresponding buffer analysis tool. This permits to process a multi-resolution filter as it is employed in medical image processing [313]. As it requires a complex graph topology containing up- and downsamplers as well as splitting and joining of data paths, buffer analysis becomes very challenging. Unfortunately, this leads to the fact that the buffer analysis by simulation leads to sub-optimal solutions due to the lack of analytical schedule determination. Existing analytical approaches as discussed in Section 3.3 on the other hand cannot be applied directly, because practically none of them operates on multidimensional data flow graphs. Furthermore, typically they neither consider up- and downsampling nor out-of-order communication. However, since this is a fundamental part of the windowed (synchronous) data flow model of computation, it has to be considered for encompassing system level design tools. Furthermore, in many cases the proposed methods rely on integer linear programing or do not consider the impact of long dependency vectors on the achievable system throughput. Both aspects, however, are critical as will be demonstrated in this chapter. Consequently, in order to circumvent these difficulties, this chapter describes a polyhedral scheduling and buffer analysis methodology that offers the following benefits:

- Support of complex graph topologies including splitting and joining of data paths, down- and upsampling as well as feedback loops and multiple, unrelated sources.
- Generation of throughput-optimized schedules for both in-order and out-of-order communication.

J. Keinert, J. Teich, *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*, Embedded Systems,
DOI 10.1007/978-1-4419-7182-1_7, © Springer Science+Business Media, LLC 2011

- Use of a computationally efficient algorithm that only requires solution of small integer linear programs that solely depend on the communication parameters of one single edge. This is important because ILP solving belongs to the class of NP-complete problems with an exponential computation effort when using currently known algorithms. In other words, by avoiding ILPs whose size increases with the number of actors contained in the data flow graph, it is possible to apply the discussed buffer analysis method to huge application specifications. Furthermore, this approach supports solution by intelligent exhaustive search in case even the simple integer linear programs become intractable. This two-folded proceeding permits to profit from the capacities of modern ILP solvers to quickly obtain the searched solution while avoiding to be trapped by unbounded computation times in case of difficult communication patterns.
- Consideration of different scheduling alternatives for so-called *multirate* systems containing up- and downsamplers. In other words, some of the actors execute less often than other system components. This permits to exploit tradeoffs between required communication memory and computation logic by applying resource sharing during actor synthesis.

The remainder of the chapter is organized as follows. Section 7.1 introduces the problem to solve in detail. Next, Section 7.2 presents a simulation approach and demonstrates that scheduling is a major challenge in order to obtain buffer estimations. Consequently, the following sections discuss an efficient analytical method for determination of throughput-optimized schedules of static data flow graphs that can serve as a base for buffer size determination. In particular, Section 7.3 illustrates how WSDF edges can be transformed into a polyhedral representation, considering out-of-order communication, up- and downsampling, pipelining, and causality of the system. Since out-of-order communication leads to non-affine dependency vectors, Section 7.4 is dedicated to the question how throughput-optimized schedules can be represented in the presence of huge dependency vectors. Next, Section 7.5 discusses how graphs with complex topology can be scheduled such that no data elements are read before being written. With this information, it is possible to analytically calculate the required communication edge buffers as elaborated in Section 7.6. The special properties of applications containing up- and downsamplers are investigated in Section 7.7. Next, Section 7.8 describes two different strategies for solving the integer linear programs required for buffer analysis and scheduling. Section 7.9 finally presents the obtained results for different application scenarios, before Section 7.10 concludes this chapter.

7.1 Problem Formulation

As already motivated in Section 3.3.1, the required communication buffer for a given edge $e \in E$ of a data flow graph $G = (A, E)$ does not only depend on its communication parameters like token and window sizes, window movement, or communication order (see also Section 6.1) but also on the graph topology. In order to illustrate this effect, Fig. 7.1 shows the block diagram of a four-stage multi-resolution filter as used in medical image processing [313]. Since it consists of completely static algorithms, it can be represented using the WSDF semantics explained in Section 5.5. On each level, the input image is decomposed into a difference image, the so-called Laplacian image, and a downscaled version whose width and height are reduced by a factor of 2. After filtering with a *bilateral filter* [283], the *reconstruct block* recombines the corresponding images. However, due to the successive decomposition and filtering operations, the downscaled image arrives at the input of the reconstruct block

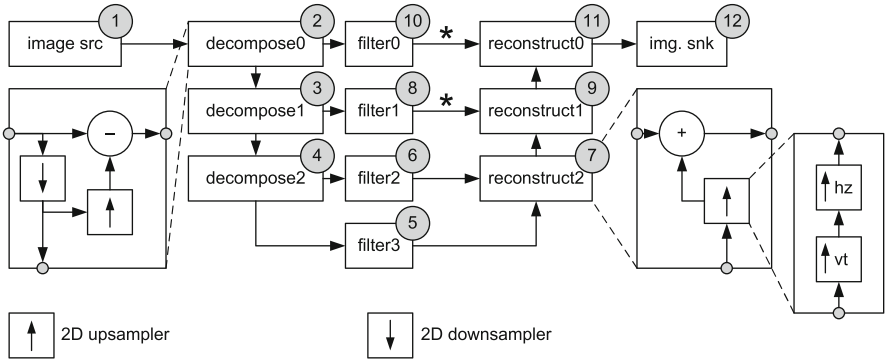


Fig. 7.1 Topology of the WSDF graph describing a four-stage multi-resolution filter. For simplicity, communication parameters have been omitted. *Decompose, reconstruct*, and the two-dimensional up- and downsamplers are hierarchical actors, which contain several sub-actors

much later than the image of bigger size. The latter has hence to be delayed on the edge marked by an asterisk, which leads to huge memory requirements.

Unfortunately, determination of the required buffer sizes is pretty time consuming even for an experienced designer, because the occurring pixel latency is influenced not only by the bilateral filter but also by the occurring up- and downsampling. Even worse, as shown later on in Section 7.7, these operations allow usage of different implementation alternatives for the filters influencing both the chip size of the hardware accelerators and the required communication memories. This, however, complicates not only buffer analysis but also hardware synthesis, as different alternatives of the accelerators have to be provided. Consequently, this chapter describes an analytical buffer analysis method that is able to handle up- and downsampling, out-of-order communication, and complex graph topologies. First, however, the following section will investigate the possibility of buffer size determination via simulation and via the techniques discussed in Chapter 6 and Appendix A.

7.2 Buffer Analysis by Simulation

As the required buffer sizes for the multi-resolution filter depicted in Fig. 7.1 depend on the graph topology, it is not sufficient anymore to simulate a single edge as done in Chapter 6. Instead, correct buffer size determination requires execution of the complete WSDF graph. In order to derive the corresponding results for the multi-resolution filter, the analysis methodology by simulation described in Chapter 6 has been exemplarily integrated into the SystemCoDesigner ESL tool discussed in Chapter 4. Furthermore, the multi-resolution filter has been implemented as a WSDF graph using the SysteMoC library described in Sections 4.1.2 and 5.7. Together with the *VPC* framework presented in Section 4.1.4 and by annotation of the corresponding action execution times, this enables combined functional and timing simulation of the complete multi-resolution filter and thus correct stimulus for the buffer analysis method by simulation.

Figure 7.2 shows a corresponding excerpt of the simulation trace. It shows the buffer analysis results for the multi-resolution filter using an input image size of 256×256 pixels and two bilateral filters. The first five lines represent the activity of the filter actors, of the

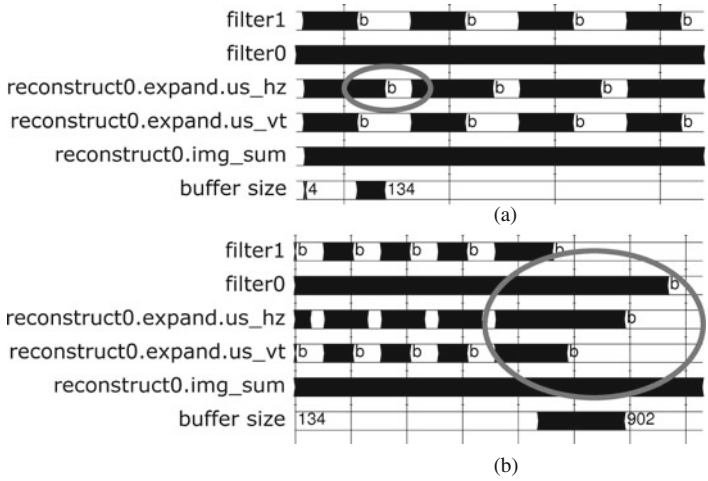


Fig. 7.2 Extract of the simulation trace showing the buffer analysis result for the multi-resolution filter. **a** Start of image. **b** End of image

horizontal and vertical upsampler of the *reconstruct0* block, and of its sum actor, where "b" means blocked due to missing input data. Furthermore, it depicts the measured buffer size on the edge between the horizontal upsampler and the adder of the *reconstruct0* block. The corresponding value has been determined using the methods described in Chapter 6 and Appendix A together with a self-timed schedule. In other words, an actor is executed whenever all required input data are available.

Figure 7.2a illustrates the behavior at the image start. The *filter0* actor processes 1 pixel per clock cycle, because it works on the full-resolution image. The *filter1* actor on the other hand shows idle lines, because the previous vertical downsampler in the *decompose0* block only generates an output each second image line. At the beginning, the buffer size amounts four data elements. This corresponds to the expected value, as for each invocation, the horizontal upsampler generates two data elements. Due to pipelining, the required buffer size doubles. Then, however, the buffer size suddenly increases.

In order to explain this behavior, Fig. 7.3 depicts a detailed extract of the WSDF graph modeling the multi-resolution filter. It encompasses the actors *filter1*, the vertical and the horizontal upsamplers, and the image sum actor. The *filter1* actor generates a downscaled image of size 128×128. In order to upsample this image in vertical direction, the vertical upsampler generates for each invocation 2 pixels simultaneously belonging to two image rows. Similarly, the horizontal upsampler produces two data elements per invocation. As the latter generates a full-resolution image, we would expect its executions to be equally distributed over time. Instead, Fig. 7.2a clearly shows an idle time. A detailed analysis furthermore shows that before this idle time, the horizontal upsampler executes every clock cycle instead of every second one as expected. This can be explained by the fact that the vertical upsampler generates 2 lines simultaneously. Hence the horizontal upsampler does not have to wait for any input data when processing the second image line. As each invocation of the horizontal upsampler generates two data elements, the resulting image line is generated faster than needed, followed by an idle period. Unfortunately, this leads to a useless increase of the measured buffer size.

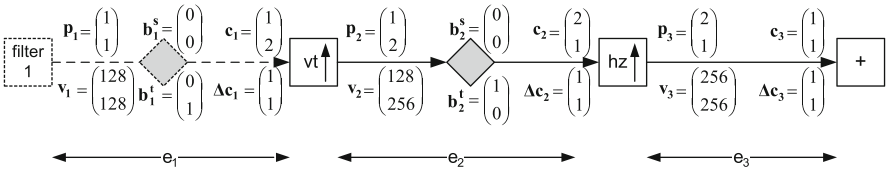


Fig. 7.3 Extract of the WSDF graph for the multi-resolution filter

The situation gets even worse when attaining the end of the image as shown in Fig. 7.2b. Here the vertical upsampler also deviates from the periodic invocation pattern, which alternates between busy and idle lines. The underlying reason can be found in the border processing causing that the last image row can be generated without waiting for any input data. Consequently, the vertical upsampler generates the data too fast, leading to a useless increase of the measured buffer size.

In other words, although the results delivered by the simulative buffer analysis will lead to a working implementation, they are not optimal due to usage of self-timed scheduling. Whereas the simulation trace clearly indicates this situation and thus helps the designer to intervene, a fully automatic approach is clearly preferable. Consequently, the next section discusses an analytic method for generation of balanced schedules by usage of lattice representations that permit to determine the time instances when an actor has to execute. Furthermore, it is even possible to quickly derive the buffer sizes assuming linearization in production order as discussed in Section 6.3.2. This significantly reduces the time required for buffer analysis while improving the quality of the obtained results. On the other hand, the supported schedule functions are restricted to affine forms and the method can only be applied to data flow graphs obeying the WSDF semantics (see Section 5.5). In other words, presence of control flow is not supported.

7.3 Polyhedral Representation of WSDF Edges

In order to derive the required communication buffer sizes resulting from the edge parameters and communication orders, it is sufficient to consider a single edge, as, for instance, done in Chapter 6. This, however, is not enough anymore for determination of memory requirements that occur due to delayed sink execution. A corresponding example has already been presented in Fig. 7.1 in form of the edges labeled by a star. In these cases, both the effective token and the sliding window consist of one single pixel. Nevertheless, a huge edge buffer is required, because the sink actors of the edges have to wait for further input data arriving much later. Consequently, this situation can only be detected when establishing a static schedule for the complete WSDF graph.

To this end, the following sections describe a lattice-based approach that exploits the regularity of the problem for fast analysis. Its major idea is to embed all actors into a common *invocation grid* such that their relative execution time is determined. This enables efficient dependency analysis and thus determination of the required buffer sizes. Note that the final hardware does not need to directly implement the determined actor schedule. Instead, a *self-timed schedule* can be used, in which the actor invocations are controlled by the availability of input data and free output space. The property of monotonic execution for data flow graphs

[302] guarantees that no deadlock occurs and that the throughput attains at least the value determined during analysis.

7.3.1 WSDF Lattice

As explained in Section 5.6, the balance equation permits to calculate for each actor of a WSDF graph the number of invocations in each dimension. Taking, for instance, the example given in Fig. 7.4, the source actor generates an image of 5 × 4 pixels. They are sampled by a vertical downsampler using a sliding window with 1 × 3 pixels. Furthermore, the image is extended by a virtual border as depicted by gray shading in Fig. 7.4. Since the window moves by 2 pixels in vertical direction, the downsampler performs only 5 × 2 invocations, while the source is executed 5 × 4 times. For each invocation, the latter generates exactly 1 pixel. Each of these invocations can be represented as a point in an n-dimensional grid, also known as *lattice*.

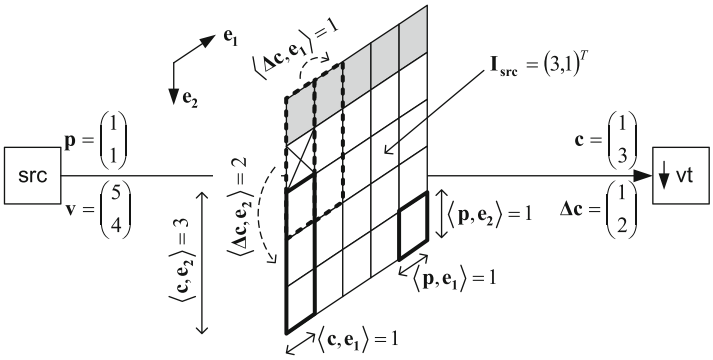


Fig. 7.4 Simple WSDF graph for illustration of the polyhedral buffer analysis method. *Gray-shaded* data elements belong to the virtually extended border

Figure 7.5a illustrates the principles assuming the example given in Fig. 7.4. The gray-shaded semicircles belong to the source invocations while the sink invocations are represented by a white color. The arrows illustrate data dependencies and will be discussed later on. Note that the extended image border is not represented in this picture, because it is not produced by the source (see Section 5.1). For the moment, it is assumed that all lattice points are executed in row-major order from left to right and from top to bottom. More complex invocation orders will be discussed in Section 7.3.3. Each semi-circle stands for the start of the corresponding actor invocation. In other words, coinciding semi-circles are executed concurrently if permitted by the data dependencies. In the opposite case, they are executed sequentially (see also Section 7.3.5). However, in any case the invocation belonging to a given semi-circle must not start before all previous circles have been terminated (this condition will be relaxed later on in Section 7.3.5 for pipelined actor execution).

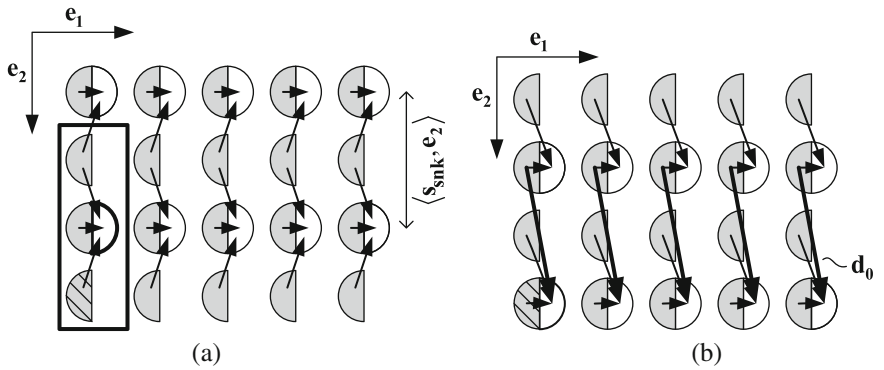


Fig. 7.5 Lattice representation of Fig. 7.4. Each *semi-circle* corresponds to an actor invocation, where the edge source is represented by a *gray shading*, while the *white color* has been assigned to the sink. The *arrows* correspond to the data dependencies and illustrate the data flow from the source to the consuming sink invocations. The *rectangle* in the *left* subfigure frames all source and sink actor invocations concerned by the *bold lower left sliding* window in Fig. 7.4. **a** Original lattice. **b** Lattice with shifted sink

7.3.2 Lattice Scaling

With the above definitions, the lattice representation defines the relative execution order of the source and sink actors required for correct buffer analysis. In particular, it determines which invocations are executed sequentially or concurrently. However, in order to obtain reasonable actor schedules, the lattices belonging to the different actors have to be scaled correctly.

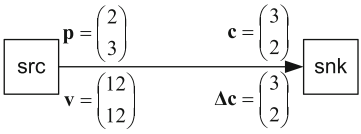
In case of the example shown in Fig. 7.5, for instance, the downsampler actor performs only half the invocations in vertical direction compared to the source. This is because the sliding window moves by 2 pixels while the source generates only 1 pixel per invocation. Consequently, the sink invocation points have to be scaled by a factor 2 relatively to the source grid as already done in Fig. 7.5a. In general, the corresponding scaling factor in dimension i can be calculated from the window movement $\Delta\mathbf{c}$ and the produced token size $\mathbf{p}$:

$$\frac{\langle \mathbf{s}_{\text{snk}}, \mathbf{e_i} \rangle}{\langle \mathbf{s}_{\text{src}}, \mathbf{e_i} \rangle} = \frac{\langle \Delta\mathbf{c}, \mathbf{e_i} \rangle}{\langle \mathbf{p}, \mathbf{e_i} \rangle} \in \mathbb{Q}. \tag{7.1}$$

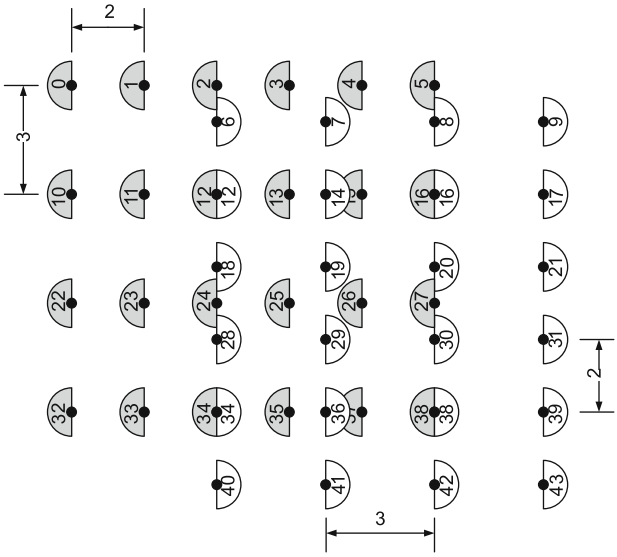
$\langle \mathbf{s}_{\text{snk}}, \mathbf{e_i} \rangle$ defines the distance of the sink lattice points in dimension $\mathbf{e_i}$. Similarly, $\langle \mathbf{s}_{\text{src}}, \mathbf{e_i} \rangle$ corresponds to the distance between two source lattice points.

Example 7.1 For the downsampler depicted in Fig. 7.5a, this leads to $\mathbf{s}_{\text{snk}} = (1, 2)^T$ when assuming $\mathbf{s}_{\text{src}} = (1, 1)^T$. Hence the sink is only active every second row as illustrated in Fig. 7.5a.

In case $\frac{\langle \mathbf{s}_{\text{snk}}, \mathbf{e_i} \rangle}{\langle \mathbf{s}_{\text{src}}, \mathbf{e_i} \rangle} \notin \mathbb{Z}$, it is assumed that the lattice points are scaled in such a way that both $\mathbf{s}_{\text{src}} \in \mathbb{N}^n$ and $\mathbf{s}_{\text{snk}} \in \mathbb{N}^n$. Figure 7.6 shows a corresponding WSDF graph and its lattice representation. In order to avoid any confusion, the sink lattice has already been shifted in such a way that parallel execution without violation of data dependencies is possible (see also Section 7.3.4). In order to fulfill Eq. (7.1) and obtain integer scaling vectors, the latter are set to $\mathbf{s}_{\text{src}} = (2, 3)^T$ and to $\mathbf{s}_{\text{snk}} = (3, 2)^T$. As a consequence, this causes that source and sink



(a) WSDF graph



(b) Lattice representation

Fig. 7.6 Rational Lattice. The *black dots* in subfigure **b** correspond to the lattice point center. Only lattice points whose centers coincide can be executed concurrently

lattice points do not necessarily coincide. The interpretation of the lattice, however, remains strictly the same, in that lattice points are executed in row-major order and in that only lattice points that exactly coincide can be executed in parallel if permitted by the data dependencies. The resulting execution order is depicted in Fig. 7.6b by means of corresponding numbers. The exact occurrence of a sink or source invocation is depicted by a small black dot. Sink and source invocations that are fired simultaneously are labeled by the same value. Note that the resulting schedule is clearly not memory optimal because invocations are delayed although all required input data are available. Invocation 6, for instance, could be executed simultaneously with invocation 2. However, such a schedule would lead to irregular distances between the different lattice points. This corresponds to non-affine schedule functions and significantly complicates memory analysis. Consequently, this is kept for further research. Instead, the following methods will derive an upper bound for the required memory size, assuming implementation of the sub-optimal schedule. Alternatively, the simulative approach discussed in Chapter 6 can be employed due to its capacity to use arbitrary schedules.

7.3.3 Out-of-Order Communication

By imposing the row-major execution order, the embedded lattices define the relative execution order of the source and sink actors required for correct buffer analysis. In particular, they determine which invocations are executed sequentially or concurrently. Unfortunately, this excludes out-of-order communication as defined in Section 5.3, because in this case the invocations are not performed in raster-scan order. Even worse, the invocation orders for the sink and source actors might differ, which completely destroys the relation between commonly executed invocations.

Figure 7.7 illustrates the problem by means of the JPEG2000 block-building process. The source generates an image line by line as indicated by the Arabic numbers in Fig. 7.7b. The block builder, however, forwards them to the entropy encoder in different order (see also Section 2.2) as depicted by the arrows. This means that both actors traverse the corresponding lattices in different orders, which means that simultaneously performed invocations are not represented by coinciding semi-circles anymore. In other words, it destroys the scheduling information and, as a consequence, all possibilities for buffer analysis.

At a first glance, one possibility to solve this problem consists in using the hierarchical iteration vectors introduced in Section 6.2. They uniquely identify an actor invocation and have the advantage to be always traversed in lexicographic order.

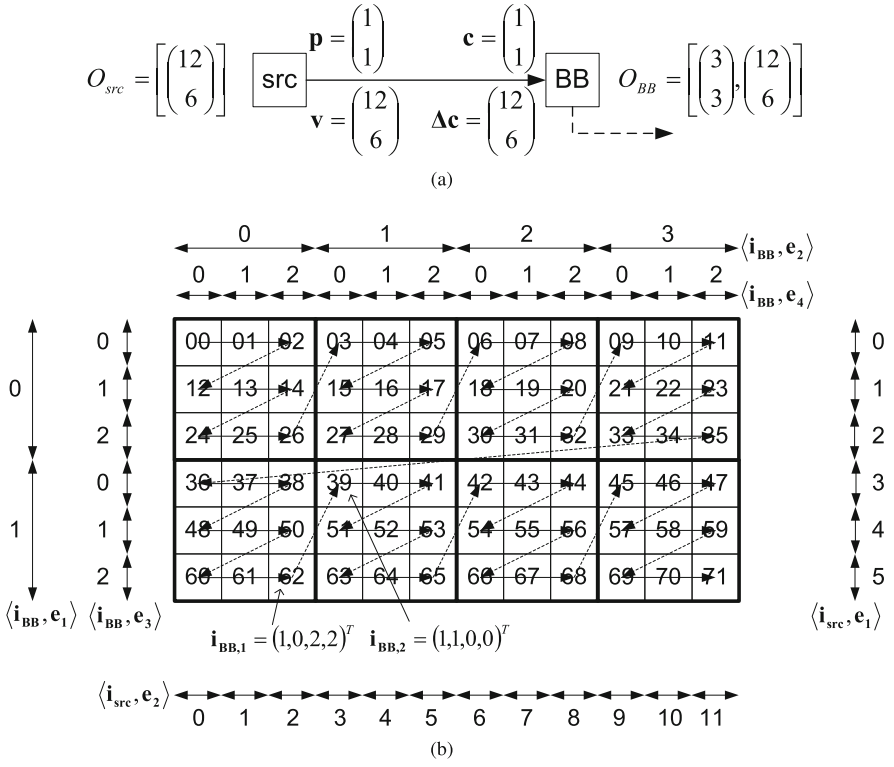


Fig. 7.7 Out-of-order communication for JPEG2000 block building. **a** WSDF graph. **b** Token space

Figure 7.7b indicates their interpretation for the JPEG2000 block building example. The source actor can be described by a two-dimensional hierarchical iteration vector while the block builder requires four dimensions:

$$0 \leq \mathbf{i}_{\text{src}} \leq \begin{pmatrix} 5 \\ 11 \end{pmatrix},$$

$$0 \leq \mathbf{i}_{\text{BB}} \leq \begin{pmatrix} 1 \\ 3 \\ 2 \\ 2 \end{pmatrix} = \mathbf{i}_{\text{BB,max}}.$$

Example 7.2 To refresh the principle of the hierarchical iteration vectors, Fig. 7.7b depicts a corresponding examples for the block builder actor. Each of its invocations is identified by a unique vector $\mathbf{i}_{\text{BB}}$, whose components are interpreted as shown in Fig. 7.7b. The maximum component values are defined by the vector $\mathbf{i}_{\text{BB,max}}$. $\mathbf{i}_{\text{BB,1}} = (1, 0, 2, 2)^T$, for instance, defines the last invocation in the first block in the second row of blocks (see Fig. 7.7b). In order to obtain the next block builder invocation $\mathbf{i}_{\text{BB,2}}$, $\mathbf{i}_{\text{BB,1}}$ has to be incremented in lexicographic order. In other words, we have to add the value 1 to the last coordinate. Since, however, $\langle \mathbf{i}_{\text{BB,1}}, \mathbf{e_4} \rangle = \langle \mathbf{i}_{\text{BB,max}}, \mathbf{e_4} \rangle$, a wraparound is performed that leads to $\langle \mathbf{i}_{\text{BB,2}}, \mathbf{e_4} \rangle = 0$ and to the necessity to increment coordinate $\mathbf{e_3}$. However, again, we have $\langle \mathbf{i}_{\text{BB,1}}, \mathbf{e_3} \rangle = \langle \mathbf{i}_{\text{BB,max}}, \mathbf{e_3} \rangle$ such that $\langle \mathbf{i}_{\text{BB,2}}, \mathbf{e_3} \rangle = 0$. Incrementation of the coordinate $\mathbf{e_2}$ finally leads to $\mathbf{i}_{\text{BB,2}} = (1, 1, 0, 0)^T$, which correctly identifies the next block builder invocation as depicted in Fig. 7.7b.

However, usage of the hierarchical iteration vectors results in lattices of different dimensions, which are clearly not optimal. This is because simultaneously executed sink and source invocations are identified by coinciding semi-circles. Since the lattices, however, have different dimensions, there will be many source invocation points without an associated sink invocation and vice versa. Since each point of the embedded lattice requires a given time for execution, this will result in bad throughput because both actors are sometimes not operating concurrently.

Figure 7.8a schematically depicts the occurring difficulties. For easier illustration, it assumes that the source lattice encompasses only one dimension while the sink lattice spans two dimensions. Consequently, although the overall number of lattice points is identical, there are many lattice points where no parallel execution of the sink and the source takes place. In other words, such a schedule leads to bad throughput.

In order to improve the situation, the hierarchical iteration vectors of the source actor can be translated into an alternative representation as illustrated in Fig. 7.9:

$$0 \leq \mathbf{i}^*_{\text{src}} \leq \begin{pmatrix} 1 \\ 2 \\ 3 \\ 2 \end{pmatrix}.$$

However, even in this case the resulting lattices have different sizes preventing perfect lattice embedding. Figure 7.8b schematically depicts the occurring difficulties assuming two-dimensional lattices. Although both of them encompass the same number of points, there are many lattice points where no parallel execution of the source and the sink actor takes place. In other words, this approach leads to schedules with bad throughput behavior as well.

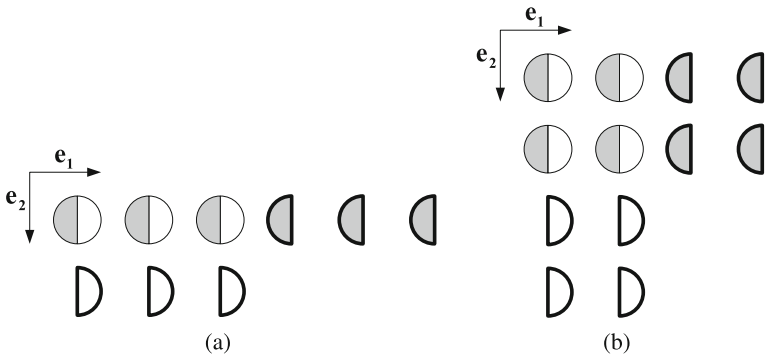


Fig. 7.8 Incompatible lattices. **a** Incompatible dimensions. **b** Incompatible extensions

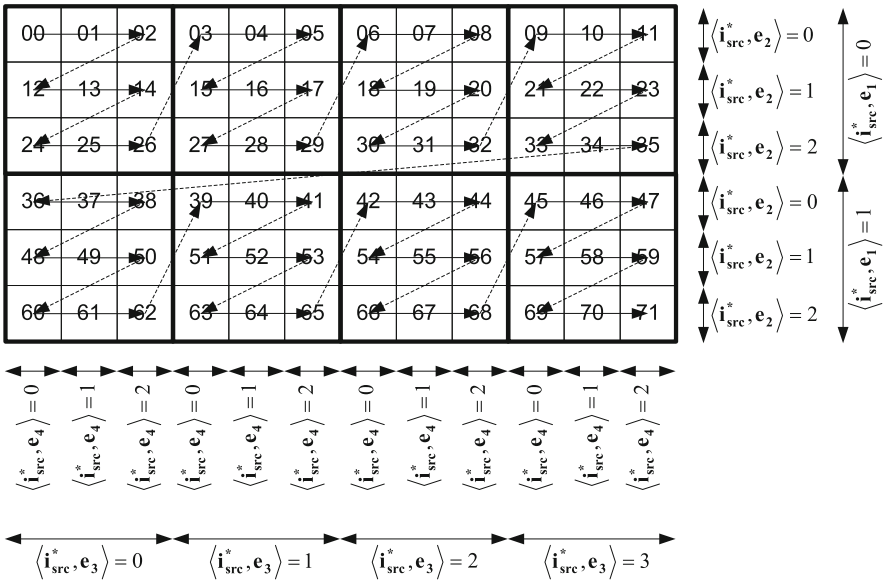


Fig. 7.9 Alternative enumeration of the source invocations

The occurring penalty can be quantified by calculating the number of coinciding lattice points. Assuming that the iteration vectors for both the source and the sink have the same number n of dimensions, then this number is given by the following equation:

$$\chi = \prod_{i=1}^{n} \left(\max \left(\langle \mathbf{i}_{\mathrm{src,max}}, \mathbf{e_i} \rangle, \langle \mathbf{i}_{\mathrm{snk,max}}, \mathbf{e_i} \rangle \right) + 1 \right).$$

In the best case, if data dependencies allow for, all coinciding lattice points are started concurrently. Furthermore, each point in the grid corresponds to a given amount of time, which can be imaged as a clock cycle. Thus, the overall amount of clock cycles necessary to invoke each lattice point is given by

$$T_\chi = \prod_{i=1}^{n} \left(\langle \mathbf{i}_{\mathrm{src,max}}, \mathbf{e_i} \rangle + 1 \right) + \prod_{i=1}^{n} \left(\langle \mathbf{i}_{\mathrm{snk,max}}, \mathbf{e_i} \rangle + 1 \right) - \chi .$$

An implementation with optimum throughput, on the other hand, would only need clock cycles.

$$T_{\mathrm{opt}} = \max \left(\prod_{i=1}^{n} \left(\langle \mathbf{i}_{\mathrm{src,max}}, \mathbf{e_i} \rangle + 1 \right), \prod_{i=1}^{n} \left(\langle \mathbf{i}_{\mathrm{snk,max}}, \mathbf{e_i} \rangle + 1 \right) \right),$$

Example 7.3 For the block building operation depicted in Fig. 7.7, the number of coinciding lattice points is given by

$$\chi = 2 \times 3 \times 3 \times 3 = 54.$$

Consequently,

$$\begin{aligned} T_\chi &= 12 \times 6 + 12 \times 6 - 54 = 90, \\ T_{\mathrm{opt}} &= 12 \times 6 = 72. \end{aligned}$$

In other words, incompatible lattice sizes cause an increment of 25% for the required execution time.

Example 7.4 Figure 7.10 shows another example in form of JPEG2000 tiling (see Section 2.2). The purpose of this operation consists in dividing an input image into different sub-images, also called tiles. The input image is produced in raster-scan order as defined by the numbers in Fig. 7.10. The sink, on the other hand, reads one tile after the other. So in case of Fig. 7.10, first the left sub-image is read, followed by the right one, as illustrated by arrows.

Such a communication behavior can be described by the following iteration vectors (see Fig. 7.10):

$$0 \leq \mathbf{i}_{\mathrm{src}} \leq \mathbf{i}_{\mathrm{src,max}} = (5, 1, 2)^T, \quad 0 \leq \mathbf{i}_{\mathrm{snk}} \leq \mathbf{i}_{\mathrm{snk,max}} = (1, 5, 2)^T .$$

Consequently, the number of coinciding lattice points is given by

$$\chi = 2 \times 2 \times 3 = 12.$$

This leads to

$$\begin{aligned} T_\chi &= 6 \times 6 + 6 \times 6 - 12 = 60, \\ T_{\mathrm{opt}} &= 36. \end{aligned}$$

In other words, for this example the execution time for all lattice points is even increased by 67%.

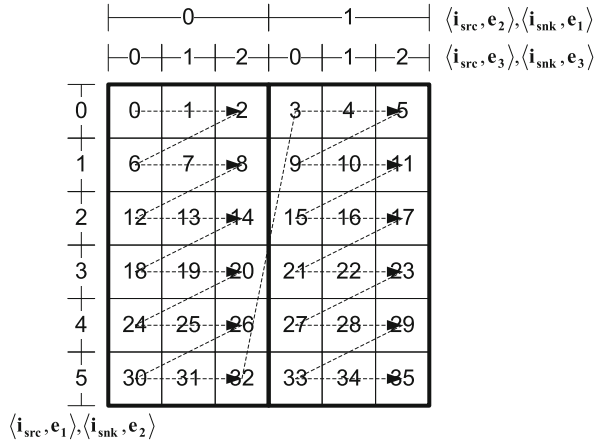


Fig. 7.10 JPEG2000 tiling operation and the resulting iteration vectors. Each *rectangle* corresponds to an image pixel that is produced and read by corresponding actor invocations. Numbers define the production order of the source, while *arrows* indicate the read order

In consequence of these observations, an alternative method is described in the following. Instead of using the hierarchical iteration vectors, the lattice dimensions are directly derived from the WSDF balance equation as described in Section 5.6. In other words, each lattice point corresponds to an actor invocation identified by a flat iteration vector **I**, which does **not** take execution order into account (see also Sections 5.1 and 6.2). For the block builder example shown in Fig. 7.7, this consequently leads to two lattices with the same dimensions and sizes. However, in order to represent the correct communication order, a remapping is performed.

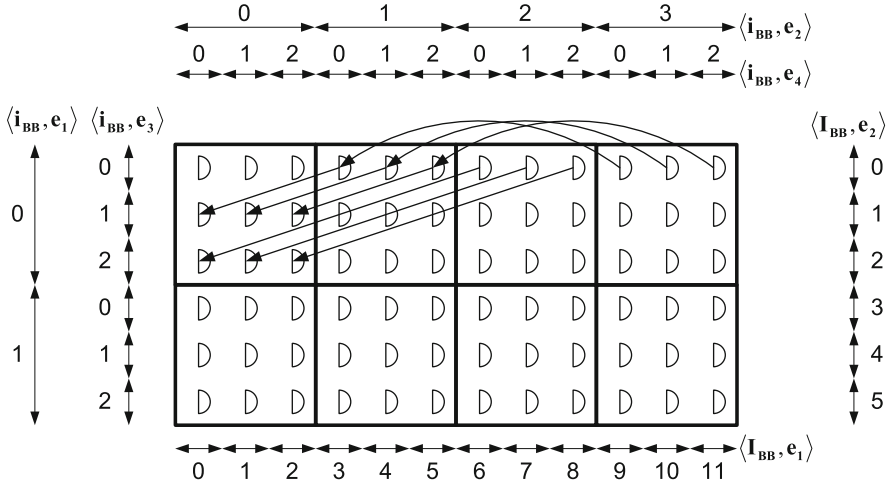


Fig. 7.11 Lattice point remapping for the JPEG2000 block building described in Fig. 7.7. Each *semi-circle* corresponds to one invocation of the block builder (BB). *Arrows* define the mapping between lattice points and actor invocations. For the given example, those are equivalent with the read data element position

Figure 7.11 shows the corresponding example for the block builder lattice where each point corresponds to an invocation of the BB actor. These lattice points are executed in row-major order from left to right and from top to bottom. The correct invocation order is taken into account by mapping each lattice point to the corresponding block builder invocation as depicted by the arrows. In order to avoid cluttering the figure, only some selected mappings are depicted. The first three lattice points are directly mapped to the corresponding BB invocation, because here no out-of-order communication occurs. The fourth lattice point, however, is remapped to the second row of the first block, and so on.

By these means, out-of-order communication does not lead to incompatible lattices. Consequently, it leads to schedules with optimized throughput. It only influences the dependency vectors as discussed in the following section.

7.3.4 Lattice Shifting Based on Dependency Vectors

In order to obtain valid schedules, a sink actor invocation must not be executed before the source actor has generated all required data elements. For this purpose, after grid scaling, for each edge $e \in E$, the *dependency vectors*[1] can be determined as shown in Fig. 7.5b. By indicating the data flow from the source lattice points to the sink, they define which source invocations a given sink invocation depends on.

Example 7.5 In order to illustrate the principles of the dependency vectors, Fig. 7.5a contains a bold rectangle, which corresponds to the lower left sliding window depicted in Fig. 7.4. The bold semi-circle depicted in Fig. 7.5a represents the corresponding sink invocation reading this sliding window. The dependency vectors start from the source invocations that generate the corresponding window pixels and point to the consuming sink lattice points. Note that due to border processing the first row of sink invocations only requires two input dependencies because border pixels are not produced by the source.

Based on those dependency vectors, it is possible to construct valid schedules by taking care that no pixel is read before being produced. Unfortunately, this is not automatically the case. In Fig. 7.5a, for instance, the bold sink execution depends on the striped source invocation although the latter is executed in the future due to the row-major execution order of the lattice. Mathematically, this corresponds to *anti-lexicographically negative* dependency vectors $\mathbf{d} \in \mathbb{Q}^n$, for which the following holds:

Definition 7.6 A vector $\mathbf{d}$ is called anti-lexicographically negative if and only if

$$\exists i : \langle \mathbf{d}, \mathbf{e_i} \rangle < 0 \wedge \forall j > i : \langle \mathbf{d}, \mathbf{e_j} \rangle \leq 0.$$

If this condition is true, the remainder of this monograph uses the following notation:

$$\mathbf{d} \underset{\leftarrow}{\prec} \mathbf{0}.$$

Such dependency vectors require to read data elements that are produced in the future. Consequently, they are not permitted, because they lead to *non-causal* behavior. Thus, in order

[1] In reference [163], the dependency vectors show the opposite direction. In other words, they specify for each sink invocation from which source invocation it depends on. This notation, however, has been changed in order to be in agreement with data flow semantics.

to construct valid schedules, the sink has to be delayed such that no dependency vector is anti-lexicographically negative. Figure 7.5b exemplarily depicts the result of this operation when applied to Fig. 7.5a.

In order to perform such a lattice shifting, it is necessary to determine the dependency vector that starts from the furthest future. Mathematically this corresponds to the anti-lexicographical minimum $\mathbf{D}_{\min}$, which can be determined by means of the following Integer Linear Program:

$$\mathbf{D}_{\min} = \min_{\overleftarrow{\prec}} \mathbf{D}, \tag{7.2}$$

$$\mathbf{0} \leq \mathbf{i}_{\text{src}} \leq \mathbf{i}_{\text{src,max}} \in \mathbb{N}_0^{n \times q_{\text{src}}}, \tag{7.3}$$

$$\mathbf{0} \leq \mathbf{I}_{\text{src}} \leq \mathbf{I}_{\text{src,max}} \in \mathbb{N}_0^{n}, \tag{7.4}$$

$$\mathbf{0} \leq \mathbf{i}_{\text{snk}} \leq \mathbf{i}_{\text{snk,max}} \in \mathbb{N}_0^{n \times q_{\text{snk}}}, \tag{7.5}$$

$$\mathbf{0} \leq \mathbf{I}_{\text{snk}} \leq \mathbf{I}_{\text{snk,max}} \in \mathbb{N}_0^{n}, \tag{7.6}$$

$$\mathbf{0} \leq \mathbf{I}_{\text{src,w}} < \mathbf{p} \in \mathbb{N}^{n}, \tag{7.7}$$

$$\mathbf{0} \leq \mathbf{I}_{\text{snk,w}} < \mathbf{c} \in \mathbb{N}^{n}, \tag{7.8}$$

$$0 = \langle \mathbf{O}_{\text{snk}}, \mathbf{I}_{\text{snk}} \rangle - \langle \mathbf{o}_{\text{snk}}, \mathbf{i}_{\text{snk}} \rangle, \tag{7.9}$$

$$\mathbf{0} = M_{\text{snk}} \times \left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right) - \mathbf{b}^{\mathbf{s}} - M_{\text{src}} \times \left(\mathbf{i}_{\text{src}}, \mathbf{I}_{\text{src,w}}\right) - \boldsymbol{\delta}, \tag{7.10}$$

$$\mathbf{0} \leq C_{\text{snk}} \times \left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right) - \mathbf{b}^{\mathbf{l}}_{\mathbf{c}}, \tag{7.11}$$

$$\mathbf{0} \geq C_{\text{snk}} \times \left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right) - \mathbf{b}^{\mathbf{h}}_{\mathbf{c}}, \tag{7.12}$$

$$0 = \langle \mathbf{O}_{\text{src}}, \mathbf{I}_{\text{src}} \rangle - \langle \mathbf{o}_{\text{src}}, \mathbf{i}_{\text{src}} \rangle, \tag{7.13}$$

$$\mathbf{D} = \text{diag}\left(\mathbf{s}_{\text{snk}}\right) \times \mathbf{I}_{\text{snk}} - \text{diag}\left(\mathbf{s}_{\text{src}}\right) \times \mathbf{I}_{\text{src}}. \tag{7.14}$$

Such an integer linear program (ILP) consists of an objective function defined by Eq. (7.2) and a set of linear equalities and inequalities given by Eqs. (7.3), (7.4), (7.5), (7.6), (7.7), (7.8), (7.9), (7.10), (7.11), (7.12), (7.13), and (7.14). In the present case, the objective function targets the anti-lexicographic minimum of all dependency vectors that are calculated in Eq. (7.14). diag $(\mathbf{s})$ defines an $n \times n$ matrix whose diagonal elements correspond to the components of $\mathbf{s}$:

$$\text{diag}\left(\mathbf{s}\right) = \begin{pmatrix} \langle \mathbf{s}, \mathbf{e_1} \rangle & 0 & & & 0 \\ 0 & \langle \mathbf{s}, \mathbf{e_2} \rangle & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \langle \mathbf{s}, \mathbf{e_{n-1}} \rangle & 0 \\ 0 & & & 0 & \langle \mathbf{s}, \mathbf{e_n} \rangle \end{pmatrix}. \tag{7.15}$$

$\mathbf{s}_{\text{src}}$ and $\mathbf{s}_{\text{snk}}$ represent the scaling vectors for the source and the sink of the considered communication edge $e \in E$ and have been derived in Section 7.3.2. $\mathbf{I}_{\text{src}}$ and $\mathbf{I}_{\text{snk}}$ define the flat iteration vectors of the considered source and sink invocation. The valid range of these vectors is defined via Eqs. (7.4) and (7.6). $\mathbf{I}_{\text{src,max}}$ and $\mathbf{I}_{\text{snk,max}}$ can be directly obtained from the WSDF balance equation (see Section 5.6). $\mathbf{0} \leq \mathbf{I}_{\text{src}} \leq \mathbf{I}_{\text{src,max}} \in \mathbb{N}_0^n$ means that each component $\langle \mathbf{I}_{\text{src}}, \mathbf{e_i} \rangle$ of $\mathbf{I}_{\text{src}}$ must be situated between 0 and $\langle \mathbf{I}_{\text{src,max}}, \mathbf{e_i} \rangle$.

Equations (7.13) and (7.9) perform the lattice point remapping required for out-of-order communication as discussed in Section 7.3.3. To this end, Eq. (7.13) maps the lattice point identified by $\mathbf{I}_{\text{src}}$ to the associated source iteration vector $\mathbf{i}_{\text{src}}$. Note that the mapping is **not** performed by the equations given in Definition 6.3, because the latter do not correspond to the remapping required for correct handling of out-of-order communication as discussed in Section 7.3.3. Instead, the remapping is performed via the order vectors $\mathbf{O}_{\text{src}}$ and $\mathbf{o}_{\text{src}}$. They are defined in such a way that $\langle \mathbf{O}_{\text{src}}, \mathbf{I}_{\text{src}} \rangle$ returns the invocation number of the lattice point $\mathbf{I}_{\text{src}}$ and that $\langle \mathbf{o}_{\text{src}}, \mathbf{i}_{\text{src}} \rangle$ returns the invocation number of the hierarchical iteration vector $\mathbf{i}_{\text{src}}$:

$$\forall 1 \leq i \leq n : \langle \mathbf{O}_{\text{src}}, \mathbf{e_i} \rangle = \prod_{j=1}^{i-1} \left(\langle \mathbf{I}_{\text{src,max}}, \mathbf{e_j} \rangle + 1 \right), \tag{7.16}$$

$$\forall 1 \leq i \leq n \times q_{\text{src}} : \langle \mathbf{o}_{\text{src}}, \mathbf{e_i} \rangle = \prod_{j=i+1}^{n \times q_{\text{src}}} \left(\langle \mathbf{i}_{\text{src,max}}, \mathbf{e_j} \rangle + 1 \right). \tag{7.17}$$

The equivalent holds for the sink invocation remapping:

$$\forall 1 \leq i \leq n : \langle \mathbf{O}_{\text{snk}}, \mathbf{e_i} \rangle = \prod_{j=1}^{i-1} \left(\langle \mathbf{I}_{\text{snk,max}}, \mathbf{e_j} \rangle + 1 \right), \tag{7.18}$$

$$\forall 1 \leq i \leq n \times q_{\text{snk}} : \langle \mathbf{o}_{\text{snk}}, \mathbf{e_i} \rangle = \prod_{j=i+1}^{n \times q_{\text{snk}}} \left(\langle \mathbf{i}_{\text{snk,max}}, \mathbf{e_j} \rangle + 1 \right). \tag{7.19}$$

The allowed range of the hierarchical source and sink iteration vectors is given by Eqs. (7.3) and (7.5), where q_{src} and q_{snk} define the number of firing levels as introduced in Definition 6.3. $\mathbf{i}_{\text{src,max}}$ and $\mathbf{i}_{\text{snk,max}}$ can be obtained from the corresponding sequence of firing blocks O_{src} and O_{snk}.

Equation (7.10) takes care that the so-determined hierarchical sink iteration vector $\mathbf{i}_{\text{snk}}$ depends on the corresponding source iteration vector $\mathbf{i}_{\text{src}}$. For this purpose, it requests that at least one data element produced by $\mathbf{i}_{\text{src}}$ is read by the sink invocation $\mathbf{i}_{\text{snk}}$. $M_{\text{snk}} \in \mathbb{N}^{n, n \times (q_{\text{snk}}+1)}$ and $M_{\text{src}} \in \mathbb{N}^{n, n \times (q_{\text{src}}+1)}$ are two matrices that map the hierarchical iteration vectors to the accessed data elements. $\mathbf{I}_{\text{snk,w}}$ and $\mathbf{I}_{\text{src,w}}$ define the considered position in the inner of the sliding window and effective token, respectively. Their allowed range is specified in Eqs. (7.8) and (7.7). The offset $\mathbf{b^S}$ results from virtual border extension, whereas $\boldsymbol{\delta}$ represents initial tokens (see Section 5.1.5). Equations (7.11) and (7.12) finally take care that only data elements not belonging to the extended border are taken into account because the latter is not produced by the source actor (see Section 5.1).

The so-defined ILP can be solved, for instance, by usage of the *PIP library* [5, 104], which is able to consider objective functions including lexicographic orders. Unfortunately, in general the solution of such ILPs can quickly become computational intensive. However, in the present case, the equalities and inequalities only depend on one single edge such that their complexity stays low. Consequently, for most application scenarios considered in this book, their solution only takes a fraction of seconds. Nevertheless, the experiments also revealed examples leading to severe difficulties in finding the correct solution, despite the simple structure of Eqs. (7.2), (7.3), (7.4), (7.5), (7.6), (7.7), (7.8), (7.9), (7.10), (7.11), (7.12), (7.13), and

(7.14). Fortunately, corresponding remedies could be found. However, their discussion has been deferred to Section 7.8. Instead, the following examples aim to clarify the interpretation of the above equations.

Example 7.7 In order to make the above ILP easier to understand, consider first the example WSDF edge exemplified in Fig. 7.4. From the number of required invocations, we can derive

$$\begin{aligned}\mathbf{I}_{\text{src,max}} &= (4,3)^T\,,\\ \mathbf{I}_{\text{snk,max}} &= (4,1)^T\,.\end{aligned}$$

Since no out-of-order communication is used, this directly leads to

$$\begin{aligned}\mathbf{i}_{\text{src,max}} &= (3,4)^T\,,\\ \mathbf{i}_{\text{snk,max}} &= (1,4)^T\,.\end{aligned}$$

Note that the coordinates have been switched, because the flat iteration vectors use the coordinate system depicted in Figs. 7.4 and 7.5, whereas the hierarchical iteration vectors are ordered such that the execution order corresponds to a lexicographic increment. Due to missing out-of-order communication, no lattice point remapping is required, leading to trivial order vectors

$$\begin{aligned}\mathbf{O}_{\text{src}} &= (1,5)^T\,,\\ \mathbf{o}_{\text{src}} &= (5,1)^T\,,\\ \mathbf{O}_{\text{snk}} &= (1,5)^T\,,\\ \mathbf{o}_{\text{snk}} &= (5,1)^T\,.\end{aligned}$$

In other words, Eq. (7.13) maps the ith hierarchical source iteration vector to the ith flat source iteration vector leading to

$$\begin{aligned}\langle \mathbf{I}_{\text{src}}, \mathbf{e_1}\rangle &= \langle \mathbf{i}_{\text{src}}, \mathbf{e_2}\rangle,\\ \langle \mathbf{I}_{\text{src}}, \mathbf{e_2}\rangle &= \langle \mathbf{i}_{\text{src}}, \mathbf{e_1}\rangle.\end{aligned}$$

Similarly, for the sink we obtain

$$\begin{aligned}\langle \mathbf{I}_{\text{snk}}, \mathbf{e_1}\rangle &= \langle \mathbf{i}_{\text{snk}}, \mathbf{e_2}\rangle,\\ \langle \mathbf{I}_{\text{snk}}, \mathbf{e_2}\rangle &= \langle \mathbf{i}_{\text{snk}}, \mathbf{e_1}\rangle.\end{aligned}$$

Next, we require the mapping matrices M_{src}, M_{snk} and mapping offset vectors $\boldsymbol{\delta}$, $\mathbf{b^s}$. Since no initial tokens are present on the edge depicted in Fig. 7.4, this leads to $\boldsymbol{\delta} = (0,0)^T$. Otherwise $\boldsymbol{\delta}$ would express the shift of the produced data elements caused by initial data elements as discussed in Section 5.1.5. As an effective token consists of only one single data element ($\mathbf{p} = (1,1)^T$, $\mathbf{I}_{\text{src,w}} = (0,0)^T$), and as the source does not use any out-of-order communication, the source mapping matrix gets trivial:

$$M_{\text{src}} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}.$$

If we consider, for instance, the source invocation $\mathbf{I}_{\text{src}} = (3, 1)^T$, this leads to $\mathbf{i}_{\text{src}} = (1, 3)^T$ and consequently to

$$\begin{aligned} & M_{\text{src}} \times \left(\mathbf{i}_{\text{src}}, \mathbf{I}_{\text{src,w}}\right) + \boldsymbol{\delta} \\ = & \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 3 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\ = & \begin{pmatrix} 3 \\ 1 \end{pmatrix}, \end{aligned}$$

which corresponds exactly to the data element marked in Fig. 7.4. Note that the extended border is excluded from this consideration because it is not produced by the source actor.

Concerning the sink, the situation is slightly more complex due to virtual border extension and downsampling:

$$\begin{aligned} \mathbf{b}^{\mathbf{S}} &= \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \\ M_{\text{snk}} &= \begin{pmatrix} 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{pmatrix}. \end{aligned}$$

$\langle M_{\text{snk}} \times \mathbf{e_1}, \mathbf{e_2} \rangle = 2$ tells that the sliding window moves by 2 pixels in vertical direction. The mapping offset vector essentially says that the first pixel of the first sliding window does not correspond to the first data element produced by the source due to the virtual border extension. This means, however, that the corresponding window data element does not cause any data dependency. Consequently, Eqs. (7.11) and (7.12) exclude data elements situated on the virtually extended border:

$$\begin{aligned} C_{\text{snk}} &= \begin{pmatrix} 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{pmatrix}, \\ \mathbf{b}_{\mathbf{c}}^{\mathbf{l}} &= \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \\ \mathbf{b}_{\mathbf{c}}^{\mathbf{h}} &= \begin{pmatrix} 4 \\ 4 \end{pmatrix}. \end{aligned}$$

Consider, for instance, the first sink invocation $\mathbf{I}_{\text{snk}} = (0, 0)^T = \mathbf{i}_{\text{snk}}$ and the first data element $\mathbf{I}_{\text{snk,w}} = (0, 0)^T$ in the sliding window. Then

$$\begin{aligned} & C_{\text{snk}} \times \left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right) - \mathbf{b}_{\mathbf{c}}^{\mathbf{l}} \\ = & \begin{pmatrix} 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ = & -\begin{pmatrix} 1 \\ 0 \end{pmatrix} < \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \end{aligned}$$

In other words, this data element is excluded by Eq. (7.11), because it is situated on the virtually extended border. Consider, on the other hand, $\mathbf{I}_{\text{snk}} = (4, 1)^T$ and the last data element $\mathbf{I}_{\text{snk,w}} = (0, 2)^T$ of the sliding window. This leads to

$$\begin{aligned}
\mathbf{i}_{\text{snk}} &= (1, 4)^T \\
&\Rightarrow C_{\text{snk}} \times \left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right) - \mathbf{b}_{\mathbf{c}}^{\mathbf{h}} \\
&= \begin{pmatrix} 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 4 \\ 0 \\ 2 \end{pmatrix} - \begin{pmatrix} 4 \\ 4 \end{pmatrix} \\
&= \begin{pmatrix} 4 \\ 4 \end{pmatrix} - \begin{pmatrix} 4 \\ 4 \end{pmatrix} \\
&\geq \begin{pmatrix} 0 \\ 0 \end{pmatrix}.
\end{aligned}$$

In other words, Eq. (7.12) correctly indicates that this pixel is not situated on any virtually extended border and thus induces data dependencies from the source actor.

With these definitions, Eq. (7.10) relates each sink invocation $\mathbf{i}_{\text{snk}}$ and hence $\mathbf{I}_{\text{snk}}$ to those source invocations which it depends on. Consequently, the dependency vectors pointing from the source invocation producing the required data element to the consuming sink invocation can be calculated as

$$\mathbf{D} = \mathbf{I}_{\text{snk}} - \mathbf{I}_{\text{src}}.$$

This, however, does not take into account that the lattice points might be scaled as, for instance, depicted in Fig. 7.5. Consequently, Eq. (7.14) scales each vector by the scale factors derived in Section 7.3.2. For the example given in Fig. 7.4, this leads to

$$\begin{aligned}
\mathbf{s}_{\text{src}} &= (1, 1)^T. \\
\mathbf{s}_{\text{snk}} &= (1, 2)^T.
\end{aligned}$$

With these definitions, the ILP solver traverses all sink and source invocations and calculates the dependency vectors as depicted in Fig. 7.5. The objective function returns the anti-lexicographic minimum:

$$\mathbf{D}_{\text{min}} = (0, -1)^T.$$

This permits to shift the sink lattice such that no data elements produced in the future will be read.

Example 7.8 After having explained the basic principles of the ILP for determination of $\mathbf{D}_{\text{min}}$, this example aims to focus on out-of-order communication by reconsidering Fig. 7.7. In this case, we obtain

$$\begin{aligned}
\mathbf{I}_{\text{src,max}} &= (11, 5)^T, \\
\mathbf{I}_{\text{snk,max}} &= (11, 5)^T.
\end{aligned}$$

In order to correctly describe the occurring out-of-order communication, the following hierarchical iteration vectors have to be used (see also Section 7.3.3):

$$\begin{aligned}\mathbf{i}_{\text{src,max}} &= (5, 11)^T\,,\\ \mathbf{i}_{\text{snk,max}} &= (1, 3, 2, 2)^T\,.\end{aligned}$$

Furthermore, this leads to the following order vectors:

$$\begin{aligned}\mathbf{O}_{\text{src}} &= (1, 12)^T\,,\\ \mathbf{o}_{\text{src}} &= (12, 1)^T\,,\\ \mathbf{O}_{\text{snk}} &= (1, 12)^T\,,\\ \mathbf{o}_{\text{snk}} &= (27, 9, 3, 1)^T\,.\end{aligned}$$

Whereas the source produces its data in raster-scan order, which corresponds to Example 7.7, the sink—in form of the block builder—merits additional consideration. Application of Eq. (7.9) leads to the following remapping:

$\mathbf{I}_{\text{snk}} = \mathbf{I}_{\mathbf{BB}}$	$\langle \mathbf{O}_{\text{snk}}, \mathbf{I}_{\text{snk}} \rangle$	$\mathbf{i}_{\text{snk}} = \mathbf{i}_{\mathbf{BB}}$	$\langle \mathbf{o}_{\text{snk}}, \mathbf{i}_{\text{snk}} \rangle$
$(0, 0)^T$	0	$(0, 0, 0, 0)^T$	0
$(1, 0)^T$	$1 \times 1 = 1$	$(0, 0, 0, 1)^T$	$1 \times 1 = 1$
$(2, 0)^T$	$2 \times 1 = 2$	$(0, 0, 0, 2)^T$	$2 \times 1 = 2$
$(3, 0)^T$	$3 \times 1 = 3$	$(0, 0, 1, 0)^T$	$1 \times 3 = 3$
$\ldots$	$\ldots$	$\ldots$	$\ldots$
$(5, 0)^T$	$5 \times 1 = 5$	$(0, 0, 1, 2)^T$	$1 \times 3 + 2 \times 1 = 5$
$(6, 0)^T$	$6 \times 1 = 6$	$(0, 0, 2, 0)^T$	$2 \times 3 = 6$
$\ldots$	$\ldots$	$\ldots$	$\ldots$
$(9, 0)^T$	$9 \times 1 = 9$	$(0, 1, 0, 0)^T$	$1 \times 9 = 9$
$\ldots$	$\ldots$	$\ldots$	$\ldots$

In other words, Eq. (7.9) performs exactly the remapping discussed in Section 7.3.3 and depicted in Fig. 7.11.

The data element mapping matrices and offsets can be calculated this way

$$\begin{aligned}M_{\text{src}} &= \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix},\\ \boldsymbol{\delta} &= (0, 0)^T\,,\\ M_{\text{snk}} &= \begin{pmatrix} 0 & 3 & 0 & 1 & 1 & 0 \\ 3 & 0 & 1 & 0 & 0 & 1 \end{pmatrix},\\ \mathbf{b}^{\mathbf{s}} &= (0, 0)^T\,.\end{aligned}$$

Considering, for instance, the block builder invocation $\mathbf{i}_{\text{snk}} = \mathbf{i}_{\mathbf{BB}} = (1, 0, 2, 2)^T$ as depicted in Fig. 7.7b, this leads to

$$M_{\text{snk}} \times \left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right) - \mathbf{b}^{\mathbf{s}}$$

$$= \begin{pmatrix} 0 & 3 & 0 & 1 & 1 & 0 \\ 3 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \\ 2 \\ 2 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$= \begin{pmatrix} 2 \\ 5 \end{pmatrix},$$

which corresponds exactly to what is illustrated in Fig. 7.7b. Equations (7.11) and (7.12) are not important as we do not have any virtually extended border.

Figure 7.12a depicts the resulting dependency vectors for the first three rows of the block builder actor. As can be seen, they are much less regular than for the sliding window algorithm

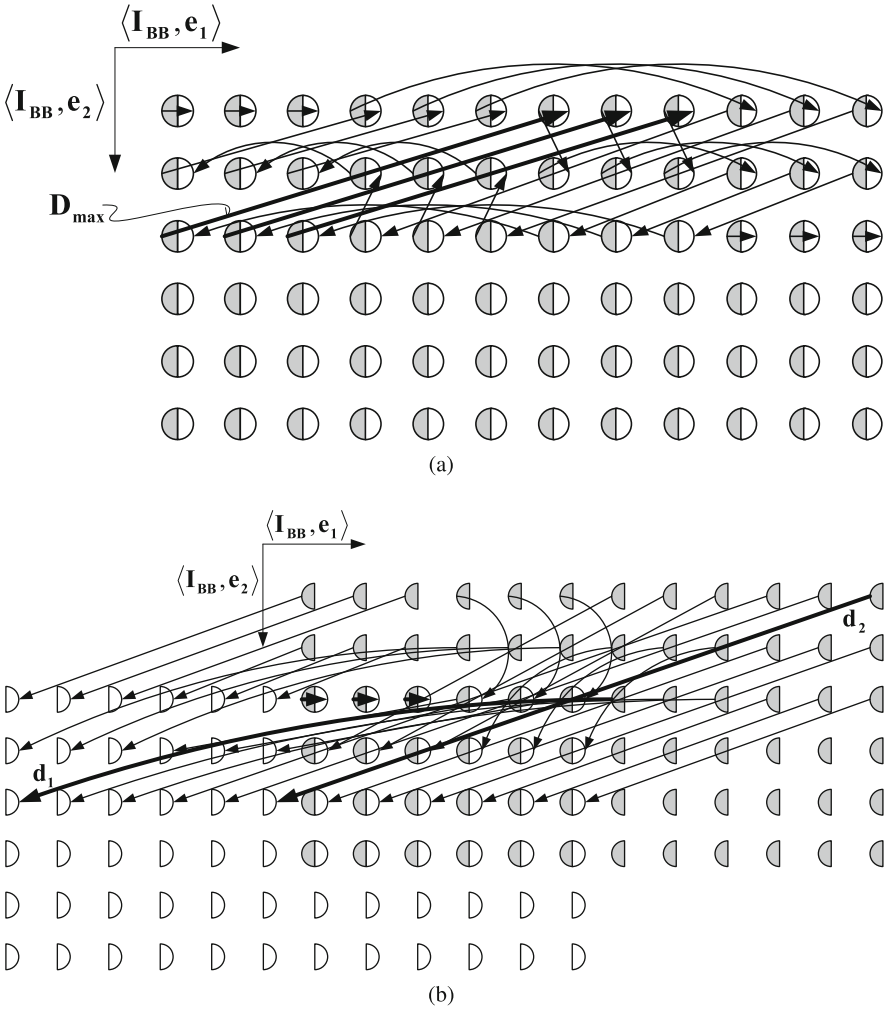


Fig. 7.12 Dependency vectors for the JPEG2000 block building. **a** Dependency vectors. **b** Shifted lattices

depicted in Fig. 7.5a. This is because of the remapping required for correct out-of-order communication. This, by the way, is the major reason for using an ILP approach when determining the dependency vectors; for ordinary sliding window applications, they could be determined in a much simpler way (see also [282]).

Figure 7.12b depicts the result when shifting the sink lattice such that no block builder requires data elements produced in the future. Obviously, the out-of-order communication requires a huge shift due to a big anti-lexicographic minimum vector $\mathbf{D}_{\min}$. As a consequence, this generates many lattice points where either only a source or a sink invocation takes place, leading to bad throughput. Fortunately, this can be avoided by a new concept of lattice wraparound as described later on in Section 7.4.

Remark 7.9 Note that in case of existing initial data elements, a sink schedule period might depend on two source schedule periods. In order to handle this situation correctly, the above ILP has to be extended by a variable describing which schedule period shall be taken into account. As this further complicates notation, this step will not be detailed. Instead, the interested reader can find the principles in Section 8.2.7.3.

7.3.5 Pipelined Actor Execution

In the previous section, it was assumed that the sink lattice is shifted by $-\mathbf{D}_{\min}$ in order to ensure that only data elements already available are read by the sink. Whereas this leads to valid schedules, it causes that certain sink lattice points coincide with the source invocations producing the required data elements. This can, for instance, be seen in Figs. 7.5b and 7.12b, which both contain dependency vectors of zero length after shifting. However, this means an increased execution time for the corresponding lattice point because the sink and source actor cannot be executed in parallel. In other words, first the source invocation has to terminate before the sink actor can start operation. As a consequence, this sequential operation leads to reduced system throughput.

Fortunately, this can be solved rather easily by additional shifting of the sink lattice. Figure 7.13 exemplarily depicts this operation for the WSDF graph shown in Figs. 7.4 and 7.5. In Fig. 7.13a, the sink lattice has additionally been shifted by one in direction $\mathbf{e_1}$. Consequently, the length of each dependency vector is larger than zero, permitting for parallel execution of the source and sink actor invocations. Note, however, that the source actor is still required to calculate the result within one invocation.

This contradicts the principle of pipelining, widely employed in efficient hardware design. In this case, an actor is allowed to spend several invocations until writing the final result while still being able to generate one result per invocation. Fortunately, this can be easily taken into account by additional shifting of the sink lattice. In Fig. 7.13b, for instance, the bold source invocation starts calculation of an output value. However, in contrast to Fig. 7.13a, the corresponding write operation needs not to be performed before the next source invocation, because the sink does not immediately require the produced data element. Instead, the source can start to calculate a second output value while still processing the first one, resulting in a pipeline length of 2.

Note, however, that additional shifting is not possible in case the overall WSDF graph contains tight feedback loops, as, for instance, shown in Fig. 5.19. There, the data element produced by the *max*-actor has to be completely processed by the *min*-actor and the *duplicator* before the next *max*-actor invocation is possible. This, however, is not a problem of the buffer analysis method, but of the underlying WSDF graph that makes it impossible to parallely execute the three involved actors.

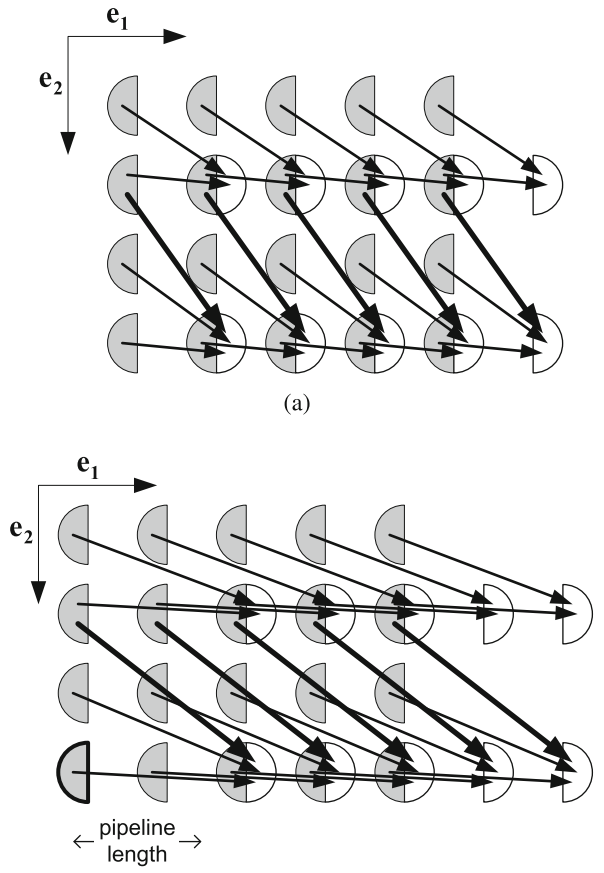


Fig. 7.13 Additional sink shifting for pipelined execution. **a** Parallel sink and source execution. **b** Pipelined source invocation

7.4 Lattice Wraparound

As shown in Example 7.8, out-of-order communication can lead to long dependency vectors requiring huge shifts in order to avoid that data elements will be read before their production. This leads, however, to many lattice points where only one of the two actors connected to an edge is executed. Considering, for instance, Fig. 7.12b, only a small part of the lattice points effectively overlap, although the overall number of source and sink invocations is identical. However, a *throughput-optimal* schedule requires that both of them operate permanently in parallel.

Definition 7.10 Given a WSDF graph $G = (A, E)$ whose actors $a \in A$ are mapped to fixed hardware resources for execution. Then a WSDF graph schedule is called *throughput optimal*, if at least one of these resources is never stalled due to missing data or full buffers. In case each actor is mapped to a dedicated resource, this also means that there exists at least one actor that is permanently busy.

Remark 7.11 For graphs with tight feed-back loops and depending on the mapping of actors to hardware resources, creation of throughput-optimal schedules might be impossible.

Example 7.12 Consider the schedule for JPEG2000 block building whose lattice representation is given in Fig. 7.12. For the following consideration we now have to take into account that multidimensional data flow graphs execute on infinite streams of arrays. Consequently, the lattice points have to be repeated infinitely in dimension $\mathbf{e_2}$, leading to the result depicted in Fig. 7.14. But even with this fact in mind, half of the sink lattice points are not coinciding with any source lattice point, although the overall number of source and sink lattice points is strictly identical. Let N_{total} be this number, then a throughput-optimal implementation needs N_{total} clock cycles for executing these lattice points, while the schedule depicted in Fig. 7.14 requires $N_{\text{total}} + \frac{N_{\text{total}}}{2}$ clock cycles. Unfortunately, this represents an increase of 50%.[2]

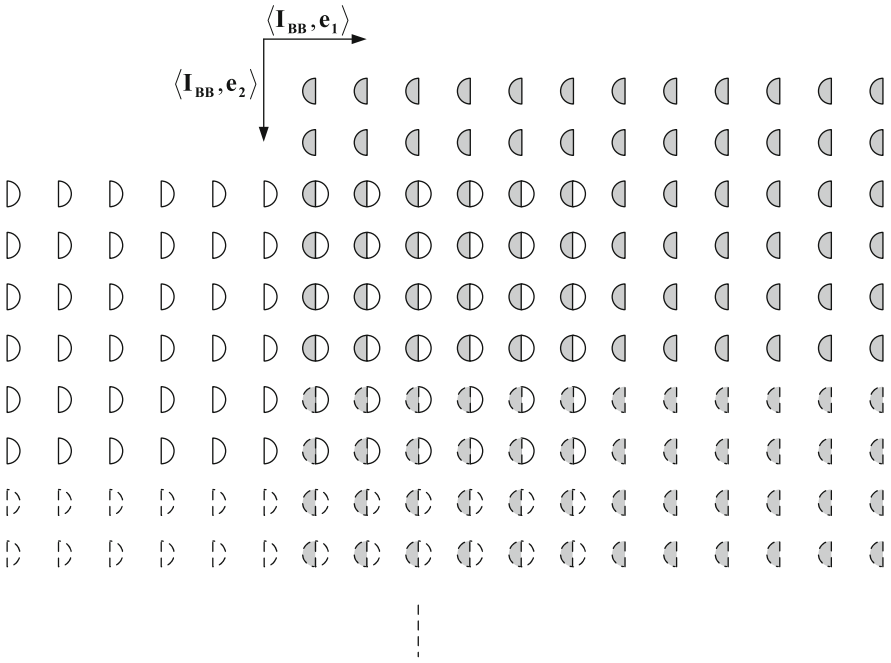


Fig. 7.14 Lattice schedule representation for JPEG2000 block building on an infinite stream of arrays (see also Figs. 7.7 and 7.12). *Dashed lines* define the actor executions belonging to the second schedule period

In order to solve this problem, the following sections present a concept of lattice wraparound that helps to describe throughput-optimal schedules. Before giving a formal description in Section 7.4.2, Section 7.4.1 summarizes its principles. Section 7.4.3 finally discusses the modifications required for scheduling of lattices with wraparound.

[2] Note that the JPEG2000 block forming operation has no feedback cycles, and thus implementation of a throughput-optimal schedule is possible.

7.4.1 Principle of Lattice Wraparound

In order to avoid schedules with bad throughput, it is assumed that the lattice has a limited extent given by a vector $\mathbf{l}_{\text{lattice}}^{\text{wrapped}}$. Since WSDF graphs operate on infinite streams of data, its last component is set to $\langle \mathbf{l}_{\text{lattice}}^{\text{wrapped}}, \mathbf{e_n} \rangle = \infty$ by definition. The other components of vector $\mathbf{l}_{\text{lattice}}^{\text{wrapped}}$ are finite. A lattice point that exceeds these dimensions is supposed to be wrapped around.

Figure 7.15 shows a corresponding example. Here, the dashed sink lattice points are assumed to be replaced by the bold ones. Consequently, both the sink and the source actor would operate in a fully parallel manner. In other words, by permitting that lattice points exceeding the lattice dimensions coincide with other lattice points, the throughput of the system can be improved.

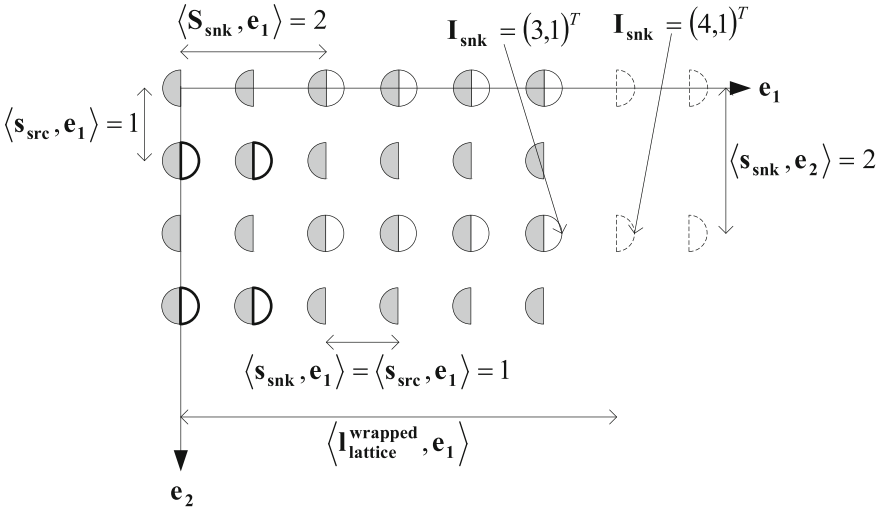


Fig. 7.15 Principle of lattice wraparound

The wraparound is performed relatively to a common reference grid that defines the common *system clock*. In Fig. 7.15, this reference grid happens to coincide with the source lattice points. Each sink invocation exceeding the lattice borders is thus moved into the next row of the reference grid.

In general, the reference grid is defined by requesting

$$\forall a \in A : \langle \mathbf{s_a}, \mathbf{e_i} \rangle \in \mathbb{N}, \tag{7.20}$$

where $\mathbf{s_a}$ is the scaling factor of each actor $a \in A$ as calculated in Section 7.3.2. As the lattice representation only defines relative execution orders, this can be achieved by simple multiplication of all scaling vectors $\mathbf{s_a}$ with a given factor. The reference grid is then the grid $\mathbb{N}^n$, n being the number of token dimensions of the WSDF graph.

Remark 7.13 As each lattice point corresponds to a time bin or clock cycle, the grid scaling should ideally be done such that

$$\forall 1 \leq i \leq n : \exists a \in A : \langle \mathbf{s_a}, \mathbf{e_i} \rangle = 1.$$

Whereas this is possible for most applications, there exist scenarios that do not permit to fulfill this condition (see, for instance, Fig. 7.6).

7.4.2 Formal Description of the Lattice Wraparound

The formal description of the lattice wraparound requires determination of the target lattice size $\mathbf{l}^{\text{wrapped}}_{\text{lattice}} \geq \mathbf{l}^{\text{wrapped}}_{\text{lattice,min}}$ after wraparound. Its minimum value $\mathbf{l}^{\text{wrapped}}_{\text{lattice,min}}$ can be derived after having scaled all actor lattices of a WSDF graph $G = (A, E)$ as described in Section 7.3.2:

$$\forall 1 \leq i < n : \langle \mathbf{l}^{\text{wrapped}}_{\text{lattice,min}}, \mathbf{e_i} \rangle = \max_{a \in A} \left(\left(\langle \mathbf{I_{a,\max}}, \mathbf{e_i} \rangle + 1 \right) \times \langle \mathbf{s_a}, \mathbf{e_i} \rangle \right), \tag{7.21}$$

$$\langle \mathbf{l}^{\text{wrapped}}_{\text{lattice,min}}, \mathbf{e_n} \rangle = \infty, \tag{7.22}$$

where $\left(\langle \mathbf{I_{a,\max}}, \mathbf{e_i} \rangle + 1 \right)$ defines the number of invocations of actor $a \in A$ in dimension $\mathbf{e_i}$ as calculated by the WSDF balance equation (see Section 5.6). $\mathbf{s_a} \in \mathbb{N}^n$ corresponds to the scaling factor of actor $a \in A$ as calculated in Section 7.3.2, n being the number of token dimensions of the WSDF graph. $\langle \mathbf{l}^{\text{wrapped}}_{\text{lattice,min}}, \mathbf{e_n} \rangle$ has been set to infinity because WSDF graphs operate on infinite streams of data.

Now, let $\mathbf{x}(\mathbf{I_a})$ be the position of the lattice point belonging to the actor invocation $\mathbf{I_a}$, where $\mathbf{I_a} \in \mathbb{N}^n$ is a flat iteration vector (see Sections 5.1.1 and 6.2):

$$\langle \mathbf{x}(\mathbf{I_a}), \mathbf{e_i} \rangle = \langle \mathbf{I_a}, \mathbf{e_i} \rangle \times \langle \mathbf{s_a}, \mathbf{e_i} \rangle + \langle \mathbf{S_a}, \mathbf{e_i} \rangle.$$

$\mathbf{S_a} \in \mathbb{N}^n$ corresponds to the lattice shift calculated in Section 7.3.4. Then, the new lattice point coordinates after the wraparound are given by $W(\mathbf{x}(\mathbf{I_a}))$ with

$$k_j = \left\lfloor \frac{\langle \mathbf{x}(\mathbf{I_a}), \mathbf{e_j} \rangle + k_{j-1}}{\langle \mathbf{l}^{\text{wrapped}}_{\text{lattice}}, \mathbf{e_j} \rangle} \right\rfloor \tag{7.23}$$

$$k_0 = 0 \tag{7.24}$$

$$k_n = 0 \tag{7.25}$$

$$\langle W(\mathbf{x}(\mathbf{I_a})), \mathbf{e_j} \rangle = \langle \mathbf{x}(\mathbf{I_a}), \mathbf{e_j} \rangle + k_{j-1} - k_j \times \langle \mathbf{l}^{\text{wrapped}}_{\text{lattice}}, \mathbf{e_j} \rangle. \tag{7.26}$$

Example 7.14 In order to illustrate the above equations, consider two example invocations of the sink actor given in Fig. 7.15

$$\mathbf{I}_{\text{snk},\mathbf{1}} = (3, 1)^T,$$
$$\mathbf{I}_{\text{snk},\mathbf{2}} = (4, 1)^T.$$

with

$$\mathbf{l}_{\text{lattice}}^{\text{wrapped}} = \langle \mathbf{l}_{\text{lattice,min}}^{\text{wrapped}} \rangle = (6, \infty)^T .$$

The position of the corresponding lattice points can be obtained from Fig. 7.15 as

$$\mathbf{x}\left(\mathbf{I}_{\text{snk},\mathbf{1}}\right) = (5, 2)^T ,$$
$$\mathbf{x}\left(\mathbf{I}_{\text{snk},\mathbf{2}}\right) = (6, 2)^T .$$

Consequently,

$$\langle W\left(\mathbf{x}\left(\mathbf{I}_{\text{snk},\mathbf{1}}\right)\right), \mathbf{e_1} \rangle = 5 - \left\lfloor \frac{5}{6} \right\rfloor \times 6 = 5,$$
$$\langle W\left(\mathbf{x}\left(\mathbf{I}_{\text{snk},\mathbf{1}}\right)\right), \mathbf{e_2} \rangle = 2 + \left\lfloor \frac{5}{6} \right\rfloor - 0 = 2,$$
$$\langle W\left(\mathbf{x}\left(\mathbf{I}_{\text{snk},\mathbf{2}}\right)\right), \mathbf{e_1} \rangle = 6 - \left\lfloor \frac{6}{6} \right\rfloor \times 6 = 0,$$
$$\langle W\left(\mathbf{x}\left(\mathbf{I}_{\text{snk},\mathbf{2}}\right)\right), \mathbf{e_2} \rangle = 2 + \left\lfloor \frac{6}{6} \right\rfloor - 0 = 3.$$

Thus, whereas $W\left(\mathbf{x}\left(\mathbf{I}_{\text{snk},\mathbf{1}}\right)\right) = \mathbf{I}_{\text{snk},\mathbf{1}}$ remains unchanged because it does not exceed the lattice borders, $\mathbf{I}_{\text{snk},\mathbf{2}}$ is wrapped around as depicted in Fig. 7.15.

7.4.3 Lattice Shifting for Lattices with Wraparound

As already discussed in Section 7.3.4, construction of valid schedules requires to take care that data elements are not read before their creation. Typically, this is done by performing a lattice shift such that no dependency vector is anti-lexicographically negative. However, in case of lattice wraparound, this condition is not sufficient anymore.

Figure 7.16 depicts a corresponding example assuming two arbitrary dependency vectors. Figure 7.16b illustrates the results after shifting the sink lattice in such a way that none of the dependency vectors is anti-lexicographically negative. Figure 7.16c finally shows the outcome of the wraparound. Obviously, vector $\mathbf{D_2}$ stays anti-lexicographically negative, thus leading to an illegal schedule.

The underlying reason can be clarified by unrolling all lattice points of Fig. 7.16c into one row as depicted in Fig. 7.16d. Application of the lattice wraparound immediately leads to the original configuration. However, in this unrolled representation the dependency vector $\mathbf{D_1}$ corresponds only to a delay of one lattice point, while dependency vector $\mathbf{D_2}$, although anti-lexicographically larger, requires a delay of 8 lattice points.

Thus, in order to generate valid schedules, the objective function of the ILP defined by s (7.2), (7.3), (7.4), (7.5), (7.6), (7.7), (7.8), (7.9), (7.10), (7.11), (7.12), (7.13), and (7.14) is replaced by the following function:

$$Delay = \max \sum_{i=1}^{n} \langle -\mathbf{D}, \mathbf{e_i} \rangle \prod_{j=1}^{i-1} \langle \mathbf{l}_{\text{lattice}}^{\text{wrapped}}, \mathbf{e_j} \rangle \tag{7.27}$$

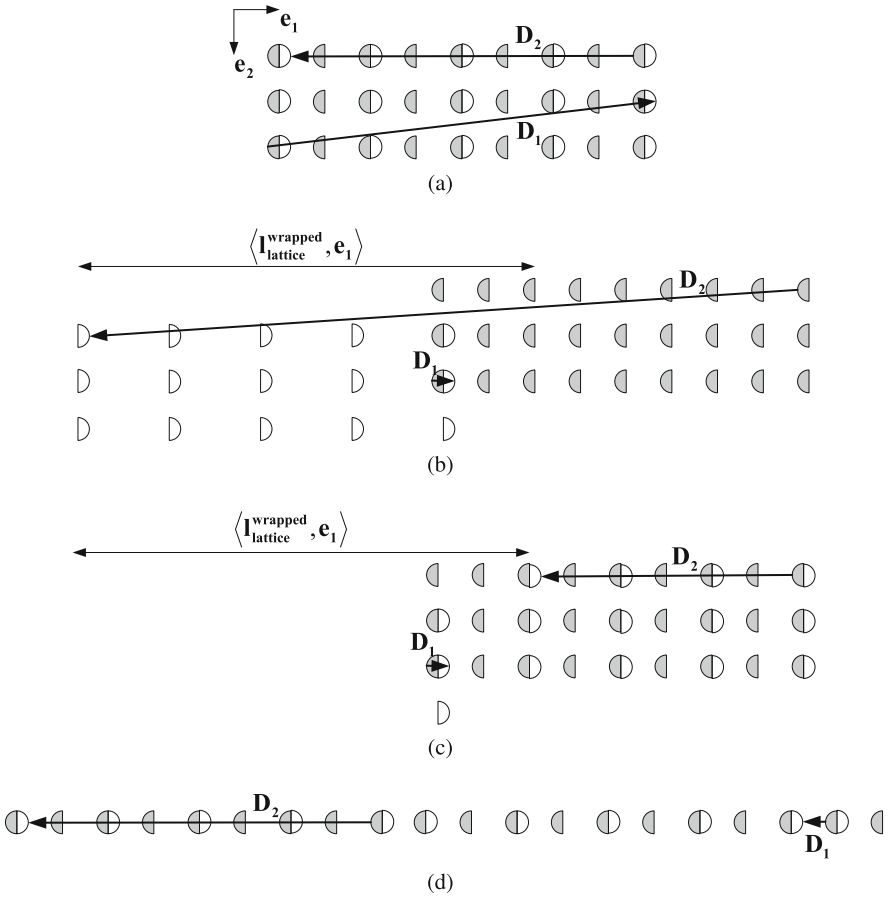


Fig. 7.16 Lattice shift in the presence of wraparound. **a** Original lattice. **b** Shifted lattice. **c** With wrap around. **d** Unrolled lattice

$$= \max\left(-\sum_{i=1}^{n}\langle \mathbf{D}, \mathbf{e_i}\rangle \prod_{j=1}^{i-1}\langle \mathbf{l}_{\text{lattice}}^{\text{wrapped}}, \mathbf{e_j}\rangle\right) \tag{7.28}$$

$$= \min\sum_{i=1}^{n}\langle \mathbf{D}, \mathbf{e_i}\rangle \prod_{j=1}^{i-1}\langle \mathbf{l}_{\text{lattice}}^{\text{wrapped}}, \mathbf{e_j}\rangle. \tag{7.29}$$

As a consequence, the solution of the ILP returns that dependency vector which requires the biggest delay in the number of reference lattice points.

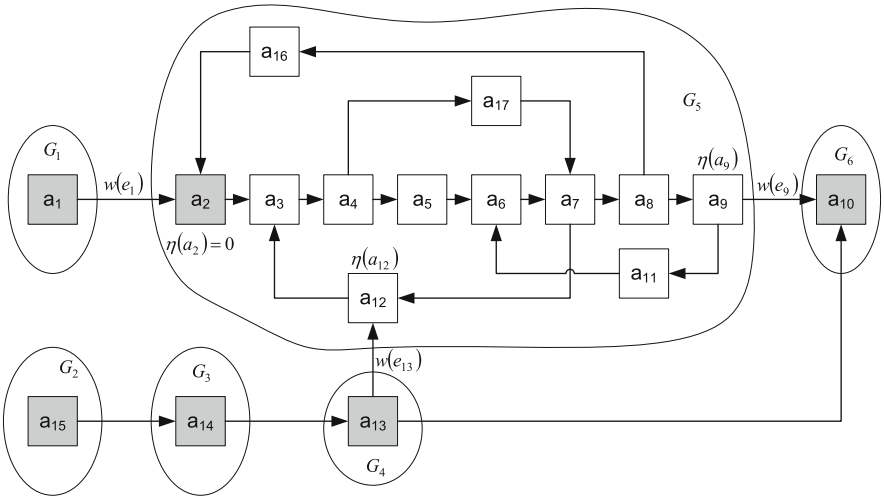
Remark 7.15 As can seen by means of Fig. 7.16, lattice wraparound can increase the required delay. Typically, this does not cause any difficulties, but ensures that the buffer size is set accordingly in order to permit throughput-optimized schedules. In the presence of tight feedback loops, however, this can lead to deadlocks. Fortunately, they can be avoided by increas-

ing $\mathbf{l}_{\text{lattice}}^{\text{wrapped}}$. Note that $\langle \mathbf{l}_{\text{lattice}}^{\text{wrapped}}, \mathbf{e_j} \rangle \to \infty$ leads to the original objective function given in Eq. (7.2).

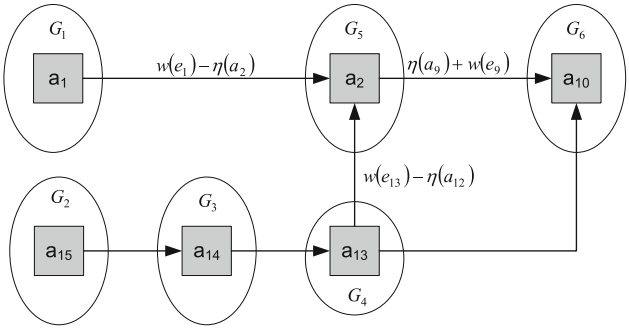
7.5 Scheduling of Complete WSDF Graphs

The previous sections have introduced the basic principles for polyhedral scheduling of WSDF graphs. In particular, it has been shown how WSDF edges can be translated into a lattice representation. The latter permits to determine the minimum delay of the sink actor guaranteeing that no data element is read before being created.

This section is devoted to the question how these concepts can be applied to complete WSDF graphs as exemplified in Fig. 7.17. Each rectangle corresponds to an actor while each



(a)



(b)

Fig. 7.17 Complete WSDF graph with complex interconnection. **a** Original WSDF graph. **b** Replaced strongly connected components

edge represents a WSDF communication channel. For illustration purposes, the corresponding WSDF parameters are omitted. In order to obtain a valid schedule for the overall graph, the lattices of all actors $a \in A$ have to be scaled and shifted such that no data elements are read before being created. For this purpose, Section 7.5.1 discusses how the lattices of all actors can be scaled correctly. Section 7.5.2 then explains an efficient method to determine the start times for each actor such that on each edge of the WSDF graph data elements are only read after being created.

7.5.1 Lattice Scaling

In order to obtain regular and short dependency vectors, and thus small buffer sizes, the different actor lattices have to be scaled according to their consumption and production behavior as discussed in Section 7.3.2 for an individual edge. Consequently, for a complete WSDF graph $G = (A, E)$, the lattice of each actor $a \in A$ has to be scaled such that for each edge $e \in E$ the following holds:

$$\frac{\langle \mathbf{s_{snk(e)}}, \mathbf{e_i} \rangle}{\langle \mathbf{s_{src(e)}}, \mathbf{e_i} \rangle} = \frac{\langle \Delta \mathbf{c}, \mathbf{e_i} \rangle}{\langle \mathbf{p}, \mathbf{e_i} \rangle} \in \mathbb{Q}. \tag{7.30}$$

For WSDF graphs without directed cycles, this does not cause any difficulties and can be performed by traversing the graph in a topological order such that each actor is visited after its sources. Algorithm 3 shows the corresponding pseudocode that calculates the corresponding scaling factor $\mathbf{s_a}$ for each actor $a \in A$ of the WSDF graph $G = (A, E)$. In case an actor does not have any predecessor, it is called *graph source* and its scaling factor is set to 1 (lines 33–36). Otherwise, its scaling factor is set relatively to the predecessor (lines 8–12). Furthermore, the `init` variable stores to which graph source the actor a_1 is related.

This helps to detect when two paths starting in different graph sources meet in an actor with more than one input edge (lines 17–32). In this case one of the paths is corrected by a vector $\mathbf{t}$ such that both input edges follow the relation given in Eq. (7.30). Lines (13)–(16) capture the special case that two paths start from the same graph source and meet in the same actor, but request different scaling factors. For almost any typical application this should never occur, but theoretically it is possible by extensive use of virtual border extension. In this case, the algorithm arbitrarily selects the minimum (line 15).

In case a WSDF graph contains cycles, Algorithm 3 can still be used by breaking all cycles first.

7.5.2 Lattice Shifting

After having scaled the actor lattices as described in the previous section, the next step for determination of a valid schedule consists in shifting the lattices in such a way that data elements are only read after creation. For this purpose, Section 7.4.3 has demonstrated how to calculate the minimal shift of the sink actor lattice relative to the source actor for each WSDF edge. Consequently, the task to solve for complete WSDF graphs $G = (A, E)$ consists in shifting each actor lattice in such a way that on edge $e \in E$ this minimum shift is not violated. This can be done by performing the following steps:

Algorithm 3 Algorithm for scaling of all actor lattices of a WSDF graph

```
(01) input: G = (A, E)
(02)
(03) sort A in topological order
(04) for each a ∈ A { init(a) = -1 }
(05) j = 0
(06) for each a_1 ∈ A {
(07)   for each input edge e of a_1 {
(08)     if (init(a_1) < 0){
(09)       for i = 1 : n {
(10)         ⟨s_{a_1}, e_i⟩ = ⟨Δc,e_i⟩ / ⟨p,e_i⟩ × ⟨s_{src(e)}, e_i⟩
(11)       }
(12)       init(a_1) = init(src(e))
(13)     }else if (init(a_1) == init(src(e)){
(14)       for i = 1 : n {
(15)         ⟨s_{a_1}, e_i⟩ = min( ⟨s_{a_1}, e_i⟩, ⟨Δc,e_i⟩ / ⟨p,e_i⟩ × ⟨s_{src(e)}, e_i⟩ )
(16)       }
(17)     }else{
(18)       for i = 1 : n {
(19)         ⟨t, e_i⟩ = ⟨Δc,e_i⟩ / ⟨p,e_i⟩ × ⟨s_{src(e)}, e_i⟩
(20)         ⟨t, e_i⟩ = ⟨s_{a_1},e_i⟩ / ⟨t,e_i⟩
(21)       }
(22)       k = init(src(e))
(23)       for each a_2 ∈ A {
(24)         if (init(a_2) == k){
(25)           for i = 1 : n {
(26)             ⟨s_{a_2}, e_i⟩ = ⟨s_{a_2}, e_i⟩ × ⟨t, e_i⟩
(27)           }
(28)           init(a_2) = init(a_1)
(29)         }
(30)       }
(31)     }
(32)   }
(33)   if (init(a_1) < 0){
(34)     init(a_1) = j;   j = j + 1
(35)     s_{a_1} = 1
(36)   }
(37) }
```

- Identification of the strongly connected components of the WSDF graph
- Shifting of all lattices of a strongly connected component
- Shifting of the strongly connected components

7.5.2.1 Determination of Strongly Connected Components

The first step consists in splitting the overall WSDF graph $G = (A, E)$ into several subgraphs $G_i = (A_i, E_i)$ forming strongly connected components:

$$A_i \subseteq A,$$

$$E_i \subseteq E,$$
$$e \in E_i \Leftrightarrow \text{src}(e) \in A_i \wedge \text{snk}(e) \in A_i.$$

Definition 7.16 Subgraph $G_i = (A_i, E_i)$ is called a *strongly connected component* if and only if

$$\forall (a_1, a_2) \in A_i \times A_i \Rightarrow \exists \text{ a directed path from } a_1 \text{ to } a_2 \text{ and from } a_2 \text{ to } a_1,$$
$$\forall (a_1, a_2) \in A_i \times (A \setminus A_i) \Rightarrow \nexists \text{ a directed path from } a_1 \text{ to } a_2 \text{ or from } a_2 \text{ to } a_1.$$

A *directed path* from a_1 to a_2 is a sequence of edges $(e_1, e_2, \ldots, e_k)$ such that

$$\begin{aligned} \text{src}(e_1) &= a_1, \\ \forall 1 < j \leq k: \text{src}\left(e_j\right) &= \text{snk}\left(e_{j-1}\right), \\ \text{snk}(e_k) &= a_2. \end{aligned}$$

Fig. 7.17a shows a corresponding example by grouping all actors belonging to the same strongly connected component with a circle. Several algorithms have been proposed in order to solve this task efficiently, like, for instance, the Tarjan's algorithm [271], which shows a complexity of $\mathcal{O}(|A| + |E|)$.

7.5.2.2 Internal Scheduling of the Strongly Connected Components

Once these strongly connected components have been identified, the next step consists in processing each of them separately by shifting all actor lattices belonging to the same strongly connected component $G_i = (A_i, E_i)$. To this end, an arbitrary actor $a_i^{\text{ref}} \in A_i$ is selected as a reference. Furthermore, each edge $e \in E_i$ is labeled by a weight that corresponds to the negative minimal delay value calculated in Section 7.4.3:

$$w(e) = -\min \sum_{i=1}^{n} \langle \mathbf{D}, \mathbf{e_i} \rangle \prod_{j=1}^{i-1} \langle \mathbf{l}_{\text{lattice}}^{\text{wrapped}}, \mathbf{e_j} \rangle.$$

Then, the scheduling problem can be reduced to a shortest path problem with possibly negative weights but without negative cycles.

Definition 7.17 A *directed cycle* is a directed path in form of a sequence of edges $(e_1, e_2, \ldots, e_k)$ such that

$$\begin{aligned} \text{src}(e_1) &= \text{snk}(e_k), \\ \forall 1 < j \leq k: \text{src}\left(e_j\right) &= \text{snk}\left(e_{j-1}\right). \end{aligned}$$

Lemma 7.18 *Let $G = (A, E)$ be a WSDF graph. Then for each directed cycle $(e_1, e_2, \ldots, e_k)$, the following holds:*

$$\sum_{i=1}^{k} w(e_i) \geq 0.$$

Otherwise, the WSDF graph deadlocks under the assumption of an affine schedule function as done in this chapter.

Proof Figure 7.18 depicts a simple cycle. As discussed in Section 7.4.3, $-w\,(e)$ corresponds to the minimum delay by which the sink has to be delayed relative to the source. Consider actor a_0 to be the reference actor. Then actor a_1 has to be delayed at least by $-w\,(e_1)$ relative to a_0. Consequently, actor a_2 has to be delayed at least by $-w\,(e_1) - w\,(e_2)$ relative to a_0. And finally, actor a_0 has to be delayed at least by $-w\,(e_1) - w\,(e_2) - w\,(e_3)$ relative to itself. This of course is only possible if

$$\sum -w\,(e_i) \leq 0.$$

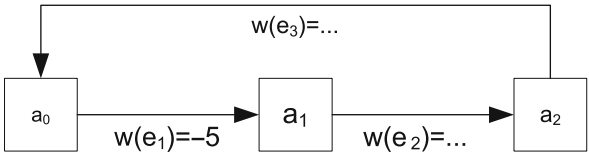


Fig. 7.18 Simple cycle

Remark 7.19 Even if $\sum_{i=1}^{k} w\,(e_i) < 0$, it might be possible to find a schedule that executes the WSDF graph and that does not deadlock. However, such a schedule would be more complex than what can be represented by the lattice notation introduced in this chapter.

Lemma 7.20 *Let $G_i = (A_i, E_i)$ be a strongly connected component and let $a_i^{\text{ref}} \in G_i$ be a reference actor. Then, for each actor $a \in A_i$, the minimal delay relative to a_i^{ref} for a valid schedule can be obtained by the negative shortest distance between a and a_i^{ref}.*

Proof By construction. To each actor $a \in A_i$, a shortest distance to a_i^{ref} can be associated, which shall be labeled by $\eta(a)$. Then the following property holds for each edge $e \in E_i$:

$$\begin{aligned} & \eta\,(\text{snk}\,(e)) \leq \eta\,(\text{src}\,(e)) + w\,(e) \\ \Leftrightarrow\ & \underbrace{-\eta\,(\text{snk}\,(e))}_{\text{delay of snk}(e)} \geq \underbrace{-\eta\,(\text{src}\,(e))}_{\text{delay of src}(e)} - w\,(e)\,. \end{aligned}$$

In other words, the above lemmas say that a valid schedule can be constructed for each strongly connected component G_i by solving a shortest path problem with non-negative cycles. This can be done efficiently using the Bellman-Ford [72] algorithm requiring a complexity of $\mathcal{O}\,(|A_i| \times |E_i|)$.

7.5.2.3 Shifting of the Strongly Connected Components

The last step in generating a valid schedule for a complete WSDF graph consists in shifting the different strongly connected components in relation to each other. For this purpose, each strongly connected component G_i is replaced by its reference actor a_i^{ref}. Figure 7.17b shows the corresponding result when applied to Fig. 7.17a. Each reference actor is shaded with a

gray color. In order to preserve the delay caused by the strongly connected component, the weights on the input and output edges have to be adapted accordingly. For an output edge e_o, the new weight becomes

$$w(e_o) \rightarrow w(e_o) + \eta(\mathrm{src}(e_o)).$$

For an input edge e_i, the new weight amounts to

$$w(e_i) \rightarrow w(e_i) - \eta(\mathrm{snk}(e_i)).$$

With these definitions, all lattices can be shifted by a simple topological sort as described in Algorithm 4. It obtains as input the WSDF graph $\tilde{G} = \left(\tilde{A}, \tilde{E}\right)$ with the replaced strongly connected components and defines for each actor $a \in \tilde{A}$ the corresponding delay S_a. This delay can be transformed into a vector representation $\mathbf{S_a}$, applying Eq. (7.26) to the vector $S_a \times \mathbf{e_1}$.

The structure of Algorithm 4 strongly resembles that of Algorithm 3. Initially, each graph source is supposed to start at time $S_a = 0$ (lines 24–27). All other actors are delayed by the minimum delay $-w(e)$ (lines 08–10). If an actor has several inputs, the maximum delay is chosen (lines 11–12). In Fig. 7.17b, this will, for instance, occur when processing actor a_{10}.

Algorithm 4 Algorithm for shifting of all actor lattices of a WSDF graph

```
(01) input: G̃ = (Ã, Ẽ)
(02)
(03) sort A in topological order
(04) for each a ∈ A { init(a) = -1; }
(05) j = 0
(06) for each a_1 ∈ A {
(07)   for each input edge e of a_1 {
(08)     if (init(a_1) < 0){
(09)       S_{a_1} = −w(e) + S_{src(e)}
(10)       init(a_1) = init(src(e))
(11)     }else if (init(a_1) == init(src(e)){
(12)       S_{a_1} = max(S_{a_1}, −w(e) + S_{src(e)})
(13)     }else{
(14)       t = S_{a_1} − (−w(e)) − S_{src(e)}
(15)       k =init(src(e))
(16)       for each a_2 ∈ A {
(17)         if (init(a_2) == k){
(18)           S_{a_2} = S_{a_2} + t
(19)           init(a_2) = init(a_1)
(20)         }
(21)       }
(22)     }
(23)   }
(24)   if (init(a_1) < 0){
(25)     init(a_1) = j;   j = j + 1
(26)     S_{a_1} = 0
(27)   }
(28) }
```

Finally, lines 13–23 cover the case that an actor has multiple inputs that are not related to the same graph source. In Fig. 7.17b, such a scenario occurs for actor a_2, which is indirectly connected to both graph source a_1 and a_{15}. In this case the previously scheduled actors are corrected such that both input edges fulfill the requirement on the minimum delay (lines 14–21).

Algorithm 4 thus calculates an ASAP schedule. As calculation of a topological sort shows a complexity of $\mathcal{O}\left(\left|\tilde{A}\right| + \left|\tilde{E}\right|\right)$, the overall complexity of Algorithm 4 amounts to

$$\mathcal{O}\left(\left|\tilde{E}\right| + \left|\tilde{A}_{\text{src}}\right| \times \left|\tilde{A}\right|\right),$$

where $\left|\tilde{A}_{\text{src}}\right|$ is the cardinality of the set of graph sources.

7.6 Buffer Size Calculation

The approaches considered so far in Sections 7.3, 7.4, and 7.5 have investigated scheduling of WSDF graphs. Hence, they answered the question when the individual actors can be executed without violating data dependencies by requiring data elements that will be created in the future. As these execution times have a severe impact on the required buffer size (see also Section 7.1), their knowledge is an important prerequisite for determination of the necessary memory sizes. This section will now tackle the next step by showing how the buffer sizes can be calculated from the lattice representation of the WSDF graph. The requirement for storing data elements in the edge buffers results from the fact that data elements produced by the source at a given time are required later on by the sink. Thus, determination of the buffer size of a given WSDF edge requires consideration of the dependency vectors that have been introduced in Section 7.3.4 and that can be calculated by means of integer linear programing as described later on in Section 7.6.1.

Figure 7.19 depicts the corresponding principles by means of an example. Each dependency vector starts at a source invocation and points to the sink invocation that requires one of the produced data elements. Since the lattice grid is executed in row-major order (see Sec-

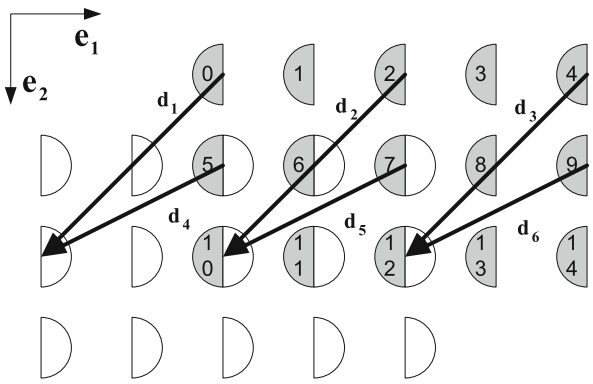


Fig. 7.19 Buffer size determination using dependency vectors

tion 7.3.1), each dependency vector thus induces a memory buffer that contains all the data elements created from the source invocation, where it starts from, up to the sink invocation, which it points to. Assuming no lattice wraparound, dependency vector $\mathbf{d_1}$, for instance, requires storage of the data elements produced by the source invocations 0−9 while for dependency vector $\mathbf{d_3}$, the data elements produced by invocations 4−12 have to be buffered.[3] The overall memory size results from the dependency vector that induces the largest number of source invocations for buffering.

Unfortunately, the corresponding calculations are pretty complex, in particular when out-of-order communication and multirate systems containing up- and downsamplers shall be supported. Whereas the latter cause that the different actors perform an unequal number of invocations, out-of-order communication leads to irregular dependency vectors making buffer analysis more difficult. Hu et al. [143], for instance, propose to derive the required buffer size from the length of a given dependency vector while ignoring its position in the token space. However, this method is not able to deliver exact results as can be clearly seen by means of Fig. 7.19. Here, dependency vector $\mathbf{d_1}$ causes a buffer requirement of 10 data elements while for dependency vector $\mathbf{d_2}$ only 9 data elements must be stored temporarily. However, both dependency vectors have the same length.

The situation gets even more complex when employing lattice wraparound in multirate applications. This is exemplified by means of Fig. 7.20, which compares two different scenarios. The first assumes a source scaling factor of $\langle \mathbf{s}_{\text{src}}, \mathbf{e_2} \rangle = 1$ leading to the depicted wraparound. Consequently, dependency vector $\mathbf{d_1}$ requires to buffer data elements 0−8. Figure 7.20b considers a quasi-identical scenario, except that the source scaling factor now amounts to $\langle \mathbf{s}_{\text{src}}, \mathbf{e_2} \rangle = 2$. As a consequence, dependency vector $\mathbf{d_1}$ now requires to buffer data elements 0−9. In other words, the buffer size caused by a dependency vector does not only depend on its length and its origin but also depend on the scaling factor.

Considering the approaches discussed in Section 3.3.2, none of them takes these effects into account. Furthermore, some of them, such as [176–178], can lead to inaccurate results due to employed approximations.

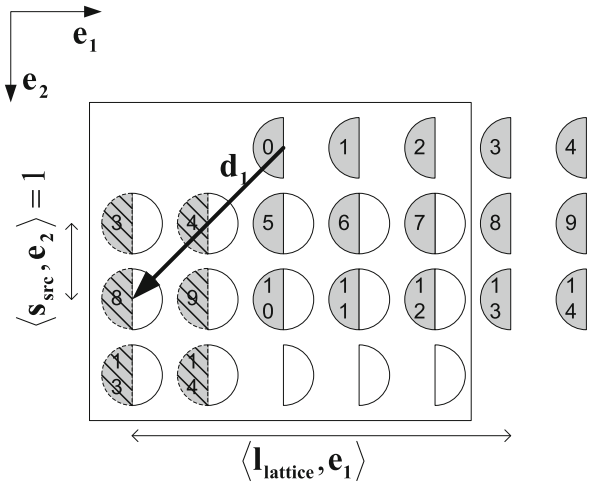
Consequently the following section describes an alternative method that uses an ILP formulation supporting both in-order and out-of-order communication, lattice wrap-around for throughput-optimized schedules, and multirate systems containing up- and downsamplers.

7.6.1 ILP Formulation for Buffer Size Calculation

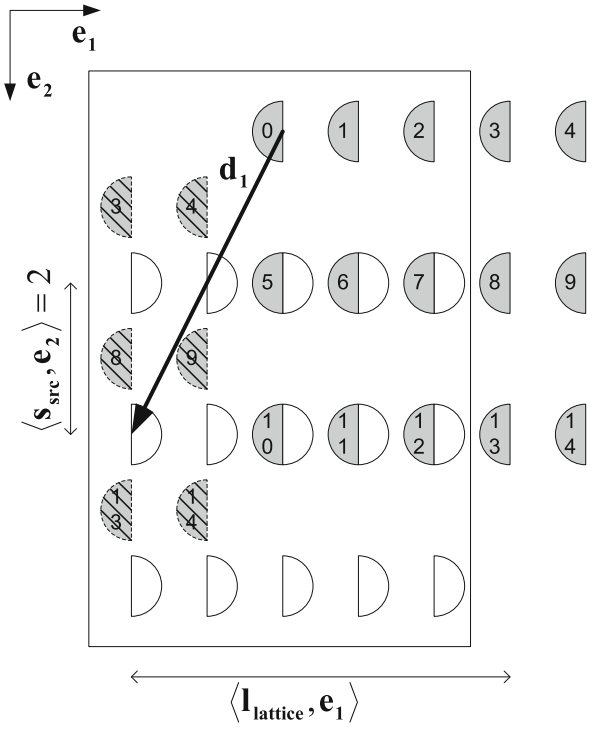
In order to calculate the required buffer size for a given WSDF edge $e \in E$, it is necessary to derive the earliest source invocation producing a required data element for each sink invocation. From this information, the necessary buffer size can be obtained by counting all source invocations occurring up to the considered sink execution. This process can be formulated with the following equations:

$$\max B, \tag{7.31}$$

[3] The source and sink lattice points are assumed to execute in parallel. Furthermore, for pipelined actor execution (see Section 7.3.5) the sink is assumed to access the data elements only during the first lattice grid and buffers the data internally for further usage. This assumption fits perfectly to typical hardware implementation strategies.



(a)



(b)

Fig. 7.20 Influence of the lattice wraparound on the required buffer size. Lattice points resulting from wraparound are shaded by *stripes*. **a** Scenario 1. **b** Scenario 2

$$\mathbf{0} \leq \mathbf{i}_{\text{src}} \leq \mathbf{i}_{\text{src,max}} \in \mathbb{N}_0^{n \times q_{\text{src}}}, \tag{7.32}$$

$$\mathbf{0} \leq \mathbf{i}_{\text{src,latest}} \leq \mathbf{i}_{\text{src,max}} \in \mathbb{N}_0^{n \times q_{\text{src}}}, \tag{7.33}$$

$$\mathbf{0} \leq \mathbf{I}_{\text{src,latest}} \leq \mathbf{I}_{\text{src,max}} \in \mathbb{N}_0^{n}, \tag{7.34}$$

$$\mathbf{0} \leq \mathbf{i}_{\text{snk}} \leq \mathbf{i}_{\text{snk,max}} \in \mathbb{N}_0^{n \times q_{\text{snk}}}, \tag{7.35}$$

$$\mathbf{0} \leq \mathbf{I}_{\text{snk}} \leq \mathbf{I}_{\text{snk,max}} \in \mathbb{N}_0^{n}, \tag{7.36}$$

$$\mathbf{0} \leq \mathbf{I}_{\text{src,w}} < \mathbf{p} \in \mathbb{N}^{n}, \tag{7.37}$$

$$\mathbf{0} \leq \mathbf{I}_{\text{snk,w}} < \mathbf{c} \in \mathbb{N}^{n}, \tag{7.38}$$

$$0 = \langle \mathbf{O}_{\text{snk}}, \mathbf{I}_{\text{snk}} \rangle - \langle \mathbf{o}_{\text{snk}}, \mathbf{i}_{\text{snk}} \rangle, \tag{7.39}$$

$$\begin{aligned} \mathbf{0} = {} & M_{\text{snk}} \times \left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}} \right) - \mathbf{b}^{\mathbf{S}} \\ & - M_{\text{src}} \times \left(\mathbf{i}_{\text{src}}, \mathbf{I}_{\text{src,w}} \right) - \boldsymbol{\delta}, \end{aligned} \tag{7.40}$$

$$\mathbf{0} \leq C_{\text{snk}} \times \left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}} \right) - \mathbf{b}_{\mathbf{c}}^{\mathbf{l}}, \tag{7.41}$$

$$\mathbf{0} \geq C_{\text{snk}} \times \left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}} \right) - \mathbf{b}_{\mathbf{c}}^{\mathbf{h}}, \tag{7.42}$$

$$\forall 1 \leq j < n:$$

$$k_j = \left\lfloor \frac{\langle \mathbf{I}_{\text{snk}}, \mathbf{e_j} \rangle \times \langle \mathbf{s}_{\text{snk}}, \mathbf{e_j} \rangle + \langle \Delta \mathbf{S}, \mathbf{e_j} \rangle + k_{j-1}}{\langle \mathbf{l}_{\text{lattice}}^{\text{wrapped}}, \mathbf{e_j} \rangle} \right\rfloor, \tag{7.43}$$

$$k_0 = 0, \tag{7.44}$$

$$k_n = 0, \tag{7.45}$$

$$\forall 1 \leq j \leq n:$$

$$\begin{aligned} \langle \mathbf{I}_{\text{snk}}^{\text{wrapped}}, \mathbf{e_j} \rangle = {} & \langle \mathbf{I}_{\text{snk}}, \mathbf{e_j} \rangle \times \langle \mathbf{s}_{\text{snk}}, \mathbf{e_j} \rangle + \langle \Delta \mathbf{S}, \mathbf{e_j} \rangle + k_{j-1}, \\ & - k_j \times \langle \mathbf{l}_{\text{lattice}}^{\text{wrapped}}, \mathbf{e_j} \rangle \end{aligned} \tag{7.46}$$

$$\text{diag}\left(\mathbf{s}_{\text{src}}\right) \times \mathbf{I}_{\text{src,latest}} \lesseqgtr \mathbf{I}_{\text{snk}}^{\text{wrapped}}, \tag{7.47}$$

$$0 = \langle \mathbf{O}_{\text{src}}, \mathbf{I}_{\text{src,latest}} \rangle - \langle \mathbf{o}_{\text{src}}, \mathbf{i}_{\text{src,latest}} \rangle, \tag{7.48}$$

$$B = \langle \mathbf{o}_{\text{src}}, \mathbf{i}_{\text{src,latest}} \rangle - \langle \mathbf{o}_{\text{src}}, \mathbf{i}_{\text{src}} \rangle + 1. \tag{7.49}$$

Equation (7.31) defines the objective function searching for the maximum buffer size B required during execution of the considered WSDF communication edge. To this end, each sink invocation $\mathbf{I}_{\text{snk}}$ is mapped to its hierarchical iteration vector $\mathbf{i}_{\text{snk}}$ via Eq. (7.39). Equations (7.41) and (7.42) exclude all those data elements, respectively, iterations that are situated on the virtually extended border. For the others, Eq. (7.40) derives the source invocation generating the considered data element. Equations (7.43), (7.44), (7.45), and (7.46) take the effect of lattice wraparound into account and correspond to Eqs. (7.23), (7.24), (7.25), and (7.26). For this purpose, it is assumed without loss of generality that only the sink invocations are influenced by a wraparound. $\mathbf{s}_{\text{snk}}$ corresponds to the sink scaling vector calculated in Sections 7.3.2 and 7.5.1. $\Delta\mathbf{S} = \mathbf{S}_{\mathbf{snk}} - \mathbf{S}_{\mathbf{src}}$ is the shift of the sink lattice relative to the source. Equation (7.47) determines the latest source invocation that is not executed after the sink invocation $\mathbf{I}_{\text{snk}}$ currently considered, where

$\mathrm{diag}(\mathbf{s}_{\mathrm{src}})$ is defined in Eq. (7.15). The symbol $\underset{\leftarrow}{\preceq}$ means anti-lexicographically smaller or equal and is defined as follows:

$$\begin{aligned} & \mathbb{N}^n \ni \mathbf{x_1} \underset{\leftarrow}{\preceq} \mathbf{x_2} \in \mathbb{N}^n \\ \Leftrightarrow\ & \Delta\mathbf{x} = \mathbf{x_1} - \mathbf{x_2} \underset{\leftarrow}{\preceq} \mathbf{0} \\ \Leftrightarrow\ & \Delta\mathbf{x} = \mathbf{0} \vee (\exists i : \langle \Delta\mathbf{x}, \mathbf{e_i} \rangle < 0 \wedge i < j \leq n : \langle \Delta\mathbf{x}, \mathbf{e_i} \rangle = 0)\,. \end{aligned}$$

This latest source invocation is translated into a hierarchical iteration vector by means of Eq. (7.48) such that Eq. (7.49) can finally calculate the resulting buffer size by counting the source iterations occurring between $\mathbf{i}_{\mathrm{src,latest}}$ and $\mathbf{i}_{\mathrm{src}}$, where $\mathbf{o}_{\mathrm{src}}$ corresponds to the ordering vector defined in Eq. (7.17).

Unfortunately, the so-defined system of equations cannot be solved directly by classical ILP solvers, because it contains two non-linear conditions in form of Eqs. (7.43) and (7.47). Fortunately, two simple transformations can easily solve these difficulties.

Corollary 7.21 *Given two integer numbers $a, z \in \mathbb{Z}$ and a natural number $b \in \mathbb{N}$. Then the following holds:*

$$z = \left\lfloor \frac{a}{b} \right\rfloor \Leftrightarrow a - b < b \times z \leq a.$$

Proof

$$\begin{aligned} & \frac{a}{b} - 1 < \left\lfloor \frac{a}{b} \right\rfloor \leq \frac{a}{b} \\ \Leftrightarrow\ & a - b < b \times \left\lfloor \frac{a}{b} \right\rfloor \leq a. \end{aligned}$$

In other words, Eq. (7.43) can be replaced by two simple inequalities. Equation (7.47), on the other hand, can be split into an or-conjunction of n linear constraints:

$$\begin{aligned} & \mathbb{N}^n \ni \mathbf{x_1} \underset{\leftarrow}{\preceq} \mathbf{x_2} \in \mathbb{N}^n \\ \Leftrightarrow\ & \langle \mathbf{x_1} - \mathbf{x_2}, \mathbf{e_n} \rangle < 0 \\ & \vee\ (\langle \mathbf{x_1} - \mathbf{x_2}, \mathbf{e_{n-1}} \rangle < 0 \wedge \langle \mathbf{x_1} - \mathbf{x_2}, \mathbf{e_n} \rangle = 0) \\ & \vee\ \ldots \\ & \vee\ (\langle \mathbf{x_1} - \mathbf{x_2}, \mathbf{e_1} \rangle \leq 0 \wedge \ldots \wedge \langle \mathbf{x_1} - \mathbf{x_2}, \mathbf{e_n} \rangle = 0)\,. \end{aligned}$$

For each component of the or-conjunction, the ILP can be solved separately. The overall solution can then be obtained by selecting the maximum of all solutions.

Remark 7.22 When initial tokens are present ($\boldsymbol{\delta} \neq \mathbf{0}$), the iterators in Eqs. (7.31), (7.32), (7.33), (7.34), (7.35), (7.36), (7.37), (7.38), (7.39), (7.40), (7.41), (7.42), (7.43), (7.44), (7.45), (7.46), (7.47), (7.48), and (7.49) have to be extended by the schedule period.

7.6.2 Memory Channel Splitting

The explanations given so far for automatic buffer analysis of individual edges assumed a linearized buffer model as introduced in Section 6.3.2. In other words, all data elements produced by the source actor are placed into one single memory or address space in the order in which they are produced. Whereas this is a reasonable assumption when using external memory chips, this can lead to sub-optimal buffer allocation in case of on-chip memory. This is because in the latter case often multiple individual memory units are available allowing for a more fine-grained memory allocation. This section shows how this fact can be exploited during polyhedral buffer analysis in order to obtain optimized buffer configurations.

In order to clarify the problem to solve, Fig. 7.21 depicts a simple vertical upsampler. As it outputs 2 pixels per invocation, the sink executes twice as often as the source and the vertical upsampler. All actors are assumed to traverse the image in raster-scan order. Figure 7.21b depicts the token space belonging to edge e_2. The Arabic numbers indicate the memory address where the individual data elements are placed to and follow the production order as discussed in Section 6.3.2. This means that for the depicted position of the effective token and sliding window, data elements 0−9 have to be stored in the edge buffer[4] leading to an overall memory size of two complete lines. Unfortunately, this is significantly more than the optimum, which amounts to 1 line and 2 pixels (in case of parallel actor execution).

This problem originates from the strict linearization in production order. This causes that data elements 0, 2, and 4 cannot be freed from the buffer, because data element 1 is still required and because holes in the address range are not supported in order to avoid complex and expensive hardware implementations. Fortunately, there is an efficient solution: releasing the constraint that all data elements have to be placed into one memory or address space. Instead, the data elements in dimension $\mathbf{e_i}$ are split into $\langle \mathbf{p_2}, \mathbf{e_i} \rangle$ memory channels as exemplified in Fig. 7.21c. In case of separate buffer management (see Chapter 8), this allows to free the data elements A:0, A:1, and A:2 leading to an overall memory requirement of 1 line and 2 pixels (due to the parallel sink and source execution). This is almost 50% less compared to the strict linearization in production order.

This benefit can be taken into account during polyhedral buffer analysis by calculating the buffer size separately for each memory channel. To this end, the ILP defined via Eqs. (7.31), (7.32), (7.33), (7.34), (7.35), (7.36), (7.37), (7.38), (7.39), (7.40), (7.41), (7.42), (7.43), (7.44), (7.45), (7.46), (7.47), (7.48), and (7.49) has to be extended by additional constraints that restrict the considered data element position in the effective token. In case of Fig. 7.21, for instance, Eqs. (7.31), (7.32), (7.33), (7.34), (7.35), (7.36), (7.37), (7.38), (7.39), (7.40), (7.41), (7.42), (7.43), (7.44), (7.45), (7.46), (7.47), (7.48), and (7.49) have to be solved twice. For memory channel A, they have to be extended by

$$\mathbf{I}_{\mathrm{src,w}} = (0, 0)^T ,$$

whereas memory channel B requires

$$\mathbf{I}_{\mathrm{src,w}} = (0, 1)^T .$$

[4] Data element 0 could be discarded. Since, however, the polyhedral buffer analysis operates on source actor invocations instead of individual data elements, this is not considered further.

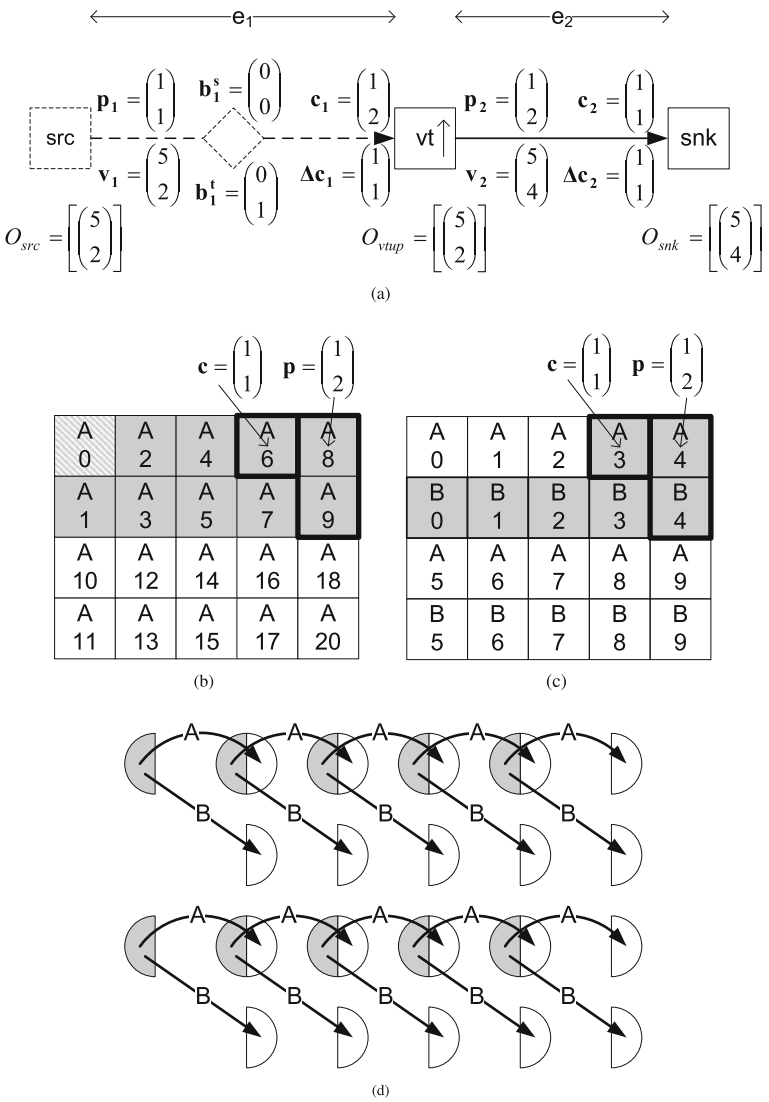


Fig. 7.21 Upsampler for illustration of memory channel splitting. Data elements *shaded gray* have to be buffered for the depicted token positions. *A* and *B* label different memory channels. **a** WSDF graph. **b** Token space for edge e_2. **c** Channel splitting for edge e_2. **d** Lattice representation including channel splitting

By these means, the ILP solver only considers those dependencies that are caused by the considered memory channel. Figure 7.21d shows the corresponding results for the upsampler using two memory channels. Each dependency vector only considers those dependencies that are caused by the associated memory channel. Taking, for instance, the first row of sink invocations, they do not induce any dependency vectors labeled by *B,* because no data elements

are read from this memory channel. Similarly, the second row of sink invocations only read from memory channel *B*.

This separation of the dependency vectors permits to calculate the required buffer size for each memory channel. In case of Fig. 7.21d, simple point counting shows that dependency vectors *A* require a buffer size of two data elements while dependency vectors *B* cause a buffer size of one complete row.

7.7 Multirate Analysis

The previous sections have introduced a novel buffer analysis technique based on a polyhedral representation of WSDF graphs. Special care has been taken in order to allow for throughput-optimized schedules together with out-of-order communication, memory channel splitting, and complex graph topologies including feedback loops. Furthermore, lattice scaling assured the support of so-called *multirate systems* where not all actors perform the same number of invocations. This typically occurs in case of image upsampling and downsampling. As a consequence, part of the actors only have to process images of smaller sizes, leading to reduced activity. Such scenarios are widespread in image processing and occur, for instance, for the wavelet transform or for the multi-resolution filter depicted in Fig. 7.1. Unfortunately, they significantly complicate system analysis because they lead to complex schedules and offer additional tradeoffs for hardware implementations, which shall be investigated in more detail in the following.

For motivation, let's consider again the vertical downsampler of Fig. 7.4 whose lattice representation is repeated in Fig. 7.22a. In order not to get trapped by a special case, the input image height has been increased by 2. Obviously, the workload of the downsampler is not equally distributed over the time, but shows bursty behavior: The phases where each source invocation also induces execution of the downsampler are followed by idle lines. This means that the underlying hardware of the downsampler has to be able to accept one new input pixel per clock cycle, in order to cope with the worst-case throughput requirements. In other words, the so-called *initiation interval,* which defines the time between two consecutive actor executions, only amounts 1. Whereas this can be easily accepted for the downsampler actor because it is very simple, the situation gets more critical for the complex bilateral filters that are part of the multi-resolution filter depicted in Fig. 7.1. Here, this small initiation interval reduces the possibilities for resource sharing in the bilateral filters, making hardware implementation more expensive.

However, from the overall throughput considerations, an initiation interval of 2 would also be possible, allowing for extended resource sharing by the synthesis tools and thus reduced hardware costs. This effect can be taken into account during buffer analysis by redistributing the sink lattice points. Figure 7.22c exemplarily illustrates the results for the downsampler actor. In contrast to the *bursty schedule* depicted in Fig. 7.22a, the workload is now equally balanced, because it is executed only every second source invocation (II $= 2$). In the rest of this monograph, such a schedule shall be called *smoothed schedule*. However, this has to be paid for by less regular dependency vectors and thus increased buffer requirements.

Figures 7.22b, d depict the shifted lattice representations for both the bursty and the smoothed schedules. In both cases, the dependency vectors show how the data flow from the source invocations to the consuming sink invocations. From these dependency vectors it can be derived that for the bursty implementation, a buffer size of 2 lines and 1 pixel, or 11 data elements, is sufficient. In contrast, vector $\mathbf{d_0}$ in Fig. 7.22d causes a buffer size

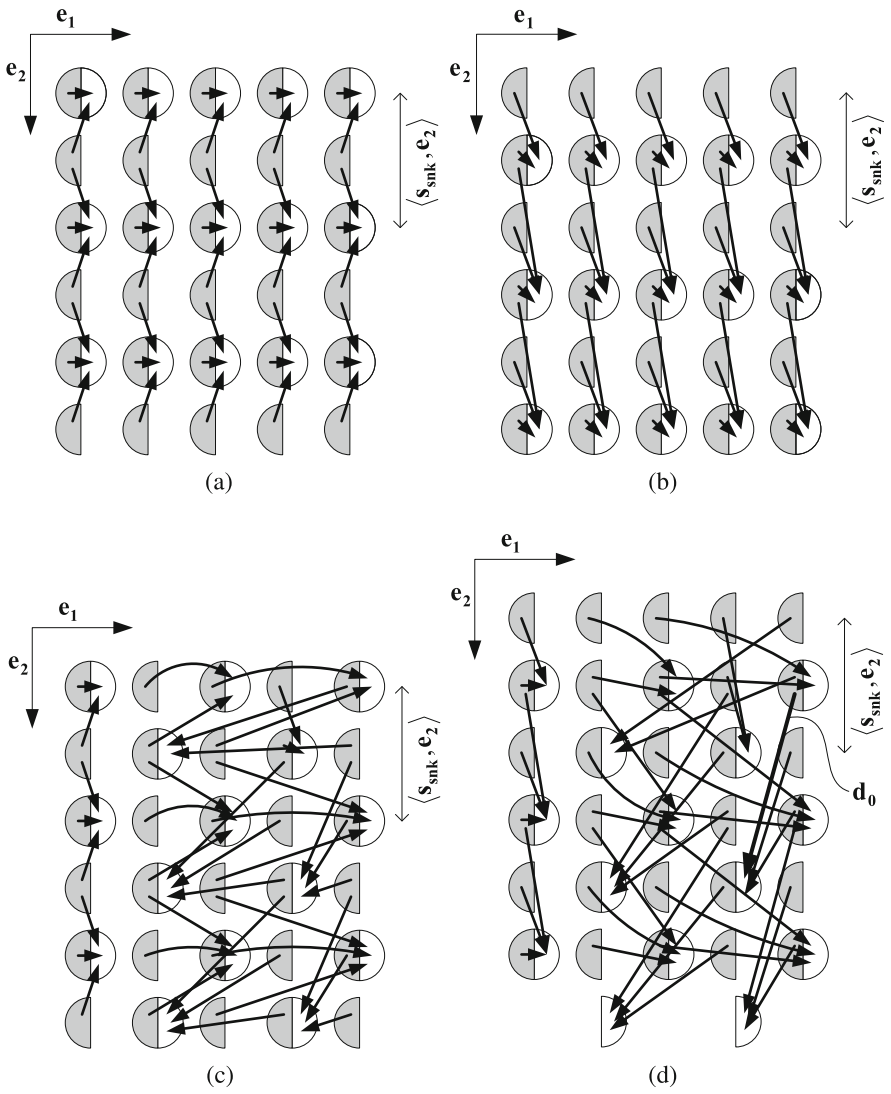


Fig. 7.22 Load-smoothing techniques for a vertical downsampler. *Gray-shaded semi-circles* correspond to the source, whereas the vertical downsampler is represented by *white semi-circles*. **a** Bursty schedule. **b** Shifted lattice for bursty schedule. **c** Smoothed schedule. **d** Shifted lattice for smoothed schedule

of 15 data elements, which corresponds to an increase of 36%. This, by the way, would be the implementation alternative chosen by [302], whereas the bursty schedule is not supported. However, the capability to analyze and synthesize both actor schedule alternatives permits to exploit a tradeoff between communication memory and hardware requirements

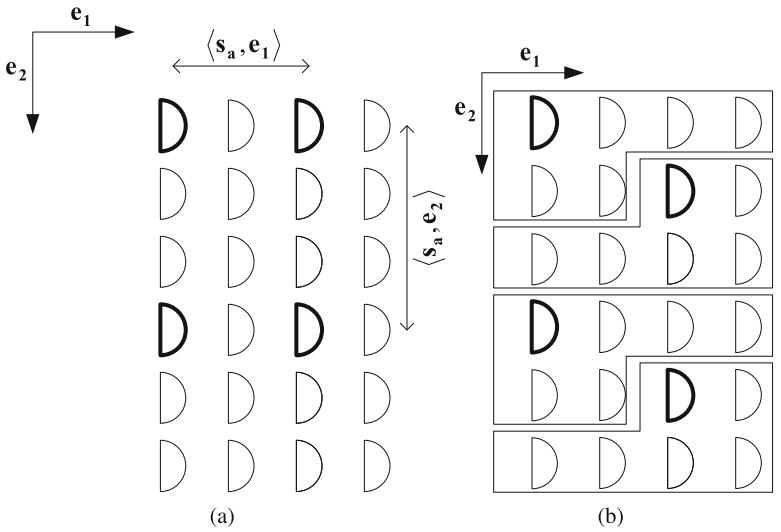


Fig. 7.23 General scenario for lattice point redistribution. *Bold semi-circles* represent the original sink invocations. **a** Duplicated actor lattice points. **b** Remapped actor lattice points

for the accelerator. This has not been done before in any other approach discussed in Chapter 3.

Mathematically, the redistribution of the sink lattice points can be achieved by a remapping similar to that employed in Section 7.3.3 for out-of-order communication. Figure 7.23 illustrates this process by means of a fictive actor invocation pattern. Originally, the actor executes two times in horizontal and two times in vertical direction, as depicted by bold semi-circles. The corresponding actor scaling factor shall amount to

$$\mathbf{s_a} = (2, 3)^T .$$

Thus, considering the overall throughput, each actor invocation is permitted to take $2 \times 3 = 6$ "clock cycles," where a clock cycle corresponds to one lattice point of the reference grid as defined in Section 7.4.1. In general, the number of clock cycles per actor invocation for a smoothed schedule is given by

$$t = \prod_{i=1}^{n} \langle \mathbf{s_a}, \mathbf{e_i} \rangle,$$

n being the number of token dimensions and a the considered actor.

In order to represent this information, the sink lattice points are now duplicated by a factor of $\langle \mathbf{s_a}, \mathbf{e_i} \rangle$ in each dimension $\mathbf{e_i}$, as already done in Fig. 7.23a. Since the lattice is executed in row-major order, a remapping similar to that performed for out-of-order communication has to be performed in order to obtain the desired semantics. To this end, t consecutive lattice points are mapped to the same sink invocation. The result of this operation is exemplified in Fig. 7.23b. All lattice points framed by a rectangle correspond to the same sink invocation.

These operations can be taken into account during scheduling by replacing Eqs. (7.4) and (7.6) by

$$\mathbf{0} \leq \mathbf{I}_{\text{src}} \leq \text{diag}\,(\mathbf{s}_{\text{src}}) \times \left(\mathbf{I}_{\text{src,max}} + \mathbf{1}\right) - \mathbf{1} \in \mathbb{N}_0^n, \tag{7.50}$$

$$\mathbf{0} \leq \mathbf{I}_{\text{snk}} \leq \text{diag}\,(\mathbf{s}_{\text{snk}}) \times \left(\mathbf{I}_{\text{snk,max}} + \mathbf{1}\right) - \mathbf{1} \in \mathbb{N}_0^n, \tag{7.51}$$

in order to consider lattice point duplication, where diag $(\mathbf{s}_{\text{src}})$ and diag $(\mathbf{s}_{\text{snk}})$ are as defined in Eq. (7.15). For correct lattice point mapping, Eqs. (7.9) and (7.13) have to be transformed to

$$0 = \langle \mathbf{O}_{\text{snk}}, \mathbf{I}_{\text{snk}} \rangle - \left(\prod_{i=1}^{n} \langle \mathbf{s}_{\text{snk}}, \mathbf{e_i} \rangle \right) \times \langle \mathbf{o}_{\text{snk}}, \mathbf{i}_{\text{snk}} \rangle - \beta_{\text{snk}}, \tag{7.52}$$

$$0 \leq \beta_{\text{snk}} < \left(\prod_{i=1}^{n} \langle \mathbf{s}_{\text{snk}}, \mathbf{e_i} \rangle \right), \tag{7.53}$$

$$0 = \langle \mathbf{O}_{\text{src}}, \mathbf{I}_{\text{src}} \rangle - \left(\prod_{i=1}^{n} \langle \mathbf{s}_{\text{src}}, \mathbf{e_i} \rangle \right) \times \langle \mathbf{o}_{\text{src}}, \mathbf{i}_{\text{src}} \rangle - \beta_{\text{src}}, \tag{7.54}$$

$$0 \leq \beta_{\text{src}} < \left(\prod_{i=1}^{n} \langle \mathbf{s}_{\text{src}}, \mathbf{e_i} \rangle \right), \tag{7.55}$$

$$\forall 1 \leq i \leq n: \quad \langle \mathbf{O}_{\text{snk}}, \mathbf{e_i} \rangle = \prod_{j=1}^{i-1} \left[\left(\langle \mathbf{I}_{\text{snk,max}}, \mathbf{e_j} \rangle + 1 \right) \times \langle \mathbf{s}_{\text{snk}}, \mathbf{e_i} \rangle \right], \tag{7.56}$$

$$\forall 1 \leq i \leq n: \quad \langle \mathbf{O}_{\text{src}}, \mathbf{e_i} \rangle = \prod_{j=1}^{i-1} \left[\left(\langle \mathbf{I}_{\text{src,max}}, \mathbf{e_j} \rangle + 1 \right) \times \langle \mathbf{s}_{\text{src}}, \mathbf{e_i} \rangle \right]. \tag{7.57}$$

$\mathbf{o}_{\text{src}}$ and $\mathbf{o}_{\text{snk}}$ remain unchanged.

Similar, for correct buffer analysis in case of smoothed schedules, Eqs. (7.36) and (7.39) have to be replaced by (7.51) and (7.52). Equation (7.34) gets to

$$\mathbf{0} \leq \mathbf{I}_{\text{src,latest}} \leq \text{diag}\,(\mathbf{s}_{\text{src}}) \times \left(\mathbf{I}_{\text{src,max}} + \mathbf{1}\right) - \mathbf{1} \in \mathbb{N}_0^n. \tag{7.58}$$

Equations (7.43), (7.44), (7.45), (7.46), (7.47), and (7.48) finally transform to

$$\forall 1 \leq j < n: \quad k_j = \left\lfloor \frac{\langle \mathbf{I}_{\text{snk}}, \mathbf{e_j} \rangle + \langle \Delta \mathbf{S}, \mathbf{e_j} \rangle + k_{j-1}}{\langle \mathbf{l}_{\text{lattice}}^{\text{wrapped}}, \mathbf{e_j} \rangle} \right\rfloor, \tag{7.59}$$

$$k_0 = 0, \tag{7.60}$$

$$k_n = 0, \tag{7.61}$$

$$\forall 1 \leq j \leq n:$$

$$\langle \mathbf{I}_{\text{snk}}^{\text{wrapped}}, \mathbf{e_j} \rangle = \langle \mathbf{I}_{\text{snk}}, \mathbf{e_j} \rangle + \langle \Delta\mathbf{S}, \mathbf{e_j} \rangle + k_{j-1}, \\ -k_j \times \langle \mathbf{l}_{\text{lattice}}^{\text{wrapped}}, \mathbf{e_j} \rangle \tag{7.62}$$

$$\mathbf{I}_{\text{src,latest}} \preceq\!\!\!\!\!\!\leftarrow \mathbf{I}_{\text{snk}}^{\text{wrapped}}, \tag{7.63}$$

$$\langle \mathbf{O}_{\text{src}}, \mathbf{I}_{\text{src,latest}} \rangle = \left(\prod_{i=1}^{n} \langle \mathbf{s}_{\text{src}}, \mathbf{e_i} \rangle \right) \langle \mathbf{o}_{\text{src}}, \mathbf{i}_{\text{src,latest}} \rangle + \beta_{\text{src}}. \tag{7.64}$$

7.8 Solution Strategies

Determination of both the required lattice shifts and the resulting edge buffer sizes requires the solution of several integer linear programs as discussed in Sections 7.3.4, 7.4.3, and 7.6. Although usage of efficient graph algorithms assures that these optimization problems only depend on one single WSDF edge, it remains a fundamental problem that their solution in general can become very computational intensive. This is because in the worst case the applied ILP solver has to perform an exhaustive search over all unknowns. For instance, in case of the integer linear program given by Eqs. (7.31), (7.32), (7.33), (7.34), (7.35), (7.36), (7.37), (7.38), (7.39), (7.40), (7.41), (7.42), (7.43), (7.44), (7.45), (7.46), (7.47), (7.48), and (7.49), these unknowns encompass $\mathbf{i}_{\text{src}} \in \mathbb{N}_0^{n \times q_{\text{src}}}$, $\mathbf{i}_{\text{src,latest}} \in \mathbb{N}_0^{n \times q_{\text{src}}}$, $\mathbf{i}_{\text{snk}} \in \mathbb{N}_0^{n \times q_{\text{snk}}}$, $\mathbf{I}_{\text{snk}} \in \mathbb{N}_0^n$, $\mathbf{I}_{\text{src,w}} \in \mathbb{N}^n$, $\mathbf{I}_{\text{snk,w}} \in \mathbb{N}^n$, $k_1, \ldots, k_{n-1} \in \mathbb{Z}$, $\mathbf{I}_{\text{snk}^{\text{wrapped}}} \in \mathbb{N}_0^n$, $\mathbf{I}_{\text{src,latest}} \in \mathbb{N}_0^n$, and $B \in \mathbb{N}$. In other words, in the worst case the ILP solver has to try all possible combinations of the unknowns in order to determine the optimum solution of the integer linear program. Since, however, the bounds of all these unknowns depend on the concrete image size, it is obvious that even for small or medium image sizes this is completely unacceptable.

Fortunately, in many cases the ILP solvers can strongly reduce the solution complexity by sophisticated mathematical manipulations, such that they deliver the desired results within fraction of seconds. Unfortunately, however, Section 7.9 will also reveal application scenarios in which standard ILP solvers could not deliver the optimal answers within reasonable time and memory requirements.

Consequently, an alternative solution strategy, called *intelligent exhaustive search* in the following, has been elaborated. It can be used in case the standard ILP solvers fail. It essentially exploits the special structure of the integer linear programs required for buffer analysis and scheduling. In case of Eqs. (7.31), (7.32), (7.33), (7.34), (7.35), (7.36), (7.37), (7.38), (7.39), (7.40), (7.41), (7.42), (7.43), (7.44), (7.45), (7.46), (7.47), (7.48), and (7.49), for instance, it is sufficient to determine the resulting value of B for each sink invocation $0 \leq \mathbf{I}_{\text{snk}} \leq \mathbf{I}_{\text{snk,max}}$ and each sliding window position $0 \leq \mathbf{I}_{\text{snk,w}} < \mathbf{c}$. This is easily possible thanks to the special structure of the used formulas. In particular, Eq. (7.39) permits to directly derive the hierarchical iteration vector $\mathbf{i}_{\text{snk}}$ associated with $\mathbf{I}_{\text{snk}}$, since $0 \leq \mathbf{i}_{\text{snk}} \leq \mathbf{i}_{\text{snk,max}}$ and since $\forall 1 \leq i \leq n \times q_{\text{snk}} : \langle \mathbf{o}_{\text{snk}}, \mathbf{e_i} \rangle = \prod_{j=i+1}^{n \times q_{\text{snk}}} \left(\langle \mathbf{i}_{\text{snk,max}}, \mathbf{e_j} \rangle + 1 \right)$:

$$\langle \mathbf{i}_{\text{snk}}, \mathbf{e_j} \rangle = \left\lfloor \frac{\langle \mathbf{O}_{\text{snk}}, \mathbf{I}_{\text{snk}} \rangle - \sum_{k=1}^{j-1} \left(\langle \mathbf{i}_{\text{snk}}, \mathbf{e_k} \rangle \times \langle \mathbf{o}_{\text{snk}}, \mathbf{e_k} \rangle \right)}{\langle \mathbf{o}_{\text{snk}}, \mathbf{e_j} \rangle} \right\rfloor.$$

With this information, Eq. (7.40) allows calculation of the accessed data element and by which source invocation it has been produced (see also Eqs. (6.1) and (6.2)):

Algorithm 5 Systematic derivation of $\mathbf{I}_{\text{src,latest}}$

```
max_value = false;
for k = n downto 1 {
  if (not max_value){
    ⟨I_src,latest, e_k⟩ = ⌊ ⟨I_snk^wrapped, e_k⟩ / ⟨s_src, e_k⟩ ⌋
    if (⟨I_snk^wrapped, e_k⟩ mod ⟨s_src, e_k⟩ ≠ 0)
      max_value = true;
  }else{
    ⟨I_src,latest, e_k⟩ = ⟨I_src,max, e_k⟩
  }
}
```

$$\mathbf{g} = M_{\text{snk}} \times \left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right) - \mathbf{b}^{\mathbf{s}},$$

$$\langle \mathbf{i}_{\text{src}}, \mathbf{e}_{\mathbf{n\times(q-k)-j+1}} \rangle = \left\lfloor \frac{\left\lfloor \frac{\langle \mathbf{g}-\boldsymbol{\delta}, \mathbf{e_j} \rangle}{\langle \mathbf{p}, \mathbf{e_j} \rangle} \right\rfloor \bmod \langle \mathbf{B}^{\text{src}}_{k+1}, \mathbf{e_j} \rangle}{\langle \mathbf{B}^{\text{src}}_{k}, \mathbf{e_j} \rangle} \right\rfloor,$$

where $\mathbf{B}^{\text{src}}_{k}$ defines the firing blocks of the source actor. Data elements or sink invocations situated on the extended border can be excluded via Eqs. (7.41) and (7.42). Since calculation of Eqs. (7.43), (7.44), (7.45), and (7.46) is trivial, they do not pose any difficulties for determination of the required buffer size. Algorithm 5 shows how to derive the latest source lattice point that is not anti-lexicographically larger than $\mathbf{I}_{\text{snk}^{\text{wrapped}}}$. Equation (7.48) can finally be solved by the following equation:

$$\langle \mathbf{i}_{\text{src,latest}}, \mathbf{e_j} \rangle = \left\lfloor \frac{\langle \mathbf{O}_{\text{src}}, \mathbf{I}_{\text{src,latest}} \rangle - \sum_{k=1}^{j-1} \left(\langle \mathbf{i}_{\text{src,latest}}, \mathbf{e_k} \rangle \times \langle \mathbf{o}_{\text{src}}, \mathbf{e_k} \rangle \right)}{\langle \mathbf{o}_{\text{src}}, \mathbf{e_j} \rangle} \right\rfloor.$$

In other words, the buffer size for a given WSDF edge can be determined by iterating over all possible sink invocations $0 \leq \mathbf{I}_{\text{snk}} \leq \mathbf{I}_{\text{snk,max}}$ and sliding window positions $0 \leq \mathbf{I}_{\text{snk,w}} < \mathbf{c}$, performing the above calculations and choosing the occurring maximum value for B. Consequently, this *intelligent exhaustive search* leads to a complexity in the order to read data elements. This is in contrast to standard ILP solvers, which, in the worst case, have to perform an exhaustive search over **all** ILP unknowns.

7.9 Results

In order to apply the previously discussed buffer size analysis techniques to different examples, they have been integrated into the ESL tool SystemCoDesigner described in Chapter 4. By this means, it is possible to describe complex applications like the multi-resolution filter depicted in Fig. 7.1 using the SystemoC library and the WSDF communication semantics. The corresponding SystemC specification can be transformed into an intermediate XML format. A corresponding library helps to transform it into a graph representation accessible to buffer analysis, which can be performed completely automatically. Based on this

setup, several experiments explain the principal benefits achievable by the discussed buffer analysis method as well as the major occurring challenges.

7.9.1 Out-of-Order Communication

One of the major benefits offered by the discussed buffer analysis approach consists in its capacities to handle out-of-order communication. Consequently, in a first step, a simple WSDF graph has been created performing the block-building operation shown in Fig. 7.7a. Using the small image sizes depicted there, the solution of the integer linear programs causes no problems at all and can be terminated within fractions of seconds. Comparison with the results delivered by the simulation approach discussed in Appendix A shows that the algorithms work correctly.

However, the troubles started when moving to bigger images. Assuming, for instance, an image size of 2048×1088 pixels, the ILP defined via Eqs. (7.31), (7.32), (7.33), (7.34), (7.35), (7.36), (7.37), (7.38), (7.39), (7.40), (7.41), (7.42), (7.43), (7.44), (7.45), (7.46), (7.47), (7.48), and (7.49) becomes intractable for the PIP library [5]. The same is valid when changing to *lp_solve* [3], even when performing several simplifications like variable elimination and splitting into several smaller sub-problems. Further investigations have shown that for some examples, even the ILP to determine the lattice shift as defined by Eqs. (7.2), (7.3), (7.4), (7.5), (7.6), (7.7), (7.8), (7.9), (7.10), (7.11), (7.12), (7.13), and (7.14) becomes intractable for the PIP library. A corresponding scenario could, for instance, be found in the shuffle operation described later on in Section 8.1, using only QCIF image size (176×144).

This observation clearly shows the danger of relying on integer linear programing, as the problem in general is NP-complete, and hence computational intensive for problems with many variables and constraints. It could, for instance, be observed that addition of a simple constraint can transform an ILP solvable within fractions of a second into an intractable problem. Furthermore, the ILP solvers seem to have difficulties in recognizing the structure of the applied equations. In case of the PIP library, for instance, it could be observed that a problem with an obvious solution is transformed in such a way that after several steps, the resulting equations are getting so complex that determination of the correct objective value gets almost impossible.

Fortunately, the *ILOG CPLEX* solver [148] does not show these difficulties in solving both the integer linear programs for calculating the lattice shift and the resulting buffer size. This tool allows processing images whose size exceeds 4096×2176 pixels for both the block building and the shuffle operation. The corresponding solution takes far less than a second. In contrast, the simulation approach discussed in Appendix A requires approximately 97 s in order to deliver the desired value for the block builder example.

Although this result clearly shows the benefits of the discussed buffer analysis method, the strong dependence on the capacities of the ILP solver remains unsatisfactory, because the user cannot be sure whether he will obtain a solution or whether the problem complexity becomes intractable for the existing software solvers. Fortunately, this can be easily solved by switching in these cases to the intelligent exhaustive search strategy presented in Section 7.8. Its application to the block-building operation mentioned above and using an image size of 4096×2176 pixels led to an overall execution time of approximately 7 s, while the determination of the lattice shift was still performed with ILOG CPLEX. Whereas this is significantly larger than the ILP solution time, the intelligent exhaustive search offers the great advantage of having a lower worst-case complexity than standard ILP solvers. Furthermore,

it can be parallelized excellently on modern multi-CPU computers. Note, additionally, that this intelligent exhaustive search requires significantly less time than the simulation approach presented in Appendix A. The underlying reason can be found in the fact that the schedule of the involved actors is already fixed and needs not to be determined during run-time. Additionally, less complex data structure manipulations are required.

To sum up, two different techniques are available in order to solve the integer linear programs required for correct and exact buffer analysis. For most applications, usage of ILOG CPLEX will permit finding solutions very quickly. However, in case an ILP will become intractable, the system level design tool can switch to the intelligent exhaustive search leading to the same results in reasonable time. On modern computer systems with several processors, it is even possible to start both solution alternatives in parallel and abort the one that requires more time. This clearly demonstrates the benefit gained by the discussed scheduling technique, which only requires local ILPs in order to deliver good results. This not only keeps the integer linear programs tractable but also permits to switch to an intelligent exhaustive search in case the ILP solver requires unbounded computation time.

7.9.2 Application to Complex Graph Topologies

After having demonstrated the ability to analyze out-of-order communication, this section considers the application of the buffer analysis described in this book to complex graph topologies. For this purpose, the multi-resolution filter depicted in Fig. 7.1 has been chosen, because it contains several up- and downsamplers together with multiple paths from the source to the sink. As already discussed in Section 7.1, this represents severe challenges not only for automatic buffer size determination but also for manual implementation.

Figure 7.24 shows the buffer analysis results for a multi-resolution filter with three filter stages and an input image size of 2048×1024 pixels. Each node corresponds to a WSDF actor, which is labeled by the corresponding actor name, the scaling vector, and the lattice shift. The source actor *top.src* starts at the lattice origin $\mathbf{S}_{\text{top.src}} = (0, 0)^T$ and uses a scaling vector of $\mathbf{S}_{\text{top.src}} = (1, 1)^T$. Each edge is annotated by the corresponding buffer size, whereas memory channel splitting has not been employed. The graph shown in Fig. 7.24 is automatically generated by the buffer analysis tool using the *Graphviz* package [2]. As can be clearly seen, the largest buffers are not required for image filtering, but for compensating the delays between the different paths from the source actor to the sink.

In order to evaluate the discussed buffer analysis technique concerning its ability to process huge application graphs, it has been furthermore applied to a multi-resolution filter with seven filter stages, leading to 69 actors and 79 edges. Table 7.1 lists the measured analysis run-time and compares it with information found in the literature about alternative approaches. Since the publications use different examples, they can only be related by means of the processed graph size. *MMAlpha* [62] is applied to a WCDMA receiver, while Feautrier chooses an equalizer consisting of a source, several parallel filters, and a mixer for evaluation of *CRP*. Finally, *IMEM* has been used to model a simple surveillance application consisting of a temporal low-pass filter, a threshold operation, several morphological filters, and a color transform.

The first conclusion that can be drawn from Table 7.1 is that buffer analysis via simulation is clearly the slowest variant. The underlying reason can be found in the slow schedule determination. Taking, for instance, the multi-resolution filter with seven filter stages, time accurate simulation without any buffer analysis already requires more than 13 min. This is

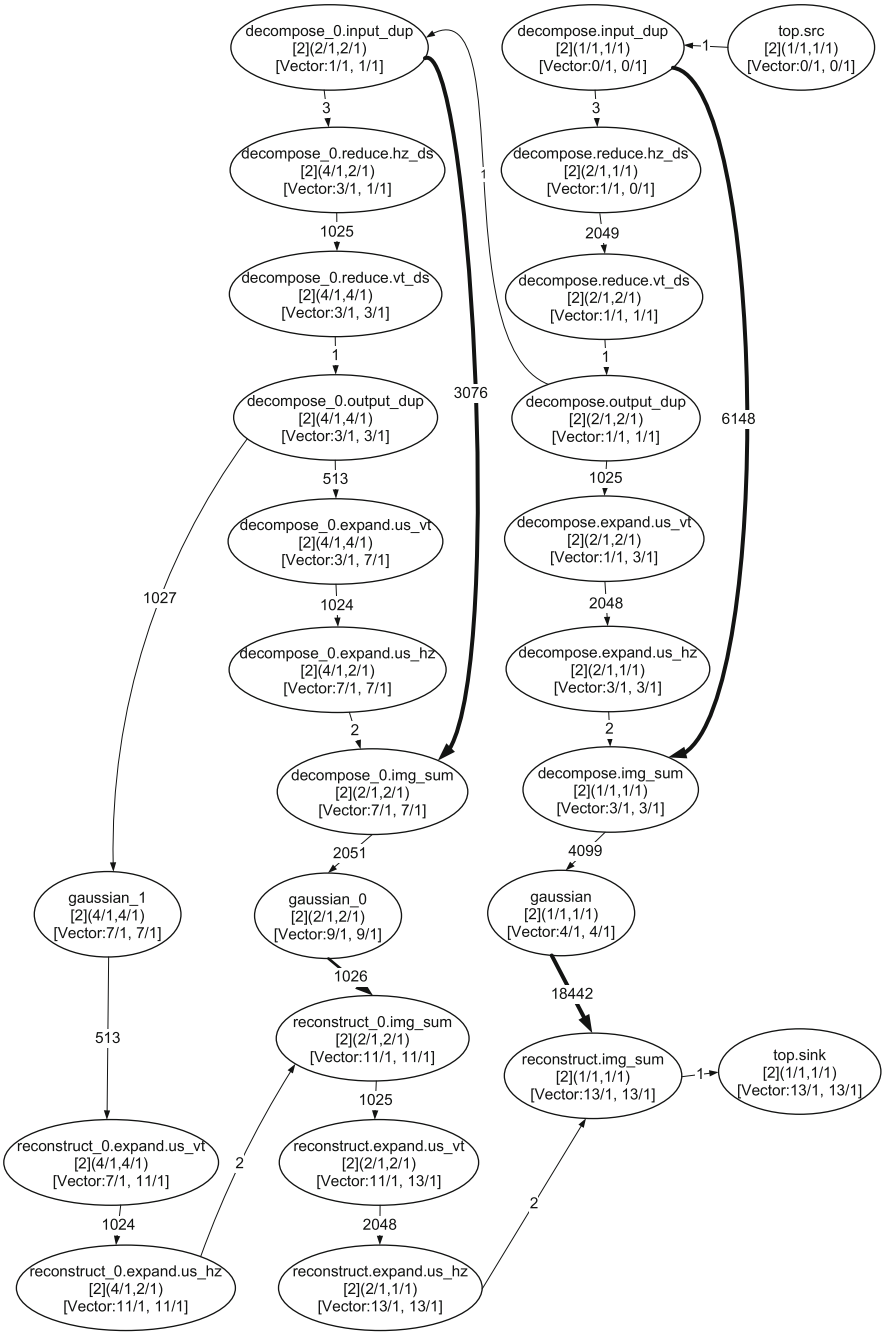


Fig. 7.24 Buffer analysis results for a multi-resolution filter with three filter stages. *Bold arrows* highlight communication edges that perform delay compensation

Table 7.1 Comparison of analysis run-times

	Actors	Edges	Time (s)	CPU
MMAlpha [62]	6	8	0.4	1.7 GHz
MMAlpha [62]	24	22	59.8	1.7 GHz
CRP [105]	7	10	73	1.4 GHz
IMEM [282] (620 × 460)	12	16	< 1	??
Simulation without buffer analysis (2048 × 1024)	69	79	13 m in 37	P4 3 GHz
Simulation with buffer analysis (2048 × 1024)	69	79	19 m in 58	P4 3 GHz
Polyhedral analysis (ILP buffer size) (2048 × 1024)	69	79	8.1	P4 3 GHz
Polyhedral analysis (buffer size by intelligent exhaustive search) (2048 × 1024)	69	79	25.5	P4 3 GHz
Polyhedral analysis (ILP buffer size) (512 × 512)	69	79	6.5	P4 3 GHz
Polyhedral analysis (buffer size by intelligent exhaustive search) (512 × 512)	69	79	4.0	P4 3 GHz
Polyhedral analysis (ILP buffer size) (4096 × 2048)	69	79	12.0	P4 3 GHz
Polyhedral analysis (buffer size by intelligent exhaustive search) (4096 × 2048)	69	79	99.0	P4 3 GHz

because each actor invocation results in generation of several *SystemC* events that have to be scheduled according to the chosen schedule strategy. On the other hand, this also means that simulation is the most flexible approach, because it is possible to employ arbitrary scheduling functions including resource sharing with different priorities or data-dependent decisions. Nevertheless, as long as this flexibility is not explicitly required, analytical buffer analysis is strongly preferable, because it is much faster and delivers better results. Considering, for instance, the sum of all communication edge buffer sizes for an input image with 2048×1024 pixels, the value returned by simulation is larger by 73% compared to the analytical approach. For individual edges, this overestimation can even reach several orders of magnitudes due to the scheduling difficulties discussed in Section 7.2.

Second, Table 7.1 also demonstrates that the analysis time depends on the processed image size. This can be explained by the fact that in the worst-case ILP solvers need to perform an exhaustive search over all unknowns whose valid ranges depend on the image size. Fortunately, the computation time does not increase linearly with the number of pixels. Taking, for instance, the two images of size 2048 × 1024 and 4096 × 2048, the latter encompasses four times as many pixels, but the analysis only takes 1.48 times longer. For the intelligent exhaustive search, the analysis time increases with the number of pixels almost linearly. Note that the relation is not completely linear in Table 7.1, because in this setup the scheduling is still performed using *ILOG CPLEX*. Interestingly, the intelligent exhaustive search is faster than using an ILP solver when using small image sizes.

Finally, Table 7.1 also demonstrates the capacity of the buffer analysis technique to process huge application graphs. Thanks to the polyhedral analysis and the usage of small integer linear programs, it is faster than MMAlpha and CRP. Only IMEM requires less processing time. However, since IMEM only supports standard sliding window algorithms without out-of-order-communication or multirate applications, analysis becomes much simpler. Furthermore, IMEM performs scheduling using an ILP whose complexity increases with the number of actors. In contrast, the approach described in this monograph only requires local integer linear programs, which is advantageous when working on application graphs with many actors and edges. Similarly, Hu et al. [143] report faster analysis speeds, as well. However, it does consider neither out-of-order communication nor lattice wraparound nor multirate

analysis. Verdoolaege et al. [292] finally report analysis times ranging from 2.5 to 46.4 s for different examples of varying complexity, where the most complex one consists of a Motion-JPEG decoder. However, this approach does also not consider out-of-order communication, multirate analysis, or lattice wraparound.

7.9.3 Memory Channel Splitting

After having demonstrated the applicability of the presented buffer analysis technique for both out-of-order communication and complex graph topologies, this section focuses on the improvement that can be achieved by memory channel splitting as discussed in Section 7.6.2. In particular, it investigates the lifting-based wavelet kernel described in Section 5.8.2 in order to demonstrate that Windowed Synchronous Data Flow together with the presented buffer analysis methodology leads to implementations with an optimal buffer configuration.

Figure 7.25a resumes the lifting scheme for a vertical 5-3 wavelet kernel, as introduced in Section 5.8.2 and as used, for instance, in JPEG2000 [152]. The rectangles labeled with numbers correspond to the input pixels, while L and H define the resulting low-pass and high-pass wavelet coefficients. Arithmetic operations are depicted by large circles, while arrows represent data flow. From the literature it is known that such a lifting scheme can be implemented by storing 3 lines [198].[5] For the bold sliding window position producing the bold L and H coefficient, for instance, the gray-shaded lines 2 and 3 have to be buffered, together with the intermediate line produced by the first arithmetic operation. In other words, the lifting-based implementation strategy reduces the required storage memory compared to ordinary sliding windows, which would buffer lines 0 and 1 instead of the intermediate results.

As memory minimization is an important aspect in embedded system design (see Chapter 6), an efficient system level design method must be able to achieve such an optimized wavelet implementation in order to get accepted by application engineers. Figure 7.25b depicts the corresponding WSDF high-level model as derived in Section 5.8.2. Since the individual actors do not need a significant amount of storage, the question of interest consists in determination of the edge buffer sizes required for implementation of the WSDF graph. To this end, it has been submitted to the buffer analysis method explained above.

Table 7.2 presents the corresponding results assuming an input image size of 2048×1088 pixels. As can be seen, without memory channel splitting, edges e_1 and e_3 have to buffer slightly more than four image lines. Unfortunately, this significantly exceeds the optimal solution needing only a storage buffer of approximately 3 lines. The underlying reason can be identified by means of Fig. 7.25a. As can be seen, generation of the bold low-pass pixel requires the data elements belonging to two effective tokens produced by WT1. As memory channel splitting has been disabled, the upper effective token has to be stored completely, leading to a buffer requirement of 2 lines.

This, however, is completely unnecessary, since only parts of the data elements belonging to effective token 1 are indeed required for calculation of the bold low-pass pixel. In order to solve this difficulty, memory channel splitting as introduced in Section 7.6.2 is required. In this case, the overall buffer requirement is 3 lines and 3 pixels, which exceeds the optimum

[5] Depending on the implementation strategy, it might be furthermore necessary to buffer the currently processed pixel of the most recent line by means of a register.

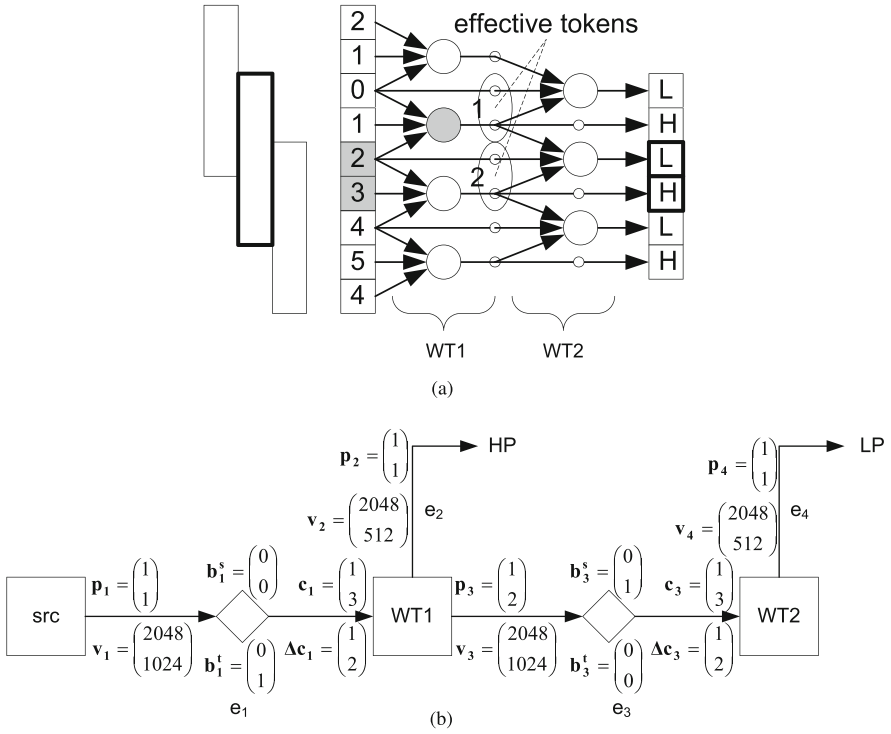


Fig. 7.25 Lifting-based vertical wavelet transform and its WSDF representation. **a** Lifting scheme. *Gray shading* identifies the pixels that have to be retrieved from a line buffer when calculating the *bold* low- and high-pass pixels. **b** Corresponding WSDF graph

Table 7.2 Edge buffer sizes for the lifting-based wavelet transform and an input image size of 2048×1088 pixels

Edge	e_1	e_2	e_3	e_4
Without channel split	$4097 = 2 \times 2048 + 1$	1	$4098 = 2 \times 2048 + 2$	1
With channel split	$4097 = 2 \times 2048 + 1$	1	$1 + 2049$	1

by only 2 pixels. This is because WSDF edges always have to buffer data elements, whereas for minimum memory Fig. 7.25a must be implemented in a pure combinatorial manner. However, for performance reasons, this is typically not done, leading to the same buffer size as determined by the buffer analysis method.

Whereas this clearly demonstrates the benefits of memory channel splitting, it has to be paid for with an additional equality constraint that has to be added to the integer linear program. Sadly, this single constraint completely overburdens available ILP solvers. Whereas the PIP library can only handle very small image sizes, ILOG CPLEX fails completely. Fortunately, this situation can be overcome by performing the intelligent exhaustive search as described in Section 7.9.1. This clearly demonstrates the disadvantages of integer linear programing as well as the benefit of being able to replace it by alternative methods, if necessary.

7.9.4 Multirate Analysis

The previous results demonstrated the applicability of the buffer analysis method to varying applications, including out-of-order communication, memory channel splitting, and complex graph topologies. This section aims to address the load smoothing techniques discussed in Section 7.7. To this end, the multi-resolution filter depicted in Fig. 7.1 has been configured with a different number of filter stages while using both a bursty and a smoothed actor schedules.

Table 7.3 depicts the corresponding results. The obtained numbers clearly show that from a buffer point of view, the bursty schedule has to be preferred. This demonstrates the usefulness of the described polyhedral buffer analysis compared to [302], which works on one-dimensional models of computation and only supports the smoothed schedule.

Table 7.3 Buffer analysis results for workload smoothing

Image size	#Stages	Bursty	Smoothed	Increase(%)
2048×1024	2	22,553	25,617	13.6
2048×1024	4	93,795	114,084	21.6
2048×1024	5	182,192	226,435	24.3
512×512	2	5657	6417	13.4
512×512	4	23,523	29,028	23.4
512×512	5	45,680	58,723	28.6

On the other hand, the smoothed schedule improves resource sharing, because it requires only smaller initiation intervals. In order to evaluate the corresponding impact, all filters, decompose, and reconstruct blocks of the multi-resolution filter depicted in Fig. 7.1 have been synthesized using the PARO behavioral compiler (see Section 3.2.7) and 32 bit fixed-point arithmetic. Table 7.4 presents the obtained results by adding the resource requirements of the different actors for both the bursty and the smoothed schedules. Evaluation has shown that 90% of the required hardware resources are assigned to the bilateral filters because they contain complex arithmetic operations like divisions and exponential functions. As this leads to important possibilities for resource sharing, the smoothed schedule reduces the FPGA resources between 2.3 and 16.1% while achieving the same system throughput. This is the case, because the bursty schedule requires initiation intervals of $1, 2, 4, \ldots$ for the filters in the different levels, whereas for the smoothed implementation $1, 4, 16, \ldots$ is sufficient. Consequently, more resources can be shared because the number of pixels that have to be processed per clock cycle decreases. In other words, high-level synthesis together with the presented buffer analysis permits to trade communication buffers for computational logic. As a consequence, it is possible to choose the implementation alternative that best fits the user requirements. This is not possible in any other related approach presented in Chapter 3.

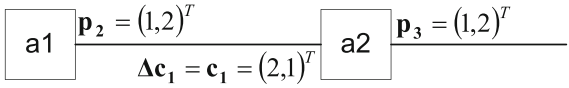
7.9.5 Limitations

As discussed in the previous sections, lattice-based scheduling not only permits to process out-of-order communication but also offers the advantage of covering both bursty and smoothed scheduling variants for multirate applications. Consequently, the designer can trade communication memory for computation memory. In contrast, for manual RTL coding, typically only one variant is implemented due to time limitations. Consequently, the presented

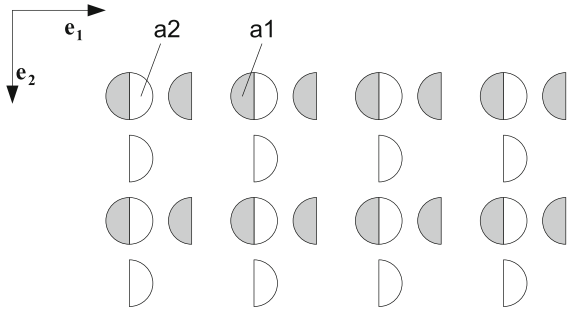
Table 7.4 FPGA resource consumption of the filters, decompose, and reconstruct blocks belonging to the multi-resolution filter

Stages		Flip-flops	Lookup tables	Multipliers
2	Bursty	38,229	37,153	292
	Smoothed	34,770	35,167	278
	Increase	−9.0%	−5.3%	−4.8%
4	Bursty	67,351	71,780	454
	Smoothed	60,144	67,877	392
	Increase	−10.7%	−5.4%	−13.7%
5	Bursty	80,494	88,748	522
	Smoothed	73,335	86,691	438
	Increase	−8.9%	−2.3%	−16.1%

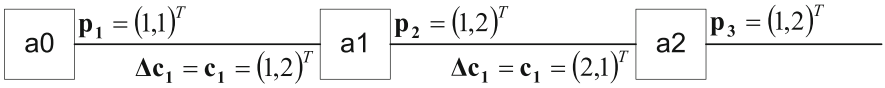
$$\mathbf{s_{a0}} = (1,1)^T,$$
$$\mathbf{s_{a1}} = (1,2)^T,$$
$$\mathbf{s_{a2}} = (2,1)^T.$$



(a)



(b)



(c)

Fig. 7.26 Example graphs demonstrating the possibility of schedules not being throughput optimal. **a** WSDF graph leading to a schedule with non-optimal throughput. **b** Corresponding lattice representation. **c** Similar WSDF graph leading to a throughput-optimal schedule

system level analysis helps to achieve better designs, since several alternatives can be quickly evaluated.

However, despite this progress, there are also examples that require further considerations, because they lead to schedules not offering the maximum possible throughput. Such a situation occurs for graphs $G = (A, E)$, where

$$\nexists a \in A : \mathbf{s_a} = \mathbf{1},$$

while $\mathbf{s_a}$ is the scaling factor of actor $a \in A$ as derived in Sections 7.3.2 and 7.5.1. Figure 7.26a depicts a corresponding example that is a simplified variant of the shuffle operation discussed later on in Section 8.1. Application of Eqs. (7.1) and (7.20) leads to

$$\begin{aligned} \mathbf{s_{a1}} &= (1, 2)^T, \\ \mathbf{s_{a2}} &= (2, 1)^T. \end{aligned}$$

Figure 7.26b depicts the corresponding lattice. As can be seen, none of the actors operates every clock cycle. Whereas actor $a1$ waits every second row, actor $a2$ is idle every second column. Consequently, this application could be implemented with a better throughput when an alternative schedule would be used, offering thus potential for further research. Note, however, that the example given in Fig. 7.26b does not show these difficulties, although only one actor has been added:

$$\begin{aligned} \mathbf{s_{a0}} &= (1, 1)^T, \\ \mathbf{s_{a1}} &= (1, 2)^T, \\ \mathbf{s_{a2}} &= (2, 1)^T. \end{aligned}$$

7.10 Conclusion

Generation of buffer efficient implementations is an important requirement for design of embedded systems, because it reduces costs and energy consumption and helps thus to build implementations that offer high throughput and long run-time in case of battery power supply. Unfortunately, determination of the required memory sizes is a laborious and complicated task, because it depends on the application parameters like window sizes, communication order, graph topology, memory organization, and scheduling decisions. Consequently, its manual solution can take a significant portion of the system development time. Corresponding challenges are, in particular, occurring out-of-order communication and data path synchronization. This second issue has been introduced by means of a multi-resolution filter, showing the need for complex scheduling and the existence of scheduling alternatives. As a consequence, automation of the buffer size determination represents an important battleground where electronic system level design tools can help developers to attain the final implementation more quickly and to eliminate possible sources of errors.

In case of application modeling with Windowed (Synchronous) Data Flow, automatic buffer size determination essentially translates to calculation of the edge buffer sizes, because they comprise dominating parts of the overall required memory. To this end, Section 3.3 reviewed several possible solution approaches. Since, however, practically none of them

operates on multidimensional data flow graphs, they cannot be applied directly. Furthermore, typically neither multirate systems including up- and downsamplers nor out-of-order communication is considered. This is, however, a fundamental part of the Windowed (Synchronous) Data Flow model of computation and has hence to be taken into account for encompassing system level design. Furthermore, most memory analysis algorithms do not consider the impact of long dependency vectors on the achievable system throughput, an aspect shown critical in this chapter.

Consequently, this chapter has presented a buffer analysis technique that is capable to handle out-of-order communication and scheduling alternatives for multirate applications. To this end, it starts from a WSDF graph that obeys the restrictions introduced in Section 5.5, and whose communication orders can be described by sequences of firing blocks. Furthermore, it is assumed that all actor invocations are executed sequentially in a possibly pipelined fashion.[6] With these assumptions, it has been shown how WSDF graphs can be translated into a polyhedral model and how lattice wraparound can help to achieve throughput-optimized schedules. Then an efficient scheduling technique has been discussed in the sense that it only requires local ILPs depending on one single edge. Based on their solution, each actor can be delayed in such a way that the resulting system stays causal, meaning that data elements are not read before being produced. Furthermore, differences in the actor activity are taken into account by corresponding lattice scaling. Employment of existing efficient graph algorithms like the Tarjan's or the Bellman–Ford algorithm helps to handle complex graph topologies including feedback loops and split and join of data paths. Once the execution times of each actor is determined, this information can be used for analytical calculation of the resulting edge buffer sizes, using once again local integer linear programs.

However, it could be demonstrated that for both scheduling and buffer size calculation, the solution of the resulting integer linear programs risks to become computationally very expensive. Unfortunately, neither the *PIP library* nor *lp_solve* have been able to process all considered application graphs. Only *ILOG CPLEX* has shown to be more robust and come out to be an efficient tool by which the buffer size determination can be performed very quickly even when the used image sizes are huge. Nevertheless, the situation is unsatisfactory, because since solving integer linear programs is an NP-complete problem, the user can never be sure whether he will obtain a result or whether the complexity leads to unacceptable solution times. Consequently, this chapter additionally described a second strategy for solving the occurring ILPs. It essentially exploits the particular structure of the used equations, which enables intelligent exhaustive search whose solution time is bounded linearly with the number of processed pixels. Whereas this typically leads to longer run-times compared with ILP solvers, it offers the great advantage of having a lower worst-case complexity than standard ILP solvers. Application to different examples demonstrated that the resulting analysis times are completely acceptable. In other words, whereas direct usage of *ILOG CPLEX* permits in most cases for quick buffer size determination, a corresponding fall-back scenario is available if the solution time of *ILOG CPLEX* gets unacceptable. This greatly enhances the practical applicability of the discussed buffer analysis method. By this means, it could be demonstrated that for small image sizes, intelligent exhaustive search is even quicker than solving integer linear programs. Furthermore, in contrast to direct usage of *ILOG CPLEX*, the intelligent exhaustive search enables memory channel splitting, a prerequisite for efficient synthesis of the lifting-based wavelet transform.

---

[6] This restriction can be easily circumvented by transforming the actor such that it outputs bigger effective tokens and reads larger sliding windows.

Apart from this application scenario, which demonstrated the benefits of WSDF also in case of non-standard sliding window algorithms, the discussed buffer analysis method also has been applied to out-of-order communication and complex graph topologies in form of a multi-resolution filter. In particular, it could be shown that the resulting analysis times are competitive compared to related approaches when considering that the presented method delivers exact results for both multirate applications and out-of-order communication. Furthermore, it can be efficiently applied to WSDF graphs consisting of many actors and edges. Additionally, it is the only known solution that can handle resulting scheduling alternatives in case of image up- and downsampling.

Whereas these are significant achievements, the experiments also revealed scenarios offering still potential for further research. Corresponding examples include, for instance, complex up- and downsampling where neither the fraction between the effective token size and the window movement nor its inverse are integers. In this case, the presented technique leads to schedules which are not memory optimal. Furthermore, if none of the actors of the WSDF graphs uses a scaling factor of 1, the resulting solution might not be throughput optimal, offering thus potential for future research. The same is valid for buffer optimal scheduling, which cannot be guaranteed by the approach described in this book. Due to complexity reasons, it first performs ASAP scheduling, followed by the resulting buffer size calculation. Whereas for all considered applications this delivered optimal or near-optimal results, there exist scenarios where ASAP scheduling does not lead to minimum buffer configurations.

Chapter 8
Communication Synthesis

The previous chapters have introduced a multidimensional data flow model of computation that can represent complex image processing applications including (i) out-of-order communication, (ii) sliding windows, (iii) parallel data access, and (iv) control flow. Furthermore, it has been shown how it helps to verify applications on a high level of abstraction assuring bounded memory execution and how required communication buffer sizes can be determined automatically either by simulation or by polyhedral buffer analysis. Corresponding tradeoffs between the required computation logic of the used hardware accelerators and the communication buffers by which they are interconnected have been evaluated in Section 7.7.

After such a system analysis and optimization, the next step in system level design consists in automatic generation of corresponding hardware–software systems. This task encompasses four different aspects, namely

1. Hardware synthesis of the actors
2. Instantiation of one or several micro-processors and generation of the required software modules
3. Generation of the communication primitives for the data exchange between different actors. This includes both hardware–hardware, hardware–software, and software–software communication
4. Assembling of the overall system

For applications described via one-dimensional models of computation, Chapter 4 has already presented a corresponding design flow employed in the system level design tool SystemCoDesigner. Among others, it performs automatic refinement of SystemoC actors into synthesizable *SystemC* modules and creation of the overall system using the Xilinx Embedded Development Kit (EDK).

Since *Windowed Data Flow (WDF)* has been integrated into SystemCoDesigner, the same approach can principally also be employed for multidimensional applications. However, communication synthesis will differ significantly, because the multidimensional communication semantics developed in Chapter 5 offer various additional features compared to ordinary FIFO data exchange:

- Overlapping sliding windows
- Virtual border extension
- Out-of-order communication
- Parallel data access on both read and write side

J. Keinert, J. Teich, *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*, Embedded Systems,
DOI 10.1007/978-1-4419-7182-1_8, © Springer Science+Business Media, LLC 2011

Consequently, this chapter will consider efficient communication synthesis for multidimensional communication in more detail. In particular, it describes a hardware primitive called *multidimensional FIFO* providing the following benefits:

- Support of both in-order communication and out-of-order communication
- FIFO-like interface for simple system integration
- Parallel read and write of all data elements belonging to one window or effective token
- Reading of one complete window and writing of a whole effective token per clock cycle
- Possibility to trade throughput against required hardware resources
- High achievable clock frequencies
- Fine-grained scheduling on pixel level for memory and latency optimized implementations
- Static analysis during compile time reducing the occurring run-time overhead

As a consequence, this multidimensional FIFO significantly outperforms related approaches discussed in Section 3.4, which either do not consider out-of-order communication at all [190], require several clock cycles per read or write operation together with low clock frequencies [318], use dynamic memory allocation with high run-time overhead [320], or cannot exploit the tradeoff between achievable throughput and required hardware resources [292]. Further details about these and other related approaches can also be found in Sections 3.5.2.2, 3.5.2.1, 3.2.2, 3.2.3, and 3.2.4.

The remainder of this chapter is as follows. Section 8.1 precisely describes the problems which shall be solved by the multidimensional hardware FIFO. Next, Section 8.2 details a corresponding solution, including memory partitioning, address generation, and fill-level control. Section 8.3 then shortly discusses how to configure the memory sizes based on the method proposed in Chapter 7. Next, Section 8.4 investigates the impact of scheduling granularity, before Section 8.5 presents several obtained synthesis results in order to demonstrate the efficiency of the multidimensional hardware FIFO. Finally, Section 8.6 concludes this chapter.

8.1 Problem Formulation

In order to describe complex image processing applications, Chapter 5 has defined a multidimensional model of computation called *Windowed Data Flow (WDF)*. It decomposes the overall system into a set of actors, which communicate via channels transporting multidimensional arrays. However, these arrays are not produced or consumed as an atomic unit, but in smaller parts. Each write operation generates a so-called effective token consisting of several data elements while each read operation accesses the data elements belonging to a sliding window of constant size. Virtual border extension permits to correctly handle border conditions. Data reordering can be performed by specification of different read and write orders via sequences of firing blocks (see Definition 5.8).

Although these communication semantics seem to have nothing in common with ordinary one-dimensional FIFOs, Section 5.4.1 illustrated that multidimensional communication can be embedded into a FIFO-like interface. This not only permits tight interaction between one- and multidimensional application parts but also enables a modular hardware design, in which actors and channels can be synthesized independently of each other and then combined to the overall system. However, compared to ordinary FIFOs, the multidimensional variant imposes two major challenges:

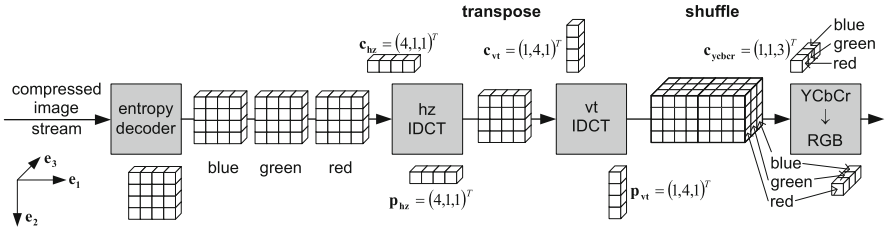


Fig. 8.1 JPEG decoder flow illustrating the read and write granularity

- Handling of data reuse due to overlapping windows
- Handling of out-of-order communication including reads and writes of several data elements in parallel

The first item addresses the fact that for overlapping windows data elements have to be read multiple times. Unfortunately, this can lead to I/O bottlenecks if the memory structure is not designed carefully. Typically, this problem is solved by instantiation of several on-chip memories that efficiently buffer data elements read for the first time such that their future access can be performed cheaply and quickly. As this aspect is rather well studied in literature, it shall not be in the focus of this monograph. Instead, the interested reader is referred to the approaches discussed in Sections 3.2.2, 3.2.3, 3.4.3, 3.5.2.1 (*IMEM*) on page 73, and 3.5.2.2. Furthermore, several behavioral compilers such as *DEFACTO* (Section 3.2.4) or *PARO* (Section 3.2.7) are already able to address data reuse internally. Consequently, in the remainder of this chapter, it is assumed that the multidimensional FIFO does not need to handle overlapping windows ($\Delta\mathbf{c} = \mathbf{c}$).

On the other hand, the second aspect, namely efficient out-of-order communication with parallel data access, is far less studied, although it plays an important role in design of image processing algorithms (see Section 2.2). As discussed previously, either it is ignored completely or slow and expensive implementations are used. As this cannot be accepted for efficient system level design of image processing applications, the following sections describe a fast, efficient, and flexible alternative.

In order to explain the problems that have to be solved in this context, Fig. 8.1 exemplarily shows parts of the process chain for a *Motion-JPEG* decoder, which processes a stream of compressed images. The latter is read by the entropy decoder and translated into a sequence of 8×8 blocks (Fig. 8.1 uses 4×4 blocks for illustration purposes). Each block is associated to the red, green, or blue color component. Next, these blocks are transformed by means of a two-dimensional *inverse discrete cosine transform* (IDCT), which is split into a horizontal and a vertical processing step. Each of them can be implemented by an efficient butterfly network [54] processing 8 pixels in parallel. Whereas the horizontal IDCT reads one row after the other ($\mathbf{c}_{\text{hz}} = \mathbf{p}_{\text{hz}} = (4, 1, 1)^T$), the vertical IDCT operates on block columns ($\mathbf{c}_{\text{vt}} = \mathbf{p}_{\text{vt}} = (1, 4, 1)^T$) as illustrated in Fig. 8.1. After the IDCT, the sequentially arriving blocks are re-ordered such that the color transform can access all three color components in parallel ($\mathbf{c}_{\text{ycbcr}} = (1, 1, 3)^T$). This reordering also helps to output the resulting image line by line as required by most display devices.

Hence, for the example shown in Fig. 8.1, two examples of out-of-order communication with parallel read and write access can be observed: The two stages of the IDCT have to be connected by means of a *transpose* operation, whereas the color transform requires that the colors are interleaved, also called *shuffled* in the remainder of this monograph. Figure 8.2

shows the corresponding access pattern together with the occurring read and write orders. Whereas the vertical IDCT generates one block after the other on the granularity of block columns, the color transform accesses all color components in parallel in a line-based manner. Thus, the order in which the data are produced and consumed differs.

Unfortunately, implementation of such communication semantics is significantly more difficult compared to simple one-dimensional FIFOs. First of all, fill-level control is much more complex. Considering, for instance, the shuffle operation depicted in Fig. 8.2, it can be seen that not each write operation induces the possibility for a corresponding sink execution. In fact, although the number of produced data elements would be sufficient, the first invocation of the *vertical IDCT* does not permit any execution of the color transform actor, because the latter requires pixels belonging to different color components while the IDCT generates columns of one single color.

Second, one-dimensional FIFOs do not permit parallel data access when the read and write tokens do not have the same sizes. In case of Fig. 8.2 this would mean that writing a complete block column would require several clock cycles. Similarly, reading three different color components could not be performed within one clock cycle. This is also true for so-called windowed FIFOs as proposed in [43, 67, 133], which principally support out-of-order communication due to random access, but which cannot handle parallel access to several data elements. This is because their used memory structure has not been designed to perform several read or write operations in parallel.

And finally, address generation is more difficult than for one-dimensional FIFOs. In the latter case, both the read and the write addresses can be obtained by a simple increment of a pointer. However, this is not true anymore for out-of-order communication. Figure 8.3, for instance, resumes the JPEG2000 tiling operation described in Fig. 5.24 of Section 5.8.2. The numbers define the order by which the source generates the input image. In case of linearization in production order as discussed in Section 6.3.2, these numbers also define the memory address where the corresponding pixels are stored. The tiler on the other hand reads the data in the order depicted by corresponding arrows. Consequently, the read address cannot be derived by a simple address increment as for the one-dimensional FIFO. Considering, for instance, the first few invocations, the sequence of read addresses equals

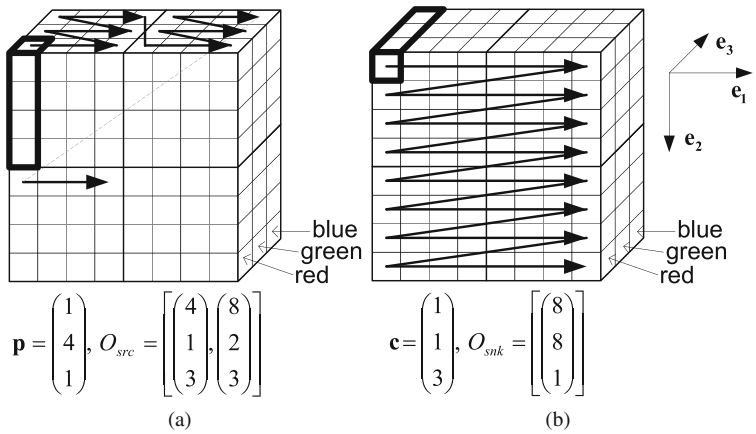


Fig. 8.2 Communication order for the shuffle operation. *Bold* cuboids define the effective token and sliding window, respectively. **a** Write order. **b** Read order

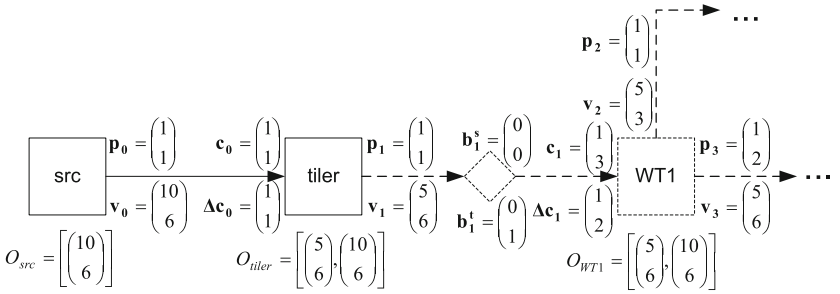


Fig. 8.3 JPEG2000 image tiling. **a** WSDF graph. **b** Token space between the src and tiler actors

$$0, 1, 2, 3, 4, 10, \ldots 53, 54, 5, 6, 7 \ldots .$$

As a consequence, the following sections describe a hardware architecture for parallel out-of-order communication, which can be parametrized with user-defined read and write orders. It provides

- an efficient address generation, which guarantees that the data are forwarded to the sink in the correct order;

- efficient fine-grained fill-level control, which takes care that the source does not overwrite data in the memory that will be required later on and that the sink only reads valid data elements; and
- a memory structure, which enables parallel reads and writes of multiple data elements.

Since this memory structure increases the hardware costs for implementation, it is designed in such a way that throughput can be traded against resource requirements by allowing different degrees of parallelism. Concerning the underlying buffer model, this chapter favors the linearization in production order against the rectangular memory organization due to the reasons discussed in Section 6.5.

- Compared to the rectangular mapping function, it requires less modulo-operations, which are expensive to implement in hardware as shown later on in Section 8.5.2. Moreover, the analysis steps are less complex.
- The analytical buffer analysis method explained in Chapter 7 can be used for determination of the required memory sizes.
- The memory sizes can be adjusted in a more fine-grained manner. In other words, the amount of used memory can be configured on a precision of individual effective tokens. In contrast, the rectangular buffer organization requires the storage size to be a multiple of images rows or even complete frames.
- Since physical memories only offer a linear address space, the rectangular mapping function needs a further linearization.
- The main focus of this book consists in parallel high-performance applications where linearization in production order typically outperforms the rectangular memory model (see also Section 6.5).

8.2 Hardware Architecture

Figure 8.4 shows the corresponding hardware architecture of a multidimensional FIFO implementing the above-mentioned tasks. It consists of two major parts, namely (i) a *controller unit* and (ii) a *memory subsystem*. The latter can be mapped to both on-chip and off-chip memory and offers different memory channels required for guaranteeing the parallel data access. The controller is responsible for the address generation and the fill-level control. The latter ensures that the sink only reads valid data and that the source does not overwrite data that will be required later on. Furthermore, it performs an optional serial to parallel and parallel to serial conversion in order to restrict the number of required memory channels and thus provides a tradeoff between throughput and resource consumption.

The interface of the multidimensional FIFO corresponds to that of an ordinary FIFO as the latter has been proven to be efficient for interconnection of processes. In other words, a full signal indicates that the next data element to write cannot be placed into the memory buffer. Similarly, the empty signal indicates that the sink has to wait for the next data element to read. Read and write counters allow predicting how many data elements can be read and written without interruption.

All these architectural elements of the multidimensional FIFO are described in more detail in the following sections.

8.2.1 Read and Write Order Control

In order to perform the data reordering necessary because of out-of-order communication, the hardware FIFO requires knowledge about the next token that will be written or read. This is done by means of two *hierarchical iteration vectors* as discussed in Section 6.2:

$$\mathbf{i}_{\text{src}} \in \mathbb{N}_0^{n_{\text{src}}},\ n_{\text{src}} \in \mathbb{N},\ 0 \leq \mathbf{i}_{\text{src}} \leq \mathbf{i}_{\text{src,max}},$$
$$\mathbf{i}_{\text{snk}} \in \mathbb{N}_0^{n_{\text{snk}}},\ n_{\text{snk}} \in \mathbb{N},\ 0 \leq \mathbf{i}_{\text{snk}} \leq \mathbf{i}_{\text{snk,max}}.$$

The relation between the current read or write operation and the accessed data elements is established via two *mapping matrices* $M_{\text{src}} \in \mathbb{N}_0^{n,(n_{\text{src}}+n)}$ and $M_{\text{snk}} \in \mathbb{N}_0^{n,(n_{\text{snk}}+n)}$, where n represents the number of token dimensions.

With these definitions, the coordinates of the data elements generated by the write operation $\mathbf{i}_{\text{src}}$ are given by $M_{\text{src}} \times \left(\mathbf{i}_{\text{src}}, \mathbf{I}_{\text{src,w}}\right)^T + \boldsymbol{\delta}$, where $\boldsymbol{\delta}$ considers possible initial data elements as defined in Section 5.1.5 on page 99. The vector $\mathbf{I}_{\text{src,w}} \in \mathbb{N}_0^n$ selects one of the data elements written in parallel (see Section 7.3.4). The possible range of $\mathbf{I}_{\text{src,w}}$ is given by $\mathbf{p} > \mathbf{I}_{\text{src,w}} \geq \mathbf{0}$, where $\mathbf{p}$ defines the effective token size whose data elements shall be written in parallel.

Similarly, $M_{\text{snk}} \times \left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right)^T - \mathbf{b}^{\mathbf{S}}$ calculates the coordinates of the data elements involved in the read operation $\mathbf{i}_{\text{snk}}$. The vector $\mathbf{I}_{\text{snk,w}} \in \mathbb{N}_0^n$, $\mathbf{c} > \mathbf{I}_{\text{snk,w}} \geq \mathbf{0}$ identifies one of the data elements read in parallel (see Section 7.3.4). The vector $\mathbf{b}^{\mathbf{S}}$ results from the virtual border extension discussed in Section 5.1.2 on page 96.

Each time the source performs a write operation, the corresponding hierarchical iteration vector $\mathbf{i}_{\text{src}}$ is lexicographically incremented:

$$\begin{cases} \langle \text{succ}\left(\mathbf{i}_{\text{src}}\left(t\right)\right), \mathbf{e_j}\rangle = \langle \mathbf{i}_{\text{src}}\left(t+1\right), \mathbf{e_j}\rangle = \\ \left(\langle \mathbf{i}_{\text{src}}\left(t\right), \mathbf{e_j}\rangle + 1\right) \bmod \left(\langle \mathbf{i}_{\text{src,max}}, \mathbf{e_j}\rangle + 1\right) & \text{if } C1 \\ \langle \mathbf{i}_{\text{src}}\left(t\right), \mathbf{e_j}\rangle & \text{if } \neg C1 \end{cases}, \\ \text{with } C1 : \forall j < k \leq n_{\text{src}} : \langle \mathbf{i}_{\text{src}}\left(t\right) - \mathbf{i}_{\text{src,max}}, \mathbf{e_k}\rangle = 0. \end{cases} \tag{8.1}$$

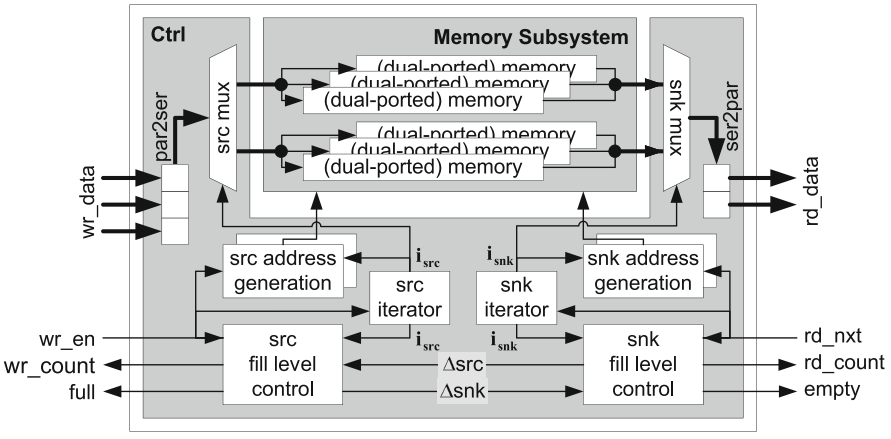


Fig. 8.4 Multidimensional FIFO hardware architecture

A similar relation can be established for the sink.

Example 8.1 In order to illustrate the above equations, consider the write order depicted in Fig. 8.2a. The latter can be described by the following sequence of firing blocks:

$$O_{\text{src}} = \left[\begin{pmatrix} 4 \\ 1 \\ 3 \end{pmatrix}, \begin{pmatrix} 8 \\ 2 \\ 3 \end{pmatrix} \right].$$

This means that four write operations in direction $\mathbf{e_1}$ and three ones in direction $\mathbf{e_3}$ are required in order to generate three 4×4 blocks of different colors. The corresponding token production is performed in raster-scan order as illustrated in Fig. 8.2a. The so-formed blocks of $4 \times 4 \times 3$ pixels are then concatenated to the overall image in raster-scan order. Altogether, this leads to 8 write operations in direction $\mathbf{e_1}$, 2 in direction $\mathbf{e_2}$, and 3 in direction $\mathbf{e_3}$. Note that for each write operation, 4 pixels are generated.

Such a write order can be described by means of the following hierarchical iteration vector and mapping matrix:

$$\mathbf{i}_{\text{src,max}} = \begin{pmatrix} 1 \\ 1 \\ 2 \\ 3 \end{pmatrix}, \; M_{\text{src}} = \begin{pmatrix} 0\,4\,0\,1\,1\,0\,0 \\ 4\,0\,0\,0\,0\,1\,0 \\ 0\,0\,1\,0\,0\,0\,1 \end{pmatrix}, \; \boldsymbol{\delta} = \mathbf{0}, \; \mathbf{p} = \begin{pmatrix} 1 \\ 4 \\ 1 \end{pmatrix}.$$

$\langle \mathbf{i}_{\text{src,max}}, \mathbf{e_4} \rangle + 1$, for instance, indicates that the source needs to execute four times in order to generate a complete 4×4 block. Similarly, $\langle \mathbf{i}_{\text{src,max}}, \mathbf{e_3} \rangle + 1$ tells that three different color components are generated. $\langle \mathbf{i}_{\text{src,max}}, \mathbf{e}_2 \rangle + 1$ and $\langle \mathbf{i}_{\text{src,max}}, \mathbf{e}_1 \rangle + 1$ finally indicate the number of complete $4 \times 4 \times 3$ blocks in order to form the overall image. Or, in other words, $\langle \mathbf{i}_{\text{src}}, \mathbf{e_4} \rangle$ and $\langle \mathbf{i}_{\text{src}}, \mathbf{e_3} \rangle$ indicate the current write position within a $4 \times 4 \times 3$ block while $\langle \mathbf{i}_{\text{src}}, \mathbf{e_2} \rangle$ and $\langle \mathbf{i}_{\text{src}}, \mathbf{e_1} \rangle$ define the current block coordinates in horizontal and vertical directions, respectively. Note that the coordinate system of the source iteration vector $\mathbf{i}_{\text{src}}$ does not correspond to that depicted in Fig. 8.2. This is necessary in order to obtain the correct write order, while the coordinate system shown in Fig. 8.2 is used to identify the individual data elements (see also Example 7.7 on page 167).

The mapping matrix is chosen such that it associates the accessed data elements to each write operation $\mathbf{i}_{\text{src}}$. Consequently, $\mathbf{i}_{\text{src}}(t) = (0, 0, 1, 3)^T$, for example, means that the source is currently writing the last column of the first green block:

$$\begin{aligned} M_{\text{src}} \times \left(\mathbf{i}_{\text{src}}(t), \mathbf{I}_{\text{src,w}} \right) &= \begin{pmatrix} 0\,4\,0\,1\,1\,0\,0 \\ 4\,0\,0\,0\,0\,1\,0 \\ 0\,0\,1\,0\,0\,0\,1 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ 1 \\ 3 \\ \langle \mathbf{I}_{\text{src,w}}, \mathbf{e_1} \rangle \\ \langle \mathbf{I}_{\text{src,w}}, \mathbf{e_2} \rangle \\ \langle \mathbf{I}_{\text{src,w}}, \mathbf{e_3} \rangle \end{pmatrix} \\ &= \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix} + \mathbf{I}_{\text{src,w}}, \end{aligned}$$

where $\mathbf{I}_{\text{src,w}}$ selects one of the data elements written in parallel:

$$0 \leq \mathbf{I}_{\text{src,w}} < \begin{pmatrix} 1 \\ 4 \\ 1 \end{pmatrix}.$$

Application of Eq. (8.1) leads to $\mathbf{i}_{\text{src}}(t+1) = (0,0,2,0)^T$, indicating that next the source will write the first column of the first blue block. Note that the higher coordinates of the iteration vector are incremented first.

8.2.1.1 Automatic Derivation of the Hierarchical Iteration Maxima and Mapping Matrices

As discussed in Chapter 5, the WDF specification of a communication edge contains, among others, the effective token and sliding window sizes, as well as the occurring read and write orders $O_{\text{snk}} = \left[\mathbf{B}_{\mathbf{1}}^{\text{snk}}, \ldots, \mathbf{B}_{\mathbf{q}_{\text{snk}}}^{\text{snk}}\right]$ and $O_{\text{src}} = \left[\mathbf{B}_{\mathbf{1}}^{\text{src}}, \ldots, \mathbf{B}_{\mathbf{q}_{\text{src}}}^{\text{src}}\right]$. Thus, for synthesis, $\mathbf{i}_{\text{src,max}}$ and $\mathbf{i}_{\text{snk,max}}$ have to be derived automatically. Fortunately, this is easily possible by application of the following equation (see also Eq. (6.1)):

$$\forall 1 \leq j \leq n,\ 0 \leq k < q_{\text{src}} : \langle \mathbf{i}_{\text{src,max}}, \mathbf{e}_{\mathbf{n \times (q_{\text{src}} - k) - j + 1}} \rangle = \frac{\langle \mathbf{B}_{\mathbf{k+1}}^{\text{src}}, \mathbf{e_j} \rangle}{\langle \mathbf{B}_{\mathbf{k}}^{\text{src}}, \mathbf{e_j} \rangle} - 1, \tag{8.2}$$

with $\mathbf{B_0} = \mathbf{1}$. In case $\frac{\langle \mathbf{B}_{\mathbf{k+1}}^{\text{src}}, \mathbf{e_j} \rangle}{\langle \mathbf{B}_{\mathbf{k}}^{\text{src}}, \mathbf{e_j} \rangle} = 1$, the resulting component $\langle \mathbf{i}_{\text{src,max}}, \mathbf{e}_{\mathbf{n \times (q-k) - j + 1}} \rangle$ amounts zero and can be simply skipped, giving thus n_{src} as defined above. Since each component $\langle \mathbf{i}_{\text{src}}, \mathbf{e_i} \rangle$ defines the belonging to a firing block, creation of the mapping matrix M_{src} is straightforward. The same holds for $\mathbf{i}_{\text{snk,max}}$ and M_{snk}.

8.2.1.2 Extended Iteration Vectors

Whereas the synthesized hardware FIFO only contains the above-discussed iteration vectors $\mathbf{i}_{\text{src}} \in \mathbb{N}^{n_{\text{src}}}$ and $\mathbf{i}_{\text{snk}} \in \mathbb{N}^{n_{\text{snk}}}$, mathematical analysis performed during compile time occasionally also requires the (local) schedule period (see Section 5.5) that is currently processed. Such a local schedule period typically corresponds to a processed image or block. In order to ease notation, this schedule period is integrated into an *extended iteration vector* $\tilde{\mathbf{i}}_{\text{src}} \in \mathbb{N}^{n_{\text{src}}+1}$ labeled by a tilde. $\langle \tilde{\mathbf{i}}_{\text{src}}, \mathbf{e_0} \rangle$ corresponds to the schedule period while $\langle \tilde{\mathbf{i}}_{\text{src}}, \mathbf{e_k} \rangle = \langle \mathbf{i}_{\text{src}}, \mathbf{e_k} \rangle,\ 1 \leq k \leq n_{\text{src}}$ defines the components available in the hardware implementation. Note that $\mathbf{e_0}$ is exceptionally used to access the first component of a vector in order to ease notation and to emphasize the special meaning of the (local) schedule period. Similarly, $\tilde{\mathbf{i}}_{\text{snk}} \in \mathbb{N}^{n_{\text{snk}}+1}$ represents an extended sink iteration vector containing the currently processed schedule period.

Example 8.2 Example 8.1 derived the maximum hierarchical source iteration vector belonging to Fig. 8.2a:

$$\mathbf{i}_{\text{src,max}} = \begin{pmatrix} 1 \\ 1 \\ 2 \\ 3 \end{pmatrix}.$$

The extended iteration vector $\tilde{\mathbf{i}}_{\text{src}} \in \mathbb{N}^5$ thus has five dimensions. The corresponding extended maximum source iteration vector amounts

$$\tilde{\mathbf{i}}_{\text{src,max}} = \begin{pmatrix} \infty \\ 1 \\ 1 \\ 2 \\ 3 \end{pmatrix}.$$

The first component $\langle \tilde{\mathbf{i}}_{\text{src,max}}, \mathbf{e_0} \rangle$ indicates the number of (local) schedule periods (see Section 5.5) that are processed by the multidimensional FIFO. Since it operates on an infinite stream of data, the iteration vector maximum $\langle \tilde{\mathbf{i}}_{\text{src,max}}, \mathbf{e_0} \rangle = \infty$ is not bounded. The data elements produced during consecutive schedule periods are assumed to be concatenated in dimension $\mathbf{e_n}$ as exemplified in Fig. 8.5, where n is the number of token dimensions. The source mapping matrix consequently becomes

$$\tilde{M}_{\text{src}} = \begin{pmatrix} 0 & 0 & 4 & 0 & 1 & 0 & 0 & 1 \\ 0 & 4 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Remark 8.3 Note that the unbounded iteration vector component $\langle \tilde{\mathbf{i}}_{\text{src,max}}, \mathbf{e_0} \rangle$ does not inhibit successful implementation in hardware, since it is only required for analysis purposes and will vanish later on.

8.2.2 Memory Partitioning

In general, classical FIFOs are built on top of a single possibly dual-ported physical memory. For parallel out-of-order communication, however, this is not possible in all cases, since parallel data access might require several read and write ports.

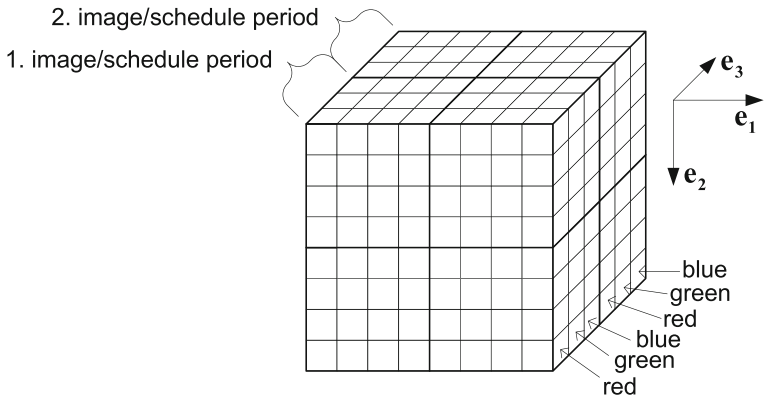
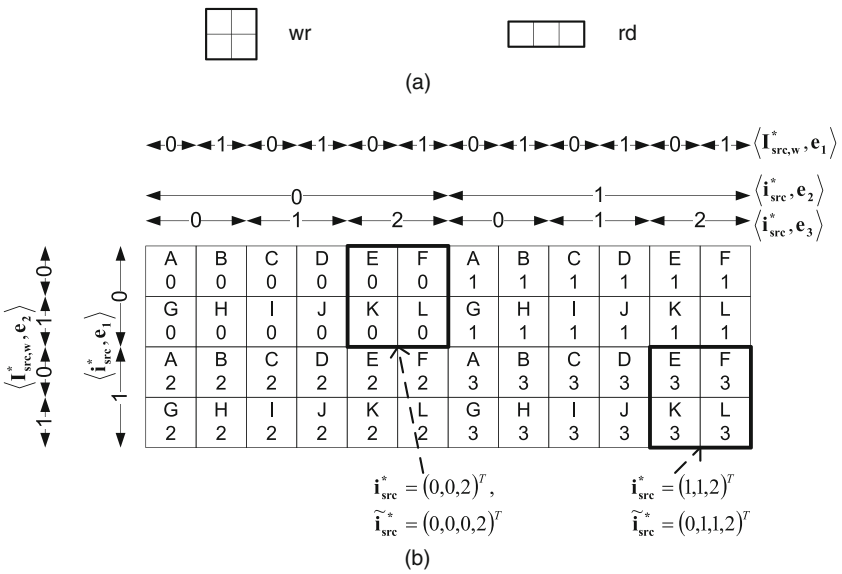


Fig. 8.5 Extension of the token space by consecutive schedule periods. In this example, a schedule period is assumed to correspond to one image

This is exemplarily illustrated in Fig. 8.6, in which both the sink and the source are assumed to scan the image in raster-scan order from left to right and from top to bottom. However, whereas the source writes 2×2 data elements in parallel, the sink tokens consist of 1 line and 3 columns, thus leading to out-of-order communication. Consequently, if all data elements of the image were placed sequentially in one single memory, it would not be possible to terminate a read or write access within one clock cycle, as both of them induce sequential accesses to several data elements. Due to the different sizes of the read and write tokens, it does furthermore not help to simply combine all data elements belonging to one source token into the same memory word, since in this case a read operation would still require to access at least two different memory cells.

In order to solve this problem, the array of data elements is partitioned in each dimension $1 \leq i \leq n$ into $\langle \mathbf{m}, \mathbf{e_i} \rangle$, $\mathbf{m} \in \mathbb{N}^n$ so-called *virtual memory channels*, which will be mapped later on to *physical memories* (see Section 8.2.4). Figure 8.6b exemplarily shows



A 0	B 0	C 0	D 0	E 0	F 0	A 1	B 1	C 1	D 1	E 1	F 1
G 0	H 0	I 0	J 0	K 0	L 0	G 1	H 1	I 1	J 1	K 1	L 1
A 2	B 2	C 2	D 2	E 2	F 2	A 3	B 3	C 3	D 3	E 3	F 3
G 2	H 2	I 2	J 2	K 2	L 2	G 3	H 3	I 3	J 3	K 3	L 3

A 0	B 1	C 4	D 5	E 8	F 9	A 12	B 13	C 16	D 17	E 20	F 21
G 2	H 3	I 6	J 7	K 10	L 11	G 14	H 15	I 18	J 19	K 22	L 23
A 24	B 25	C 28	D 29	E 32	F 33	A 36	B 37	C 40	D 41	E 44	F 45
G 26	H 27	I 30	J 31	K 34	L 35	G 38	H 39	I 42	J 43	K 46	L 47

Fig. 8.6 Memory partitioning for parallel data access. *Letters* indicate the virtual memory channel; *Arabic numbers* the address of the data element in the memory. **a** Write and read tokens. **b** Memory partition with linear address function for write accesses.**c** Memory partition with address generation by linearization in production order

a corresponding partitioning where each virtual memory channel is identified by a unique letter.

This, however, means that the address generation scheme following linearization in production order as presented in Section 6.3.2 has to be adapted, because Chapter 6 assumed one single memory module and address space per WDF communication edge (see Section 6.3). This is exemplified in Fig. 8.6c, which employs Eq. (6.7) for address generation. As can be clearly seen, this approach leads to very bad memory utilization. Considering, for instance, the virtual memory channel "B," only addresses 1, 13, 25, and 37 are used resulting in an enormous waste of memory.[1]

Consequently, a modified address generation scheme has to be employed that still follows linearization in production order, but considers each virtual memory channel individually. Whereas the mathematical details will be presented in Section 8.2.3, Fig. 8.6b already shows the corresponding principle for the scenario given in Fig. 8.6a, assuming token production in raster-scan order from left to right and from top to bottom. As can be seen, the memory addresses principally follow the production order, but consider each virtual memory channel individually.

By setting $\langle \mathbf{m}, \mathbf{e_i} \rangle = \text{scm}\left(\langle \mathbf{p}, \mathbf{e_i} \rangle, \langle \mathbf{c}, \mathbf{e_i} \rangle\right) \geq \max\left(\langle \mathbf{p}, \mathbf{e_i} \rangle, \langle \mathbf{c}, \mathbf{e_i} \rangle\right)$, it can be guaranteed that each read and write access can be terminated within one clock cycle. The smallest common multiple has been selected instead of the maximum operation because the latter can lead to non-linear address relations as shown in the following lemma. Since this complicates hardware implementation, the smallest common multiple is preferred instead, because it permits an efficient address generation as described in the next section.

Lemma 8.4 *Using* $\max\left(\langle \mathbf{p}, \mathbf{e_i} \rangle, \langle \mathbf{c}, \mathbf{e_i} \rangle\right)$ *instead of* $\text{scm}\left(\langle \mathbf{p}, \mathbf{e_i} \rangle, \langle \mathbf{c}, \mathbf{e_i} \rangle\right)$ *virtual memory channels can lead to non-linear dependencies between the write address and the corresponding hierarchical source iteration vector.*

Proof In order to prove the above lemma, consider the example given in Fig. 8.6a. For any given source invocation $\mathbf{i}_{\text{src}}$, $a\left(\mathbf{i}_{\text{src}}\right)$ shall return the corresponding write address. As the function a is requested to be linear, the following holds

$$a\left(\mathbf{i}_{\text{src},\mathbf{1}} + \mathbf{i}_{\text{src},\mathbf{2}}\right) = a\left(\mathbf{i}_{\text{src},\mathbf{1}}\right) + a\left(\mathbf{i}_{\text{src},\mathbf{2}}\right). \tag{8.3}$$

Now, consider an extract of the produced image that is shown in Fig. 8.7. It illustrates the resulting write addresses when using $\max\left(\langle \mathbf{p}, \mathbf{e_1} \rangle, \langle \mathbf{c}, \mathbf{e_1} \rangle\right) = 3$ virtual memory channels in dimension $\mathbf{e_1}$. In the remainder of this proof, special attention shall be paid on the upper left pixel of each source invocation. Then for $\Delta\mathbf{i}_{\text{src}} = \mathbf{i}_{\text{src},\mathbf{2}} - \mathbf{i}_{\text{src},\mathbf{1}}$, Eq. (8.3) leads to $a\left(\Delta\mathbf{i}_{\text{src}}\right) = a\left(\mathbf{i}_{\text{src},\mathbf{2}}\right) - a\left(\mathbf{i}_{\text{src},\mathbf{1}}\right) = 0 - 0 = 0$ as illustrated in Fig. 8.7. This, however, implies that $\mathbf{i}_{\text{src},3} \neq \mathbf{i}_{\text{src},2} + \Delta\mathbf{i}_{\text{src}}$, since $a\left(\mathbf{i}_{\text{src},\mathbf{3}} - \mathbf{i}_{\text{src},\mathbf{2}}\right) = a\left(\mathbf{i}_{\text{src},\mathbf{3}}\right) - a\left(\mathbf{i}_{\text{src},\mathbf{2}}\right) = 1 - 0 = 1 \neq 0$. As the invocations $\mathbf{i}_{src,\mathbf{j}}$, $1 \leq j \leq 4$ are successive invocations, Eq. (8.1) requires that $\mathbf{i}_{\text{src},\mathbf{2}} - \mathbf{i}_{\text{src},\mathbf{1}} = \mathbf{i}_{\text{src},\mathbf{4}} - \mathbf{i}_{\text{src},\mathbf{3}} = \Delta\mathbf{i}_{\text{src}}$. Since, however, $a\left(\mathbf{i}_{\text{src},\mathbf{4}} - \mathbf{i}_{\text{src},\mathbf{3}}\right) = a\left(\mathbf{i}_{\text{src},\mathbf{4}}\right) - a\left(\mathbf{i}_{\text{src},\mathbf{3}}\right) = 2 - 1 = 1 \neq 0$, $a\left(\mathbf{i}_{\text{src}}\right)$ cannot be a linear function.

In other words, when using $\max\left(\langle \mathbf{p}, \mathbf{e_i} \rangle, \langle \mathbf{c}, \mathbf{e_i} \rangle\right)$ virtual memory channels for the example shown in Fig. 8.6, there does not exist any source iteration vector scheme that leads to a linear address function $a\left(\mathbf{i}_{\text{src}}\right)$. Instead, a modulo-function has to be employed, which makes

[1] Note that the rectangular memory model would show the same difficulties.

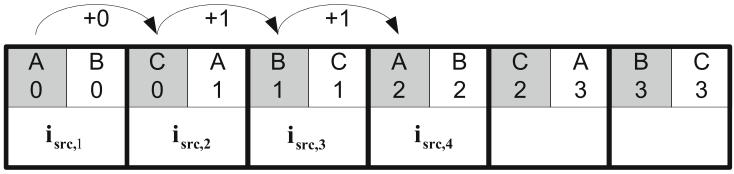


Fig. 8.7 Alternative memory mapping for Fig. 8.6 using only max ($\langle \mathbf{p}, \mathbf{e_i} \rangle$, $\langle \mathbf{c}, \mathbf{e_i} \rangle$) virtual memory channels

hardware implementation significantly more expensive if $\langle \mathbf{m}, \mathbf{e_i} \rangle$ is not a power of 2 (see also Section 8.5.2).

8.2.3 Source Address Generation

As discussed in Section 8.1, the data elements to store shall be mapped to the individual memory modules using linearization in production order, which has been introduced in Section 6.3.2. However, whereas Chapter 6 assumed one single address space per communication edge, parallel access to several data elements requires to distribute them to several memory modules. Consequently, a modified address generation scheme has to be employed that still follows linearization in production order but that considers each virtual memory channel separately as exemplified in Fig. 8.6b. It illustrates the memory partitioning of the processed image together with the addresses where each data element is written to. Considering, for instance, memory channel "B," it can be seen that the write address increments by 1 for each effective token involving that memory channel. The same observation can be made for the other memory channels, whereas each memory channel is treated completely independent of the other ones.

This, however, leads to quite complex address calculation rules. Considering, for instance, Fig. 8.6b, it can be seen that the first three source invocations write to address zero before the latter is incremented. If the read and write token sizes were interchanged, the source would even write two tokens to address zero, followed by two tokens for address one, before falling back to address zero again. In other words, a simple address increment as for ordinary FIFOs is not sufficient for address generation.

Fortunately, this problem can be solved efficiently in hardware. First of all, it can be seen that all data elements of one source token are stored at the same address. This is due to the fact that the number of virtual memory channels corresponds to the smallest common multiple of the source and sink token sizes. Second, an address function can be derived that depends linearly on the source iteration vector $\tilde{\mathbf{i}}_{\text{src}}$.

For this purpose, the components $\langle \tilde{\mathbf{i}}_{\text{src}}, \mathbf{e_k} \rangle$ of the hierarchical iteration vector $\tilde{\mathbf{i}}_{\text{src}}$ are divided into two sets, namely those contributing to the source address and those deciding to which virtual memory channel to write and thus controlling the *src-mux* shown in Fig. 8.4. Due to the tight relation between $\tilde{\mathbf{i}}_{\text{src}}$ and the firing blocks (see Section 8.2.1), this of course is only possible if there exists a firing block $\mathbf{B}_\mathbf{k}^{\text{src}}$ whose extent matches the number of virtual memory channels:

$$\forall 1 \leq i \leq n \, \exists 1 \leq k \leq q_{\text{src}} : \langle \mathbf{B}_\mathbf{k}^{\text{src}}, \mathbf{e_i} \rangle \times \langle \mathbf{p}, \mathbf{e_i} \rangle = \langle \mathbf{m}, \mathbf{e_i} \rangle \tag{8.4}$$

$$\Rightarrow \quad \forall 1 \leq i \leq n \, \exists 0 \leq k \leq n_{\text{src}} : \langle \tilde{M}_{\text{src}} \times \mathbf{e_k}, \mathbf{e_i} \rangle = \langle \mathbf{m}, \mathbf{e_i} \rangle. \tag{8.5}$$

This equation ensures that the components of the hierarchical iteration vector can be split into those indicating to which virtual memory channel to write and those that contribute to the source address. Consequently, the memory address where to write to can be calculated for each virtual memory channel $\mathbf{C}$ as

$$\tilde{a}\left(\tilde{\mathbf{i}}_{\text{src}}\right) = a\left(\tilde{\mathbf{i}}_{\text{src}}\right) \bmod B\left(\mathbf{C}\right), \tag{8.6}$$

$$a\left(\tilde{\mathbf{i}}_{\text{src}}\right) = \left(\sum_{j=0}^{n_{\text{src}}} A_j \times \langle \tilde{\mathbf{i}}_{\text{src}}, \mathbf{e_j} \rangle\right), \tag{8.7}$$

with

$$A_k = \begin{cases} 0 & \text{if } \tilde{M}_{\text{src}} \times \mathbf{e_k} < \mathbf{m} \text{ (a)} \\ \max\left\{\{1\} \cup H\left(k\right)\right\} & \text{otherwise} \quad \text{(b)} \end{cases}, \tag{8.8}$$

$$H\left(k\right) = \left\{ A_j \times \left(\langle \tilde{\mathbf{i}}_{\text{src,max}}, \mathbf{e_j} \rangle + 1\right) \middle| \, k < j \leq n_{\text{src}} \right\},$$

and $\tilde{\mathbf{i}}_{\text{src}}$ being the extended iteration vector as introduced in Section 8.2.1.2. Despite $\langle \tilde{\mathbf{i}}_{\text{src}}, \mathbf{e_0} \rangle = \infty$, this does not impact the applicability for hardware implementation, because $\langle \tilde{\mathbf{i}}_{\text{src}}, \mathbf{e_0} \rangle$ will vanish later on.

$a\left(\tilde{\mathbf{i}}_{\text{src}}\right)$ is called *linear part of the write address* and increments in production order while considering each virtual memory channel individually. $B\left(\mathbf{C}\right)$ represents the size of each virtual memory channel $\mathbf{C}$, with $0 \leq \mathbf{C} < \mathbf{m}$. Thus, for each value of $B\left(\mathbf{C}\right)$, a corresponding source address generator has to be instantiated as depicted in Fig. 8.4. Its value is forwarded to those memory modules that make part of the virtual memory channel $\mathbf{C}$. Thus, whenever the source writes to a given virtual memory channel $\mathbf{C}$, the determined address is used to decide where to place the data element. Note that $B\left(\mathbf{C}\right)$ is typically a power of 2 in order to permit efficient hardware implementation. However, the source address generator depicted in Fig. 8.4 has been implemented such that also arbitrary values can be chosen.

In case Eq. (8.4) does not directly hold, formula (8.6) can still be applied as long as the following condition is fulfilled:

$$\frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\langle \mathbf{p}, \mathbf{e_i} \rangle \times \beta_i^{\leq}} \in \mathbb{N} \wedge \frac{\langle \mathbf{p}, \mathbf{e_i} \rangle \times \beta_i^{\geq}}{\langle \mathbf{m}, \mathbf{e_i} \rangle} \in \mathbb{N}, \tag{8.9}$$

with

$$\beta_i^{\leq} = \max\left(\left\{ \langle \mathbf{B_k^{src}}, \mathbf{e_i} \rangle \middle| \, 1 \leq k \leq q_{\text{src}} \wedge \langle \mathbf{B_k^{src}}, \mathbf{e_i} \rangle \leq \frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\langle \mathbf{p}, \mathbf{e_i} \rangle} \right\} \cup \{1\}\right),$$

$$\beta_i^{\geq} = \min\left(\left\{ \langle \mathbf{B_k^{src}}, \mathbf{e_i} \rangle \middle| \, 1 \leq k \leq q_{\text{src}} \wedge \langle \mathbf{B_k^{src}}, \mathbf{e_i} \rangle \geq \frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\langle \mathbf{p}, \mathbf{e_i} \rangle} \right\} \cup \{1\}\right).$$

$O_{\text{src}} = \left[\mathbf{B_1^{src}}, \ldots, \mathbf{B_{q_{src}}^{src}}\right]$ is supposed to be the sequence of firing blocks describing the write order as discussed in Definition 5.8, and $\mathbf{p}$ defines the number of data elements written in parallel.

The above condition ensures that the virtual memory channels are covered by complete firing blocks and that the data elements produced within the firing blocks are a multiple of the virtual memory channels. In other words, the hierarchical source iteration vector $\mathbf{i}_{\text{src}}$ can be transformed in such a way that Eq. (8.5) holds.

Example 8.5 In order to illustrate the above equations, consider again the example shown in Fig. 8.6, leading to

$$\mathbf{p} = (2,2)^T\ ,\ O_{\text{src}} = \left[(6,2)^T\right],\ \mathbf{c} = (3,1)^T\ ,\ O_{\text{snk}} = \left[(4,4)^T\right],$$
$$\mathbf{i}_{\text{src,max}} = (1,5)^T\ ,\ M_{\text{src}} = \begin{pmatrix} 0 & 2 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{pmatrix},\ \mathbf{m} = (6,2)^T\ .$$

Whereas Eq. (8.9) is already fulfilled, condition (8.5) is violated for $i = 1$. This is because the six horizontal write operations of the source are covered by one single iteration level $\langle \mathbf{i}_{\text{src,max}}, \mathbf{e_2} \rangle$. However, since the virtual memory channels are completely covered by three write operations, this means that the second component of the hierarchical iteration vector both decides to which virtual memory channel to write and contributes to the source address. Unfortunately, this leads to non-linear address dependencies, which are to avoid.

Consequently, a transformation of the hierarchical source iteration vector is required that splits the 6 invocations of the second component $\langle \mathbf{i}_{\text{src,max}}, \mathbf{e_2} \rangle$ into 2×3 ones:

$$\mathbf{i}^*_{\text{src,max}} = (1,1,2)^T\ ,\ M^*_{\text{src}} = \begin{pmatrix} 0 & 6 & 2 & 1 & 0 \\ 2 & 0 & 0 & 0 & 1 \end{pmatrix}. \tag{8.10}$$

Note that this transformation is always possible as condition (8.9) holds. The extended iteration vector as introduced in Section 8.2.1.2 thus becomes

$$\tilde{\mathbf{i}}^*_{\text{src,max}} = (\infty,1,1,2)^T\ ,\ \tilde{M}^*_{\text{src}} = \begin{pmatrix} 0 & 0 & 6 & 2 & 1 & 0 \\ 4 & 2 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Figure 8.6b shows the resulting interpretation of the iteration vector for one single (local) schedule period. It can be clearly seen that $\langle \mathbf{i}^*_{\text{src,max}}, \mathbf{e_3} \rangle$ does not contribute to the source address but can be used to decide to which virtual memory channel to write. This is captured by condition (8.8a). Condition (8.8b) leads to a recursive equation whose solution leads to $A_3 = 0$, $A_2 = 1$, $A_1 = 2$, $A_0 = 4$. Hence, for the first processed image/schedule period ($\langle \tilde{\mathbf{i}}_{\text{src}^*}, \mathbf{e_0} \rangle = 0$), this leads to the addresses shown in Fig. 8.6b. Thanks to the iteration vector transformation in (8.10), address generation in Eq. (8.6) does not contain any modulo-operation with a divisor not being a power of 2 although the firing blocks are not powers of 2. Only the buffer size B (**C**) is part of a modulo-operation, but as the latter in general is a power of 2, its hardware implementation is for free.

Although Eq. (8.7) seems to contain a huge amount of multiplications, it can be implemented very efficiently in hardware. This is, because $\tilde{\mathbf{i}}_{\text{src}}$ does not vary arbitrarily, but in a predefined manner as described in Eq. (8.1). Hence, instead of performing a huge amount of multiplications, it is possible to determine $\Delta a\left(\tilde{\mathbf{i}}_{\text{src}}\right) = a\left(\text{succ}\left(\tilde{\mathbf{i}}_{\text{src}}\right)\right) - a\left(\tilde{\mathbf{i}}_{\text{src}}\right)$, where $\text{succ}\left(\tilde{\mathbf{i}}_{\text{src}}\right)$ has been defined in Eq. (8.1). $\Delta a\left(\tilde{\mathbf{i}}_{\text{src}}\right)$ can be easily derived by some multiplexers, because it only depends on whether $\langle \tilde{\mathbf{i}}_{\text{src}}, \mathbf{e_k} \rangle = \langle \tilde{\mathbf{i}}_{\text{src,max}}, \mathbf{e_k} \rangle$ or not $\forall 1 \leq k \leq n_{\text{src}}$:

$$\Delta a\left(\tilde{\mathbf{i}}_{\mathrm{src}}\right) = f\left(\langle\tilde{\mathbf{i}}_{\mathrm{src}}, \mathbf{e_k}\rangle = \langle\tilde{\mathbf{i}}_{\mathrm{src,max}}, \mathbf{e_k}\rangle \, \forall 1 \leq k \leq n_{\mathrm{src}}\right).$$

Thus, instead of direct application of Eq. (8.7), the write address can be calculated by

$$\tilde{a}\left(\mathrm{succ}\left(\tilde{\mathbf{i}}_{\mathrm{src}}\right)\right) = a\left(\mathrm{succ}\left(\tilde{\mathbf{i}}_{\mathrm{src}}\right)\right) \bmod B\,(\mathbf{C}), \tag{8.11}$$

$$a\left(\mathrm{succ}\left(\tilde{\mathbf{i}}_{\mathrm{src}}\right)\right) = \left(a\left(\tilde{\mathbf{i}}_{\mathrm{src}}\right) + \Delta a\left(\tilde{\mathbf{i}}_{\mathrm{src}}\right)\right). \tag{8.12}$$

Note that $\langle\tilde{\mathbf{i}}_{\mathrm{src}}, \mathbf{e_0}\rangle$ occurring in Eq. (8.6) is not required anymore. Thus, the above equations can be rewritten as

$$\tilde{a}\left(\mathrm{succ}\left(\mathbf{i}_{\mathrm{src}}\right)\right) = a\left(\mathrm{succ}\left(\mathbf{i}_{\mathrm{src}}\right)\right) \bmod B\,(\mathbf{C}),$$
$$a\left(\mathrm{succ}\left(\mathbf{i}_{\mathrm{src}}\right)\right) = \left(a\left(\mathbf{i}_{\mathrm{src}}\right) + \Delta a\left(\mathbf{i}_{\mathrm{src}}\right)\right).$$

Hence, instead of several multiplications, a simple adder together with some multiplexers are sufficient to calculate Eq. (8.11). Moreover, $\langle\tilde{\mathbf{i}}_{\mathrm{src}}, \mathbf{e_0}\rangle$ vanishes, which is very important, because its value is not bounded.

Example 8.6 Example 8.5 calculated the following equation for address generation:

$$\begin{aligned}\tilde{a}\left(\tilde{\mathbf{i}}_{\mathrm{src}^*}\right) &= a\left(\tilde{\mathbf{i}}_{\mathrm{src}^*}\right) \bmod B\,(\mathbf{C}) \\ &= \left(\sum_{j=0}^{3} A_j \times \langle\tilde{\mathbf{i}}_{\mathrm{src}^*}, \mathbf{e_j}\rangle\right) \bmod B\,(\mathbf{C}) \\ &= \left(4 \times \langle\tilde{\mathbf{i}}_{\mathrm{src}^*}, \mathbf{e_0}\rangle + 2 \times \langle\tilde{\mathbf{i}}_{\mathrm{src}^*}, \mathbf{e}_1\rangle + 1 \times \langle\tilde{\mathbf{i}}_{\mathrm{src}^*}, \mathbf{e}_2\rangle\right) \bmod B\,(\mathbf{C}).\end{aligned}$$

Due to the lexicographic increment defined by Eq. (8.1), this can be rewritten as

$$\begin{aligned} a\,(\mathbf{0}) &= 0 \\ a\left(\mathrm{succ}\left(\tilde{\mathbf{i}}_{\mathrm{src}^*}\right)\right) &= \left(a\left(\tilde{\mathbf{i}}_{\mathrm{src}^*}\right) + \Delta a\left(\tilde{\mathbf{i}}_{\mathrm{src}^*}\right)\right) \\ &= \left(a\left(\tilde{\mathbf{i}}_{\mathrm{src}^*}\right) + \begin{cases} 4 & \text{if } \langle\tilde{\mathbf{i}}_{\mathrm{src}^*}, \mathbf{e_1}\rangle = \langle\tilde{\mathbf{i}}^*_{\mathrm{src,max}}, \mathbf{e_1}\rangle \\ & \quad \wedge\langle\tilde{\mathbf{i}}_{\mathrm{src}^*}, \mathbf{e_2}\rangle = \langle\tilde{\mathbf{i}}^*_{\mathrm{src,max}}, \mathbf{e_2}\rangle \\ & \quad \wedge\langle\tilde{\mathbf{i}}_{\mathrm{src}^*}, \mathbf{e_3}\rangle = \langle\tilde{\mathbf{i}}^*_{\mathrm{src,max}}, \mathbf{e_3}\rangle \\ 2 & \text{if } \langle\tilde{\mathbf{i}}_{\mathrm{src}^*}, \mathbf{e_1}\rangle \neq \langle\tilde{\mathbf{i}}^*_{\mathrm{src,max}}, \mathbf{e_1}\rangle \\ & \quad \wedge\langle\tilde{\mathbf{i}}_{\mathrm{src}^*}, \mathbf{e_2}\rangle = \langle\tilde{\mathbf{i}}^*_{\mathrm{src,max}}, \mathbf{e_2}\rangle \\ & \quad \wedge\langle\tilde{\mathbf{i}}_{\mathrm{src}^*}, \mathbf{e_3}\rangle = \langle\tilde{\mathbf{i}}^*_{\mathrm{src,max}}, \mathbf{e_3}\rangle \\ 1 & \text{if } \langle\tilde{\mathbf{i}}_{\mathrm{src}^*}, \mathbf{e_2}\rangle \neq \langle\tilde{\mathbf{i}}^*_{\mathrm{src,max}}, \mathbf{e_2}\rangle \\ & \quad \wedge\langle\tilde{\mathbf{i}}_{\mathrm{src}^*}, \mathbf{e_3}\rangle = \langle\tilde{\mathbf{i}}^*_{\mathrm{src,max}}, \mathbf{e_3}\rangle \\ 0 & \text{otherwise} \end{cases}\right) \end{aligned}$$

$$= \left(a\left(\mathbf{i}^*_{\text{src}}\right) + \begin{cases} 4 & \text{if } \langle \mathbf{i}^*_{\text{src}}, \mathbf{e_1} \rangle = \langle \mathbf{i}^*_{\text{src,max}}, \mathbf{e_1} \rangle \\ & \wedge \langle \mathbf{i}^*_{\text{src}}, \mathbf{e_2} \rangle = \langle \mathbf{i}^*_{\text{src,max}}, \mathbf{e_2} \rangle \\ & \wedge \langle \mathbf{i}^*_{\text{src}}, \mathbf{e_3} \rangle = \langle \mathbf{i}^*_{\text{src,max}}, \mathbf{e_3} \rangle \\ 2 & \text{if } \langle \mathbf{i}^*_{\text{src}}, \mathbf{e_1} \rangle \neq \langle \mathbf{i}^*_{\text{src,max}}, \mathbf{e_1} \rangle \\ & \wedge \langle \mathbf{i}^*_{\text{src}}, \mathbf{e_2} \rangle = \langle \mathbf{i}^*_{\text{src,max}}, \mathbf{e_2} \rangle \\ & \wedge \langle \mathbf{i}^*_{\text{src}}, \mathbf{e_3} \rangle = \langle \mathbf{i}^*_{\text{src,max}}, \mathbf{e_3} \rangle \\ 1 & \text{if } \langle \mathbf{i}^*_{\text{src}}, \mathbf{e_2} \rangle \neq \langle \mathbf{i}^*_{\text{src,max}}, \mathbf{e_2} \rangle \\ & \wedge \langle \mathbf{i}^*_{\text{src}}, \mathbf{e_3} \rangle = \langle \mathbf{i}^*_{\text{src,max}}, \mathbf{e_3} \rangle \\ 0 & \text{otherwise} \end{cases} \right)$$

$$\tilde{a}\left(\text{succ}\left(\tilde{\mathbf{i}}_{\text{src}^*}\right)\right) = a\left(\text{succ}\left(\tilde{\mathbf{i}}_{\text{src}^*}\right)\right) \bmod B\,(\mathbf{C})\,.$$

Consequently, the write address calculation is possible by one adder, one multiplexer, and several comparators, instead of using several multipliers and additions required for direct evaluation of Eq. (8.6). This is particularly beneficial in case not all operands are powers of 2 as shown by the following example.

Remark 8.7 Note that although $a\left(\mathbf{i}^*_{\text{src}}\right)$ is not bounded, address calculation can be performed efficiently in hardware by directly taking the modulo-operation with $B\,(\mathbf{C})$ into account.

Example 8.8 In order to illustrate the possibility for operands not being powers of 2, let's modify Example 8.5 as follows:

$$\mathbf{p} = (2,2)^T\,,\ O_{\text{src}} = \left[(9,2)^T\right],\ \mathbf{c} = (3,1)^T\,,\ O_{\text{snk}} = \left[(6,4)^T\right],$$
$$\mathbf{i}_{\text{src,max}} = (1,8)^T\,,\ M_{\text{src}} = \begin{pmatrix} 0 & 2 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{pmatrix},\ \mathbf{m} = (6,2)^T\,.$$

Transformation of the hierarchical source iteration vector leads to

$$\mathbf{i}^*_{\text{src,max}} = (1,2,2)^T\,,\ M^*_{\text{src}} = \begin{pmatrix} 0 & 6 & 2 & 1 & 0 \\ 2 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Application of Eq. (8.8) leads to

$$A_3 = 0,\ A_2 = 1,\ A_1 = 1 \times (2+1) = 3,\ A_0 = 3 \times (1+1) = 6.$$

In other words, direct evaluation of Eq. (8.6) requires multiplications with operands not being powers of 2, which is relatively expensive in hardware. Equation (8.11) on the other hand only needs one adder, a multiplexer, and several comparators.

Remark 8.9 The above methodology for the source address calculation can only be employed if Eq. (8.9) holds. For most real-world examples, this should indeed be the case. Otherwise, concatenation of two multidimensional FIFOs as depicted in Fig. 8.8 is a valid solution as long as

$$\frac{\langle \mathbf{B}^{\text{src}}_{\mathbf{q}_{\text{src}}}, \mathbf{e_i} \rangle \times \langle \mathbf{p}, \mathbf{e_i} \rangle}{\langle \mathbf{m}, \mathbf{e_i} \rangle} \in \mathbb{N}. \tag{8.13}$$

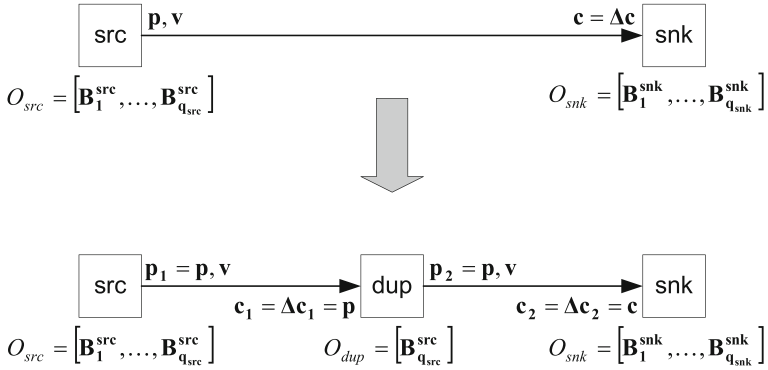


Fig. 8.8 Concatenation of two FIFOs in order to fulfill Eq. (8.9)

In other words, at least the data elements produced by the biggest firing blocks must be a multiple of the virtual memory channels. This, for instance, holds when no virtual border extension occurs. Then, the first FIFO depicted in Fig. 8.8 uses $\mathbf{p_1} = \mathbf{c_1} = \mathbf{p}$ and reorders the effective tokens generated by the source actor into raster-scan order. The second FIFO extracts the tokens required by the sink: $\mathbf{p_2} = \mathbf{p}, \mathbf{c_2} = \mathbf{c}$.

If even condition (8.13) is violated, $\langle \mathbf{m}, \mathbf{e_i} \rangle$ can be reduced by serial to parallel conversion as discussed in Section 8.2.5.

Remark 8.10 Although the factor B ($\mathbf{C}$) occurring in Eq. (8.11) is typically a power of 2, it is possible to create hardware implementations that can handle arbitrary values of B ($\mathbf{C}$). To this end, the fact can be exploited that $\Delta a\left(\mathbf{i}_{\mathrm{src}}^{*}\right)$ only has a well-defined set of possible values. Consequently, the modulo-operation can be replaced by an addition and several comparisons. Although the achievable clock frequencies are pretty large, the designer has to be aware that values of B ($\mathbf{C}$) being not a power of 2 significantly increase the required hardware resources.

8.2.4 Virtual Memory Channel Mapping

Whereas the memory partitioning scheme discussed in Section 8.2.2 can be efficiently implemented in hardware, it can lead to a huge amount of *virtual memory channels*. The shuffle FIFO, for instance, depicted in Fig. 8.1 needs 24 virtual memory channels for 8×8 blocks ($\mathbf{m} = (1, 8, 3)^T$) when 8 pixels are written in parallel as done by efficient butterfly implementations for the IDCT [54, 113]. This even increases to 48 channels if the possible read and write throughput shall be assimilated by performing the color transform for two consecutive pixels in parallel ($\mathbf{c} = (2, 1, 3)^T$, $\mathbf{m} = (2, 8, 3)^T$).

Since this is very expensive, the multidimensional FIFO is able to combine several virtual memory channels to the same physical memory when the latter offers large word widths. In Xilinx FPGAs, for instance, word widths of internal block RAMs (BRAMs) can attain up to 36 bits. Beginning with the *Virtex4* family [308], these words can be split into up to 4 bytes controlled by an individual byte write enable. These features can be used advantageously to reduce the number of required physical memories. Since all data elements belonging to the same source token are written simultaneously, they can be placed into the same byte if the

latter offers a sufficient width. Virtual memory channels belonging to different write tokens have to be placed into distinct bytes.

Such a combination is, however, only valid if it does not destroy required read parallelism on the sink side of the multidimensional FIFO. In order to illustrate the problem, Fig. 8.9 depicts a simple example. Both the source and the sink use a token size of 1×2 data elements and are supposed to traverse the token space in raster-scan order. Consequently, according to Section 8.2.2, $\mathbf{m} = (1, 2)^T$ virtual memory channels are required in order to accept one read and write operation per clock cycle.

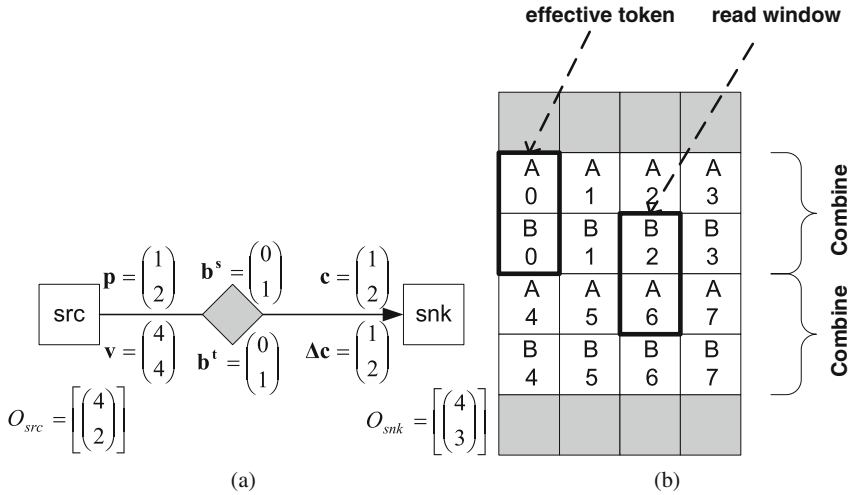


Fig. 8.9 Destruction of read parallelism. *A* and *B* define virtual memory channels. **a** WSDF graph. **b** Token space

Figure 8.9b depicts the corresponding mapping of data elements to virtual memory channels labeled by corresponding letters. From the source point of view, both virtual memory channels A and B could be combined into 1 byte, because the corresponding data elements are written simultaneously. Unfortunately such an operation destroys the required read parallelism. Considering, for instance, the window position depicted in Fig. 8.9b, the latter would require accessing two different memory addresses if the virtual memory channels were combined as indicated. Consequently, a read operation would require two clock cycles, which prohibits such an operation.

Consequently, the following sections develop a mathematical condition for valid virtual memory channel mapping.

8.2.4.1 Strategy for Virtual Memory Channel Mapping

Given the number of virtual memory channels $\prod_{i=1}^{n} \langle \mathbf{m}, \mathbf{e_i} \rangle$, arbitrarily complex memory sharing schemes could be developed in order to avoid destruction of the required read parallelism. However, in order to keep the memory interconnect structure simple, a simple two-staged strategy shall be discussed in this book. In the first stage, it checks which virtual memory channels can be placed into the same *logical memory byte* or *word* without destroying the required read parallelism. This operation is performed without considering the parameters

of the available physical memory modules. Instead, it assumes the existence of a memory module whose byte and word widths can be adjusted as required. Furthermore, each byte shall be controllable by a corresponding byte write enable signal. A second step then maps the logical bytes and words built in the previous step to the actual memory architecture.

Figure 8.10 graphically depicts the first step of the virtual memory channel mapping, assuming $\mathbf{m} = (8, 8)^T$. It investigates whether it is possible to combine α_i virtual memory channels in dimension $\mathbf{e_i}$ into the same *logical memory byte* or *word* without destroying the read parallelism. In order to obtain regular memory structures, only those values of α_i are accepted for which the following holds:

$$\frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\alpha_i} \in \mathbb{N}.$$

In case of Fig. 8.10, this leads to

$$\alpha_{1,2} \in \{1, 2, 4, 8\}.$$

Furthermore, all virtual memory channels $\mathbf{C}$ combined to one logical byte or word must share the same value $B(\mathbf{C})$. From these possible values for α_i, the biggest one not destroying the required read parallelism is selected as discussed in Section 8.2.4.2, leading to a required logical byte and word width.

Figure 8.10 shows the combination of virtual memory channels into logical bytes and words, assuming the following values:

$$\begin{aligned} \mathbf{p} &= (4, 4)^T, \\ \mathbf{m} &= (8, 8)^T, \\ \alpha_1 &= 2, \\ \alpha_2 &= 8. \end{aligned}$$

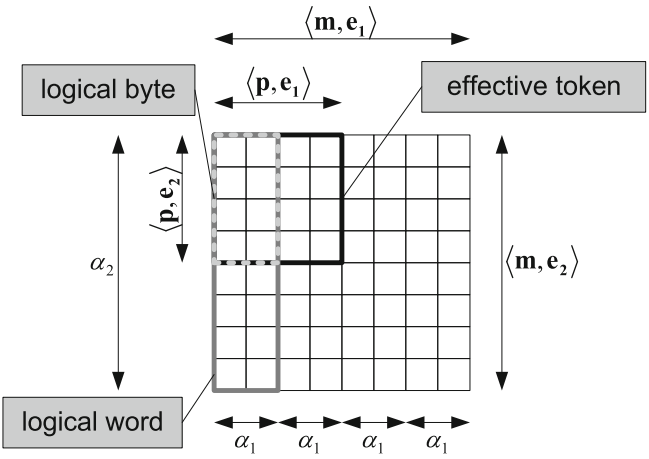


Fig. 8.10 Combination of virtual memory channels into logical bytes and words

The logical word thus contains $\alpha_1 \times \alpha_2$ virtual memory channels. All those that are situated in the intersection between the effective token and the logical word are assigned to one logical byte, which can be controlled by a corresponding byte write enable signal. This guarantees that a write operation only changes the affected virtual memory channels. Thus, the logical memory word depicted in Fig. 8.10 consists of two logical bytes. Altogether, four logical memory words are required.

Remark 8.11 The above simple approach only works if

$$\frac{\alpha_i}{\langle \mathbf{p}, \mathbf{e_i} \rangle} \in \mathbb{N} \vee \frac{\langle \mathbf{p}, \mathbf{e_i} \rangle}{\alpha_i} \in \mathbb{N}. \tag{8.14}$$

Otherwise, the virtual memory channels have to be mapped to individual bytes due to symmetry reasons. As the extension is straightforward, further details are omitted.

Since the above proceeding typically leads to logical byte and word widths larger than those provided by the selected physical memory, the second stage of the virtual channel mapping consists in combining several physical bytes provided by the memory to one unit to obtain the required logical byte width. Furthermore, the byte write enable signals have to be connected accordingly.

After termination of the second stage, it is thus known how many physical memories are required in order to permit parallel data access during read and write operations. Whereas on-chip storage typically provides a sufficient number of memory modules, printed circuit boards often offer only a limited number of available memory chips. This aspect will be addressed in Section 8.2.5. First, however, the following section will derive a mathematical condition that allows to detect destruction of the required read parallelism.

Remark 8.12 The above approach is voluntarily kept simple in order to permit direct implementation in VHDL. However, it might result in sub-optimal solutions, which could be avoided by more complex memory mapping strategies like those presented in [225, 310]. This, however, is out of scope for the present monograph.

8.2.4.2 Condition for Valid Virtual Memory Channel Mapping

After having introduced the principle strategy for virtual memory channel mapping, this section derives a mathematical condition in order to check whether or not the required read parallelism is destroyed.

Lemma 8.13 *Combination of α_i virtual memory channels into one physical is permitted if the following does not hold*

$$0 \leq \langle \mathbf{m}, \mathbf{e_i} \rangle - \alpha_i + 1 + \zeta_{\min} < \langle \mathbf{c}, \mathbf{e_i} \rangle,$$

where

$$\zeta_{\min} = \min_{\substack{0 \leq \beta < \frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\langle \mathbf{c}, \mathbf{e_i} \rangle} \\ 0 \leq j < \frac{\mathrm{scm}(\langle \mathbf{v}, \mathbf{e_i} \rangle, \alpha_i)}{\langle \mathbf{v}, \mathbf{e_i} \rangle}}} \zeta(\beta, j),$$

with

$$\zeta(\beta, j) = \begin{cases} 0 & \text{if } C1 \\ \alpha_i - \left(\langle j \times \mathbf{v} - \mathbf{b^s} - \boldsymbol{\delta} + \beta \times \mathbf{c}, \mathbf{e_i} \rangle + 1\right) \bmod \alpha_i & \text{otherwise} \end{cases}$$

$$C1 : \left(\langle j \times \mathbf{v} - \mathbf{b^s} - \boldsymbol{\delta} + \beta \times \mathbf{c}, \mathbf{e_i} \rangle + 1\right) \bmod \alpha_i = 0.$$

Proof Consider again Fig. 8.10. As can be seen, the data elements are mapped to $\frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\alpha_i}$ logical memory channels. Consequently, given the coordinates $\mathbf{x}$ of a data element from the source point of view, the following equation calculates the memory ID where it is placed to:

$$\left\lfloor \frac{\langle \mathbf{x}, \mathbf{e_i} \rangle}{\alpha_i} \right\rfloor \bmod \frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\alpha_i}.$$

Destruction of read memory parallelism occurs if the sink accesses two data elements that result in identical memory IDs but that require different read addresses. From Section 8.2.3, it is known that a hierarchical source iteration vector component $\langle \mathbf{i}_{\text{src}}, \mathbf{e_i} \rangle$ contributes either to the write address or to selection of a virtual memory channel. Thus, the first $\langle \mathbf{m}, \mathbf{e_i} \rangle$ data elements in dimension $\mathbf{e_i}$ are associated with the same write address. Consequently, destruction of read memory parallelism occurs if

$$\left\lfloor \frac{\langle \mathbf{x}_{\text{snk}} + \Delta\mathbf{x}_{\text{snk},\mathbf{1}}, \mathbf{e_i} \rangle}{\alpha_i} \right\rfloor \bmod \frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\alpha_i} = \left\lfloor \frac{\langle \mathbf{x}_{\text{snk}} + \Delta\mathbf{x}_{\text{snk},\mathbf{2}}, \mathbf{e_i} \rangle}{\alpha_i} \right\rfloor \bmod \frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\alpha_i}$$
$$\left\lfloor \frac{\langle \mathbf{x}_{\text{snk}} + \Delta\mathbf{x}_{\text{snk},\mathbf{1}}, \mathbf{e_i} \rangle}{\alpha_i} \right\rfloor \neq \left\lfloor \frac{\langle \mathbf{x}_{\text{snk}} + \Delta\mathbf{x}_{\text{snk},\mathbf{2}}, \mathbf{e_i} \rangle}{\alpha_i} \right\rfloor.$$

$\mathbf{x}_{\text{snk}}$ defines the "upper left" data element of an arbitrary read operation. More precisely, given an arbitrary read window, $\mathbf{x}_{\text{snk}}$ is that data element with the smallest coordinates in all dimensions $1 \leq i \leq n$. $0 \leq \Delta\mathbf{x}_{\text{snk},\mathbf{1}} < \mathbf{c}$, $0 \leq \mathbf{x}_{\text{snk},\mathbf{2}} < \mathbf{c}$ take into account that a read window consists of several data elements.[2]

This can be transformed into the following equivalent condition:

$$\left\lfloor \frac{\langle \mathbf{x}_{\text{snk}} + \Delta\mathbf{x}_{\text{snk},\mathbf{2}}, \mathbf{e_i} \rangle}{\alpha_i} \right\rfloor = \left\lfloor \frac{\langle \mathbf{x}_{\text{snk}} + \Delta\mathbf{x}_{\text{snk},\mathbf{1}}, \mathbf{e_i} \rangle}{\alpha_i} \right\rfloor + k \times \frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\alpha_i}.$$
$$k \neq 0$$

The worst case occurs for $k = 1$ and $\Delta\mathbf{x}_{\text{snk},\mathbf{1}} = \mathbf{0}$:

$$\left\lfloor \frac{\langle \mathbf{x}_{\text{snk}} + \Delta\mathbf{x}_{\text{snk},\mathbf{2}}, \mathbf{e_i} \rangle}{\alpha_i} \right\rfloor = \left\lfloor \frac{\langle \mathbf{x}_{\text{snk}}, \mathbf{e_i} \rangle}{\alpha_i} \right\rfloor + \frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\alpha_i}. \tag{8.15}$$

Furthermore, since the memory partitioning defined in Section 8.2.2 is performed relatively to the produced source tokens ignoring virtual border extension and initial data

[2] In order to ease analysis, a pessimistic approximation is performed by omitting virtual border extension.

elements, $\langle \mathbf{x}_{\text{snk}}, \mathbf{e_i} \rangle$ can be calculated for the $(\beta + 1)$th read operation in the $(j + 1)$th virtual token as

$$\langle \mathbf{x}_{\text{snk}}, \mathbf{e_i} \rangle = j \times \langle \mathbf{v}, \mathbf{e_i} \rangle + \langle -\mathbf{b^S} - \boldsymbol{\delta}, \mathbf{e_i} \rangle + \langle \Delta\mathbf{c}, \mathbf{e_i} \rangle \times \beta.$$

With $\Delta\mathbf{c} = \mathbf{c}$ (see Section 8.1), Eq. (8.15) becomes

$$\begin{aligned} &\left\lfloor \frac{j \times \langle \mathbf{v}, \mathbf{e_i} \rangle + \langle -\mathbf{b^S} - \boldsymbol{\delta}, \mathbf{e_i} \rangle + \langle \mathbf{c}, \mathbf{e_i} \rangle \times \beta + \langle \Delta\mathbf{x}_{\mathbf{snk,2}}, \mathbf{e_i} \rangle}{\alpha_i} \right\rfloor \\ &= \left\lfloor \frac{j \times \langle \mathbf{v}, \mathbf{e_i} \rangle + \langle -\mathbf{b^S} - \boldsymbol{\delta}, \mathbf{e_i} \rangle + \langle \mathbf{c}, \mathbf{e_i} \rangle \times \beta}{\alpha_i} \right\rfloor + \frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\alpha_i}. \end{aligned} \tag{8.16}$$

As

$$\frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\langle \mathbf{c}, \mathbf{e_i} \rangle} \in \mathbb{N},$$

the above condition has only to be checked for

$$0 \leq \beta < \frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\langle \mathbf{c}, \mathbf{e_i} \rangle}.$$

Similarly,

$$0 \leq j < \frac{\text{scm}\left(\langle \mathbf{v}, \mathbf{e_i} \rangle, \alpha_i\right)}{\langle \mathbf{v}, \mathbf{e_i} \rangle}.$$

Thus, for each virtual token j, a worst-case β can be derived by minimizing ζ ($\alpha_i > \zeta \geq 0$, $\zeta \in \mathbb{N}_0$):

$$j \times \langle \mathbf{v}, \mathbf{e_i} \rangle + \langle -\mathbf{b^S} - \boldsymbol{\delta}, \mathbf{e_i} \rangle + \beta \times \langle \mathbf{c}, \mathbf{e_i} \rangle + \zeta = k \times \alpha_i - 1. \tag{8.17}$$

This leads to the following condition for k:

$$k = \left\lceil \frac{j \times \langle \mathbf{v}, \mathbf{e_i} \rangle + \langle -\mathbf{b^S} - \boldsymbol{\delta}, \mathbf{e_i} \rangle + \beta \times \langle \mathbf{c}, \mathbf{e_i} \rangle + 1}{\alpha_i} \right\rceil.$$

Hence

$$\begin{aligned} \zeta = &\left\lceil \frac{j \times \langle \mathbf{v}, \mathbf{e_i} \rangle + \langle -\mathbf{b^S} - \boldsymbol{\delta}, \mathbf{e_i} \rangle + \beta \times \langle \mathbf{c}, \mathbf{e_i} \rangle + 1}{\alpha_i} \right\rceil \times \alpha_i \\ &-1 - \left(j \times \langle \mathbf{v}, \mathbf{e_i} \rangle + \langle -\mathbf{b^S} - \boldsymbol{\delta}, \mathbf{e_i} \rangle + \beta \times \langle \mathbf{c}, \mathbf{e_i} \rangle\right). \end{aligned}$$

Since

$$\left\lceil \frac{x}{\alpha} \right\rceil \times \alpha = \begin{cases} x & \text{if } x \bmod \alpha = 0 \\ x + \alpha - x \bmod \alpha & \text{otherwise} \end{cases},$$

this leads to

$$\zeta = \begin{cases} 0 & \text{if } C1 \\ \alpha_i - \left(j \times \langle \mathbf{v}, \mathbf{e_i} \rangle + \langle -\mathbf{b^s} - \boldsymbol{\delta}, \mathbf{e_i} \rangle + \beta \times \langle \mathbf{c}, \mathbf{e_i} \rangle + 1\right) \bmod \alpha_i & \text{otherwise} \end{cases}$$

$$C1 : \left(j \times \langle \mathbf{v}, \mathbf{e_i} \rangle + \langle -\mathbf{b^s} - \boldsymbol{\delta}, \mathbf{e_i} \rangle + \beta \times \langle \mathbf{c}, \mathbf{e_i} \rangle + 1\right) \bmod \alpha_i = 0.$$

As both $\frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\langle \mathbf{c}, \mathbf{e_i} \rangle}$ and $\frac{\text{scm}(\langle \mathbf{v}, \mathbf{e_i} \rangle, \langle \mathbf{m}, \mathbf{e_i} \rangle)}{\langle \mathbf{v}, \mathbf{e_i} \rangle}$ are typically small, the minimum value of $\zeta_{\min}$ can be obtained by exhaustive search over j and β.

Insertion of Eq. (8.17) in (8.16) leads to

$$\begin{aligned} &\left\lfloor \frac{k \times \alpha_i - 1 - \zeta_{\min} + \langle \Delta \mathbf{x}_{\text{snk},\mathbf{2}}, \mathbf{e_i} \rangle}{\alpha_i} \right\rfloor = \left\lfloor \frac{k \times \alpha_i - 1 - \zeta_{\min}}{\alpha_i} \right\rfloor + \frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\alpha_i} \\ \Leftrightarrow &\left\lfloor \frac{\alpha_i - 1 - \zeta_{\min} + \langle \Delta \mathbf{x}_{\text{snk},\mathbf{2}}, \mathbf{e_i} \rangle}{\alpha_i} \right\rfloor = \left\lfloor \frac{\alpha_i - 1 - \zeta_{\min}}{\alpha_i} \right\rfloor + \frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\alpha_i} \\ \Leftrightarrow &\left\lfloor \frac{\alpha_i - 1 - \zeta_{\min} + \langle \Delta \mathbf{x}_{\text{snk},\mathbf{2}}, \mathbf{e_i} \rangle}{\alpha_i} \right\rfloor = \frac{\langle \mathbf{m}, \mathbf{e_i} \rangle}{\alpha_i} \\ \Leftarrow &\ \alpha_i - 1 - \zeta_{\min} + \langle \Delta \mathbf{x}_{\text{snk},\mathbf{2}}, \mathbf{e_i} \rangle = \langle \mathbf{m}, \mathbf{e_i} \rangle \\ \Leftrightarrow &\ \langle \Delta \mathbf{x}_{\text{snk},\mathbf{2}}, \mathbf{e_i} \rangle = \langle \mathbf{m}, \mathbf{e_i} \rangle - \alpha_i + 1 + \zeta_{\min}. \end{aligned}$$

Example 8.14 In order to explain the principle of the above lemma, consider the example given in Fig. 8.9

$$\begin{aligned} \mathbf{p} &= (1, 2)^T, \\ \mathbf{c} &= (1, 2)^T, \\ \mathbf{b^s} &= (0, 1)^T, \\ \mathbf{m} &= (1, 2)^T, \\ \alpha &= (\alpha_1, \alpha_2)^T = (1, 2)^T. \end{aligned}$$

This leads to

$$\boldsymbol{\zeta}_{\min} = (0, 0)^T.$$

Consequently

$$\begin{aligned} &\mathbf{m} - \boldsymbol{\alpha} + \mathbf{1} + \boldsymbol{\zeta}_{\min} \\ &= \begin{pmatrix} 1 \\ 2 \end{pmatrix} - \begin{pmatrix} 1 \\ 2 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \not< \mathbf{c}. \end{aligned}$$

Thus, according to Lemma 8.13, this virtual channel mapping is not valid, which confirms the observation of Section 8.2.4.

8.2.4.3 Application to the JPEG Shuffle Operation

In order to illustrate the benefits of the virtual channel mapping, consider the shuffle operation assuming 8×8 blocks of 12-bit data elements where the color transform shall generate two consecutive pixels in parallel for a more balanced relation between possible read and write throughput:

$$\begin{aligned} \mathbf{p} &= (1, 8, 1)^T, \\ \mathbf{c} &= (2, 1, 3)^T, \\ \mathbf{m} &= (2, 8, 3)^T. \end{aligned}$$

For the memory mapping, we assume identical sizes for all virtual memory channels and choose

$$\boldsymbol{\alpha} = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 8 \\ 3 \end{pmatrix}.$$

This is permitted because

$$\boldsymbol{\zeta}_{\min} = (1, 0, 2).$$

Thus,

$$\begin{aligned} &\mathbf{m} - \alpha + \mathbf{1} + \boldsymbol{\zeta}_{\min} \\ &= \begin{pmatrix} 2 \\ 8 \\ 3 \end{pmatrix} - \begin{pmatrix} 2 \\ 8 \\ 3 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix} \geq \mathbf{c}. \end{aligned}$$

A corresponding analysis shows that each of the $2 \times 8 \times 3 = 48$ virtual memory channels needs to have a capacity of 256 data elements for correct operation when using QCIF images. In this case, a Xilinx BRAM can be configured to consist of 4 bytes with 9 bits each. Hence, the 8 data elements corresponding to $8 \times 12 = 96$ bits, which are written in parallel, can be placed into 11 bytes. As a consequence, the 48 virtual memory channels can be mapped to $11 \times 2 \times 3 = 66$ bytes corresponding to 17 physical memories. In other words, the number of required physical memories can be reduced by 65%.

8.2.5 Trading Throughput Against Resource Requirements

Considering, for instance, the JPEG shuffle operation, the virtual channel mapping introduced in Section 8.2.4 can significantly reduce the amount of physical memories. However, their resulting number might still be too large, especially if external memories shall be used. The reasons can be found in the IDCT as many efficient implementations such as those proposed in [54, 113] read and write 8 data elements in parallel. On the other hand, the capacity to process 8 data elements in parallel is completely useless, if the application requirements can be satisfied by a throughput of 1 pixel per clock cycle, each consisting of three color components.

This is the reason for introducing parallel-to-serial and serial-to-parallel converters in Fig. 8.4. Instead of writing a complete column of data elements, the latter one can be split into two sequentially accessed parts having 4 data elements each. Although this results in an access time of two clock cycles per block column, the overall throughput is still sufficient to process 1 pixel per clock cycle. The number of required BRAMs for the shuffle FIFO, however, reduces to only 9, thus leading to savings of 47%.

8.2.6 Sink Address Generation

Section 8.2.3 has presented an efficient method for generation of the write addresses. This is possible thanks to linearization in production order together with the splitting of the hierarchical iteration vector into two different sets of components either contributing to the write address or selecting the virtual memory channel to use. Furthermore, incremental address generation permits to avoid expensive arithmetic operations like multiplications or multiple additions.

A similar strategy shall also be employed in order to calculate the read address for each virtual memory channel $\mathbf{C}$

$$\begin{aligned} \tilde{a}\left(\mathbf{i}_{\mathrm{snk}}, \mathbf{I}_{\mathrm{snk,w}}\right) &= a\left(\mathbf{i}_{\mathrm{snk}}, \mathbf{I}_{\mathrm{snk,w}}\right) \bmod B\left(\mathbf{C}\right), \\ a\left(\mathrm{succ}\left(\mathbf{i}_{\mathrm{snk}}\right), \mathbf{I}_{\mathrm{snk,w}}\right) &= a\left(\mathbf{i}_{\mathrm{snk}}, \mathbf{I}_{\mathrm{snk,w}}\right) + \Delta a\left(\mathbf{i}_{\mathrm{snk}}, \mathbf{I}_{\mathrm{snk,w}}\right). \end{aligned} \tag{8.18}$$

In analogy to Section 8.2.3, $a\left(\mathbf{i}_{\mathrm{snk}}, \mathbf{I}_{\mathrm{snk,w}}\right)$ is derived from the previous read operation in order to avoid expensive arithmetic operations. The final read address $\tilde{a}\left(\mathbf{i}_{\mathrm{snk}}, \mathbf{I}_{\mathrm{snk,w}}\right)$ is built by means of a modulo-operation that takes the available size of the virtual memory channel $\mathbf{C}$ into account. However, whereas the linear part of the write address has been constant for all data elements of an effective token, this is not true anymore for the read windows as exemplified in Fig. 8.9. There, the depicted read window reads from both addresses $a\left(\mathbf{i}_{\mathrm{snk}}, \mathbf{I}_{\mathrm{snk,w}}\right) = 2$ and $a\left(\mathbf{i}_{\mathrm{snk}}, \mathbf{I}_{\mathrm{snk,w}}\right) = 6$. Consequently, $a\left(\mathbf{i}_{\mathrm{snk}}, \mathbf{I}_{\mathrm{snk,w}}\right)$ depends not only on the read operation $\mathbf{i}_{\mathrm{snk}}$ but also on the position $\mathbf{I}_{\mathrm{snk,w}}$ of the data element in the inner of the sliding window. In such cases, several read address generation units have to be instantiated as depicted in Fig. 8.4. Furthermore, out-of-order communication complicates determination of $a\left(\mathbf{i}_{\mathrm{snk}}, \mathbf{I}_{\mathrm{snk,w}}\right)$. Due to the linearization in production order, it is first necessary to calculate the producing write operation, from which the memory address can be derived using Eq. (8.6).

In order to solve this task, several possibilities exist. One strategy consists in establishing a closed form expression that maps the considered read operation $\mathbf{i}_{\mathrm{snk}}$ and data element position $\mathbf{I}_{\mathrm{snk,w}}$ within the read window to the write operation $\mathbf{i}_{\mathrm{src}}$ that produces the required data element:

$$\tilde{P} : \left(\mathbf{i}_{\mathrm{snk}}, \mathbf{I}_{\mathrm{snk,w}}\right) \mapsto \mathbf{i}_{\mathrm{src}}.$$

Such a function can be established using the data element mapping relation given in Eq. (7.10). Although the latter is linear, its solution is not easy and requires in general solution of a Parametric Integer Program [104] using, for instance, the PIP library [5]. As the latter returns the solution in a very complex form containing several modulo-functions, it has to be transformed into piecewise linear functions (which is possible because $0 \leq \mathbf{i}_{\mathrm{snk}} \leq \mathbf{i}_{\mathrm{snk,max}}$). The resulting piecewise linear function $\tilde{P}\left(\mathbf{i}_{\mathrm{snk}}, \mathbf{I}_{\mathrm{snk,w}}\right)$ can then be inserted into Eq. (8.6)

in order to calculate the required read address. Similar to Section 8.2.3, the knowledge about the lexicographic increment of $\mathbf{i}_{\text{snk}}$ can then be used to derive the address increment $\Delta a\left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right) = a\left(\text{succ}\left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right)\right) - a\left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right)$ in order to achieve efficient hardware implementation. Since, however, the PIP library often tends to produce complex solutions, this might lead to very huge formulas for $\Delta a\left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right)$.

Alternatively, the address increment $\Delta a\left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right)$ can be directly derived via a table-based approach. To this end, consider, for instance, the tiling operation given in Fig. 8.1. Due to the linearization in production order, the address of a data element simply corresponds to the number of the producing write operations, which is represented by Arabic numerals in Fig. 8.1. By following the sink invocations in the order indicated by the dashed arrows, the address increment $\Delta a\left(\mathbf{i}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right)$ can be easily obtained. For the given example, the address increment simply amounts one, as long as we stay in the same line of the same tile. If we move to the next line in the same tile, the address is increased by six. Moving to the next tile in horizontal direction means an address decrement of 49 and so on.

As, however, very huge images shall be processed, it is not possible to just synthesize a lookup table that associates to each sink iteration vector the corresponding address offset, as this would be extremely expensive in terms of hardware resources and synthesis time. Instead, as many identical address offsets as possible have to be grouped together.

Figure 8.11a shows the pseudocode for the corresponding algorithm generating nested if-then-else statements expressing the correct address increment. It is started with $j = 1$ and obtains a table $T\left(\mathbf{i}_{\text{snk}}\right)$ that assigns to each sink invocation $\mathbf{i}_{\text{snk}}$ the corresponding address increment required in order to calculate the data element address of the next read operation. Based on this table, the algorithm groups identical address increment values in order to obtain a compact representation in form of nested conditionals. For this purpose, line (05) checks whether there exist two table entries that only differ in coordinate j and that do not have the

```
(00) create_cond(j,z){
(01)  if j = n_snk+1
(02)   →addr_inc := T(z)
(03)  else
(04)   for ⟨z,e_j⟩=0 :⟨i_snk,max,e_j⟩−1
(05)    if { ∃k1,k2 ∈ I_snk(j,z),
             ⟨k1,e_j⟩ = ⟨z,e_j⟩,
               k2 = k1 + e_j :
             T(k1) ≠ T(k2) }
(06)     →IF i_snk[j]<=⟨z,e_j⟩
(07)     create_cond(j+1,z);
(08)     →ELSE
(09)    end if
(10)   end for
(11)   ⟨z,e_j⟩ = ⟨i_snk,max,e_j⟩;
(12)   create_cond(j+1,z);
(13)  end if
(14) }
```

(a)

```
IF i_snk[2]<=0
 IF i_snk[3]<=4
  IF i_snk[4]<=3
   addr_inc := 1
  ELSE
   addr_inc := 6
 ELSIF i_snk[4]<=3
  addr_inc := 1
 ELSE
  addr_inc := -49
ELSIF i_snk[3]<=4
 IF i_snk[4]<=3
  addr_inc := 1
 ELSE
  addr_inc := 6
ELSE
 addr_inc := 1
```

(b)

Fig. 8.11 **a** Coding of the address offsets by nested conditionals. The resulting code output is indicated by the "→"-sign. $I_{\text{snk}}(j, \mathbf{z}) = \{0 \leq \mathbf{i}_{\text{snk}} \leq \mathbf{i}_{\text{snk,max}} \mid \forall 1 \leq k < j : \langle \mathbf{i}_{\text{snk}} - \mathbf{z}, \mathbf{e}_{\mathbf{k}} \rangle = 0\}$. **b** The (reformatted) code generated by the algorithm for the example given in Fig. 8.3. *Else*-statements immediately followed by an *if*-statement are replaced by *elsif*-constructs

same value. If this is the case, a corresponding distinction in form of a conditional has to be introduced. The latter is generated in lines (06) and (08), whereas line (02) outputs the assignment of the result variable *addr_inc*.

In part b of Fig. 8.11, the code generated by the above algorithm for the tiling example in Fig. 8.3 is printed. As can be seen, the number of required if-statements is much smaller than the number of pixels forming the image. They can hence be efficiently synthesized in hardware. In case the read window consists of several data elements reading from different addresses, a corresponding address generation unit has to be synthesized for each of them using the algorithm given in Fig. 8.11a.

The next section shows, how based on the linearized memory model, efficient flow control can be realized. In other words, it solves the question when the source can write an effective token or when the sink can read a (sliding) window.

8.2.7 Fill-Level Control

Hardware implementations typically employ self-timed scheduling. In other words, a module verifies all input FIFOs whether the required input data are available. Furthermore, the output FIFOs have to be checked whether they can accept the expected result values. If only one of these conditions is not fulfilled, then the module is stalled. Since this implementation strategy has been proven efficient in many designs, the multidimensional FIFO elaborated in this chapter shall support the same operation semantics. To this end, it is necessary to provide a fill-level indication telling how many windows can be read by the sink and how many effective tokens can be written by the source.

Whereas for one-dimensional FIFOs, the fill-level control is rather easy to implement, out-of-order communication makes this task more challenging. Consider, for instance, once again the tiling operation illustrated in Fig. 8.3. Then, it can be easily seen that each of the first five source invocations immediately permits the sink to execute once. The source invocations 5–9, however, do not allow the sink to continue, because due to out-of-order communication, the latter requires the data element produced by the source invocation 10. On the other hand, once the sink has processed pixel 54, it can immediately continue with pixels 5–49 without waiting for the source, because they are already available.

A similar reasoning is valid for freeing buffer elements and hence for the question, whether the source can still execute or has to wait due to a full buffer. Because of the linearization in production order, pixels stored in the buffer can only be freed in the same order in which they have been produced. In other words, in Fig. 8.3 it is not possible to discard data elements 10–14 before 5–9 in order to avoid holes in the address space, which would be too complex to handle in hardware. However, this also means that no buffer elements can be freed, when the sink processes the pixels 10–14, 20–24, 30–34, 40–44, and 50–54. On the other hand, when discarding pixel 9, also pixels 10–14 can be freed because they have already been processed.

This example clearly shows that in contrast to one-dimensional FIFOs, it is not sufficient anymore to count the tokens stored in the buffer in order to derive whether the source or the sink can execute. Consequently, in the architecture shown in Fig. 8.4, both the source and the sink fill-level control modules contain their own counters indicating the number of possible write, respectively, read operations. Both are initialized with the correct values during startup. If neither initial tokens nor virtual border extension occur, the sink counter is set to zero, whereas the source counter depends on the buffer size. Each time the source

writes an effective token or the sink reads a (sliding) window, the corresponding counter is decreased by one. Additionally, whenever the source executes, it communicates the number of additional sink invocations to the sink-level fill control and vice versa.

8.2.7.1 Sink Fill-Level Control

In order to determine the possible amount of read operations, the multidimensional FIFO depicted in Fig. 8.4 has to calculate how many additional possible sliding windows $\Delta\text{snk}(\tilde{\mathbf{i}}_{\text{src}})$ become available after termination of the write operation $\tilde{\mathbf{i}}_{\text{src}}$. This question can be answered by means of a *Parametric Integer Program* (*PIP*) [104] that minimizes a system of linear inequalities depending on one or more parameters. In other words, the linear inequalities not only contain the unknowns that have to be selected in such a way that the objective function gets minimal but also depend on parameter values whose possible range is specified by the so-called *context* [104]. Consequently, the solution of a parametric integer program consists not only of a single number defining the optimal objective value but also of a function that maps to each parameter value the corresponding optimal objective. This function can be expressed by means of nested if-then-else conditionals leading to a piecewise affine form. The minimization itself is performed in the sense of the lexicographic order $\prec$ (see Definition 6.4).

This mathematical tool can be used for calculating $\Delta\text{snk}(\tilde{\mathbf{i}}_{\text{src}})$ by considering the current write operation $0 \leq \tilde{\mathbf{i}}_{\text{src},\mathbf{0}} \leq \tilde{\mathbf{i}}_{\text{src,max}}$ as a parameter. Then the following PIP searches the first read operation that requires a data element produced after $\tilde{\mathbf{i}}_{\text{src},\mathbf{0}}$:

$$\min_{\prec}\left(\tilde{\mathbf{i}}_{\text{snk}}\right) \tag{8.19}$$

$$\tilde{\mathbf{i}}_{\text{src,max}} \geq \tilde{\mathbf{i}}_{\text{src}} \geq \mathbf{0} \tag{8.20}$$

$$\tilde{\mathbf{i}}_{\text{snk,max}} \geq \tilde{\mathbf{i}}_{\text{snk}} \geq \mathbf{0} \tag{8.21}$$

$$\mathbf{p} > \mathbf{I}_{\text{src,w}} \geq \mathbf{0} \tag{8.22}$$

$$\mathbf{c} > \mathbf{I}_{\text{snk,w}} \geq \mathbf{0} \tag{8.23}$$

$$\underbrace{\tilde{M}_{\text{src}} \times \left(\tilde{\mathbf{i}}_{\text{src}}, \mathbf{I}_{\text{src,w}}\right) + \boldsymbol{\delta}}_{(a)} - \underbrace{\left(\tilde{M}_{\text{snk}} \times \left(\tilde{\mathbf{i}}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right) + \mathbf{b}^{\mathbf{s}}\right)}_{(b)} = 0 \tag{8.24}$$

$$\tilde{\mathbf{i}}_{\text{src}} \succ \tilde{\mathbf{i}}_{\text{src},0} \tag{8.25}$$

$$\tilde{C}_{\text{snk}} \times \left(\tilde{\mathbf{i}}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right) - \mathbf{b}^{\mathbf{l}}_{\mathbf{c}} \geq \mathbf{0} \tag{8.26}$$

$$\tilde{C}_{\text{snk}} \times \left(\tilde{\mathbf{i}}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right) - \mathbf{b}^{\mathbf{h}}_{\mathbf{c}} \leq \mathbf{0} \tag{8.27}$$

$$\tilde{\mathbf{i}}_{\text{src,max}} \geq \tilde{\mathbf{i}}_{\text{src},0} \geq \mathbf{0}. \tag{8.28}$$

Equation (8.28) forms the *PIP context* and specifies the possible range of the PIP parameter $\tilde{\mathbf{i}}_{\text{src},\mathbf{0}}$. In other words, the PIP solver does not search the value of $\tilde{\mathbf{i}}_{\text{src},\mathbf{0}}$ leading to the lexicographically smallest objective $\min_{\prec}(\tilde{\mathbf{i}}_{\text{snk}})$. Instead, it returns a function that associates to each possible value of $\tilde{\mathbf{i}}_{\text{src},0}$ the corresponding lexicographic minimum for $\tilde{\mathbf{i}}_{\text{snk}}$. To this end, the PIP solver searches for those unknowns $\tilde{\mathbf{i}}_{\text{snk}}$, $\tilde{\mathbf{i}}_{\text{src}}$, $\mathbf{I}_{\text{src,w}}$ and $\mathbf{I}_{\text{snk,w}}$ that lead to the lexicographically smallest value for the objective function without violating Eqs. (8.19), (8.20), (8.21), (8.22), (8.23), (8.24), (8.25), (8.26), and (8.27).

Equation (8.24) describes the data element mapping. Part (a) calculates the coordinates of the pixel produced by the write operation $\tilde{\mathbf{i}}_{\text{src}}$, part (b) the pixel coordinates accessed by the read operation $\tilde{\mathbf{i}}_{\text{snk}}$. $\tilde{M}_{\text{src}}$ and $\tilde{M}_{\text{snk}}$ are extended mapping matrices including the (local) schedule period (see Section 8.2.1.2). $\mathbf{I}_{\text{src,w}}$ corresponds to the data element in the inner of the effective token. Analogously $\mathbf{I}_{\text{snk,w}}$ defines the data element in the inner of the sliding window. The range of both vectors is given in Eqs. (8.22) and (8.23). Together, parts (a) and (b) of Eq. (8.24) establish a relation between the read operation $\tilde{\mathbf{i}}_{\text{snk}}$ and the corresponding write operation $\tilde{\mathbf{i}}_{\text{src}}$ producing the required data element, while Eqs. (8.26) and (8.27) exclude those data elements situated on the extended border (see also Section 7.3.4). Equations (8.20) and (8.21) specify the possible range of the hierarchical iteration vectors. Equation (8.25) finally takes care that only write operations that do occur after $\tilde{\mathbf{i}}_{\text{src},\mathbf{0}}$ are taken into account (see also Section 6.2).

The overall system of inequalities thus searches the earliest read operation (Eq. (8.19)) requiring a data element (Eq. (8.24)) that is not already available because it will be produced after the current write operation $\tilde{\mathbf{i}}_{\text{src},\mathbf{0}}$ (Eq. (8.25)). In other words, let $\tilde{\mathbf{i}}_{\text{snk}} = f_{\text{snk}}(\tilde{\mathbf{i}}_{\text{src},\mathbf{0}})$ be the solution of the above PIP. Then, after termination of the write operation $\tilde{\mathbf{i}}_{\text{src},\mathbf{0}}$, $f_{\text{snk}}(\tilde{\mathbf{i}}_{\text{src},\mathbf{0}})$ is the first read operation that cannot be executed anymore. Thus, the number of additional read operations possible due to termination of $\tilde{\mathbf{i}}_{\text{src},\mathbf{0}}$ amounts

$$\Delta\text{snk}\left(\tilde{\mathbf{i}}_{\text{src},\mathbf{0}}\right) = Q_{\text{snk}}\left(f\left(\tilde{\mathbf{i}}_{\text{src},\mathbf{0}}\right)\right) - Q_{\text{snk}}\left(f\left(\text{pred}\left(\tilde{\mathbf{i}}_{\text{src},\mathbf{0}}\right)\right)\right). \tag{8.29}$$

$\text{pred}(\tilde{\mathbf{i}}_{\text{src},\mathbf{0}})$ returns the predecessor value of $\tilde{\mathbf{i}}_{\text{src},0}$ by performing the inverse operation of the lexicographical increment defined in Eq. (8.1). Q_{snk} is a function that enumerates all read operations:

$$Q_{\text{snk}}\left(\tilde{\mathbf{i}}_{\text{snk}}\right) = \sum_{i=0}^{n_{\text{snk}}} \left(\langle \tilde{\mathbf{i}}_{\text{snk}}, \mathbf{e_i} \rangle \times \prod_{j=i+1}^{n_{\text{snk}}} \left(\langle \tilde{\mathbf{i}}_{\text{snk,max}}, \mathbf{e_j} \rangle + 1 \right) \right). \tag{8.30}$$

Remark 8.15 Due to technical reasons, the PIP defined by Eqs. (8.19), (8.20), (8.21), (8.22), (8.23), (8.24), (8.25), (8.26), (8.27), and (8.28) contains the extended iteration vectors introduced in Section 8.2.1.2. Otherwise, the system of inequalities might have no solution in presence of initial data elements or when $\mathbf{i}_{\text{src},\mathbf{0}} = \mathbf{i}_{\text{src,max}}$. However, as the WDF edge returns into its initial state after termination of one schedule period (see Section 5.2 and Theorem 5.14), $\Delta\text{snk}(\tilde{\mathbf{i}}_{\text{src},\mathbf{0}})$ will be periodic with $\langle \tilde{\mathbf{i}}_{\text{src},0}, \mathbf{e_0} \rangle$. In order words, the PIP has to be solved for only one (local) schedule period. This not only means that the non-extended iteration vector $\mathbf{i}_{\text{src},0}$ is sufficient for a correct hardware implementation but also solves the problem that $\langle \tilde{\mathbf{i}}_{\text{src,max}}, \mathbf{e_0} \rangle = \langle \tilde{\mathbf{i}}_{\text{snk,max}}, \mathbf{e_0} \rangle = \infty$ is not supported by typical PIP solvers. Instead, $\langle \tilde{\mathbf{i}}_{\text{src,max}}, \mathbf{e_0} \rangle$ and $\langle \tilde{\mathbf{i}}_{\text{snk,max}}, \mathbf{e_0} \rangle$ can be set to sufficiently large values.

8.2.7.2 Solution of the PIP

Solutions of parametric integer programs can be expressed symbolically by help of nested conditionals that can be obtained by means of the PIP library [5]. In other words, the hardware synthesis of a multidimensional FIFO requires to solve the PIP defined by Eqs. (8.19), (8.20), (8.21), (8.22), (8.23), (8.24), (8.25), (8.26), (8.27), and (8.28) exactly once in order to determine $\Delta\text{snk}\,(\mathbf{i}_{\text{src}})$ for all possible time instances. However, this requires to first rewrite

Eq. (8.25) because it corresponds to an or-combination of inequalities, which is not supported by the PIP library:

$$\begin{aligned} x \prec y \Leftrightarrow\ & \langle \mathbf{x}, \mathbf{e}_1 \rangle < \langle \mathbf{y}, \mathbf{e}_1 \rangle \\ & \vee\ (\langle \mathbf{x}, \mathbf{e}_1 \rangle = \langle \mathbf{y}, \mathbf{e}_1 \rangle \wedge \langle \mathbf{x}, \mathbf{e}_2 \rangle < \langle \mathbf{y}, \mathbf{e}_2 \rangle) \\ & \vee\ (\langle \mathbf{x}, \mathbf{e}_1 \rangle = \langle \mathbf{y}, \mathbf{e}_1 \rangle \wedge \langle \mathbf{x}, \mathbf{e}_2 \rangle = \langle \mathbf{y}, \mathbf{e}_2 \rangle \wedge \langle \mathbf{x}, \mathbf{e}_3 \rangle < \langle \mathbf{y}, \mathbf{e}_3 \rangle) \\ & \vee\ \ldots . \end{aligned}$$

In principle, this can be achieved by two means. The mathematically more elegant solution enumerates the source iterations in order to transform Eq. (8.25):

$$\tilde{\mathbf{i}}_{\mathrm{src}} \succ \tilde{\mathbf{i}}_{\mathrm{src},0} \Leftrightarrow Q_{\mathrm{src}}\left(\tilde{\mathbf{i}}_{\mathrm{src}}\right) > O_{\mathrm{src}}\left(\tilde{\mathbf{i}}_{\mathrm{src},0}\right), \tag{8.31}$$

with

$$Q_{\mathrm{src}}\left(\tilde{\mathbf{i}}_{\mathrm{src}}\right) = \sum_{i=0}^{n_{\mathrm{src}}} \left(\langle \tilde{\mathbf{i}}_{\mathrm{src}}, \mathbf{e}_{\mathbf{i}} \rangle \times \prod_{j=i+1}^{n_{\mathrm{src}}} \left(\langle \tilde{\mathbf{i}}_{\mathrm{src,max}}, \mathbf{e}_{\mathbf{j}} \rangle + 1 \right) \right). \tag{8.32}$$

Unfortunately, when trying to solve the resulting PIP for realistic examples, this leads to severe difficulties in form of tremendous calculation efforts and extremely huge memory requirements even for very small image sizes. Even worse, sometimes the PIP library [5] failed completely.

Alternatively, Eq. (8.25) can be split into a disjunction of $n_{\mathrm{src}} + 1$ linear conjunctions. For each of these linear conjunctions, Eqs. (8.19), (8.20), (8.21), (8.22), (8.23), (8.24), (8.25), (8.26), (8.27), and (8.28) can be solved individually. In order to obtain the overall lexicographic minimum, the so-obtained solutions in form of nested if-then-else conditionals have to be merged. Although this is of exponential complexity, the measured run-times are much smaller than when using Eq. (8.31).

However, even in this case the solutions returned by the PIP library are highly complex and contain integer divisions whose quotients are not powers of 2, huge amounts of multiplications, and multiple if-then-else statements. Fortunately, this problem can be rather easily solved by generation of a table $T\left(\mathbf{i}_{\mathrm{src},0}\right)$ that assigns to each write operation the resulting number of additional read operations. By means of the algorithm shown in Fig. 8.11, this table can then be written as nested conditionals, which can be efficiently synthesized in hardware.

Example 8.16 Figure 8.12 exemplifies the corresponding result for calculation of $\Delta\mathrm{snk}\left(\mathbf{i}^*_{\mathrm{src}}\right)$ in case of the tiling operation depicted in Fig. 8.3. `ci_iterator` corresponds to the hierarchical source iterator $\mathbf{i}^*_{\mathrm{src}} \in \mathbb{N}^3$ as shown in Fig. 8.1. Note that the latter has already been processed as discussed later on in Section 8.2.8. `ci_delta_snk` equals $\Delta\mathrm{snk}\left(\mathbf{i}^*_{\mathrm{src}}\right)$. Thus, the result is very simple and can be efficiently synthesized in hardware.

Remark 8.17 Instead of solving a Parametric Integer Program, $\Delta\mathrm{snk}\left(\tilde{\mathbf{i}}_{\mathrm{src}}\right)$ can also be derived via self-timed simulation of an individual edge.

Remark 8.18 Technically, also Eq. (8.29) can be integrated in the PIP. However, in this case, the PIP library failed for some application scenarios. Consequently, the PIP is restricted to Eqs. (8.19), (8.20), (8.21), (8.22), (8.23), (8.24), (8.25), (8.26), (8.27), and (8.28). Equation (8.29) is taken into account when establishing the table $T\left(\mathbf{i}_{\mathrm{src},\mathbf{0}}\right)$.

```
IF (ci_iterator(1)<=4) THEN
 IF (ci_iterator(2)=0) THEN
  ci_delta_snk := 1;
 ELSE
  ci_delta_snk := 0;
 END IF ;
ELSIF (ci_iterator(2)=0) THEN
 IF (ci_iterator(3)<=3) THEN
  ci_delta_snk := 1;
 ELSE
  ci_delta_snk := 26;
 END IF ;
ELSE
 ci_delta_snk := 1;
END IF ;
```

Fig. 8.12 Generated VHDL code for determination of $\Delta\text{snk}\left(\tilde{\mathbf{i}}^{*}_{\text{src}}\right)$ belonging to Fig. 8.3. `ci_iterator` corresponds to the hierarchical source iterator $\mathbf{i}_{\text{src}}$ generated by *source iterator* block in Fig. 8.4. `ci_delta_snk` is the VHDL notation for Δsnk as depicted in Fig. 8.4

8.2.7.3 Source Fill-Level Control

Similar to the sink fill-level control, determination of the number of possible write operations is performed by calculating for each read operation the additional number of allowed write operations $\Delta\text{src}\left(\tilde{\mathbf{i}}_{\text{snk}}\right)$ using the following parametric integer program (PIP):

$$\min_{\prec}\left(\tilde{\mathbf{i}}_{\text{src}}\right) \tag{8.33}$$

$$\tilde{\mathbf{i}}_{\text{src,max}} \geq \tilde{\mathbf{i}}_{\text{src}} \geq \mathbf{0} \tag{8.34}$$

$$\tilde{\mathbf{i}}_{\text{snk,max}} \geq \tilde{\mathbf{i}}_{\text{snk}} \geq \mathbf{0} \tag{8.35}$$

$$\mathbf{p} > \mathbf{I}_{\text{src,w}} \geq \mathbf{0} \tag{8.36}$$

$$\mathbf{c} > \mathbf{I}_{\text{snk,w}} \geq \mathbf{0} \tag{8.37}$$

$$\tilde{\mathbf{i}}_{\text{snk}} \succeq \tilde{\mathbf{i}}_{\text{snk},\mathbf{0}} \tag{8.38}$$

$$\underbrace{\tilde{M}_{\text{src}} \times \left(\tilde{\mathbf{i}}_{\text{src}}, \mathbf{I}_{\text{src,w}}\right) + \boldsymbol{\delta}}_{(a)} - \underbrace{\tilde{M}_{\text{snk}} \times \left(\tilde{\mathbf{i}}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right) + \mathbf{b}^{\mathbf{s}}}_{(b)} = \mathbf{0} \tag{8.39}$$

$$\tilde{C}_{\text{snk}} \times \left(\tilde{\mathbf{i}}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right) - \mathbf{b}^{\mathbf{l}}_{\mathbf{c}} \geq \mathbf{0} \tag{8.40}$$

$$\tilde{C}_{\text{snk}} \times \left(\tilde{\mathbf{i}}_{\text{snk}}, \mathbf{I}_{\text{snk,w}}\right) - \mathbf{b}^{\mathbf{h}}_{\mathbf{c}} \leq \mathbf{0} \tag{8.41}$$

$$\tilde{\mathbf{i}}_{\text{snk,max}} \geq \tilde{\mathbf{i}}_{\text{snk},0} \geq \mathbf{0}. \tag{8.42}$$

Given the current read operation $0 \preceq \tilde{\mathbf{i}}_{\text{snk},0} \preceq \tilde{\mathbf{i}}_{\text{snk,max}}$ in form of a PIP parameter (Eq. 8.42), it searches the earliest write operation (Eq. (8.33)) whose produced data element is required by a read operation $\tilde{\mathbf{i}}_{\text{snk}}$ (Eq. (8.39)) that occurs not before $\tilde{\mathbf{i}}_{\text{snk},0}$. In other words, let $\tilde{\mathbf{i}}_{\text{src}} = f_{\text{src}}\left(\tilde{\mathbf{i}}_{\text{snk},0}\right)$ be the solution of the above PIP. Then all data elements produced before $f_{\text{src}}\left(\tilde{\mathbf{i}}_{\text{snk},0}\right)$ are not required anymore and can be discarded.

Whereas for the sink fill-level control the PIP solution could be directly used to derive the number of additional read operations (see Eq. (8.29)), the situation is more complex on the write side of the multidimensional FIFO, because it has to take the available buffer sizes into account. For this reason, the PIP defined via Eqs. (8.33), (8.34), (8.35), (8.36), (8.37), (8.38), (8.39), (8.40), (8.41), and (8.42) has to be solved individually for each virtual memory channel $\mathbf{C}$, $0 \leq \mathbf{C} < \mathbf{m}$, by restricting $\tilde{\mathbf{i}}_{\text{src}}$ and $\mathbf{I}_{\text{src,w}}$ to values associated with the virtual memory channel $\mathbf{C}$:

$$\tilde{M}_{\text{src}} \times \left(H_{\text{src}} \times \left(\tilde{\mathbf{i}}_{\text{src}}, \mathbf{I}_{\text{src,w}} \right) \right) = \mathbf{C}. \tag{8.43}$$

$H_{\text{src}} \in \mathbb{N}^{(n_{\text{src}}+n+1),(n_{\text{src}}+n+1)}$ is a diagonal helper matrix with the following values:

$$\langle H \times \mathbf{e_i}, \mathbf{e_j} \rangle = \begin{cases} 0 & i \neq j \\ 1 & i = j \wedge j > n_{\text{src}} + 1 \\ 1 & i = j \wedge A_j = 0 \\ 0 & \text{otherwise} \end{cases}.$$

A_j is the address factor defined in Eq. (8.8), n identifies the number of token dimensions, and $n_{\text{src}} + 1$ corresponds to the number of components of the extended hierarchical source iteration vector (see Section 8.2.1.2). Since these components are split such that they contribute either to the write address or to the selection of the considered virtual memory channel (see Section 8.2.3), Eq. (8.43) ensures that only those write operations are taken into account that access the given memory channel $\mathbf{C}$. In other words, let $\tilde{\mathbf{i}}_{\text{src}} = f_{\text{src}} \left(\tilde{\mathbf{i}}_{\text{snk},0}; \mathbf{C} \right)$ be the solution of the resulting PIP. Then all data elements produced before $f_{\text{src}} \left(\tilde{\mathbf{i}}_{\text{snk},0}; \mathbf{C} \right)$ and stored in virtual channel $\mathbf{C}$ and are not required anymore and can be discarded.

Example 8.19 In order to enable parallel data access and linear address generation, Fig. 8.6 proposed $\mathbf{m} = (6, 2)^T$ virtual memory channels. In order to ensure that an iteration vector component either contributes to the source address or selects the virtual memory channel where to write to, Example 8.5 derived the following extended source iteration vector:

$$\tilde{\mathbf{i}}^*_{\text{src,max}} = (\infty, 1, 1, 2)^T, \; \tilde{M}^*_{\text{src}} = \begin{pmatrix} 0 & 0 & 6 & 2 & 1 & 0 \\ 4 & 2 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

The associated address factors equal:

$$A_3 = 0, \; A_2 = 1, \; A_1 = 2, \; A_0 = 4.$$

When considering now exemplarily the virtual memory channel labeled by K ($\mathbf{C} = (4, 1)^T$) in Fig. 8.6, this leads to the following helper matrix:

$$H_{\text{src}} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Equation (8.43) consequently becomes:

$$\begin{pmatrix} 0 & 0 & 6 & 2 & 1 & 0 \\ 4 & 2 & 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ 0 \\ \langle \tilde{\mathbf{i}}_{\mathrm{src}^*}, \mathbf{e_3} \rangle \\ \langle \mathbf{I}_{\mathrm{src,w}}, \mathbf{e_1} \rangle \\ \langle \mathbf{I}_{\mathrm{src,w}}, \mathbf{e_2} \rangle \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \end{pmatrix}.$$

With consideration of the values of $\tilde{\mathbf{i}}^*_{\mathrm{src,max}}$ and of $\mathbf{p}$, this leads to

$$\begin{aligned} \langle \mathbf{I}_{\mathrm{src,w}}, \mathbf{e_2} \rangle &= 1, \\ \langle \mathbf{I}_{\mathrm{src,w}}, \mathbf{e_1} \rangle &= 0, \\ \langle \tilde{\mathbf{i}}_{\mathrm{src}^*}, \mathbf{e_3} \rangle &= 2, \end{aligned}$$

and corresponds to what has been depicted in Fig. 8.6.

Since the address increments by 1 for each write access of channel $\mathbf{C}$, the first write operation $\tilde{\mathbf{i}}_{\mathrm{src}}(\mathbf{C})$ not possible anymore with regard to channel $\mathbf{C}$ can be calculated as

$$\tilde{\mathbf{i}}_{\mathrm{src}}(\mathbf{C}) = \mathrm{lex\_inc}\left(f_{\mathrm{src}}\left(\tilde{\mathbf{i}}_{\mathrm{snk},0}; \mathbf{C}\right), B(\mathbf{C})\right). \tag{8.44}$$

$\mathrm{lex\_inc}\left(\tilde{\mathbf{i}}_{\mathrm{src}}, B\right)$ increments $\tilde{\mathbf{i}}_{\mathrm{src}}$ by B write operations contributing to the same virtual memory channel:

$$\mathrm{lex\_inc} : \left(\mathbb{N}^{n_{\mathrm{src}}+1}, \mathbb{N}\right) \rightarrow \mathbb{N}^{n_{\mathrm{src}}+1}$$

$$\forall 0 \leq j \leq n_{\mathrm{src}} : \langle \mathrm{lex\_inc}\left(\tilde{\mathbf{i}}_{\mathrm{src}}, B\right), \mathbf{e_j} \rangle = \begin{cases} \langle \tilde{\mathbf{i}}_{\mathrm{src}}, \mathbf{e_j} \rangle & \text{if } A_j = 0 \\ \left\lfloor \frac{a\left(\tilde{\mathbf{i}}_{\mathrm{src}}\right)+B}{A_j} \right\rfloor \bmod \left(\langle \tilde{\mathbf{i}}_{\mathrm{src,max}}, \mathbf{e_j} \rangle + 1\right) & \text{otherwise} \end{cases}. \tag{8.45}$$

$a\left(\tilde{\mathbf{i}}_{\mathrm{src}}\right)$ is the write address function given in Eq. (8.7) and enumerates all write operations contributing to the same virtual memory channel $\mathbf{C}$ in production order. A_j equals the corresponding address factors defined in Eq. (8.8). Equation (8.45) thus essentially calculates the source iteration that writes to address $a(\tilde{\mathbf{i}}_{\mathrm{src}}) + B$ in memory channel $\mathbf{C}$. This corresponds exactly to the Bth iteration after $\tilde{\mathbf{i}}_{\mathrm{src}}$ writing to the same virtual memory channel $\mathbf{C}$ than $\tilde{\mathbf{i}}_{\mathrm{src}}$.

Example 8.20 In order to ease understanding of Eq. (8.44), consider again the scenario depicted in Fig. 8.6b. From Example 8.5 it is known that

$$\mathbf{i}^*_{\mathrm{src,max}} = (1, 1, 2)^T,$$

$$\begin{aligned}\tilde{\mathbf{i}}^{*}_{\mathrm{src,max}} &= (\infty, 1, 1, 2)^T,\\ A_0 &= 4,\\ A_1 &= 2,\\ A_2 &= 1,\\ A_3 &= 0.\end{aligned}$$

Consider now, for instance, virtual memory channel K ($\mathbf{C} = (4, 1)^T$) and assume a buffer size of $B(\mathbf{C}) = 3$ as well as $f_{\mathrm{src}}(\tilde{\mathbf{i}}_{\mathrm{snk},\mathbf{0}}; \mathbf{C}) = (0, 0, 0, 2)^T$. Remember that $f_{\mathrm{src}}(\tilde{\mathbf{i}}_{\mathrm{snk},\mathbf{0}}; \mathbf{C})$ returns an extended iteration vector, such that the first component equals the (local) schedule period, which has been set to zero. Consequently, $f_{\mathrm{src}}\left(\tilde{\mathbf{i}}_{\mathrm{snk},0}; \mathbf{C}\right) = (0, 0, 0, 2)^T$ corresponds to the write operation highlighted in Fig. 8.6b. Application of Eq. (8.7) leads to

$$a\left(f_{\mathrm{src}}\left(\tilde{\mathbf{i}}_{\mathrm{snk},0}; \mathbf{C}\right)\right) = 0.$$

Consequently,

$$\mathrm{lex\_inc}\left(f_{\mathrm{src}}\left(\tilde{\mathbf{i}}_{\mathrm{snk},0}; \mathbf{C}\right), B(\mathbf{C})\right) = \begin{pmatrix}0\\1\\1\\2\end{pmatrix}.$$

Thus, concerning memory channel K, $\tilde{\mathbf{i}}_{\mathrm{src}}(\mathbf{C}) = (0, 1, 1, 2)^T$ is the first write operation which cannot be executed anymore, because the data elements generated by the write operation $f_{\mathrm{src}}\left(\tilde{\mathbf{i}}_{\mathrm{snk},0}; \mathbf{C}\right) = (0, 0, 0, 2)^T$ are still required. This is exactly what can be expected from Fig. 8.6b, because due to raster-scan order of the write operations, a buffer size of $B(\mathbf{C}) = 4$ would be required when keeping both the data elements produced by write operation $(0, 0, 0, 2)^T$ and $(0, 1, 1, 2)^T$. Consequently, $(0, 1, 1, 2)^T$ cannot be executed anymore.

Considering now all memory channels $\mathbf{C}$, with $0 \leq \mathbf{C} < \mathbf{m}$, and given the current read operation $\tilde{\mathbf{i}}_{\mathrm{snk},0}$, the first write operation that cannot be executed anymore can be obtained by

$$\tilde{\mathbf{i}}_{\mathrm{src}} = h_{\mathrm{src}}\left(\tilde{\mathbf{i}}_{\mathrm{snk},0}\right) := \min_{\prec}\left\{\mathrm{lex\_inc}\left(f_{\mathrm{src}}\left(\tilde{\mathbf{i}}_{\mathrm{snk},0}; \mathbf{C}\right), B(\mathbf{C})\right), 0 \leq \mathbf{C} < \mathbf{m}\right\}.$$

Consequently, termination of the read operation $\tilde{\mathbf{i}}_{\mathrm{snk},0}$ enables

$$\Delta\mathrm{src}\left(\tilde{\mathbf{i}}_{\mathrm{snk},0}\right) = Q_{\mathrm{src}}\left(h_{\mathrm{src}}\left(\mathrm{succ}\left(\tilde{\mathbf{i}}_{\mathrm{snk},0}\right)\right)\right) - Q_{\mathrm{src}}\left(h_{\mathrm{src}}\left(\tilde{\mathbf{i}}_{\mathrm{snk},0}\right)\right) \tag{8.46}$$

additional write operations. Q_{src} is defined in Eq. (8.32), and $\mathrm{succ}(\tilde{\mathbf{i}}_{\mathrm{snk},0})$ returns the successor of $\tilde{\mathbf{i}}_{\mathrm{snk},0}$ as described by Eq. (8.1).

Remark 8.21 In case all virtual memory channels have the same buffer capacity $B(\mathbf{C}) = B$, $\tilde{\mathbf{i}}_{\mathrm{src}}(\mathbf{C})$ calculated in Eq. (8.44) adds the same number of write operations to each

$f_{\text{src}}(\tilde{\mathbf{i}}_{\text{snk},0}; \mathbf{C})$ returned by the PIP solver. Consequently,

$$\begin{aligned} & h_{\text{src}}\left(\tilde{\mathbf{i}}_{\text{snk},0}\right) \\ = & \min_{\prec}\left\{\text{lex\_inc}\left(f_{\text{src}}\left(\tilde{\mathbf{i}}_{\text{snk},0}; \mathbf{C}\right), B\left(\mathbf{C}\right)\right), 0 \leq \mathbf{C} < \mathbf{m}\right\} \\ = & \text{lex\_inc}\left(\min_{\prec}\left\{f_{\text{src}}\left(\tilde{\mathbf{i}}_{\text{snk},0}; \mathbf{C}\right), 0 \leq \mathbf{C} < \mathbf{m}\right\}, B\right), \end{aligned}$$

and

$$\begin{aligned} & \Delta\text{src}\left(\tilde{\mathbf{i}}_{\text{snk},0}\right) \\ = & Q_{\text{src}}\left(\min_{\prec}\left\{f_{\text{src}}\left(\text{succ}\left(\tilde{\mathbf{i}}_{\text{snk},0}\right); \mathbf{C}\right), 0 \leq \mathbf{C} < \mathbf{m}\right\}\right) - \\ & Q_{\text{src}}\left(\min_{\prec}\left\{f_{\text{src}}\left(\tilde{\mathbf{i}}_{\text{snk},0}; \mathbf{C}\right), 0 \leq \mathbf{C} < \mathbf{m}\right\}\right). \end{aligned}$$

This, however, can be obtained much easier by solving the parametric integer program (PIP) defined via Eqs. (8.33), (8.34), (8.35), (8.36), (8.37), (8.38), (8.39), (8.40), (8.41), and (8.42) independently of the virtual memory channels leading to a similar formula as obtained for the sink fill-level control:

$$\Delta\text{src}\left(\tilde{\mathbf{i}}_{\text{snk},0}\right) = Q_{\text{src}}\left(f_{\text{src}}\left(\text{succ}\left(\tilde{\mathbf{i}}_{\text{snk},0}\right)\right)\right) - Q_{\text{src}}\left(f_{\text{src}}\left(\tilde{\mathbf{i}}_{\text{snk},0}\right)\right).$$

8.2.8 Elimination of Modular Dependencies

The previous sections have introduced techniques for efficient address generation and fill-level control. They essentially exploit that both the address increment, $\Delta\text{snk}\left(\mathbf{i}_{\text{src}}\right)$ and $\Delta\text{src}\left(\mathbf{i}_{\text{snk}}\right)$ can be efficiently precomputed once. Then, the result can be compactly represented via nested conditionals, thus eliminating the need to store this information in huge lookup tables.

This benefit can be even further boosted by eliminating modular dependencies. Figure 8.13a, for instance, shows a tiling operation similar to those depicted in Fig. 8.3, but using different tile and image sizes. The production order is represented in form of numbers, while the arrows highlight the read order. Figure 8.13b illustrates for each write operation $\mathbf{i}_{\text{src}}$ the number of additionally possible read operations $\Delta\text{snk}\left(\mathbf{i}_{\text{src}}\right)$. As it can be clearly seen, these values are identical for all tiles except the first and the last one. In particular, whenever the source has generated the last pixel of a tile, the sink can read not only this pixel but also all lines of the next tile except for the last line. Hence, in order to determine the value of $\Delta\text{snk}\left(\mathbf{i}_{\text{src}}\right)$, the multidimensional FIFO has to know whether the source has produced the last pixel of a tile. Mathematically, this is nothing else than checking whether

$$\langle \mathbf{i}_{\text{src}}, \mathbf{e}_1 \rangle = 2, \tag{8.47}$$

$$\langle \mathbf{i}_{\text{src}}, \mathbf{e}_2 \rangle \bmod 3 = 2. \tag{8.48}$$

Unfortunately, this modular dependency increases the complexity of the resulting hardware implementation. Since the nested conditionals generated by the algorithm shown in

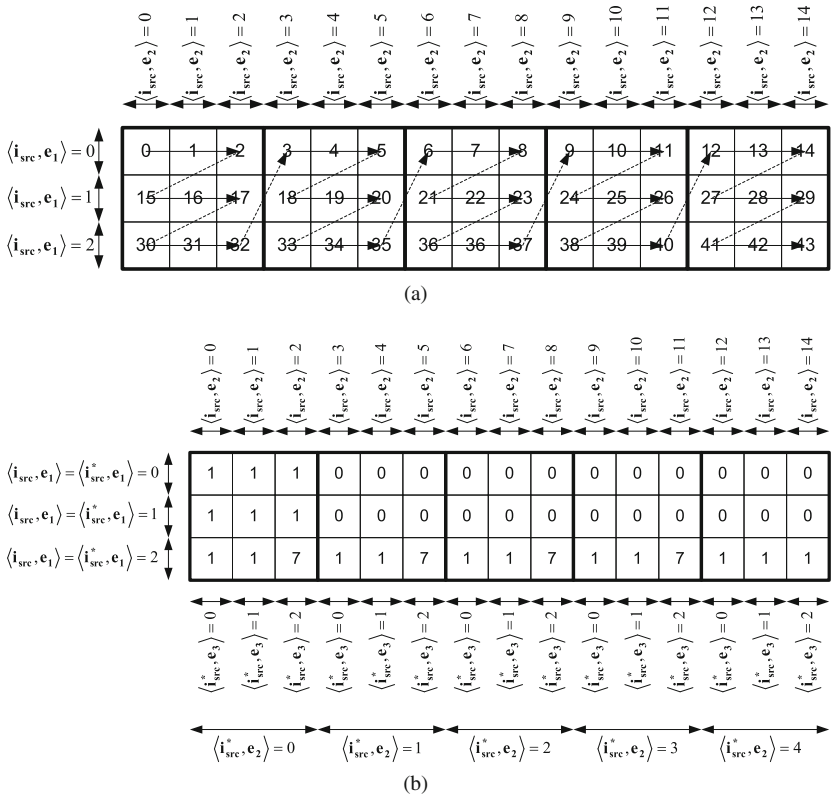


Fig. 8.13 Occurrence of modular dependencies when calculating $\Delta\text{snk}\,(\mathbf{i}_{\text{src}})$ for image tiling. **a** Read and write orders. The write order is depicted by *numbers*, while *arrows* represent the read order. $\mathbf{i}_{\text{src}}$ correspond to the iteration vector describing which data element is currently written. **b** Number of enabled additional sink invocations $\Delta\text{snk}\,(\mathbf{i}_{\text{src}})$ for each write operation. Obviously, all tiles show the same pattern, except the first and the last one. This, however, leads to modular dependencies as described by Eqs. (8.47) and (8.48). Fortunately they can be eliminated by introduction of the source iteration vector $\mathbf{i}^*_{\text{src}}$

Fig. 8.11 do not contain any modulo function, they are translated into a possible huge amount of conditions. Equation (8.48), for instance, can be represented by $\langle \mathbf{i}_{\text{src}}, \mathbf{e}_2 \rangle = 2 \vee \langle \mathbf{i}_{\text{src}}, \mathbf{e}_2 \rangle = 5 \vee \ldots$. However, this increases the required resources for a hardware implementation. Furthermore, even if the conditions included a modulo function, this would not help much, since its hardware implementation is expensive due to the necessary instantiation of a hardware divider. Unfortunately, such a module already consumes 130 flip-flops and 100 lookup tables for a 12-bit dividend and a 6-bit divisor and shows a latency of 14 clock cycles to calculate the result when using the Xilinx Core Generator [306].[3]

[3] On the other hand, Section 8.5.2 will show that less than 300–400 flip-flops and lookup tables are sufficient to synthesize a heavily pipelined high-speed multidimensional FIFO.

Fortunately, it is easily possible to get rid of the modulo-operation by replacing $\mathbf{i}_{\text{src}}$ with a three-dimensional vector: $\mathbf{i}^*_{\text{src}} = \left(y^*_{\text{src}}, x^*_{\text{src},1}, x^*_{\text{src},2}\right)$ with $\mathbf{0} \leq \mathbf{i}^*_{\text{src}} \leq (2,4,2)$. $\langle \mathbf{i}^*_{\text{src}}, \mathbf{e}_2 \rangle$ defines the horizontal tile coordinate while $\langle \mathbf{i}^*_{\text{src}}, \mathbf{e}_3 \rangle$ corresponds to the horizontal coordinate of the produced data element relative to the tile border. This removes the modular dependencies because they are now already occurring in the source iterator. In other words, Eq. (8.48), for instance, can be replaced by $x^*_{\text{src},2} = \langle \mathbf{i}^*_{\text{src}}, \mathbf{e}_3 \rangle = 2$, which can be efficiently represented in hardware.

This kind of hierarchical iteration vector transformation can be performed automatically by introduction of redundant firing blocks as explained in Algorithm 6. It obtains the current sequence of firing blocks O, which is assumed to begin with $\mathbf{B_1} = 1$. As the latter does not have any effect on the resulting read or write order, this does not impact the applicability of the presented algorithm. In addition, the algorithm obtains the desired firing block size B^{dest} that shall be inserted into the sequence of firing blocks in dimension $\mathbf{e_i}$. The other dimensions are chosen such that this operation does not change the resulting read or write order.

Example 8.22 The tiling example given in Fig. 8.13 can be described by

$$O^{\text{src}} = \left[\begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}15\\3\end{pmatrix}\right].$$

Elimination of modular dependencies requires a firing block of size $B^{\text{dest}} = 3$ in dimension $\mathbf{e_1}$. Application of Algorithm 6 leads to $k = 2$ and $\mathbf{B}_{\text{new}} = (3,1)$:

$$O^{\text{src}}_{\text{new}} = \left[\begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}3\\1\end{pmatrix}, \begin{pmatrix}15\\3\end{pmatrix}\right].$$

Algorithm 6 Insertion of redundant firing blocks

```
InsertRedundantFiringBlock{
  inputs: O = [B_1 = 1, B_2, ..., B_q], i, B^dest
  if (B^dest ≤ 1) return;
  k = 2; //because B_1 = 1
  while((k ≤ q) && (⟨B_k, e_i⟩ < B^dest)) k = k + 1;
  if (k > q)
    return; //New reads or writes would be added
  if (⟨B_k, e_i⟩ = B^dest)
    return; //Firing block already exists
  // from here on, ⟨B_k, e_i⟩ > B^dest, ⟨B_{k-1}, e_i⟩ < B^dest
  if ((⟨B_k, e_i⟩ mod B^dest ≠ 0) || (B^dest mod ⟨B_{k-1}, e_i⟩ ≠ 0))
    return; //Incompatible firing block
            //see Definition 5.8
  for(j = 1 : i − 1){
    ⟨B_new, e_j⟩ = ⟨B_k, e_j⟩
  }
  ⟨B_new, e_i⟩ = B^dest
  for(j = i + 1 : n){
    ⟨B_new, e_j⟩ = ⟨B_{k-1}, e_j⟩
  }
  insert B_new before B_k in O
}
```

With Eq. (8.2), it follows:

$$\begin{aligned}\mathbf{i}_{\text{src,max}} &= \left(\frac{3}{1}-1, \frac{15}{3}-1, \frac{1}{1}-1, \frac{3}{1}-1, \frac{1}{1}-1, \frac{1}{1}-1\right)^T \\ &= (2,4,0,2,0,0)^T\,.\end{aligned}$$

Elimination of all components amounting zero finally leads to

$$\mathbf{i}^*_{\text{src,max}} = (2,4,2)^T\,.$$

This corresponds exactly to the expected result.

Algorithm 7 shows a simplified method for elimination of modular dependencies. To this end, it essentially tries to insert the sink firing blocks into the sequence of source firing blocks and vice versa. Lines (04) and (11) take into account that due to different window movement and effective token sizes the number of required read and write operations for entering periodic patterns might differ.

Algorithm 7 Simplified elimination of modular dependencies

```
(00) EliminateModularDependencies{
(01)   inputs: O^src, O^snk, p ∈ ℕ^n, Δc ∈ ℕ^n
(02)   for(k = 1 : |O^src|) {
(03)     for(i = 1 : n) {
(04)       B^dest = ⟨B_k^src, e_i⟩ × ⟨p,e_i⟩ / ⟨Δc,e_i⟩
(05)       if (B^dest ∉ ℕ) continue;
(06)       InsertRedundantFiringBlock(O_snk, i, B^dest);
(07)     }
(08)   }
(09)   for(k = 1 : |O^snk|) {
(10)     for(i = 1 : n) {
(11)       B^dest = ⟨B_k^snk, e_i⟩ × ⟨Δc,e_i⟩ / ⟨p,e_i⟩
(12)       if (B^dest ∉ ℕ) continue;
(13)       InsertRedundantFiringBlock(O_src, i, B^dest);
(14)     }
(15)   }
(16) }
```

8.3 Determination of Channel Sizes

The above-presented FIFO architecture takes care that the sink only reads valid data in the correct order and that the source does not overwrite data that will still be required later on. As a consequence, the source stops operation as soon as the buffer gets full. This, however, means that the hardware implementation would deadlock if the memory sizes of the different memory channels are not large enough.

Fortunately, their size can be easily calculated by the techniques presented in Section 7.6.2. The only difference consists in the fact that not only $\mathbf{p}$ but also $\mathbf{m} = \text{scm}(\mathbf{p}, \Delta\mathbf{c})$ different memory channels have to be distinguished. Fortunately, this is straightforward.

8.4 Granularity of Scheduling

The concepts given so far in Section 8.2.7 are able to perform scheduling on pixel granularity. In other words, after each data element produced by the source, it is checked whether this permits additional sink executions. Similarly, for each read operation the number of possible write accesses is updated.

Fortunately, such an approach does not induce any throughput penalty, since the presented communication primitive is synthesized into hardware where all these checks can be performed in parallel, such that still one read and write access per clock cycle is possible. Furthermore, the techniques presented in Section 8.2.7 take care that the scheduling information can be synthesized efficiently in few nested if-then-else conditionals. Nevertheless, scheduling on pixel granularity of course requires that the nested if-then-else conditionals cover more situations than when updating the fill-level information, for instance only after every produced image. Consequently, there exists a tendency that fine-grained scheduling requires more complex if-then-else statements than a coarse-grained approach.

Consequently, an alternative approach consists in updating the fill-level information not for every produced and read data element, but only at the end of every row, block, or even image. Unfortunately, such an approach is not always possible, since it might introduce deadlocks. This happens if the source waits after a produced pixel for some feedback generated by the sink. If this feedback information depends on the just generated pixel, and the multidimensional FIFO holds back this value since the fill-level control is not updated, a deadlock arises. Furthermore, delayed fill-level update can increase the latency of the application implementation as well as of the required memory size.

In order to clarify these issues, the following sections will first investigate the impact of a delayed fill-level control on the resulting latency and memory requirements. Next, a strategy is shortly discussed that allows to adjust the granularity for the fill-level update. This permits to measure the impact on resource requirements later on in Section 8.5.6.

8.4.1 Latency Impact of Coarse-Grained Scheduling

In order to illustrate the possible impact of coarse-grained scheduling on the achievable latency, let's reconsider the tiling example of Fig. 8.3. This example has been chosen because there the consequences are particularly pronounced.

Figure 8.14a depicts the corresponding polyhedral representation as discussed in Chapter 7. Each semi-circle corresponds to an actor invocation, vectors represent data flow from the source to the sink. Since tiling performs data reordering, a lattice point remapping as discussed in Section 7.3.3 has to be applied. Consequently, the first five sink invocations depend on the first five source executions. The next five sink firings, however, require data generated by the source executions 10–15 of Fig. 8.3. In order to obtain valid schedules, the sink and source lattices have to be shifted in such a way that no sink execution depends

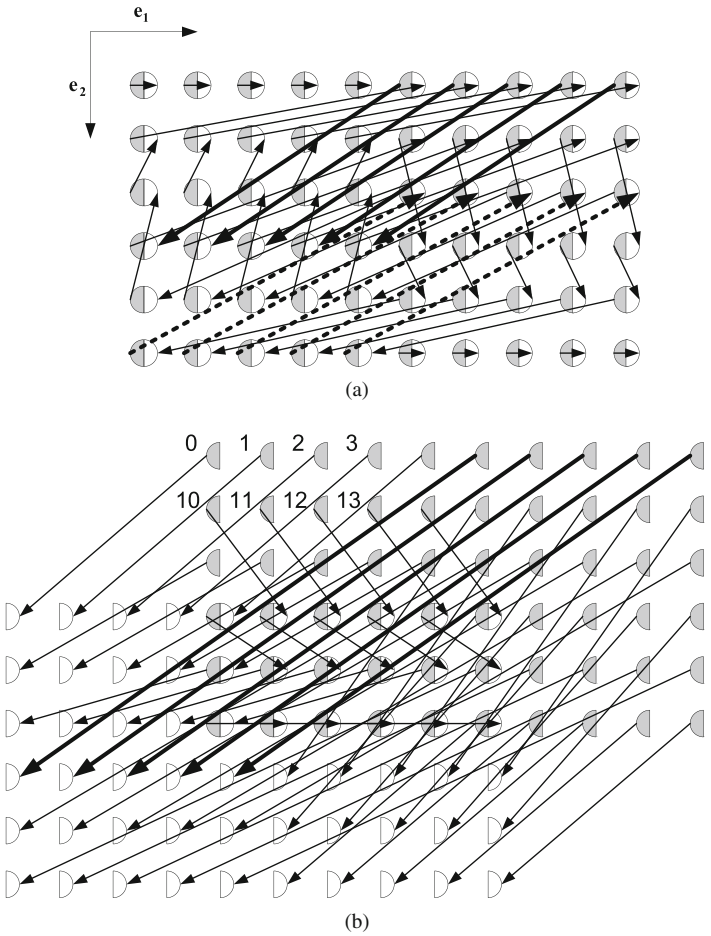


Fig. 8.14 Dependency graph for fine-grained scheduling. *White semi-circles* represent sink actor invocations, the source is identified by a *gray filling*. *Arrows* define data flowing from the source to the sink. **a** Initial lattice representation. **b** Shifted result

on source invocations taking place in the future (see Section 7.3.4). Since coinciding lattice points are executed concurrently (see Section 7.3.1), the required lattice shift is determined by the bold dashed dependency vectors: Obviously, the data required by the sink executions that those arrow heads point to are generated much later by the source. The corresponding invocations are identified by the origin of the bold dashed dependency vectors. Taking the principles of actor pipelining as described in Section 7.3.5 into account, this finally leads to the result depicted in Fig. 8.14. The underlying reason is that a dependency vector with length zero prevents concurrent execution of the source and sink, which contradicts our requirement of a high-performance implementation. In other words, the sink lattice has to be shifted such that all data elements required by a sink execution are effectively produced by previous source lattice points. Consequently, Fig. 8.14 and the principles of lattice wraparound (see

Section 7.4) tell us that the sink obviously starts execution after the source has generated a little bit less than half of the image.

Figure 8.15 depicts the corresponding lattice representation for coarse-grained scheduling. In order to improve readability, only those dependency vectors important for the following discussion are shown. They result from the fact that the fill-level control is supposed to be updated on tile granularity. In other words, the source only informs the sink about new data elements when the first tile has been completely generated. This happens when executing lattice point 54 of Fig. 8.1. Consequently, their is a virtual dependency vector $\mathbf{d_1}$ in Fig. 8.15a, pointing from this source invocation to the first sink execution. Similarly, the sink can only start reading the second tile when the source has terminated its production. This is expressed by dependency vector $\mathbf{d_3}$.

Figure 8.15b shows the result after lattice shifting required to obtain a valid schedule, where data elements are not read before being produced. Comparing Figs. 8.15b and 8.14b clearly shows that the sink starts operation much later for coarse-grained scheduling. Consequently, latency is approximately increased by the equivalent of half an image.

8.4.2 Memory Size Impact of Coarse-Grained Scheduling

The example presented in the previous section can also be used to clarify the consequences of coarse-grained scheduling on memory size requirements. As explained in Section 7.6, each dependency vector induces a certain memory requirement, whose value is given by the number of source invocations occurred from the origin to the arrowhead of the dependency vector. The overall buffer size is then the maximum of all dependency vectors.

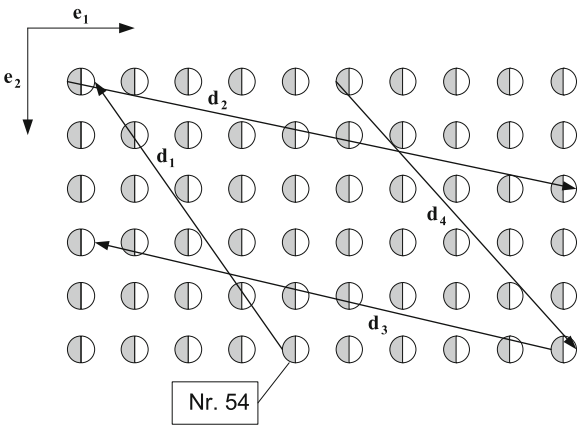
For fine-grained scheduling as depicted in Fig. 8.14b, the overall necessary buffer size results from the bold dependency vectors. Simple point counting taking lattice wraparound into account (see Section 7.4) leads to an overall buffer size of 52 data elements.

Figure 8.15b shows the corresponding scenario for coarse-grained scheduling. There, the overall buffer size results from the bold dependency vector $\mathbf{d_4}$. It takes into account that the fill-level information is only updated on tile granularity. In other words, the sink informs the source only after complete consumption of a tile that the corresponding memory space can be freed. Consequently, there are two virtual dependency vectors $\mathbf{d_2}$ and $\mathbf{d_4}$, pointing from the source invocation producing the first tile data element to the sink invocation consuming the corresponding last data element.

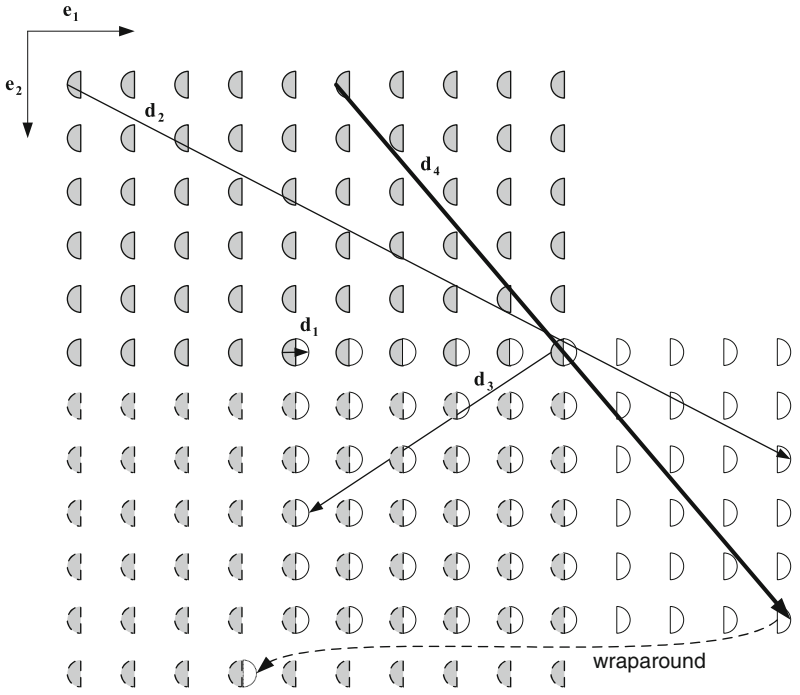
This fact in mind and assuming that the source produces one image after the other, point counting under consideration of lattice wraparound leads to a required memory size of 109 data elements. This is more than twice compared to the fine-grained scheduling. The underlying reason is that for coarse-grained scheduling, the sink waits longer before processing the data and because data elements are freed only after a complete tile has been processed.

8.4.3 Controlling the Scheduling Granularity

As described in the previous sections, coarse-grained scheduling can have a significant impact on achievable system latency and required memory size. Nevertheless, for some applications



(a)



(b)

Fig. 8.15 Impact of coarse-grained scheduling. It is assumed that the fill-level information is updated only on tile granularity. *White semi-circles* correspond to the sink, *gray* ones to the source. **a** Lattice representation with the most important data dependencies. **b** Shifted lattice for valid schedule. *Dashed semi-circles* correspond to the production of the second image

the resulting increase of these two values might be acceptable. Furthermore, not for all application scenarios the impact is as important as for the tiler discussed above. Consequently, it might be beneficial to reduce the scheduling granularity in order to simplify the resulting nested if-then-else statements.

Fortunately, this is easily possible by slight modifications of the techniques presented in Section 8.2. The basic idea consists in introducing constraints for the iteration vectors describing when the fill-level information shall be updated.

Example 8.23 Consider again the tiler example described in Fig. 8.1 and suppose that the fill-level control shall be updated using tile granularity. In other words, whenever the source has finished production of a complete tile, it shall inform the sink about the existence of new data elements. Mathematically, this can be expressed by the following conditions (see Fig. 8.1):

$$\begin{aligned}\langle \mathbf{i}^*_{\mathrm{src}}, \mathbf{e}_1 \rangle &= 5,\\ \langle \mathbf{i}^*_{\mathrm{src}}, \mathbf{e}_3 \rangle &= 4.\end{aligned}$$

Similarly, whenever the sink has completely consumed a tile, the memory can be released to the source:

$$\begin{aligned}\langle \mathbf{i}_{\mathrm{snk}}, \mathbf{e}_2 \rangle &= 5,\\ \langle \mathbf{i}_{\mathrm{snk}}, \mathbf{e}_3 \rangle &= 4.\end{aligned}$$

More generally, this can be expressed by matrix conditions:

$$\begin{aligned}R_{\mathrm{snk}} \times \tilde{\mathbf{i}}_{\mathrm{snk}} &= \boldsymbol{\xi}_{\mathrm{snk}},\\ R_{\mathrm{src}} \times \tilde{\mathbf{i}}_{\mathrm{src}} &= \boldsymbol{\xi}_{\mathrm{src}},\end{aligned}$$

where R_{snk} and R_{src} are diagonal matrices and $\tilde{\mathbf{i}}_{\mathrm{src}}$ and $\tilde{\mathbf{i}}_{\mathrm{snk}}$ are the extended iteration vectors (see Section 8.2.1.2). In case fill-level information shall be updated only for a given value $\langle \boldsymbol{\xi}, \mathbf{e_j} \rangle \neq 0$ of an iterator component $\langle \mathbf{i}, \mathbf{e_j} \rangle$, then $\langle R \times \mathbf{e_j}, \mathbf{e_j} \rangle = 1$, otherwise $\langle R \times \mathbf{e_j}, \mathbf{e_j} \rangle = \langle \boldsymbol{\xi}, \mathbf{e_j} \rangle = 0$.

For the sink fill-level control discussed in Section 8.2.7.1, it is then sufficient to replace Eq. (8.29) by

$$\Delta \mathrm{snk}\left(\tilde{\mathbf{i}}_{\mathrm{src},0}\right) = \begin{cases} 0 & \text{if } R_{\mathrm{src}} \times \tilde{\mathbf{i}}_{\mathrm{src},0} \neq \boldsymbol{\xi}_{\mathrm{src}} \\ Q_{\mathrm{snk}}\left(f\left(\tilde{\mathbf{i}}_{\mathrm{src},0}\right)\right) - Q_{\mathrm{snk}}\left(f\left(\widehat{\mathrm{pred}}\left(\tilde{\mathbf{i}}_{\mathrm{src},0}\right)\right)\right) & \text{otherwise} \end{cases}.$$

The function $\widehat{\mathrm{pred}}$ returns the predecessor iteration vector such that

$$R_{\mathrm{src}} \times \widehat{\mathrm{pred}}\left(\tilde{\mathbf{i}}_{\mathrm{src},0}\right) = \boldsymbol{\xi}_{\mathrm{src}}.$$

Similarly, for the source fill-level control explained in Section 8.2.7.3, Eq. (8.46) has to be replaced by

$$\Delta\mathrm{src}\left(\tilde{\mathbf{i}}_{\mathrm{snk},0}\right) = \begin{cases} 0 & \text{if } R_{\mathrm{snk}} \times \tilde{\mathbf{i}}_{\mathrm{snk},0} \\ & \neq \boldsymbol{\xi}_{\mathrm{snk}} \\ Q_{\mathrm{src}}\left(h_{\mathrm{src}}\left(\widehat{\mathrm{succ}}\left(\tilde{\mathbf{i}}_{\mathrm{snk},0}\right)\right)\right) - Q_{\mathrm{src}}\left(h_{\mathrm{src}}\left(\tilde{\mathbf{i}}_{\mathrm{snk},0}\right)\right) & \text{otherwise} \end{cases}.$$

The function $\widehat{\mathrm{succ}}$ returns the successor iterator vector such that

$$R_{\mathrm{snk}} \times \widehat{\mathrm{succ}}\left(\tilde{\mathbf{i}}_{\mathrm{snk},0}\right) = \boldsymbol{\xi}_{\mathrm{snk}}.$$

8.5 Results

After discussion of the multidimensional FIFO architecture, this section aims to present some implementation results obtained from a corresponding VHDL design in order to give the reader an idea about the required hardware resources. Several synthesis runs will help to evaluate the achievable performance and to quantify the effect of out-of-order communication, parallel data access, the influence of different channel sizes, and of scheduling granularities. Furthermore, combination with data reuse is shortly discussed. However, first the following section shortly explains the chosen implementation strategy permitting high clock frequencies.

8.5.1 Implementation Strategy for High Clock Frequencies

In order to be able to process large images in real time, critical operations have to be pipelined over several clock cycles. This holds in particular for the calculation of Δsnk and Δsrc necessary for correct fill-level control. Fortunately, this is easily possible for the architecture depicted in Fig. 8.4, because calculation of Δsnk and Δsrc is allowed to take several clock cycles without violating the requirement of processing one token per clock cycle. Additionally, a pipelined memory access similarly to what can be found for many high-speed memories can be employed: Since for high clock frequencies it is impossible to retrieve the desired data word within one clock cycle, it might take more than one clock cycle until the requested data word effectively arrives. Nevertheless, the interface permits to perform one read and write request per clock cycle. The same principle can be employed for the sink address generation in the multidimensional FIFO. Whereas both the sink address generation and the data access might take more than one clock cycle, the FIFO interface is designed in such a way that the sink can issue one read-request per clock cycle. All pipeline lengths can be chosen flexibly such that it is easily possible to adapt the multidimensional FIFO to the desired clock frequency. Since the latter can be very huge, large images can be processed in real time as demonstrated in the following section.

8.5.2 Out-of-Order Communication

Table 8.1 shows the resource requirements of the multidimensional FIFO in different configurations when performing the JPEG2000 tiling operation as depicted in Fig. 8.3, using a very large image with 4096×2140 pixels. Each multidimensional FIFO has been synthesized, placed, and routed independently by means of the Xilinx ISE tools. Register balancing helped to achieve balanced routing and combinatorial delays, increasing the overall possible operation frequency. In order to avoid extensive routing delays, registers have not been placed into IOB cells. Two different configurations are tested, using a big and a small tile size. The latter allows using internal Block RAM (BRAM) as FIFO buffer, whereas this is infeasible for the big tile size. In the latter case, an external memory is assumed by assigning the address bits to FPGA pins. Note that an external memory controller is not taken into account, because the generation of FPGA output signals and sampling of FPGA input signals significantly complicate the achievement of high frequencies due to tight phase requirements. Since, however, the results shall show the properties of the multidimensional FIFO and not of the memory controller, the latter is omitted in order to avoid influence on the synthesis results.

Table 8.1 Achievable frequency and required hardware resources for the JPEG2000 tiling after place and route for an image size of 4096×2140 pixels

	Virtex4 LX25-12			Virtex2 XC2V6000-6		
Tile size	MHz	FF	LUT	MHz	FF	LUT
128×5 (BRAM)	370	323 (1.5%)	379 (1.8%)	250	344 (< 1%)	366 (< 1%)
128×5 (BRAM, size not a power of 2)	291	335 (1.6%)	605 (2.8%)	200	353 (< 1%)	587 (1.0%)
128×5 (ext. memory)	405	264 (1.2%)	289 (1.4%)	245	275 (< 1%)	276 (< 1%)
1024×1070 (ext. memory)	360	380 (1.8%)	404 (1.9%)	225	367 (< 1%)	398 (< 1%)

As can be seen by means of Table 8.1, the discussed architecture for the multidimensional FIFO achieves both for recent and for older FPGA technology very good operation frequencies. In the case of a Virtex4 device, the frequencies are even sufficient to process an image with 4096×2140 pixels at more than 30 frames per second, which is a considerable throughput. Moreover, even with a Virtex2, 20 frames per second are possible. Furthermore, the resource consumption is low, needing not more than 1% of a Virtex2 6000 or 2% of a Virtex4 LX25, which is a rather small chip. Additionally, Table 8.1 clearly shows the negative impact of buffer sizes not being a power of 2. In this case, implementation of the modulo-operation in Eqs. (8.6) and (8.18) gets more expensive, increasing thus the required hardware resources while reducing attainable clock frequencies.

Table 8.2 compares the multidimensional FIFO with an ordinary FIFO generated by the *Xilinx CORE Generator* [306]. In order to allow a fair comparison, the multidimensional FIFO is configured in such a way that data production and consumption occur in the same order. Consequently, this scenario can also be realized by an ordinary FIFO. Both FIFOs are equipped with the same amount of memory equaling 16,384 items. The multidimensional FIFO is synthesized in two variants. One uses the same pipeline settings required to achieve the high synthesis frequencies of Table 8.1. The second takes into account that an identical consumption and production order are less complex than out-of-order communication. Consequently, fewer clock cycles are required for address calculation and fill-level control, which lead to a reduced pipeline length.

Table 8.2 Comparison of the multidimensional FIFO with an ordinary FIFO generated by the Xilinx CORE Generator for a Virtex4 LX25-12 device

	Frequency (MHz)	FF	LUT
Multidimensional, heavily pipelined	310	203 (1%)	303 (1.4%)
Multidimensional, less pipelined	435	63 (< 1%)	84 (< 1%)
Multidimensional, less pipelined, buffer size not a power of 2	300	146 (< 1%)	332 (< 1.5%)
CORE Generator	394	89 (< 1%)	114 (< 1%)

From these results, several consequences can be drawn. First of all, Table 8.2 confirms that the multidimensional FIFO can be as efficient as an ordinary FIFO in case of in-order communication. This essentially demonstrates the quality of the generated code. However, in case the buffer sizes are not powers of 2, the implementation gets tremendously more expensive. This is because the address generation becomes more complex and the available implementation requires a larger minimum pipeline depth. Note that this change has not been necessary in Table 8.1, since there this minimum pipeline depth has been met in any case.

Furthermore, Table 8.2 illustrates the influence of pipelining. As can be seen, it significantly increases the amount of required hardware resources. Whereas for simple in-order communication, this is completely useless and even reduces the achievable clock frequency, out-of-order communication makes use of more complex address generation and fill-level control. Consequently, pipelining is an important prerequisite for achieving high clock frequencies. On the other hand, this also means that out-of-order communication is more expensive than its in-order counterpart. This trend can additionally be confirmed by comparison of Tables 8.1 and 8.2.

Nevertheless, the multidimensional FIFO proves to achieve high throughput with acceptable resource consumption. Thanks to the static compile time analysis, higher clock frequencies can be attained compared to [318, 320]. Furthermore, the multidimensional FIFO does not need any valid bit and accepts one read and write access per clock cycle, which is not possible in [318, 320]. On the other hand, as [320] uses dynamic memory allocation, it has an increased flexibility and possibly better memory utilization.

After presentation of the JPEG2000 tiling operation, the next section evaluates the impact of parallel data access.

8.5.3 Out-of-Order Communication with Parallel Data Access

Whereas the JPEG2000 tiling operation only reads and writes 1 pixel per clock cycle, the transpose and shuffle operations depicted in Fig. 8.1 require parallel reads and writes of multiple data elements. Table 8.3a shows the corresponding synthesis results for the transpose and shuffle FIFO together with the number of available resources for a Virtex4-LX60 FPGA. It assumes a block size of 8×8 pixels with 12 bits each and QCIF images and compares different tradeoffs between resource and throughput requirements by usage of different serial-to-parallel and parallel-to-serial conversion settings. In case of T.4, for instance, the rows and columns, which consist of 8 pixels each, are divided into two parts that are read and written sequentially. Each of these parts encompasses 4 pixels. The buffer size of the virtual memory channels has been chosen to the smallest power of 2 allowing for full throughput. The results clearly indicate that also for parallel data access the multidimensional FIFO can

be efficiently realized in hardware supporting very high clock frequencies. Furthermore, they demonstrate the benefit of trading throughput against resource requirements for out-of-order communication. In case of the transpose operation, for instance, the full throughput variant (T.8) requires between 3 and 22 times more resources than the slowest variant (T.1). Note that all multidimensional FIFOs have the same interface and can be automatically generated from the same WDF specification.

Table 8.3 presents the results for the same multidimensional FIFOs, but this time with disabled register balancing during synthesis. Obviously, this significantly decreases the amount of instantiated flip-flops. On the other hands, the achievable clock frequency gets smaller, since the synthesis tool does not try to move the flip-flops in such a way that all combinatorial paths have similar delay. Note that the shuffle FIFO S.1 deviates from this rule, where register balancing even decrements the achievable frequency. Here it seems that the synthesis tool has difficulties in estimating the final delays after place and route.

Table 8.3 Synthesis results for the transpose (T) and the shuffle (S) FIFO. The numbers in the table caption indicate the quantity of data elements written in parallel. **(a)** With register balancing during synthesis. **(b)** Without register balancing

(a)

	T. 1	T. 4	T. 8	S. 1	S. 4	S. 8
LUT (59904)	397	560	1,382	495	698	1,268
FF (59904)	608	1,104	1,830	647	1,067	1,597
BRAM (160)	1	6	22	6	9	17
MHz	398	355	339	337	356	352
# Parallel writes	1	4	8	1	4	8
# Parallel reads	1	4	8	1	3	6

(b)

	T. 1	T. 4	T. 8	S. 1	S. 4	S. 8
LUT (59904)	376	647	1,355	484	653	1,236
FF (59904)	354	582	486	380	492	507
BRAM (160)	1	6	22	6	9	17
MHz	359	324	255	369	311	260
# Parallel writes	1	4	8	1	4	8
# Parallel reads	1	4	8	1	3	6

In order to clarify the benefits of multidimensional modeling and communication synthesis, a further experiment has been performed, where the multidimensional transpose FIFO has been configured such that it fits to the Motion-JPEG decoder discussed in Chapter 4. Compared to Table 8.3, this leads to completely different implementation targets, because a rather low frequency of 50 MHz is sufficient while very huge bit widths with 32 bits per data element shall be used. The latter requirement essentially results from the fact that the high-level model uses integer variables.

As the case study in Chapter 4 uses a one-dimensional model of computation, the data reordering occurring for the transpose operation cannot be represented directly but has to be implemented in form of a CSDF actor (see Section 3.1.3.2) as depicted in Fig. 8.16. It has eight input and output FIFOs. Each input FIFO belongs to one column of the 8×8 block, each output FIFO to one row. In each of its eight phases, the actor reads one token from each input FIFO and writes all of them into one output FIFO. By means of this operation, the vertical

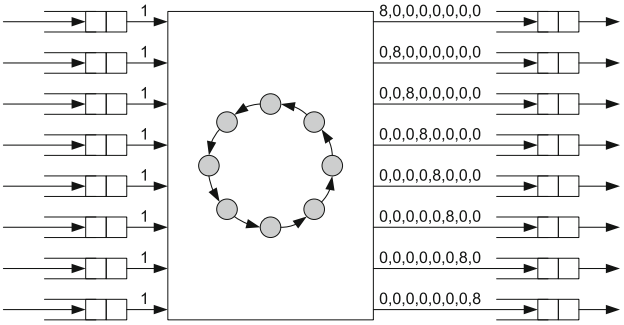


Fig. 8.16 One-dimensional transpose actor working on 8×8 blocks

IDCT can access the pixels of one column in parallel. Unfortunately, this operation pattern does not allow parallel processing of several pixels. In other words, an 8×8 block requires at least 64 clock cycles.

Table 8.4 summarizes the obtained synthesis results. The one-dimensional transpose actor has been translated automatically from the SystemoC code into an RTL implementation using the Cynthesizer behavioral compiler. As can be seen, one 8×8 block even takes 224 clock cycles, thus showing the difficulty to optimize the SystemoC code, which contains several FIFO accesses and address calculations. Consequently, even the slowest variant of the multidimensional transpose FIFO outperforms the one-dimensional actor in terms of throughput, because the multidimensional modeling permits aggressive optimization. Furthermore, it requires significantly less Block RAM, because it is able to store all data within one memory unit while the one-dimensional actor requires eight input and output FIFOs.

Table 8.4 Comparison of the multidimensional FIFO with an implementation derived from a one-dimensional model of computation. MD stands for multidimensional, whereas T.1 and T.4 define two transpose configurations investigated in Table 8.3

		MD (T.1)		MD (T.4)		MD (w#8, r#1)	
	1D actor	All	Serializer	All	Serializer	All	Serializer
LUT	1644	670	495 + 56	1111	394 + 141	407	56
FF	415	711	295 + 294	1011	389 + 388	386	294
BRAM	16	1	0	15	0	8	0
$\frac{\text{cycles}}{8\times8\,\text{block}}$	224	64	–	16	–	64	–

On the other hand, it can also be seen that the balanced configurations (T.1, T.4) defined in Table 8.3 need more flip-flops. There, the number of simultaneous reads and writes is identical. An analysis has shown that important parts of the resource consumption occur in the parallel-to-serial and serial-to-parallel converters. Although astonishingly in the first moment, it can be easily explained by the huge bit depths. Storing, for instance, one block column already requires $8 \times 32 = 256$ flip-flops. Consequently, Table 8.4 also depicts a non-balanced solution that writes eight data elements in parallel while reading them sequentially. Although this, of course, increases the required number of Block RAMs, it significantly reduces the necessary lookup tables and flip-flops, because the input serializer is economized. As a consequence, it outperforms the one-dimensional actor in both throughput and neces-

sary hardware resources. Since all of these variants can be generated automatically from the same communication specification, this clearly demonstrates the benefits of multidimensional modeling. However, it also shows the challenge of selecting the correct synthesis parameters.

8.5.4 Influence of Different Memory Channel Sizes

The examples presented so far used the same capacity for each virtual memory channel. Consequently, this section aims to investigate the impact of different memory channel sizes. For this purpose, a simple vertical downsampler as depicted in Fig. 8.17 shall be considered. According to Section 8.2.2, the number of required virtual memory channels $\mathbf{m_1}$ on edge e_1 amounts

$$\mathbf{m_1} = \text{scm}\,(\mathbf{p_1}, \mathbf{c_1}) = (1, 2)^T\,.$$

In case of equal channel sizes, an overall buffer of two image lines is required whereas otherwise only one line and a few pixels (depending on the pipeline depth) are sufficient. For this experiment, the corresponding value has been set to 8, in order to take the effect of pipelining into account.

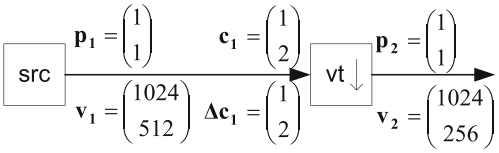


Fig. 8.17 Simple vertical downsampler

Table 8.5 shows the corresponding synthesis results for two different image widths and a bit depth of 16. In case of 1024 pixels, the obtained solutions are almost similar, except for the necessary Block RAMs. Whereas for identical sizes, each channel is mapped to one Block BRAM, this is not necessary anymore for different memory sizes, because the second virtual memory channel is so small that it can be placed into distributed logic. The overhead of multiple address generation, on the other hand, can be neglected. This is because for buffer sizes being powers of 2, the address offsets Δa as defined in Sections 8.2.3 and 8.2.6 remain the same whether identical memory channel sizes are used or not. Only the modulo-operations in Eqs. (8.11) and (8.18) have to be performed several times in case of different memory sizes. Furthermore, $\Delta\text{src}\left(\tilde{\mathbf{i}}_{\text{snk}}\right)$ as defined in Section 8.2.7.3 typically contain more nested condi-

Table 8.5 Synthesis results for the downsampler with equal and different virtual memory channel sizes

	1024		512	
Image width	Identical	Different	Identical	Different
LUT	215	221	134	163
FF	158	151	122	125
BRAM	2	1	1	1
MHz	350	392	458	420

tionals as solution of the more complex parametric integer programs (PIPs). However, both issues do not have severe impact on the required hardware resources as shown in Table 8.5.

Thus, whereas different sizes for the virtual memory channels are beneficial for an image width of 1024 pixels due to the reduced BRAM consumption, the situation changes for an image width of 512 pixels. In this case, the two image rows can be stored in the same Block RAM. Unfortunately, this is not possible anymore for different memory channel sizes. Consequently, the number of required RAM blocks does not reduce while the multidimensional FIFO needs more lookup tables due to the distributed storage of the second memory channel.

This, once again, shows the benefits of behavioral communication synthesis as introduced in this chapter, because both variants discussed above can be generated automatically from the same WSDF specification. However, it also illustrates the challenge of selecting the best implementation alternative.

Based on these results, the next section discusses how the basic downsampler functionality shown in Fig. 8.17 can be extended with data reuse in order to design more complex filters leading to superior image quality.

8.5.5 Combination with Data Reuse

The downsampler presented in Fig. 8.17 is very simple in that the used window height amounts only 2. However, since downsampling is a critical operation concerning the resulting image quality, typically larger window sizes are used, leading to overlapping windows.

Principally, such a downsampler can be implemented be extending the multidimensional hardware FIFO with support for overlapping windows. However, as already discussed in Fig. 8.1, this is out of scope for this book, since the problem is rather well investigated in literature. Instead, it will be shown how data reuse can be implemented independently of the inter-module communication. To this end, a vertical downsampler with a window size of 1×3 pixels has been described in *PARO* (see Section 3.2.7), which handles data reuse internally.

Figure 8.18a exemplifies the occurring window movement in vertical direction. The basic idea consists in dividing the sliding window into two parts as depicted in Fig. 8.18b. The lower part of size $\Delta\mathbf{c}$ reads the data elements generated by the source actor, whereas the remaining pixels of the sliding window only operate on reused data (or on extended border pixels). Consequently, such a downsampler can be realized by a module which reads $\Delta\mathbf{c}$ pixels for each invocation and handles the data reuse internally, leading directly to the graph depicted Fig. 8.17.

Figure 8.19 depicts the corresponding PARO code. Lines (4) and (5) declare the two inputs as illustrated in Fig. 8.18b. Lines (14) and (15) indicate how often the loop body defined by lines (16)–(28) has to be executed. The variable $g\left[x, y, \ldots\right]$ stands for the sliding window at position (x, y) and depends on whether it is situated on the extended border ($y = 0$) or not ($y > 0$). Line (26) finally calculates a simple mean filter in order to avoid artifacts occurring due to downsampling.

The so-described downsampler can now be synthesized using the PARO behavioral compiler. It generates a highly pipelined module that assumes FIFO communication for both its inputs and its outputs. As the multidimensional FIFO offers a FIFO like interface, both can be easily connected. Note that for odd image heights, also the lower image border has to be extended. Fortunately, this does not cause any difficulties, because both the multidimensional FIFO presented above and *PARO* support the corresponding operations.

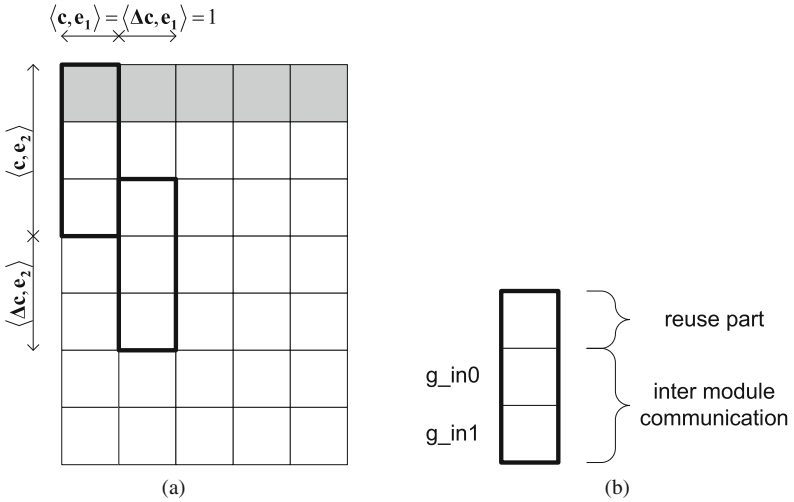


Fig. 8.18 Downsampler with data reuse. **a** Vertical downsampling with a 1×3 sliding window. **b** Division of the sliding window

```
1  program downsampler_filter_block {
2    ...
3    // input images
4    variable g_in0 2 in input_t;
5    variable g_in1 2 in input_t;
6
7    variable g 3          internal_t;
8    variable f out 2 out output_t;
9
10   // image size
11   parameter IMGSIZE_X = 1024;
12   parameter IMGSIZE_Y = 256;
13
14   pb_main: par (x >= 0 and x <= IMGSIZE_X-1 and
15                 y >= 0 and y <= IMGSIZE_Y-1) {
16
17      // inner image
18      g[x,y,0] = g_in1[x-1,y-1] if (y>0);
19      g[x,y,1] = g_in0[x-1,y  ] if (y>0);
20      g[x,y,2] = g_in1[x-1,y  ] if (y>0);
21
22      // upper border
23      g[x,y,0] = g_in1[x-1,y  ] if (y==0);
24      g[x,y,1] = g_in0[x-1,y  ] if (y==0);
25      g[x,y,2] = g_in1[x-1,y  ] if (y==0);
26      fout[x,y] = (1*g[x,y,0]+2*g[x,y,1]+1*g[x,y,2])/4;
27
28   }
29 }
```

Fig. 8.19 PARO code for vertical downsampler

8.5.6 Impact of Scheduling Granularity

All examples discussed so far in Section 8.5 employed fine-grained scheduling. In other words, for each produced data element, the multidimensional FIFO checks whether this enables additional sink invocations. Equivalently, each time a data element is read, the source fill level is updated. As explained in Section 8.4, this approach delivers the best possible latency and smallest possible memory size while avoiding the risk of deadlock. On the other hand, this leads to more complex nested if-then-else conditionals, whose impact shall be considered in this section.

To this end, Table 8.6a shows the corresponding synthesis results for the tiler discussed in Section 8.4.1. In order to increase the difference between the fine-grained and the coarse-grained scheduling alternative even further, the latter does not operate on tile granularity, but on image granularity. Nevertheless, both the coarse-grained and the fine-grained solution lead to a quasi-identical amount of lookup tables. Only the amount of flip-flops gets smaller when enabling register balancing during synthesis and when switching from fine-grained to coarse-grained scheduling. However, this difference also vanishes when disabling register balancing. Note, furthermore, that the attainable clock frequencies after place and route are almost identical when register balancing is enabled and disabled. Only the values estimated after synthesis show significant differences, but the Xilinx place and route tools cannot profit from these differences. In other words, there is no significant difference between fine-grained and coarse-grained scheduling in terms of hardware resource requirements. Since on the other hand, fine-grained scheduling offers various benefits such as reduced latency and memory requirements as well as deadlock avoidance, this demonstrates the benefit of the discussed multidimensional FIFO.

Table 8.6 Synthesis results for evaluation of the scheduling granularity. Two different frequency results report the estimated clock frequency after synthesis (Syn) and place and route (PR). **(a)** Tiler as depicted in Fig. 8.3. **(b)** Shuffle operation discussed in Fig. 8.2

(a)

Register balancing	Scheduling	FF	LUT	MHz (PR)	MHz (Syn)
On	Fine-grained	231	199	435	466
On	Coarse-grained	193	196	451	435
Off	Fine-grained	129	203	442	370
Off	Coarse-grained	125	199	455	393

(b)

Register balancing	Scheduling	FF	LUT	MHz (PR)	MHz (Syn)
On	Fine-grained	647	495	337	367
On	Coarse-grained	618	455	288	394
Off	Fine-grained	377	484	364	338
Off	Coarse-grained	355	443	323	344

In order to further investigate this observation, Table 8.6 shows the same experiment for the shuffle operation explained in Section 8.1, when only one data element can be written and read per clock cycle (W1/R1). It has been chosen since due to the three-dimensional token space the nested if-then-else conditionals are relatively complicated. For the coarse-grained scheduling approach, fill-level update is performed on granularity of macro block rows. In other words, only when the source has produced a complete row of macro blocks for each of

the three color components, the sink is informed about the existence of new data elements. Similarly, only when the sink has read all these data, the source fill level is updated.

From Table 8.6b it gets obvious that in this case even when register balancing is disabled, 5% less flip-flops and 8% less lookup tables are necessary for the coarse-grained scheduling. On the other hand, also the achievable clock frequency is reduced by 11%. Note that the estimated clock frequencies after synthesis show the inverse relation. Hence, again the place and route step have difficulties in achieving optimum results. With register balancing enabled, the differences between the two scheduling alternatives are in the same order of magnitude.

Hence, to sum up, depending on the application scenario, switching from coarse-grained to fine-grained scheduling can be beneficial. Nevertheless, the gains concerning required resources are rather small and might be paid by increased latency and memory requirements or even deadlocks (see Section 8.4).

8.6 Conclusion and Future Work

System implementation on a high level of abstraction encompasses several aspects, among others generation of the hardware accelerators for the different actors, instantiation of microprocessors, and generation of the communication infrastructure that allows exchanging data between the individual modules. Since image processing applications work on huge amounts of data, this communication infrastructure has to be highly efficient in terms of attainable throughput and clock frequencies. Moreover, in order to promote module reuse, the interface of the communication modules shall be such that it can interconnect preexisting hardware accelerators.

Typically, these tasks are solved by classical FIFO communication. However, the latter is not able to solve many important challenges in system level design of image processing applications, such as different read and write orders, parallel data access, or exploitation of tradeoffs between attainable throughput and different hardware resource types like Block RAMs, flip-flops, and lookup tables. Consequently, this chapter has described an alternative communication primitive, the so-called multidimensional FIFO. Its FIFO-like interface eases module reuse by dividing the overall system into a set of communicating actors. Furthermore, it takes care that the data are forwarded to the sink in the correct order, independently how the source actually produces them. This is possible by supporting both in-order communication and out-of-order communication with parallel data access. Although this leads to much more challenging address calculation and fill-level control compared to classical, one-dimensional FIFOs, it can be efficiently synthesized in hardware as demonstrated in this chapter. Additionally, the presented approach permits to automatically generate different implementation alternatives trading throughput against hardware resources. Together with the capability of supporting high clock frequencies and one read and write operation per clock cycle, the multidimensional FIFO thus represents a very powerful technique for design of high-speed image processing applications. Its usefulness has been demonstrated by several examples covering huge and small image sizes, medium and large bit depths, in-order and out-of-order communication, and parallel data access. A detailed comparison with results obtained from behavioral synthesis of one-dimensional models of computation demonstrated the benefit of multidimensional communication.

Although the multidimensional FIFO can also be used in manual system design, it profits particularly from the overall design methodology discussed in this monograph, because the latter offers powerful methods for modeling, simulation, and analysis. Especially the derived

buffer sizes can be used to parametrize the hardware communication primitive. Furthermore, since the multidimensional FIFO implements the WDF communication semantics, it can be directly used for hardware synthesis of multidimensional applications in the context of the SYSTEMCODESIGNER design flow or similar ESL tools.

As a consequence, the architecture presented in this chapter delivers important basics for future research. In particular, more sophisticated virtual channel mapping techniques, which have been voluntarily kept simple in this book, promise for further improvements in the obtained synthesis results. In case of the transpose operation, this could, for instance, result in a full throughput implementation needing less block RAMs. Furthermore, for applications with low-throughput requirements, it would be beneficial either to support the rectangular memory model or to perform a more sophisticated linearization that divides the image into several address spaces. By these means, scenario (8) in Section 6.4 could be synthesized more efficiently. Direct inclusion of data reuse techniques for overlapping windows would help in case the selected behavioral synthesis tool does not offer a corresponding support.

Chapter 9
Conclusion

Novel semiconductor devices offer more and more functionality per chip, which makes design of powerful systems possible. However, as discussed in Chapter 1, the inability of available design tools to handle the resulting complexity risks to break the long-standing trend of progress in semiconductor industry within the next 15 years. Consequently, new design methodologies are urgently required.

To this end, this monograph has investigated the system level design of image processing applications. As discussed in Chapter 2, they are characterized by the transport, storage, and manipulation of huge amounts of data and pronounced potential for parallelism. Furthermore, they require a combination of regular and static algorithms with highly data-dependent and control-oriented operations. Additionally, they employ complex data access patterns ranging from individual pixels, over sampling with overlapping windows to manipulation of complete images. These properties not only make manual implementation a challenging undertaken but also represent a hurdle for application of system level design tools, because they are required to generate highly optimized implementations.

9.1 Multidimensional System Design

Whereas there exist many industrial and academic tools and methodologies partly offering those capabilities, their combination into one uniform design methodology is still missing. In this context, multidimensional data flow seems to be a promising technique, because it combines the advantages of block-based specification, data flow-related system analysis, and polyhedral optimization on different levels of granularity. Since, however, the underlying techniques are still not very widespread in both industrial and academic design tools, this monograph aimed to give a corresponding overview by presenting a multidimensional design flow for high-performance image processing. By these means it has been possible to depict some of the inherent potential of this new field of research. Special attention has been put on out-of-order communication, multirate sliding window applications, high-speed communication synthesis, efficient system level analysis, and tight interaction with data-dependent application parts. By describing available solutions for all these aspects, this monograph aimed to contribute to their spreading in order to close the design gap between available semiconductor devices and manageable complexity. In particular, the book demonstrated that multidimensional data flow is able to combine the advantages of data flow models of computation with the achievements obtained from polyhedral analysis and synthesis. This includes intuitive representation of complex applications in a block-diagram-like structure containing sliding

J. Keinert, J. Teich, *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*, Embedded Systems,
DOI 10.1007/978-1-4419-7182-1_9, © Springer Science+Business Media, LLC 2011

window algorithms, border processing, out-of-order communication, parallel data access, and control flow. Furthermore, efficient system level verification and analysis in form of automatic buffer size calculation can be performed. Automatic synthesis of high-speed communication primitives allow to generate powerful implementations. In particular, support of out-of-order communication, parallel data access, high clock frequencies, and tradeoffs between required hardware resources and achievable throughput contribute to efficient behavioral synthesis techniques.

9.2 Discussed Design Steps and Their Major Benefits

In order to profit from the above-described benefits, the application has to be described by a multidimensional model of computation. To this end, this monograph introduced the so-called *windowed data flow (WDF)* and, its static counter part, the *windowed synchronous data flow (WSDF)*. Their major idea consists in modeling the processing of images via possibly overlapping windows. This enables fine-grained scheduling and thus memory-efficient implementation and delivers important information about parallel data access and occurring communication orders. Integration into the ESL tool SystemCoDesigner and usage of a finite state machine for communication control enable tight interaction with one-dimensional models of computation on pixel granularity. Consequently, this helps to represent complex applications like a JPEG2000 encoder, which contains both regular and static algorithms, such as the wavelet transform, as well as highly data-dependent operations like entropy encoding. Application to two different application scenarios, namely a lifting-based wavelet transform including image tiling, block building, and resource sharing, as well as a binary morphological reconstruction, demonstrated that the described multidimensional model of computation is a powerful base also for applications exceeding simple rectangular sliding windows.

Since WDF explicitly models point, local, and global algorithms including sampling with overlapping windows and out-of-order communication, it makes the resulting expensive memory buffers for intermediate data storage directly available to analysis tools. Consequently, the designer can be supported in determining the required buffer sizes. This is highly desirable as demonstrated in Chapter 7 by means of a multi-resolution filter employed in medial image processing, because the manual solution is laborious and error prone. To this end, this monograph compared two different memory organization schemes, namely a rectangular buffer model and linearization in production order. Whereas both offer advantages and drawbacks, application to different examples demonstrated that for high-performance parallel execution, linearization in production order leads to superior results. Based on these results, a polyhedral method for automatic buffer size determination has been discussed in Chapter 7. Special care has been taken to handle out-of-order communication, up- and downsampling including the resulting scheduling alternatives, and throughput-optimized scheduling. Usage of efficient graph algorithms permits to process graph topologies including feed-back loops and split and joins of data paths. In particular, they avoid creation of a huge integer linear program whose size depends on the number of graph actors. This has shown to be important, since the required integer linear programs for exact solution risk to get computationally very expensive. Consequently, in addition to the usage of powerful ILP solvers such as *ILOG CPLEX*, an alternative solution strategy has been described in this book, which bases on exhaustive search with linear complexity. This two-folded strategy permits to exploit the advantages of both approaches. Whereas direct solution of the integer linear programs typically results in shorter analysis times, the exhaustive search avoids the failure of the buffer

analysis algorithm in case the ILP gets intractable. Application to different examples demonstrated that in both cases the analysis times permit an application to huge data flow graphs and are competitive compared to alternative approaches. Furthermore, the experiments demonstrated the effectiveness of WDF to model complex image processing algorithms like the lifting-based wavelet transform.

Once system analysis is terminated, the next step consists in generation of a highly efficient hardware–software implementation. While behavioral compilers are able to synthesize individual modules and multi-processor systems can be generated using SYSTEMCODESIGNER and its automatic design space exploration, this monograph has focused on high-performance communication in hardware. To this end, the so-called *multidimensional FIFO* has been discussed. It is able to interconnect two different modules by a FIFO-like interface independently of the used read and write order. As a consequence, it significantly simplifies the module reuse, which is considered important for overcoming the complexity trap mentioned in Chapter 1. Furthermore, since the multidimensional FIFO matches the communication semantics introduced by WDF, it represents an important prerequisite for refinement of WDF specifications into working hardware implementations. Support of high clock frequencies, one read and write operation per clock cycle, pipelined in-order and out-of-order communication, parallel data access, and the capability to trade required hardware resources against attainable throughput guarantees that WDF graphs can be translated into high-performance implementations. This could be demonstrated by means of several experiments including large and small image sizes, different bit depths, and varying communication scenarios. Comparison with behavioral synthesis applied to a one-dimensional model of computation demonstrated that the multidimensional FIFO is able to perform communication not only much faster but also with less hardware resources.

Appendix A
Buffer Analysis by Simulation

Chapter 6 has presented two different memory mapping functions for WDF communication edges and showed the basic principles for determination of the required buffer sizes. This chapter aims to provide the details of a buffer analysis method that can be employed during simulation. To this end, Section A.1 is dedicated to the rectangular memory model while Section A.2 discusses linearization in production order. Section A.3 finally describes the simulation techniques used for sequential ASAP and self-timed parallel scheduling of an individual WDF edge.

A.1 Efficient Buffer Parameter Determination for the Rectangular Memory Model

In order to be valid, both mapping functions defined in Section 6.3 require that the buffer parameters are chosen such that two live data elements are never mapped to the same memory cell. In other words, the mapping parameters $\mathbf{m_e}$, respectively, B_e have to be chosen accordingly. For this purpose, the graph can be simulated using a given schedule that decides when to execute which actor. Monitoring the set of live data elements then permits to choose the correct mapping parameters. Since, however, the set of live data elements might be pretty huge when processing large images, efficient techniques have to be used for this task. This section consequently presents a corresponding method for the rectangular memory mapping function.

A.1.1 Monitoring of Live Data Elements

Due to the WDF communication semantics introduced in Section 5.1, the required edge memory can only increase with token production. Hence, each time the source actor generates some output data elements, it has to be checked whether the edge buffer must be enlarged or not for storage of these new data elements. Consequently, after each write operation $\mathbf{I}_{\text{src}}$ of the source actor the set of live data elements $\mathscr{L}(\mathbf{I}_{\text{src}})$ and the vector $\mathbf{m}(\mathbf{I}_{\text{src}})$ have to be determined, such that the memory mapping function $\sigma(\mathbf{g_e}) = \mathbf{g_e} \bmod \mathbf{m}(\mathbf{I}_{\text{src}})$ assigns to each live data element $\mathbf{g_e} \in \mathscr{L}(\mathbf{I}_{\text{src}})$ a unique memory cell.

Doing so without profiting from special properties of the *hierarchical line-based (HLB)* communication order defined in Section 5.3 would lead to very bad run-time behavior,

J. Keinert, J. Teich, *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*, Embedded Systems,
DOI 10.1007/978-1-4419-7182-1, © Springer Science+Business Media, LLC 2011

because for each produced effective token the complete token space has to be scanned for live data elements. This is particularly true, when large image sizes shall be used. The following sections thus present a method how buffer analysis can be achieved more efficiently for the HLB communication order.

Since its mathematical notation for token spaces with arbitrary dimensions is quite lengthy and complex to understand and since the method becomes computational expensive for more than two dimensions, this chapter is restricted to the two-dimensional case, which permits illustrative explanations.

A.1.2 Table-Based Buffer Parameter Determination

As already discussed in Section 6.3.1, determination of $\mathbf{m_e}(\mathbf{I}_{\text{src}})$ can be realized by the so-called *successive moduli technique*, calculating one component of $\mathbf{m_e}(\mathbf{I}_{\text{src}})$ after the other. Given the set of live data elements $\mathscr{L}(\mathbf{I}_{\text{src}})$ after the source invocation $\mathbf{I}_{\text{src}}$, this can be achieved by the help of the following formulas:

$$\langle \mathbf{m_e}(\mathbf{I}_{\text{src}}), \mathbf{e_2} \rangle = 1 + \max\{\langle \Delta\mathbf{l}, \mathbf{e_2} \rangle \mid \langle \Delta\mathbf{l}, \mathbf{e_1} \rangle = 0\}, \tag{A.1}$$

$$\langle \mathbf{m_e}(\mathbf{I_{src}}), \mathbf{e_1} \rangle = 1 + \max\{\langle \Delta\mathbf{l}, \mathbf{e_1} \rangle\}, \tag{A.2}$$

where $\Delta\mathbf{l} = \mathbf{l_2} - \mathbf{l_1}$ and $\mathbf{l_1}, \mathbf{l_2} \in \mathscr{L}(\mathbf{I}_{\text{src}})$. Alternatively, the order by which the components of $\mathbf{m_e}$ are calculated can be exchanged as discussed in Section 6.3.1.

Application of these techniques during graph simulation results in one vector $\mathbf{m_e}(\mathbf{I}_{\text{src}})$ per write invocation. Since the different components of $\mathbf{m_e}(\mathbf{I}_{\text{src}})$ are determined independently of each other, the overall memory parameter $\mathbf{m_e}$ is given by the component-wise maximum:

$$\mathbf{m_e} = \max_{\mathbf{I}_{\text{src}}} (\mathbf{m_e}(\mathbf{I}_{\text{src}})). \tag{A.3}$$

Unfortunately, calculation of both Eq. (A.1) and (A.2) requires to scan all live data elements in order to determine the corresponding maximum. However, this leads to a bad runtime behavior. In order to solve this problem, Fig. A.1 explains a table-based approach.

It uses for both coordinates $\mathbf{e_1}$ and $\mathbf{e_2}$ a maximum and a minimum table $\mathscr{T}^{\mathbf{e_i}}_{\max}$ and $\mathscr{T}^{\mathbf{e_i}}_{\min}$ ($i = 1, 2$) holding the current largest and smallest coordinates of the live data elements:

$$\begin{aligned}
\mathscr{T}^{\mathbf{e_1}}_{\max}(y) &= \max\{\langle \mathbf{l}, \mathbf{e_1} \rangle \mid \mathbf{l} \in \mathscr{L}(\mathbf{I_{src}}) \wedge \langle \mathbf{l}, \mathbf{e_2} \rangle = y\}, \\
\mathscr{T}^{\mathbf{e_1}}_{\min}(y) &= \min\{\langle \mathbf{l}, \mathbf{e_1} \rangle \mid \mathbf{l} \in \mathscr{L}(\mathbf{I_{src}}) \wedge \langle \mathbf{l}, \mathbf{e_2} \rangle = y\}, \\
\mathscr{T}^{\mathbf{e_2}}_{\max}(x) &= \max\{\langle \mathbf{l}, \mathbf{e_2} \rangle \mid \mathbf{l} \in \mathscr{L}(\mathbf{I_{src}}) \wedge \langle \mathbf{l}, \mathbf{e_1} \rangle = x\}, \\
\mathscr{T}^{\mathbf{e_2}}_{\min}(x) &= \min\{\langle \mathbf{l}, \mathbf{e_2} \rangle \mid \mathbf{l} \in \mathscr{L}(\mathbf{I_{src}}) \wedge \langle \mathbf{l}, \mathbf{e_1} \rangle = x\}.
\end{aligned}$$

The content of these tables is updated each time a data element is produced or read the last time (consumed) as described later on in Sections A.1.3 and A.1.4. Based on this information it is possible to efficiently determine the required buffer sizes. The latter can only increase due to token production. Hence, in accordance with Eqs. (A.1) and (A.3), the currently required buffer size in dimension $\mathbf{e_2}$ can be obtained by calculating for each produced data element $\mathbf{q}$:

$$\langle \mathbf{m}, \mathbf{e_2} \rangle = \max(\langle \mathbf{m}, \mathbf{e_2} \rangle, \langle \tilde{m}(\mathbf{q}), \mathbf{e_2} \rangle),$$

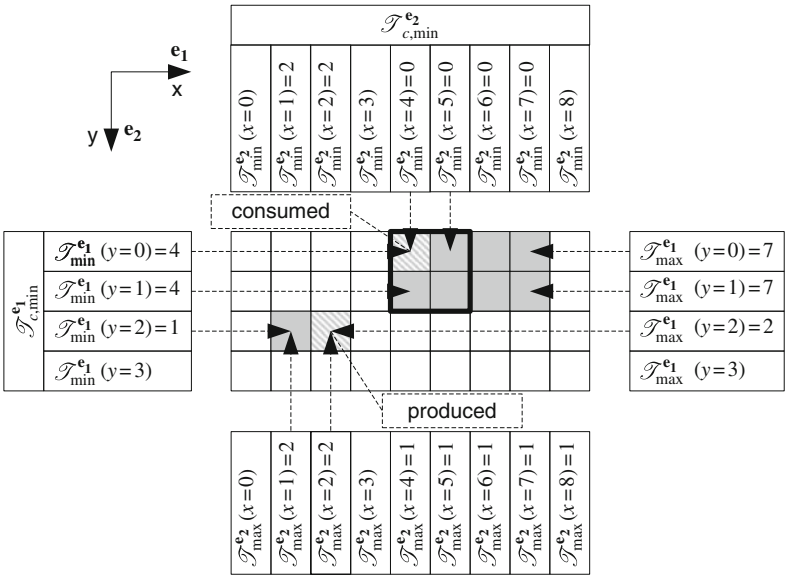


Fig. A.1 Table-based buffer parameter determination. Live data elements are *gray shaded*

with

$$\langle \tilde{m}\,(\mathbf{q})\,, \mathbf{e_2} \rangle = 1 + \mathscr{T}^{\mathbf{e_2}}_{\max}\left(\langle \mathbf{q}, \mathbf{e_1} \rangle\right) - \mathscr{T}^{\mathbf{e_2}}_{\min}\left(\langle \mathbf{q}, \mathbf{e_1} \rangle\right).$$

q stands for the coordinates of the considered produced data element. In other words, for each produced data element, only one comparison and two additive operations are required in order to update the buffer quantity in dimension $\mathbf{e_2}$. For the situation given in Fig. A.1, only one data element has been produced. Hence, the update of the required buffer size is given by $\langle \mathbf{m}, \mathbf{e_2} \rangle = \max\left(\langle \mathbf{m}, \mathbf{e_2} \rangle, 1 + 2 - 2\right) = 2$.

For determination of $\langle \mathbf{m}, \mathbf{e_1} \rangle$, such an approach is not sufficient anymore. In contrast to the calculation of $\langle \mathbf{m}, \mathbf{e_2} \rangle$, the relevant table entry cannot be derived from the coordinates **q**, because calculation of $\langle \mathbf{m}\,(\mathbf{I}_{\text{src}})\,, \mathbf{e_1} \rangle$ requires an exhaustive search in dimension $\mathbf{e_2}$:

$$\langle \mathbf{m}\,(\mathbf{I}_{\text{src}})\,, \mathbf{e_1} \rangle = \max\left\{\langle \mathbf{l_1}, \mathbf{e_1} \rangle \mid \mathbf{l_1} \in \mathscr{L}\,(\mathbf{I}_{\text{src}})\right\} - \min\left\{\langle \mathbf{l_2}, \mathbf{e_1} \rangle \mid \mathbf{l_2} \in \mathscr{L}\,(\mathbf{I}_{\text{src}})\right\}. \tag{A.4}$$

In order to avoid this exhaustive search, the algorithm further can keep track of the following variables:

$$\mathscr{T}^{\mathbf{e_i}}_{c,\min} = \min\left\{\langle \mathbf{l}, \mathbf{e_i} \rangle \mid \mathbf{l} \in \mathscr{L}\,(\mathbf{I}_{\text{src}})\right\},$$

$$\mathscr{T}^{\mathbf{e_i}}_{c,\max} = \max\left\{\langle \mathbf{l}, \mathbf{e_i} \rangle \mid \mathbf{l} \in \mathscr{L}\,(\mathbf{I}_{\text{src}})\right\}.$$

Algorithm 8 Modification of the minimum tables due to token production

```
(00) for each produced data element g_e
(01)    T^{e1}_min(⟨g_e, e2⟩) = min(T^{e1}_min(⟨g_e, e2⟩), ⟨g_e, e1⟩)
(02)    T^{e2}_min(⟨g_e, e1⟩) = min(T^{e2}_min(⟨g_e, e1⟩), ⟨g_e, e2⟩)
(03)    T^{e2}_{c,min} = min(T^{e2}_{c,min}, ⟨g_e, e2⟩)
(04)    if (T^{e2}_{c,min} = ⟨g_e, e2⟩)
(05)       T^{e2}_{c,min} is valid
(06)    end if
(07)    T^{e1}_{c,min} = min(T^{e1}_{c,min}, ⟨g_e, e1⟩)
(08)    if (T^{e1}_{c,min} = ⟨g_e, e1⟩)
(09)       T^{e1}_{c,min} is valid
(10)    end if
(11) end for
```

By these means, Eq. (A.4) simply gets

$$\langle \mathbf{m}(\mathbf{I}_{\text{src}}), \mathbf{e_1} \rangle = \mathscr{T}^{\mathbf{e_1}}_{c,\max} - \mathscr{T}^{\mathbf{e_1}}_{c,\min}, \tag{A.5}$$

where the derivation of $\mathscr{T}^{\mathbf{e_i}}_{c,\min}$ and $\mathscr{T}^{\mathbf{e_i}}_{c,\max}$ is described in the following sections.

A.1.3 Determination of the Minimum Tables

During the execution of the edge source and sink actors, the minimum tables are influenced by both token production and token consumption. Whereas the latter reduces the number of live data elements and therefore possibly increases the values of the minimum tables, token production adds new live data elements, which might reduce the values of the minimum tables.

Algorithm 8 describes the impact of the token production on the minimum tables. Lines (04)–(06) and (08)–(10) are only relevant in combination with Algorithm 9 and will consequently be described later on. The meaning of the remaining lines are illustrated by means of Fig. A.2. The data elements labeled with "2" are live, whereas those labeled with "1" are already consumed by the sink. The newly produced data element, labeled with "3," has the smallest coordinate in dimension $\mathbf{e_1}$ and hence influences the corresponding minimum tables $\mathscr{T}^{\mathbf{e_1}}_{\min}$ and $\mathscr{T}^{\mathbf{e_1}}_{c,\min}$. Alternatively, the source could also produce data element "4" modifying thus $\mathscr{T}^{\mathbf{e_2}}_{\min}$ and $\mathscr{T}^{\mathbf{e_2}}_{c,\min}$.

The influence of the token consumption on the minimum tables is given in Algorithm 9, which works for zero or positive border extension. The determination which data elements are read the last time can be easily performed due to the properties of the *hierarchical line-based (HLB)* communication order. Suppose, for instance, that the window in Fig. A.1 propagates by 1 pixel in horizontal and vertical directions ($\Delta\mathbf{c} = (1, 1)^T$), the pixel situated in the upper left corner of the window will not be read by future actor invocations.

Whereas $\mathscr{T}^{\mathbf{e_i}}_{\min}$ is directly influenced by these consumed tokens, the situation is more complex for $\mathscr{T}^{\mathbf{e_i}}_{c,\min}$. This is exemplarily depicted in Fig. A.3 for $\mathscr{T}^{\mathbf{e_2}}_{c,\min}$. First of all, the latter is only allowed to change if the consumed token is actually situated in the row $\mathscr{T}^{\mathbf{e_2}}_{c,\min}$.

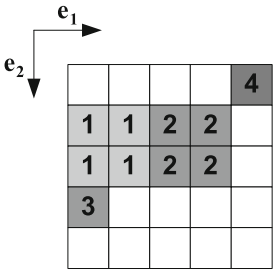


Fig. A.2 Modification of the minimum table due to token production. The data elements labeled with "2" are live, whereas those labeled with "1" are already consumed by the sink. The newly produced data element, labeled with "3," is the most left one and hence influences the corresponding minimum table

Algorithm 9 Modification of the minimum tables due to token consumption

```
(00) for each consumed data element g_e (in raster scan order)
(01)   T^{e1}_min(⟨g_e, e_2⟩) = max(T^{e1}_min(⟨g_e, e_2⟩), ⟨g_e, e_1⟩ + 1)
(02)   T^{e2}_min(⟨g_e, e_1⟩) = max(T^{e2}_min(⟨g_e, e_1⟩), ⟨g_e, e_2⟩ + 1)
(03)   if (⟨g_e, e_1⟩ = T^{e1}_{c,min} ∧ ⟨g_e, e_2⟩ = T^{e2}_max(⟨g_e, e_1⟩))
(04)     T^{e1}_{c,min} = ⟨g_e, e_1⟩ + 1
(05)     if (T^{e2}_min(T^{e1}_{c,min}) > T^{e2}_max(T^{e1}_{c,min}))
(06)       //Column empty
(07)       T^{e1}_{c,min} not valid
(08)     end if
(09)   end if
(10)   if (⟨g_e, e_2⟩ = T^{e2}_{c,min} ∧ ⟨g_e, e_1⟩ = T^{e1}_max(⟨g_e, e_2⟩))
(11)     T^{e2}_{c,min} = ⟨g_e, e_2⟩ + 1
(12)     if (T^{e1}_min(T^{e2}_{c,min}) > T^{e1}_max(T^{e2}_{c,min}))
(13)       //Row empty
(14)       T^{e2}_{c,min} not valid
(15)     end if
(16)   end if
(17) end for
```

Furthermore, it must be the last data element in this row. Both conditions are checked in line (10) of Algorithm 9. While they are fulfilled in Fig. A.3b, this is not the case in Fig. A.3a. Consequently, $\mathcal{T}^{\mathbf{e_2}}_{c,\min}$ is not changed.

Lines (12)–(15) correspond to a special case that is depicted in Fig. A.4. It assumes that the currently read data element is the last one situated in the row $\mathcal{T}^{\mathbf{e_2}}_{c,\min}$. Consequently, all other live data elements must be placed below the bold dashed line. However, it is not guaranteed that they are directly placed below this dashed line, but there might be a larger distance as depicted in Fig. A.4. In this case, line (11) of Algorithm 9 calculates an invalid value for $\mathcal{T}^{\mathbf{e_2}}_{c,\min}$.

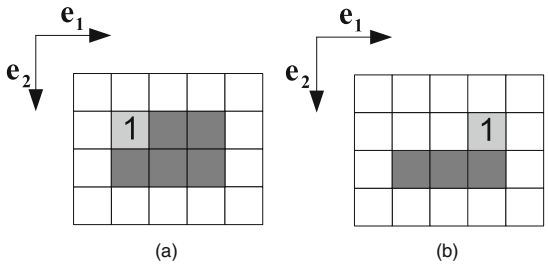


Fig. A.3 Two different token consumption scenario for illustration of their influence on the minimum table $\mathscr{T}^{\mathbf{e_2}}_{c,\min}$. The live data elements are shaded by a *gray color*. That one being read the last time is labeled by a 1

Fortunately, this is not a problem due to the properties of the HLB communication order. It ensures that the sink will consume next one (or several) of the crossed data elements.[1] In other words, before the sink can execute next, there will be a source invocation that produces one of the crossed data elements. As a consequence, $\mathscr{T}^{\mathbf{e_2}}_{c,\min}$ will either become correct automatically or it will be corrected by Algorithm 8. Until this occurs, $\mathscr{T}^{\mathbf{e_i}}_{c,\max}$ can only increase. Consequently, when $\mathscr{T}^{\mathbf{e_i}}_{c,\min}$ becomes valid, Eq. (A.5) can be used to calculate the correct buffer size.

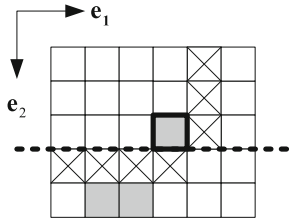


Fig. A.4 Invalid value for $\mathscr{T}^{\mathbf{e_2}}_{c,\min}$. Live data elements are *gray shaded*. The *bold* frame indicates the currently consumed data element. The *crossed data* elements are those that can be consumed by the sink next due to the properties of the HLB communication order

A.1.4 Determination of the Maximum Tables

Compared to the previous section, determination of the maximum tables $\mathscr{T}^{\mathbf{e_i}}_{\max}$ is simpler in that they are only influenced by token production. Furthermore, using the following corollaries, $\mathscr{T}^{\mathbf{e_i}}_{c,\max}$ can be derived directly from $\mathscr{T}^{\mathbf{e_i}}_{\max}$ and $\mathscr{T}^{\mathbf{e_i}}_{c,\min}$ thanks to the properties of the *hierarchical line-based (HLB)* communication order.

Corollary A.1 *Let $a, b, q \in \mathbb{Z}$. Then the following holds:*

[1] In case of negative virtual border extension, the crossed data elements also might have a larger distance to the data element with the bold border.

$$\left\lfloor \frac{a}{q} \right\rfloor = \left\lfloor \frac{b}{q} \right\rfloor \Rightarrow a \bmod q - b \bmod q = a - b.$$

Proof $a \bmod q - b \bmod q = a - \left\lfloor \frac{a}{q} \right\rfloor \times q - b + \left\lfloor \frac{b}{q} \right\rfloor \times q = a - b.$

Corollary A.2 *Let $a, k, q \in \mathbb{Z}$. Then the following holds:*

$$(a \bmod (k \times q)) \bmod q = a \bmod q.$$

Proof

$$\begin{aligned}
(a \bmod (k \times q)) \bmod q &= \left(a - \left\lfloor \frac{a}{k \times q} \right\rfloor \times k \times q\right) \bmod q \\
&= \left(a - \left\lfloor \frac{a}{k \times q} \right\rfloor \times k \times q\right) - \left\lfloor \frac{a - \left\lfloor \frac{a}{k \times q} \right\rfloor \times k \times q}{q} \right\rfloor \times q \\
&= \left(a - \left\lfloor \frac{a}{k \times q} \right\rfloor \times k \times q\right) - \left\lfloor \frac{a}{q} \right\rfloor \times q + \left\lfloor \frac{a}{k \times q} \right\rfloor \times k \times q = a \bmod q.
\end{aligned}$$

Corollary A.3 *Let $\mathbf{I_0}$ be a read or write operation. Then for all read or write operations $\mathbf{I} \in \mathbb{N}_0^{n_e}$ with $\langle \mathbf{I}, \mathbf{e_j} \rangle \leq \langle \mathbf{I_0}, \mathbf{e_j} \rangle \, \forall 1 \leq j \leq n_e$ the following holds:*

$$\theta_e(\mathbf{I}) \preceq \theta_e(\mathbf{I_0}),$$

where θ_e maps flat to hierarchical iteration vectors as defined in Definition 6.3.

Proof Since by definition $\langle \mathbf{I}, \mathbf{e_j} \rangle < \langle \mathbf{B_q}, \mathbf{e_j} \rangle$ for each dimension $\forall 1 \leq j \leq n_e$, the following holds ($k = q - 1$):

$$\begin{aligned}
& \langle \mathbf{I}, \mathbf{e_j} \rangle \leq \langle \mathbf{I_0}, \mathbf{e_j} \rangle \\
\Rightarrow \left\lfloor \frac{\langle \mathbf{I}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_q}, \mathbf{e_j} \rangle}{\langle \mathbf{B_{q-1}}, \mathbf{e_j} \rangle} \right\rfloor &= \left\lfloor \frac{\langle \mathbf{I}, \mathbf{e_j} \rangle}{\langle \mathbf{B_{q-1}}, \mathbf{e_j} \rangle} \right\rfloor \leq \left\lfloor \frac{\langle \mathbf{I_0}, \mathbf{e_j} \rangle}{\langle \mathbf{B_{q-1}}, \mathbf{e_j} \rangle} \right\rfloor = \left\lfloor \frac{\langle \mathbf{I_0}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_q}, \mathbf{e_j} \rangle}{\langle \mathbf{B_{q-1}}, \mathbf{e_j} \rangle} \right\rfloor \\
\Rightarrow & \langle \mathbf{i}, \mathbf{e_{n_e - j+1}} \rangle \leq \langle \mathbf{i_0}, \mathbf{e_{n_e - j+1}} \rangle.
\end{aligned}$$

If at least for one j $\langle \mathbf{i}, \mathbf{e_{n_e - j+1}} \rangle < \langle \mathbf{i_0}, \mathbf{e_{n_e - j+1}} \rangle$, then $\theta_e(\mathbf{I}) \prec \theta_e(\mathbf{I_0})$. Otherwise application of Corollary A.1 leads to

$$\begin{aligned}
& \left(\left(\langle \mathbf{I}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_q}, \mathbf{e_j} \rangle\right) \bmod \langle \mathbf{B_{q-1}}, \mathbf{e_j} \rangle\right) \\
- & \left(\left(\langle \mathbf{I_0}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_q}, \mathbf{e_j} \rangle\right) \bmod \langle \mathbf{B_{q-1}}, \mathbf{e_j} \rangle\right) \\
= & \left(\langle \mathbf{I}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_q}, \mathbf{e_j} \rangle\right) - \left(\langle \mathbf{I_0}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_q}, \mathbf{e_j} \rangle\right) \leq 0.
\end{aligned}$$

With Corollary A.2 and Definition 5.8, this becomes

$$\left(\langle \mathbf{I}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_{q-1}}, \mathbf{e_j} \rangle\right) - \left(\langle \mathbf{I_0}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_{q-1}}, \mathbf{e_j} \rangle\right) \leq 0.$$

Algorithm 10 Modification of maximum tables due to token production

```
for each produced data element g_e
  T^{e1}_max(⟨g_e, e2⟩) = max(T^{e1}_max(⟨g_e, e2⟩), ⟨g_e, e1⟩)
  T^{e2}_max(⟨g_e, e1⟩) = max(T^{e2}_max(⟨g_e, e1⟩), ⟨g_e, e2⟩)
end for
```

From $a \leq b \Rightarrow \lfloor a \rfloor \leq \lfloor b \rfloor$ it follows that

$$\left\lfloor \frac{\left(\langle \mathbf{I}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_{q-1}}, \mathbf{e_j} \rangle\right)}{\langle \mathbf{B_{q-2}}, \mathbf{e_j} \rangle} \right\rfloor \leq \left\lfloor \frac{\left(\langle \mathbf{I_0}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_{q-1}}, \mathbf{e_j} \rangle\right)}{\langle \mathbf{B_{q-2}}, \mathbf{e_j} \rangle} \right\rfloor.$$

If at least for one $j \leq$ can be replaced by $<$, then $\theta_e(\mathbf{I}) \prec \theta_v(\mathbf{I_0})$.

Otherwise Corollary A.1 can be applied again resulting in

$$\begin{aligned} & \left(\langle \mathbf{I}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_{q-1}}, \mathbf{e_j} \rangle\right) \bmod \langle \mathbf{B_{q-2}}, \mathbf{e_j} \rangle \\ - & \left(\langle \mathbf{I_0}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_{q-1}}, \mathbf{e_j} \rangle\right) \bmod \langle \mathbf{B_{q-2}}, \mathbf{e_j} \rangle \\ = & \left(\langle \mathbf{I}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_{q-1}}, \mathbf{e_j} \rangle\right) - \left(\langle \mathbf{I_0}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_{q-1}}, \mathbf{e_j} \rangle\right) \leq 0. \end{aligned}$$

Application of Corollary A.2 and of Definition 5.8 leads to

$$\left\lfloor \frac{\left(\langle \mathbf{I}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_{q-2}}, \mathbf{e_j} \rangle\right)}{\langle \mathbf{B_{q-3}}, \mathbf{e_j} \rangle} \right\rfloor \leq \left\lfloor \frac{\left(\langle \mathbf{I_0}, \mathbf{e_j} \rangle \bmod \langle \mathbf{B_{q-2}}, \mathbf{e_j} \rangle\right)}{\langle \mathbf{B_{q-3}}, \mathbf{e_j} \rangle} \right\rfloor.$$

This can be continued until $k = 0$.

In other words, having executed a write operation $\mathbf{I}_{\text{src},\mathbf{0}}$, Corollary A.3 says that all write operations $\mathbf{I}_{\text{src}}$ with $\langle \mathbf{I}_{\text{src}}, \mathbf{e_j} \rangle \leq \langle \mathbf{I}_{\text{src},\mathbf{0}}, \mathbf{e_j} \rangle \, \forall 1 \leq j \leq n_e$ must have been executed too. The same holds for the read operations. Consequently, this property can be used to derive $\mathscr{T}^{\mathbf{e_i}}_{c,\max}$ directly from $\mathscr{T}^{\mathbf{e_i}}_{\max}$ and $\mathscr{T}^{\mathbf{e_i}}_{c,\min}$:

$$\begin{aligned} \mathscr{T}^{\mathbf{e_1}}_{c,\max} &= \mathscr{T}^{\mathbf{e_1}}_{\max}\left(\mathscr{T}^{\mathbf{e_2}}_{c,\min}\right), \\ \mathscr{T}^{\mathbf{e_2}}_{c,\max} &= \mathscr{T}^{\mathbf{e_2}}_{\max}\left(\mathscr{T}^{\mathbf{e_1}}_{c,\min}\right). \end{aligned}$$

This is shown exemplarily in Fig. A.5 for the value $\mathscr{T}^{\mathbf{e_1}}_{c,\max}$. If both data elements "B" and "C" are live, then also "A" must be live: "A" must have been produced, if "B" has been produced, and "A" will be read in future, if "C" will be read in future, supposing that each data element will be read at least once.[2] Consequently, only $\mathscr{T}^{\mathbf{e_1}}_{\max}$ has to be monitored during execution of the edge actors leading to Algorithm 10.

[2] If this is not the case, it leads to a super-approximation for the required buffer space.

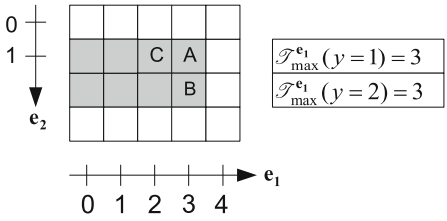


Fig. A.5 Illustration of the search space reduction

A.1.5 Complexity

For each produced and consumed data element, it is necessary to update the table values. Furthermore, after each data element production a check must be performed whether the required buffer space increases or not. All these operations can be performed by simple comparisons without performing any exhaustive search in form of loops. Hence, with z being the number of produced data elements during one schedule period, the buffer analysis technique has a complexity of $\mathscr{O}(z)$ for two-dimensional problems.

Considering the required buffer space for storing the tables, it can be seen that $\mathscr{T}^{\mathbf{e_1}}_{\min}$ and $\mathscr{T}^{\mathbf{e_1}}_{\max}$ theoretically require unbounded buffer space due to the infinite size of the token space. Fortunately, the number of table entries that must be valid simultaneously is limited by the number of simultaneously live data element lines. Hence, it is possible to work with finite tables when using simple modulo-addressing. Since the table size, however, is not known a priori, the implementation can start with an arbitrary value. If during simulation it comes out that these sizes are not sufficient, the corresponding tables are copied into larger ones. By usage of an efficient pointer data structure, the expense of these copy operations can be kept very small.

A.2 Efficient Buffer Parameter Determination for the Linearized Buffer Model

In comparison with the rectangular buffer model, determination of the buffer parameters for the linearized buffer model is much simpler, because it only offers one degree of freedom in form of $B_e \in \mathbb{N}$. Its value has to be chosen such that two live data elements are not mapped to the same memory cell. The following corollary helps in achieving this goal.

Corollary A.4 *Given two numbers $z_1, z_2 \in \mathbb{Z}$ and $n \in \mathbb{N}$. Then the following holds:*

$$0 < |z_1 - z_2| < n \Rightarrow z_1 \bmod n \neq z_2 \bmod n.$$

Proof Let $z_2 = z_1 + \Delta z$. Then

$$\begin{aligned} & z_2 \bmod n - z_1 \bmod n \\ =& z_1 + \Delta z - \left\lfloor \frac{z_1 + \Delta z}{n} \right\rfloor \times n - z_1 + \left\lfloor \frac{z_1}{n} \right\rfloor \times n \end{aligned}$$

$$= n \times \left(\frac{\Delta z}{n} + \left\lfloor \frac{z_1}{n} \right\rfloor - \left\lfloor \frac{z_1 + \Delta z}{n} \right\rfloor \right).$$

Since

$$0 < |\Delta z| < n, \quad \left\lfloor \frac{z_1}{n} \right\rfloor - \left\lfloor \frac{z_1 + \Delta z}{n} \right\rfloor \in \{0, -1\}.$$

As $0 < \frac{\Delta z}{n} < 1$, $z_2 \bmod n - z_1 \bmod n \neq 0$.

In other words, it is sufficient to choose

$$\begin{aligned} B_e(t) &= \max \left\{ \left| \Theta_e\left(\mathbf{g^h_{e,1}}\right) - \Theta_e\left(\mathbf{g^h_{e,2}}\right) \right| \mid \mathbf{g^h_{e,1}}, \mathbf{g^h_{e,2}} \in \mathscr{L}^h(t) \right\} \\ &= \max \left\{ \Theta_e\left(\mathbf{g^h_{e,1}}\right) \mid \mathbf{g^h_{e,1}} \in \mathscr{L}^h(t) \right\} - \min \left\{ \Theta_e\left(\mathbf{g^h_{e,2}}\right) \mid \mathbf{g^h_{e,2}} \in \mathscr{L}^h(t) \right\}, \end{aligned}$$

where $\mathscr{L}^h(t)$ contains the hierarchical data element identifiers of all live data elements (see also Section 6.3.2) at time t:

$$\mathscr{L}^h(t) = \{\mathbf{g^h_e} \in G^h_e \mid \mathbf{g^h_e} \text{ is live at time } t\}.$$

Together with Corollary 6.9, this leads to

$$\begin{gathered} B_e(t) = \Theta_e\left(\max_{\prec} \mathscr{L}^h(t)\right) - \Theta_e\left(\min_{\prec} \mathscr{L}^h(t)\right) + 1 \\ B_e = \max_t B_e(t). \end{gathered} \tag{A.6}$$

Determination of $\max_{\prec} \mathscr{L}^h(t)$ is very easy, because due to the linearization in production order, it is simply the last produced data element. The value of $\min_{\prec} \mathscr{L}^h(t)$ identifies the live data element that has been produced earliest. However, due to out-of-order communication, this does not necessarily correspond to the data element to consume next. This can, for instance, be seen in Fig. A.6, which illustrates an exemplary live data element distribution for a JPEG2000 tiling operation (see Section 2.2). In the assumed scenario, data element 21 will be consumed next, but data element 16 is the lexicographically smallest live data element.

Thus, more complex methods are required for the determination of $\min_{\prec} \mathscr{L}^h(t)$. Whereas scanning of the token space would be a possible solution, its run-time is far too high. Instead alternative techniques like efficient tree structures can be employed as described in the following section.

A.2.1 Tree Data Structure for Tracking of Live Data Elements

Due to the special properties of the *hierarchical line-based (HLB)* communication order scheme, $\min_{\prec} \mathscr{L}^h(t)$ can be efficiently calculated by means of a tree data structure. It has $n_e \times \left(q_e^{\text{src}} + 1\right)$ levels and is illustrated in Fig. A.7a. The symbol n_e corresponds to the number of token dimensions while q_e^{src} represents the number of write firing levels as explained in Definition 5.8. Each node on level $1 \leq i \leq n_e \times \left(q_e^{\text{src}} + 1\right)$ is identified by an $(i-1)$-tuple $(k_1, \ldots, k_{i-1})$, with $\langle \min_{\mathbf{g^h_e} \in G^h_e} \left(\mathbf{g^h_e}\right), \mathbf{e_j} \rangle \leq k_j \leq \langle \max_{\mathbf{g^h_e} \in G^h_e} \left(\mathbf{g^h_e}\right), \mathbf{e_j} \rangle$,

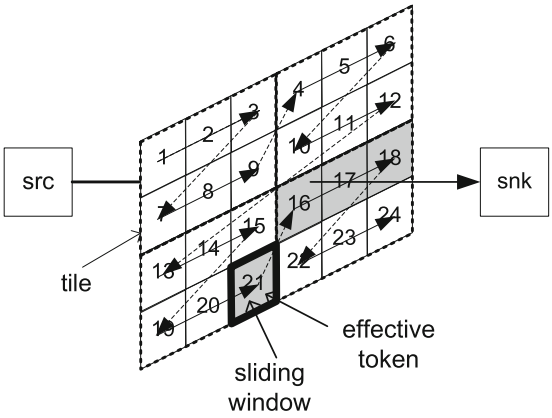


Fig. A.6 Exemplary live data element distribution for a JPEG2000 tiling operation. *Arabic numbers* correspond to the production order while *arrows* indicate the read order. *Gray shaded rectangles* represent the live pixels that have to be buffered at least in memory for the shown window positions

whereas the empty tuple $(\cdot)$ belongs to the root node. The parent of node $(k_1, \ldots, k_{i-1})$ is $(k_1, \ldots, k_{i-2})$. Element k_i of such a tuple is associated with the ith coordinate of the hierarchical data element identifiers defined in Section 6.3.2. Taking, for instance, the edge shown in Fig. A.6, node (0) is associated with the first row of image tiles, node (0, 0) with the first tile in this row of tiles, node (0, 0, 0) with the first row of this tile.

The value of a node on level i at time t corresponds to the smallest coordinate $\langle \mathbf{g_e^h}, \mathbf{e_i} \rangle$ of all live data elements $\mathbf{g_e^h} \in \mathscr{L}^h(t)$, in dependency of the smaller tree levels $1, \ldots, i-1$:

$$\begin{gathered} \text{value}\left((k_1, \ldots, k_{i-1}), t\right) = \\ \min\left(\{\langle \mathbf{g_e^h}, \mathbf{e_i} \rangle \mid \mathbf{g_e^h} \in \mathscr{L}_i^h(t)\} \cup \{\min_{\mathbf{g_e^h} \in G_e^h} \langle \mathbf{g_e^h}, \mathbf{e_i} \rangle\}\right) \\ \mathscr{L}_i^h(t) := \{\mathbf{g_e^h} \in \mathscr{L}^h(t) \mid \langle \mathbf{g_e^h}, \mathbf{e_j} \rangle = k_j,\ j = 1 \ldots i-1\}. \end{gathered} \tag{A.7}$$

In other words, the value of a node $(k_1, \ldots, k_{i-1})$ on level i identifies the smallest coordinate $\langle \mathbf{g_e^h}, \mathbf{e_i} \rangle$ of all live data elements for which the hierarchical coordinates are set to $\langle \mathbf{g_e^h}, \mathbf{e_j} \rangle = k_j$ for all $1 \leq j < i$. Hence, for the example above, value $((0), t)$ specifies the first live image tile in the first row of image tiles and value $((0, 0), t)$ identifies the first live row in the very first image tile.

A.2.2 Determination of the Lexicographically Smallest Live Data Element

The following lemma shows, how the tree introduced in Section A.2.1 helps to determine the lexicographic minimum of all live data elements.

Lemma A.5 *Let* $\{k_1, \ldots, k_{n_e \times (q_e^{\text{src}}+1)}\}$ *be a set, such that*

Algorithm 11 Update of the tree data structure for determination of $\min_{\prec} \mathscr{L}^h(t)$

```
01 η = ( ⟨g_e^h, e_1⟩, ..., ⟨g_e^h, e_{n_e × (q_e^src + 1) − 1}⟩ )
02 for(i=n_e × (q_e^src + 1) downto 1){
03    value(η) = value(η) + 1
04    if (value(η) > max_{γ_e^h ∈ G_e^h} ⟨γ_e^h, e_i⟩){
05       value(η) = min_{γ_e^h ∈ G_e^h} ⟨γ_e^h, e_i⟩ //init for next run.
06       η ← parent(η) //move to parent
07    }else break;
08 }
```

$$k_i = \text{value}\left(\left(k_1, \ldots, k_{i-1}\right), t\right).$$

Then $\langle \min_{\prec} \mathscr{L}^h(t), \mathbf{e_i} \rangle = k_i$.

Proof By construction, $k_1 = \text{value}((\cdot))$ is the smallest possible value $\langle \mathbf{g_e^h}, \mathbf{e_1} \rangle$ of all live data elements $\mathbf{g_e^h} \in \mathscr{L}^h(t)$; $k_2 = \text{value}((k_1))$ is the smallest possible value $\langle \mathbf{g_e^h}, \mathbf{e_2} \rangle$ under the condition that $\langle \mathbf{g_e^h}, \mathbf{e_1} \rangle = k_1$; and so on. This leads directly to the definition of the lexicographic minimum.

Lemma A.5 says that $\min_{\prec} \mathscr{L}^h(t)$ can be determined by simply traversing the tree, starting from the root. Arrived at node $\left(k_1, \ldots, k_{i-1}\right)$, the next child to visit is $\left(k_1, \ldots, k_{i-1}, \text{value}\left(\left(k_0, \ldots, k_{i-1}\right)\right)\right)$. This proceeding is illustrated in Fig. A.7a, showing parts of the tree belonging to the edge depicted in Fig. A.6. The first two levels identify the lexicographically smallest image tile that contains at least one live data element. The values of the third level nodes equal the smallest row number of an image tile containing a live data element, etc. Levels 5 and 6 are omitted, because their node values are always zero. The shown node values correspond to the live data element distribution given in Fig. A.7c.

The above-described tree traversal leads to the bold path and hence to

$$\min_{\prec} \mathscr{L}^h(t) = (0, 0, 1, 2, \ldots)^T,$$

which identifies the striped data element.

A.2.3 Tree Update

Due to the linearization in production order, $\min_{\prec} \mathscr{L}^h(t)$ can only change due to consumption. Each time a data element is read the last time, it will not be live anymore. Hence the tree must be updated, because $\min_{\prec} \mathscr{L}^h(t)$ can increase. The update must be performed in such a way that Eq. (A.7) is always true.

Algorithm 11 shows the corresponding algorithm when HLB communication orders are used. It starts from a tree leaf and moves toward the root. η stands for the currently processed tree node and $\mathbf{g_e^h}$ for the data element that is read the last time. If several data elements are read the last time, then they have to be processed in lexicographic order.

For illustration of the algorithm, suppose the live data element distribution shown in Fig. A.7c belonging to the edge depicted in Fig. A.6. The striped data element shall be read

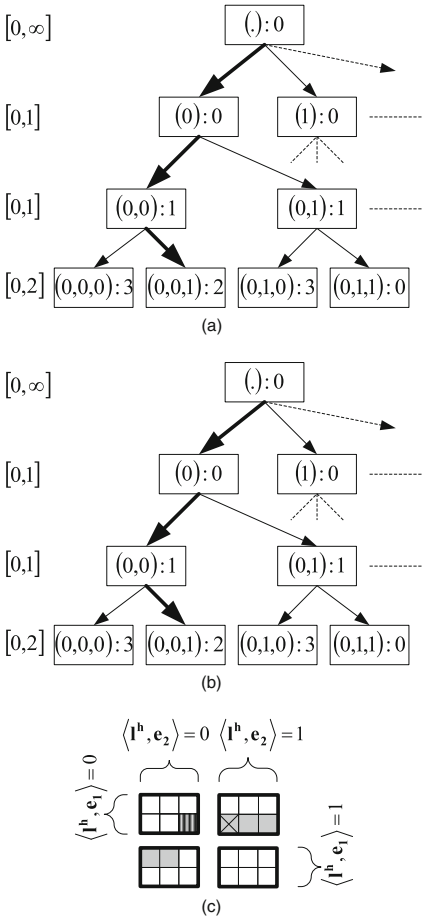


Fig. A.7 **a** Data structure for determination of $\min_{\prec} \mathscr{L}^h(t)$ before the update. $(0, 0) : 1$ is a short notation for value $((0, 0), t) = 1$. The intervals indicate the possible node values for each level. **b** Data structure for determination of $\min_{\prec} \mathscr{L}^h(t)$ after the update. **c** Corresponding token space. Live data elements are *gray hatched*

the last time. If Algorithm 11 arrives at the tree node $(0, 0, 1)$, its value is increased by 1. Since $3 > 2$, it is detected by means of line (04) that the second line of the first image tile is completely processed. Hence, the algorithm moves to the parent node and increases its value. Once again line (04) indicates that also the whole upper left image tile has been processed and the algorithm moves to node (0). Its value is increased by 1, in order to indicate that the lexicographically smallest data element is now expected in the upper right image tile. Note that this is always true due to the special properties of the HLB communication order, because the latter makes it impossible for the sink to read the crossed data element before the striped one. Hence, the last line of the upper right image tile must be live (as in case of Fig. A.7c) or will become live. Furthermore, $\min_{\prec} \mathscr{L}^h(t)$ can never decrease.

Figure A.7b shows the tree after its update. It now indicates that the data element $\mathbf{g_e^h} = (0, 1, 1, 0, \ldots)$, marked by a cross in Fig. A.7c, is the lexicographically smallest live data element. If $\langle \Delta\mathbf{c_e}, \mathbf{e_i} \rangle > \langle \mathbf{c_e}, \mathbf{e_i} \rangle$, then some data elements are never read but simply skipped. In this case, Algorithm 11 can be applied, nevertheless, by calling it for both the read and the skipped data elements.

A.2.4 Complexity of the Algorithm

The required memory size for the algorithm is dominated by the tree data structure. Since $\langle \mathbf{g_e^h}, \mathbf{e_1} \rangle$ is not bounded, in principle an infinite tree must be stored. Fortunately, only the nodes describing simultaneously live tokens must be hold at the same time in the memory. This leads to a tree size in the order of the number of simultaneously live data elements.

The computational complexity of this algorithm comprises two parts: tree establishment and determination of the buffer parameters. The complexity of the tree establishment is in the order of tree nodes. Because an edge buffer can only increase with the token production, the buffer size B_e is updated with each invocation of the source actor. For each consumption, the tree must be updated. Hence, the overall complexity is $\mathcal{O}(z)$, z being the number of produced and consumed data elements during simulation.

A.3 Stimulation by Simulation

The actually required buffer size heavily depends on the graph schedule, hence the time when the different actors of the WDF edge are executed. Whereas all analytic methods discussed in Section 3.3.2 are restricted to affine or at most piecewise affine scheduling functions, the above presented approach can be applied to arbitrary scheduling strategies. In other words, it is not required that the actor execution time is an affine function of the iteration vectors. Instead, schedule determination and buffer analysis is performed simultaneously by simulation. This allows to easily employ different scheduling criteria, as, for instance, *self-timed parallel execution* and *sequential ASAP (as soon as possible)* scheduling with resource sharing. The self-timed parallel execution is more hardware related where an actor fires as soon as enough input data are available. The sequential ASAP scheduling is more software oriented where different actors have to share the same processing resource. As a consequence, the different actors are influencing each other due to resource conflicts.

Figure A.8 illustrates the simulation flow used for schedule determination and buffer analysis of a single WDF edge. Three different cycles can be distinguished, depending whether only the sink, the source, or both actors should be fired. This decision depends not only on the available data elements but also on the scheduling strategy. For instance, if a sequential ASAP schedule is applied, it is never possible to fire both the source and the sink actor. For the self-timed parallel schedule, the decision whether to execute an actor or not principally depends on the availability of the required data elements as well as on the fact, whether an actor is still occupied with a previous invocation or not.

Each time the source actor produces some data, the maximum buffer size is determined as described in Sections A.2 and A.1. In case of the rectangular buffer model, this requires a preceding update of the table structures as discussed in Algorithms 8 and 10. After each read operation performed by the sink actor, only the corresponding data structures are updated as defined by Algorithm 9 for the rectangular buffer model and Algorithm 11 in case of

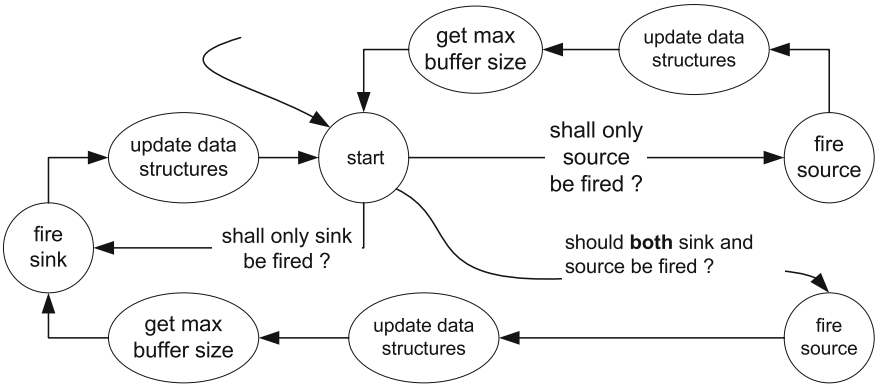


Fig. A.8 Simulation state diagram for a single WDF edge

linearized buffer organization. However, the required buffer size is not updated as it cannot increase. When both actors fire, the buffer determination is realized before the execution of the sink, in order to take into account that the data elements accessed by the source and the sink must be simultaneously available.

Appendix B
Abbreviations

ASAP	As Soon As Possible (Scheduling)
ASIC	Application-Specific Integrated Circuit
BB	Block Builder
BDF	Boolean-Controlled Data Flow
BLDF	Blocked Data Flow
BRAM	Block RAM
CAM	Context-Addressable Memory
CDFG	Control Data Flow Graph
CRP	Communicating Regular Processes
CSDF	Cyclo-Static Data Flow
CSP	Communicating Sequential Processes
DDF	Dynamic Data Flow
DDR	Double Data Rate (memory)
DSE	Design Space Exploration
DSP	Digital Signal Processor
ESL	Electronic System Level
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
FRDF	Fractional Rate Data Flow
FSL	Fast Simplex Link
HLB	Hierarchical Line Based (communication order)
HSDF	Homogeneous Synchronous Data Flow
IDCT	Inverse Discrete Cosine Transform
IDF	Integer-Controlled Data Flow
II	Initiation Interval
ILP	Integer Linear Program
KPN	Kahn Process Networks
LUT	Lookup Table
MDSDF	Multidimensional Synchronous Data Flow
MoC	Model of Computation
MOEA	Multi-Objective Evolutionary Algorithm
MPSoC	Multi-Processor Systems on Chip
NoC	Network on Chip
NPA	Non-Programmable Accelerator
NRE	Non-Recurring Engineering (costs)

PE	Processing Element
PPM	Portable Pixmap
PSDF	Parameterized Synchronous Data Flow
PSF	Progressive Segmented Frame
QDR	Quad Data Rate (memory)
RAM	Random Access Memory
RTL	Register Transfer Level
TLM	Transaction-Level Model
SDF	Synchronous Data Flow
SoC	System on Chip
SPDF	Synchronous Piggybacked Data Flow
SPM	Scratch Pad Memory
UML	Unified Modeling Language
USM	Unified Specification Model
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WCET	Worst-Case Execution Time
WDF	Windowed Data Flow
WSDF	Windowed Synchronous Data Flow

Appendix C
Formula Symbols

$\langle \mathbf{a}, \mathbf{b} \rangle$	scalar or dot product between vectors $\mathbf{a}$ and $\mathbf{b}$
$\mathbf{e_i}$	Cartesian base vector in dimension i, also called elementary vectors
$\mathbf{a} \prec \mathbf{b}$	vector $\mathbf{a} \in \mathbb{R}^n$ is lexicographically smaller than vector $\mathbf{b} \in \mathbb{R}^n$. In other words, $\exists i : \forall 1 \leq j < i,\ \langle \mathbf{b}, \mathbf{e_j} \rangle \geq \langle \mathbf{a}, \mathbf{e_j} \rangle \wedge \langle \mathbf{b}, \mathbf{e_i} \rangle > \langle \mathbf{a}, \mathbf{e_j} \rangle$
$\mathbf{a} \underset{\leftarrow}{\prec} \mathbf{b}$	vector $\mathbf{a} \in \mathbb{R}^n$ is anti-lexicographically smaller than vector $\mathbf{b} \in \mathbb{R}^n$. In other words, $\exists i : \forall i < j \leq n,\ \langle \mathbf{b}, \mathbf{e_j} \rangle \geq \langle \mathbf{a}, \mathbf{e_j} \rangle \wedge \langle \mathbf{b}, \mathbf{e_i} \rangle > \langle \mathbf{a}, \mathbf{e_j} \rangle$
$\mathbf{a} \succ 0$	vector $\mathbf{a} \in \mathbb{R}^n$ is lexicographically positive
$\mathbf{a} > \mathbf{b}$	component-wise comparison of two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$: $\mathbf{a} > \mathbf{b} \Leftrightarrow \forall 1 \leq i \leq n : \langle \mathbf{a}, \mathbf{e_i} \rangle > \langle \mathbf{b}, \mathbf{e_i} \rangle$
$\mathbf{p}$	size of (effective) token produced at invocation of a multidimensional actor
$\mathbf{c}$	size of (sliding) window consumed at invocation of a multidimensional actor
$\Delta\mathbf{c}$	movement of sliding window in each dimension
$\boldsymbol{\delta}$	initial data elements
$\mathbf{b^s}$, $\mathbf{b^t}$	virtual border extension
$\mathbf{D}$, $\mathbf{d}$	dependency vectors
$\mathbf{i}_{\text{src}}$, $\mathbf{i}_{\text{snk}}$	hierarchical iteration vectors
$\mathbf{I}_{\text{src}}$, $\mathbf{I}_{\text{snk}}$	flat iteration vectors
n	number of token dimensions
$\text{src}\,(e)$	source actor of edge e
$\text{snk}\,(e)$	sink actor of edge e
$\text{scm}\,(x, y)$	smallest common multiple of x and y
$\text{scm}\,(X)$	smallest common multiple of all elements $x \in X$. If $X = \emptyset$, then we define $\text{scm}\,(X) = 1$
$\gcd\,(x, y)$	greatest common divisor of x and y
$\mathbb{N}$	set of natural numbers not including zero
$\mathbb{N}_0$	$\mathbb{N}_0 = \mathbb{N} \cup \{0\}$
$\text{num}\left(\frac{a}{b}\right)$	numerator of a fraction: $\text{num}\left(\frac{a}{b}\right) = a,\ a, b \in \mathbb{Z}$
$\text{denom}\left(\frac{a}{b}\right)$	denominator of a fraction: $\text{denom}\left(\frac{a}{b}\right) = b,\ a, b \in \mathbb{Z}$
$\lfloor x \rfloor$	floor function, returning the greatest integer not exceeding x: $$\lfloor x \rfloor \in \mathbb{Z},\ 0 \leq x - \lfloor x \rfloor < 1$$
$\lceil x \rceil$	ceiling function, returning the smallest integer not exceeded by x: $$\lceil x \rceil \in \mathbb{Z},\ 0 \leq \lceil x \rceil - x < 1$$

$A \in \Re^{m,n}$	matrix with m rows and n columns
$\prod_{i=1}^{n} f(i)$	$f(1) \times \cdots \times f(n)$. Note that $\prod_{i=1}^{0} f(i) = 1$
$\|\mathbf{x}\|_\infty$	maximum norm of $\mathbf{x} \in \Re^n$, $\|\mathbf{x}\|_\infty = \max_{1 \le i \le n} \{\langle \mathbf{x}, \mathbf{e_i} \rangle\}$

References

1. AutoPilot. http://www.autoesl.com/products.html
2. Graphviz - graph visualization software. http://www.graphviz.org/
3. lp_solve. http://lpsolve.sourceforge.net/5.5/
4. The Omega project. http://www.cs.umd.edu/projects/omega/
5. Piplib. http://www.piplib.org/
6. International technology roadmap for semiconductors – design. Tech. rep., International Technology Roadmap for Semiconductors (2007)
7. Absar, J., Catthoor, F.: Reuse analysis of indirectly indexed arrays. ACM Trans. Des. Autom. Electron. Syst. **11**(2), 282–305 (2006)
8. Acharya, T.: VLSI algorithms and architectures for JPEG2000. Ubiquity **7**(35), 1–42 (2006)
9. Adé, M.: Data memory minimization for synchronous dataflow graphs emulated on DSP-FPGA targets. Ph.D. thesis, Katholieke Universiteit Leuven (1996)
10. Adé, M., Lauwereins, R., Peperstraete, J.: Buffer memory requirements in DSP applications. In: Proceedings of the 5th International Workshop on Rapid System Prototyping, pp. 108–123. Grenoble, France (1994)
11. Adé, M., Lauwereins, R., Peperstraete, J.A.: Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets. In: DAC '97: Proceedings of the 34th Annual Conference on Design Automation, pp. 64–69. ACM Press, New York, NY, (1997)
12. Agility: SC Compiler. http://www.europractice.stfc.ac.uk/vendors/agility_sc_datasheet_01000_hq_screen.pdf
13. Agrawal, A., Bakshi, A., Davis, J., Eames, B., Ledeczi, A., Mohanty, S., Mathur, V., Neema, S., Nordstrom, G., Prasanna, V., Raghavendra, C., Singh, M.: MILAN: A model based integrated simulation framework for design of embedded systems. In: Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2001), pp. 82–93. Snowbird, UT, (2001)
14. Aho, E., Vanne, J., Hamalainen, T.: Parallel memory architecture for arbitrary stride accesses. In: Proceedings of the 2006 IEEE Design and Diagnostics of Electronic Circuits and systems, pp. 63–68. Prague, Czech Republic (2006)
15. Ambler, S.W.: The Elements of UML(TM) 2.0 Style. Cambridge University Press, New York, NY (2005)
16. Ashenden, P.J.: The Designer's Guide to VHDL. Morgan Kaufmann Publishers, San Francisco, CA (1991)
17. Azizi-Mazreah, A., Manzuri-Shalmani, M.T., Barati, H., Barati, A.: Delay and energy consumption analysis of conventional SRAM. In: Proceedings of World Academy of Science, Engineering and Technology, vol. 27, pp. 35–39. Paris, France (2008)
18. Balarin, F., Chiodo, M., Hsieh, H., Jureska, A., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A., Sentovich, E., Suzuki, K., Tabbara, B.: Hardware-Software Co-design of Embedded System: The POLIS Approach. Kluwer, Norwell, MA (1997)

19. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.: Metropolis: An integrated electronic system design environment. Computer **36**(4), 45–52 (2003)
20. Balasa, F., Catthoor, F., Man, H.D.: Dataflow-driven memory allocation for multi-dimensional signal processing systems. In: ICCAD '94: Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design, pp. 31–34. IEEE Computer Society Press, Los Alamitos, CA, (1994)
21. Balasa, F., Kjeldsberg, P.G., Palkovic, M., Vandecappelle, A., Catthoor, F.: Loop transformation methodologies for array-oriented memory management. In: ASAP '06: Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors, pp. 205–212. IEEE Computer Society, Washington, DC, (2006)
22. Balasa, F., Zhu, H., Luican, I.I.: Computation of storage requirements for multi-dimensional signal processing applications. IEEE Trans. Very Large Scale Integr. Syst. **15**(4), 447–460 (2007)
23. Banerjee, P., Shenoy, N., Choudhary, A., Hauck, S., Bachmann, C., Haldar, M., Joisha, P., Jones, A., Kanhare, A., Nayak, A., Periyacheri, S., Walkden, M., Zaretsky, D.: A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In: FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, p. 39. IEEE Computer Society, Washington, DC, (2000)
24. Baradaran, N., Diniz, P.: Exploiting data reuse in modern FPGAs: Opportunities and challenges for compilers. In: International Workshop on Applied Reconfigurable Computing (ARC2005), pp. 1–10. Algarve, Portugal (2005). Keynote Lecture
25. Baradaran, N., Diniz, P.C.: A register allocation algorithm in the presence of scalar replacement for fine-grain configurable architectures. In: DATE '05: Proceedings of the conference on Design, Automation and Test in Europe, pp. 6–11. IEEE Computer Society, Washington, DC, (2005)
26. Baradaran, N., Diniz, P.C.: Memory parallelism using custom array mapping to heterogeneous storage structures. In: International Conference on Field Programmable Logic and Applications (FPL), pp. 1–6. Madrid, Spain (2006)
27. Baradaran, N., Diniz, P.C., Park, J.: Extending the applicability of scalar replacement to multiple induction variables. In: Languages and Compilers for High Performance Computing, vol. 3602, pp. 455–469. Springer, New York, NY (2005)
28. Baradaran, N., Park, J., Diniz, P.C.: Compiler reuse analysis for the mapping of data in FPGAs with RAM blocks. In: Proceedings of IEEE International Conference on Field-Programmable Technology (FPT), pp. 145–152 (2004)
29. Baumstark, L., Guler, M., Wills, L.: Extracting an explicitly data-parallel representation of image-processing programs. In: Proceedings of 10th Working Conference on Reverse Engineering (WCRE), pp. 24–34. Victoria, B.C., Canada (2003)
30. Baumstark, L.B., Wills, L.M.: Retargeting sequential image-processing programs for data parallel execution. IEEE Trans. Softw. Eng. **31**(2), 116–136 (2005)
31. Beierlein, T., Fröhlich, D., Steinbach, B.: Model-driven compilation of UML-models for reconfigurable architectures. In: 2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES '04). Toronto, Canada (2004)
32. Bekooij, M., Wiggers, M., van Meerbergen, J.: Efficient buffer capacity and scheduler setting computation for soft real-time stream processing applications. In: SCOPES '07: Proceedings of the 10th International Workshop on Software & Compilers for Embedded Systems, pp. 1–10. ACM Press, New York, NY (2007)
33. Benkrid, K., Crookes, D., Smith, J., Benkrid, A.: High level programming for FPGA based image and video processing using hardware skeletons. In: The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01), pp. 219–226. Rohnert Park, Canada (2001)
34. Bergeron, J., Cerny, E., Hunter, A., Nightingale, A.: Verification Methodology Manual for SystemVerilog. Springer, New York, NY (2005)
35. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Sci. Comput. Programming **19**, pp. 87–152 (1992)

36. Beux, S.L.: Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'ingénierie dirigée par les modèles. Ph.D. thesis, Université des Sciences et Technologies de Lille (2007)
37. Beux, S.L., Marquet, P., Dekeyser, J.L.: A design flow to map parallel applications onto FPGAs. In: 17th IEEE International Conference on Field Programmable Logic and Applications (FPL2007), pp. 605–608. Amsterdam, The Netherlands (2007)
38. Beux, S.L., Marquet, P., Dekeyser, J.L.: Multiple abstraction views of FPGA to map parallel applications. In: Reconfigurable Communication-centric SoCs 2007 (ReCoSoC'07), pp. 90–97. Montpellier, France (2007)
39. Bhattacharya, B.: Parameterized modeling and scheduling for dataflow graphs. Master's thesis, Department of Electrical and Computer Engineering, University of Maryland (1999)
40. Bhattacharya, B., Bhattacharyya, S.: Parameterized dataflow modeling of DSP systems. IEEE Trans. Signal Process. **49**(10), 2408–2421 (2001)
41. Bhattacharya, B., Bhattacharyya, S.S.: Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems. In: Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000), p. 84. IEEE Computer Society, Washington, DC (2000)
42. Bhattacharyya, S., Murthy, P., Lee, E.: APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. In: Design Automation for Embedded Systems, pp. 33–60. Kluwer Academic Publishers, Boston, MA (1997)
43. Bijlsma, T., Bekooij, M.J.G., Smit, G.J.M., Jansen, P.G.: Efficient inter-task communication for nested loop programs on a multiprocessor system. In: Proceedings of the ProRISC 2007 Workshop, pp. 122–127. Utrecht, Technology Foundation, Veldhoven, The Netherlands (2007)
44. Bijlsma, T., Bekooij, M.J.G., Smit, G.J.M., Jansen, P.G.: Omphale: Streamlining the communication for jobs in a multi processor system on chip. Technical Report TR-CTIT-07-44, University of Twente, The Netherlands, Enschede (2007)
45. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.: Cyclo-static dataflow. IEEE Trans. Signal Process. **44**(2), 397–408 (1996)
46. BINACHIP: BINACHIP. http://www.binachip.com/products.htm (2010)
47. Boulet, P.: Array-OL revisited, multidimensional intensive signal processing specification. Tech. Rep. 6113v2, Unité de recherche INRIA Futurs Parc Club Orsay Université, ZAC des Vignes, 4, rue Jacques Monod, 91893 ORSAY Cedex (France) (2007)
48. Boulet, P., Marquet, P., Éric Piel, Taillard, J.: Repetitive allocation modeling with MARTE. In: Forum on Specification and Design Languages (FDL'07). Barcelona, Spain (2007). Invited paper
49. Buck, J., Ha, S., Lee, E.A., Messerschmitt, D.G.: Ptolemy: A framework for simulating and prototyping heterogenous systems. Int. J. Comput. Simulat. **4**(2), 155–182 (1994)
50. Buck, J.T.: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. thesis, University of California at Berkeley (1993)
51. Buck, J.T.: Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In: 28th Asilomar Conference on Signals, Systems, and Computers. Pacific Grove, CA (1994)
52. Budiu, M.: Spatial computation. Ph.D. thesis, Carnegie Mellon School of Computer Science, Pittsburgh, PA (2003)
53. Budiu, M., Venkataramani, G., Chelcea, T., Goldstein, S.C.: Spatial computation. SIGOPS Oper. Syst. Rev. **38**(5), 14–26 (2004)
54. Bukhari, K.Z., Kuzmanov, G., Vassiliadis, S.: DCT and IDCT implementations on different FPGA technologies. In: Proceedings of ProRISC 2002, pp. 232–235. Veldhofen, Netherlands (2002)
55. Calvez, J.P., Perrier, V.: MPEG-2 encoder-decoder illustrative example. Tech. rep., CoFluent Design (2005)
56. Caspi, E.: Design automation for streaming systems. Ph.D. thesis, Electrical Engineering and Computer Sciences, University of California at Berkeley (2005)

57. Catthoor, F., Danckaert, K., Kulkarni, K., Brockmeyer, E., Kjeldsberg, P., Achteren, T., Omnes, T.: Data Access and Storage Management for Embedded Programmable Processors. Springer, New York, NY (2002)
58. Catthoor, F., Danckaert, K., Wuytack, S., Dutt, N.D.: Code transformations for data transfer and storage exploration preprocessing in multimedia processors. IEEE Des. Test **18**(3), 70–82 (2001)
59. Catthoor, F., Franssen, F., Wuytack, S., Nachtergaele, L., De Man, H.: Global communication and memory optimizing transformations for low power signal processing systems. In: Workshop on VLSI Signal Processing **VII**, pp. 178–187. IEEE Press, La Jolla, CA (1994)
60. Catthoor, F., Wuytack, S., Greef, E.D., Balasa, F., Nachtergaele, L., Vandecappelle, A.: Custom Memory Management Methodology – Exploration of Memory Organisation for Embedded Multimedia System Design. Kluwer, Boston, MA (1998)
61. Celoxica: Handel-C language reference manual. http://www.celoxica.com (2010). Accessed 25 Sep 2010
62. Charot, F., Nyamsi, M., Quinton, P., Wagner, C.: Modeling and scheduling parallel data flow systems using structured systems of recurrence equations. In: Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '04), pp. 6–16. IEEE Computer Society, Washington, DC (2004)
63. Chawala, N., Guizzetti, R., Meroth, Y., Deleule, A., Gupta, V., Kathail, V., Urard, P.: Multimedia application specific engine design using high level synthesis. In: DesignCon, pp. 1–24. Santa Clara, CA (2008)
64. Chen, M., Lee, E.: Design and implementation of a multidimensional synchronous dataflow environment. Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers, vol. **1**, pp. 519–524. Pacific Grove, CA (1994)
65. Chen, M.J.: Developing a multidimensional synchronous dataflow domain in Ptolemy. Tech. Rep. UCB/ERL M94/16, University of California, Berkeley (1994)
66. Cheung, E., Hsieh, H., Balarin, F.: Automatic buffer sizing for rate-constrained KPN applications on multiprocessor system-on-chip. In: IEEE International Workshop on High Level Design Validation and Test (HLVDT), pp. 37–44. Irvine, CA (2007)
67. Cockx, J., Denolf, K., Vanhoof, B., Stahl, R.: SPRINT: a tool to generate concurrent transaction-level models from sequential code. EURASIP J. Appl. Signal Process. **2007**(1), 213–213 (2007)
68. Colorado State University: Cameron. http://www.cs.colostate.edu/cameron/index.html (2002). Accessed 25 Sep 2010
69. Cong, J., Fan, Y., Han, G., Jiang, W., Zhang, Z.: Behavior and communication co-optimization for systems with sequential communication media. In: DAC '06: Proceedings of the 43rd Annual Conference on Design Automation, pp. 675–678. ACM Press, New York, NY (2006)
70. Cong, J., Fan, Y., Han, G., Jiang, W., Zhang, Z.: Platform-based behavior-level and system-level synthesis. In: IEEE International SOC Conference, pp. 199–202. Austin, TX (2006)
71. Cong, J., Han, G., Jiang, W.: Synthesis of an application-specific soft multiprocessor system. In: Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays (FPGA'07), pp. 99–107. ACM Press, New York, NY (2007)
72. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edition, chap. Section 24.1: The Bellman-Ford Algorithm, pp. 588–592. MIT Press and McGraw-Hill, Cambridge, MA (2001)
73. Coussy, P.: Synthèse d'interface de communication pour les composants virtuels. Ph.D. thesis, Université de Bretagne Sud, Laboratoire d'Electronique des Systèmes Temps Réel (LESTER) (2003)
74. CoWare: CoWare platform architect. http://www.coware.com/products/platformarchitect.php (2010). Accessed 25 Sep 2010
75. CriticalBlue: Cascade. http://www.criticalblue.com/criticalblue_products/cascade.shtml

76. Cronquist, D., Gleason, C., Turean, F., Kathail, V.: Design of an H.264 encoder in five months using application engine synthesis. In: The 4th International Signal Processing Conference, pp. 1–7. Santa Clara, CA (2006)
77. Crookes, D., Benkrid, K., Bouridane, A., Alotaibi, K., Benkrid, A.: Design and implementation of a high level programming environment for FPGA-based image processing. IEE Proc. Vision Image Signal Process. **147**(4), 377–384 (2000)
78. Danckaert, K., Catthoor, F., Man, H.D.: A preprocessing step for global loop transformations for data transfer optimization. In: Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '00), pp. 34–40. ACM Press, New York, NY (2000)
79. Darte, A., Schreiber, R., Villard, G.: Lattice-based memory allocation. Tech. Rep. RR2004-23, ENS-Lyon (2004)
80. Darte, A., Schreiber, R., Villard, G.: Lattice-based memory allocation. IEEE Trans. Comput. **54**(10), 1242–1257 (2005)
81. Daubechies, I., Sweldens, W.: Factoring wavelet transforms into lifting steps. J. Fourier Anal. Appl. **4**(3), 247–269 (1998)
82. Davare, A., Densmore, D., Meyerowitz, T., Pinto, A., Sangiovanni-Vincentelli, A., Yang, G., Zeng, H., Zhu, Q.: A next-generation design framework for platform-based design. In: Conference on Using Hardware Design and Verification Languages (DVCon). San Jose, CA (2007)
83. Dekeyser, J., Beux, S.L., Marquet, P.: Une approche modèle pour la conception conjointe de systèmes embarqués hautes performances dédiés au transport. In: International Workshop on Logistique & Transport (LT' 2007). Sousse, Tunisie (2007)
84. Demeure, A., Gallo, Y.D.: An array approach for signal processing design. In: Sophia-Antipolis Conference on Micro-Electronics (SAME'98), System-on-Chip Session. France (1998)
85. Denolf, K., Bekooij, M., Cockx, J., Verkest, D., Corporaal, H.: Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations. EURASIP J. Adv. Signal Process. **2007**(84078), 14 (2007)
86. Densmore, D., Passerone, R., Sangiovanni-Vincentelli, A.: A platform-based taxonomy for ESL design. IEEE Des. Test **23**(5), 359–374 (2006)
87. Diet, F., D'Hollander, E., Beyls, K., Devos, H.: Embedding smart buffers for window operations in a stream-oriented C-to-VHDL compiler. In: 4th IEEE International Symposium on Electronic Design, Test and Applications (DELTA), pp. 142–147. Hong Kong (2008)
88. Diniz, P., Hall, M., Park, J., So, B., Ziegler, H.: Automatic mapping of C to FPGAs with the DEFACTO compilation and synthesis system. Microprocess. Microsyst. Special Issue FPGA Tools Tech. **29**(2–3), 51–62 (2005)
89. Diniz, P.C.: Evaluation of code generation strategies for scalar replaced codes in fine-grain configurable architectures. In: FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 73–82. IEEE Computer Society, Washington, DC (2005)
90. Dömer, R., Gerstlauer, A., Peng, J., Shin, D., Cai, L., Yu, H., Abdi, S., Gajski, D.D.: System-on-chip environment: A SpecC-based framework for heterogeneous MPSoC design. EURASIP J. Embedded Syst. **2008**(647953), 13 (2008)
91. Draper, B., Najjar, W., Bohm, W., Hammes, J., Rinker, B., Ross, C., Chawathe, M., Bins, J.: Compiling and optimizing image processing algorithms for FPGAs. In: Proceedings of 5th IEEE International Workshop on Computer Architectures for Machine Perception, pp. 222–231. Padova, PD (2000)
92. Dumont, P., Boulet, P.: Another multidimensional synchronous dataflow: Simulating Array-OL in Ptolemy II. Tech. Rep. 5516, Institut National de Recherche en Informatique et en Automatique, Cité Scientifique, 59 655 Villeneuve d'Ascq Cedex (2005)
93. Edwards, S.A.: SHIM: A language for hardware/software integration. In: SYNCHRON. Germany (2004)

94. Edwards, S.A.: The challenges of synthesizing hardware from C-like languages. IEEE Des. Test Comput. **23**(5), 3765–386 (2006)
95. Edwards, S.A., Tardieu, O.: SHIM: a deterministic model for heterogeneous embedded systems. In: Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT '05), pp. 264–272. ACM Press, New York, NY (2005)
96. Edwards, S.A., Tardieu, O.: Efficient code generation from SHIM models. In: LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems, pp. 125–134. ACM Press, New York, NY (2006)
97. Edwards, S.A., Vasudevan, N., Tardieu, O.: Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In: Design, Automation and Test in Europe, 2008. DATE '08, pp. 1498–1503. Munich, Germany (2008)
98. Eker, J., Janneck, J.W.: Embedded system components using the CAL actor language. University of California, Berkeley (2002)
99. Eker, J., Janneck, J.W.: CAL language report. language version 1.0 | document edition 1, University of California at Berkeley (2003)
100. Engels, M., Bilsen, G., Lauwereins, R., Peperstraete, J.: Cyclo-static data flow: Model and implementation. In: Proceedings of the 28th Asilomar Conference on Signals, Systems, and Computers, pp. 503–507. Pacific Grove, CA (1994)
101. Erbas, C., Pimentel, A.D., Thompson, M., Polstra, S.: A framework for system-level modeling and simulation of embedded systems architectures. EURASIP J. Embedded Syst. **2007**(1), 2–2 (2007)
102. Falk, J., Haubelt, C., Teich, J.: Efficient representation and simulation of model-based designs in SystemC. In: Proceedings of FDL'06, Forum on Design Languages 2006, pp. 129–134. Darmstadt, Germany (2006)
103. Falk, J., Keinert, J., Haubelt, C., Teich, J., Bhattacharyya, S.: A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications. In: Proceedings of the 8th ACM International Conference on Embedded Software (EMSOFT'08), pp. 189–198. Atlanta, GA (2008)
104. Feautrier, P.: Parametric integer programming. Oper. Res. **22**(3), 243–268 (1988)
105. Feautrier, P.: Scalable and structured scheduling. Int. J. Parallel Program. **34**(5), 459–487 (2006)
106. Fischaber, S., Woods, R., McAllister, J.: SoC memory hierarchy derivation from dataflow graphs. In: IEEE Workshop on Signal Processing Systems, pp. 469–474. Shanghai, China (2007)
107. Forte Design Systems: Forte cynthesizer. http://www.forteds.com (2010). Accessed 25 Sep 2010
108. Fowler, M., Scott, K.: UML Distilled: Applying the Standard Object Modeling Language. Addison-Wesley, Reading, MA (1997)
109. GadelRab, S., Bond, D., Reynolds, D.: Fight the power – power reduction ideas for ASIC designers and tool providers. Tech. Rep., Tundra Semiconductor Corporation 603 March Road, Ottawa, Ontario, K2K 2M5, Canada (2005)
110. Gamatié, A., Beux, S.L., Éric Piel, Etien, A., Ben-Atitallah, R., Marquet, P., Dekeyser, J.L.: A model driven design framework for high performance embedded systems. Tech. Rep. 6614, Institut National de Recherche en Informatique et en Automatique (2008)
111. Geilen, M., Basten, T., Stuijk, S.: Minimising buffer requirements of synchronous dataflow graphs with model checking. In: DAC '05: Proceedings of the 42nd Annual Conference on Design Automation, pp. 819–824. ACM Press, New York, NY (2005)
112. Ghamarian, A., Geilen, M., Stuijk, S., Basten, T., Moonen, A., Bekooij, M., Theelen, B., Mousavi, M.: Throughput analysis of synchronous data flow graphs. In: Proceedings of the 6th International Conference on Application of Concurrency to System Design (ACSD'06), pp. 25–36. IEEE Computer Society, Washington, DC (2006)
113. Ghosh, S., Venigalla, S., Bayoumi, M.: Design and implementation of a 2D-DCT architecture using coefficient distributed arithmetic. In: Proceedings of the IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI '05), pp. 162–166. IEEE Computer Society, Washington, DC (2005)

114. Gokhale, M.B., Stone, J.M.: Automatic allocation of arrays to memories in FPGA processors with multiple memory banks. In: FCCM '99: Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 63–69. IEEE Computer Society, Marriott at Napa Valley, Napa, CA (1999)
115. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. SIGARCH Comput. Archit. News **34**(5), 151–162 (2006)
116. Govindarajan, R., Gao, G.: A novel framework for multi-rate scheduling in DSP applications. In: Proceedings of the 1993 International Conference on Application Specific Array Processors, pp. 77–88. Venice, Italy (1993)
117. Govindarajan, R., Gao, G., Desai, P.: Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. J. VLSI Signal Process. **31**, 207–229. Kluwer, Dordrecht (2002)
118. Greef, E.D., Catthoor, F., Man, H.D.: Memory size reduction through storage order optimization for embedded parallel multimedia applications. Parallel Comput. **23**(12), 1811–1837 (1997)
119. Guillou, A.C.: Synthèse architecturale basée sur le modèle polyhédrique : Validation et extensions de la méthodologie mmalpha. Ph.D. thesis, L'université de Rennes (2003)
120. Guillou, A.C., Quinton, P., Risset, T.: Hardware synthesis for multi-dimensional time. Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors, pp. 40–50. The Hague, The Netherlands (2003)
121. Guo, Z., Najjar, W., Buyukkurt, B.: Efficient hardware code generation for FPGAs. ACM Trans. Archit. Code Optim. **5**(1), 1–26 (2008)
122. Gupta, S., Dutt, N., Gupta, R., Nicolau, A.: SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In: Proceedings of the 16th International Conference on VLSI Design (VLSID '03), p. 461. IEEE Computer Society, Washington, DC (2003)
123. Gupta, S., Gupta, R.K., Dutt, N.D., Nicolau, A.: Coordinated parallelizing compiler optimizations and high-level synthesis. ACM Trans. Des. Autom. Electron. Syst. **9**(4), 441–470 (2004)
124. Ha, S., Kim, S., Lee, C., Yi, Y., Kwon, S., Joo, Y.P.: PeaCE: A hardware-software codesign environment for multimedia embedded systems. ACM Trans. Des. Autom. Electron. Syst. **12**(3), 1–25 (2007)
125. Ha, S., Lee, C., Yi, Y., Kwon, S., Joo, Y.P.: Hardware-software codesign of multimedia embedded systems: the PeaCE approach. In: RTCSA '06: Proceedings of the 12th IEEE International Conference on Embedded in Real-Time Computing Systems and Applications, pp. 207–214. IEEE Computer Society, Washington, DC (2006)
126. van Haastregt, S., Kienhuis, B.: Automated synthesis of streaming C applications to process networks in hardware. In: Proceedings of Design, Automation & Test in Europe, pp. 890–893. ACM Press, Nice, France (2009)
127. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language Lustre. In: Proceedings of the IEEE, vol. 79, pp. 1305–1320 (1989)
128. Hammes, J., Rinker, B., Bohm, W., Najjar, W., Draper, B., Beveridge, R.: Cameron: High level language compilation for reconfigurable systems. In: PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, p. 236. IEEE Computer Society, Washington, DC (1999)
129. Han, S.I., Guerin, X., Chae, S.I., Jerraya, A.A.: Buffer memory optimization for video codec application modeled in Simulink. In: DAC '06: Proceedings of the 43rd Annual Conference on Design Automation, pp. 689–694. ACM Press, New York, NY (2006)
130. Hannig, F., Ruckdeschel, H., Dutta, H., Teich, J.: PARO: Synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications. In: Proceedings of the Fourth International Workshop on Applied Reconfigurable Computing (ARC), Lecture Notes in Computer Science (LNCS), pp. 287–293. Springer, London, United Kingdom (2008)

131. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
132. Harel, D.: Algorithmics: The Spirit of Computing, vol. 2nd edition. Addison-Wesley, Harlow, England (1992)
133. Haubelt, C., Falk, J., Keinert, J., Schlichter, T., Streubühr, M., Deyhle, A., Hadert, A., Teich, J.: A SystemC-based design methodology for digital signal processing systems. EURASIP J. Embedded Syst. Special Issue Embedded Digital Signal Process. Syst. **2007**, Article ID 47,580, 22 pages (2007)
134. Haubelt, C., Meredith, M., Schlichter, T., Keinert, J.: SystemCoDesigner: Automatic design space exploration and rapid prototyping from behavioral models. In: Proceedings of the 45th Design Automation Conference (DAC'08), pp. 580–585. Anaheim, CA (2008)
135. Haubelt, C., Schlichter, T., Teich, J.: Improving automatic design space exploration by integrating symbolic techniques into multi-objective evolutionary algorithms. Int. J. Comput. Intell. Res. (IJCIR), Special Issue Multiobject. Optim. Appls. **2**(3), 239–254 (2006)
136. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
137. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Upper Saddle River, NJ (2004)
138. Hsu, C.J., Keceli, F., Ko, M.Y., Shahparnia, S., Bhattacharyya, S.S.: DIF: An interchange format for dataflow-based design tools. Comput. Syst. Architect. Modeling, Simulat. 3133, 423–432. Samos, Greece (2004)
139. Hsu, C.J., Ko, M.Y., Bhattacharyya, S.S.: Software synthesis from the dataflow interchange format. In: Proceedings of the 2005 Workshop on Software and Compilers for Embedded Systems (SCOPES '05), pp. 37–49. ACM Press, New York, NY (2005)
140. Hu, Q.: Hierarchical memory size estimation for loop transformation and data memory platform optimization. Ph.D. thesis, Norwegian University of Science and Technology, Faculty of Information Technology, Mathematics and Electrical Engineering, Department of Electronics and Telecommunications (2007)
141. Hu, Q., Kjeldsberg, P.G., Vandecappelle, A., Palkovic, M., Catthoor, F.: Incremental hierarchical memory size estimation for steering of loop transformations. ACM Trans. Des. Autom. Electron. Syst. **12**(50), 1–25 (2007)
142. Hu, Q., Palkovic, M., Kjeldsberg, P.G.: Memory requirement optimization with loop fusion and loop shifting. In: DSD '04: Proceedings of the Digital System Design, EUROMICRO Systems, pp. 272–278. IEEE Computer Society, Washington, DC (2004)
143. Hu, Q., Vandecappelle, A., Kjeldsberg, P.G., Catthoor, F., Palkovic, M.: Fast memory footprint estimation based on maximal dependency vector calculation. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE '07), pp. 379–384. EDA Consortium, San Jose, CA (2007)
144. Hu, Q., Vandecappelle, A., Palkovic, M., Kjeldsberg, P.G., Brockmeyer, E., Catthoor, F.: Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications. In: ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation, pp. 606–611. IEEE Press, Piscataway, NJ (2006)
145. Hutchings, B., Bellows, P., Hawkins, J., Hemmert, S., Nelson, B., Rytting, M.: A CAD suite for high-performance FPGA design. In: Proceedings of Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '99), pp. 12–24. Marriott at Napa Valley, Napa, CA (1999)
146. IEEE: IEEE Standard VHDL Language Reference Manual. IEEE, IEEE Std. 1076–1987 edn. (1987)
147. IEEE: IEEE Standard VHDL Language Reference Manual. IEEE, IEEE Std. 1076–1993 edn. (1993)
148. ILOG: cplex. http://www.ilog.com/products/cplex/ (2010). Accessed 19 Sep 2010
149. IMEC: CleanC. http://www.imec.be/CleanC/ (2010). Accessed 19 Sep 2010

150. Impoco, G.: JPEG2000 – a short tutorial. Tech. Rep., Visual Computing Lab - ISTI-CNR Pisa, Italy (2004)
151. Impulse Accelerated Technologies: ImpulseC. http://www.impulsec.com/ (2010)
152. ISO/IEC JTC1/SC29/WG1: JPEG2000 Part I Final Committee Draft Version 1.0 (2002). N1646R
153. ITU: Digital Compression and Coding of Continuous-Tone Still Images – Requirements and Guidelines. CCITT, T.81 edn. (1992)
154. Jantsch, A., Sander, I.: Models of computation and languages for embedded system design. IEE Proc. Comput. Digital Tech. **152**(2), 114–129 (2005)
155. Jha, P.K., Dutt, N.D.: High-level library mapping for memories. ACM Trans. Des. Autom. Electron. Syst. **5**(3), 566–603 (2000)
156. Jung, H., Ha, S.: Hardware synthesis from coarse-grained dataflow specification for fast HW/SW cosynthesis. In: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '04), pp. 24–29. IEEE Computer Society, Washington, DC (2004)
157. Kahn, G.: The semantics of a simple language for parallel programming. In: Proceedings of IFIP Congress 74, pp. 471–475. Stockholm, Sweden (1974)
158. Kangas, T., Kukkala, P., Orsila, H., Salminen, E., Hännikäinen, M., Hämäläinen, T.D., Riihimäki, J., Kuusilinna, K.: UML-based multiprocessor SoC design framework. ACM Trans. Embedded Comput. Syst. **5**(2), 281–320 (2006)
159. Karczmarek, M., Thies, W., Amarasinghe, S.: Phased scheduling of stream programs. In: Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded systems (LCTES '03), pp. 103–112. ACM Press, New York, NY (2003)
160. Karp, R.M., Miller, R.E.: Properties of a model for parallel computations: Determinacy, termination and queuing. SIAM J. Appl. Math. **14**(6), 1390–1411 (1966)
161. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-integrated development of embedded software. Proc. IEEE **91**(1), 145–164 (2003)
162. Kathail, V., Aditya, S., Schreiber, R., Rau, B.R., Cronquist, D.C., Sivaraman, M.: PICO: Automatically designing custom computers. Computer **35**(9), 39–47 (2002)
163. Keinert, J., Dutta, H., Hannig, F., Haubelt, C., Teich, J.: Model-based synthesis and optimization of static multi-rate image processing algorithms. In: Proceedings of Design, Automation & Test in Europe, pp. 135–140. Nice, France (2009)
164. Keinert, J., Falk, J., Haubelt, C., Teich, J.: Actor-oriented modeling and simulation of sliding window image processing algorithms. In: Proceedings of the 2007 IEEE/ACM/IFIP Workshop of Embedded Systems for Real-Time Multimedia (ESTIMEDIA 2007), pp. 113–118. Salzburg, Austria (2007)
165. Keinert, J., Haubelt, C., Teich, J.: Windowed Synchronous Data Flow (WSDF). Tech. Rep. 02-2005, University of Erlangen-Nuremberg, Institut for Hardware-Software-Co-Design (2005)
166. Keinert, J., Haubelt, C., Teich, J.: Modeling and analysis of windowed synchronous algorithms. ICASSP2006 **III**, 892–895 (2006)
167. Keinert, J., Haubelt, C., Teich, J.: Simulative buffer analysis of local image processing algorithms described by Windowed Synchronous Data Flow. In: IC-SAMOS, pp. 161–168. Samos, Greece (2007)
168. Keinert, J., Haubelt, C., Teich, J.: Automatic synthesis of design alternatives for fast stream-based out-of-order communication. In: Proceedings of the 2008 IFIP/IEEE WG 10.5 International Conference on Very Large Scale Integration (VLSI-SoC), pp. 265–270. Rhodes Island, Greece (2008)
169. Keinert, J., Haubelt, C., Teich, J.: Synthesis of multi-dimensional high-speed FIFOs for out-of-order communication. In: Architecture of Computing Systems (ARCS 2008), vol. 4934/2008, pp. 130 – 143. Springer, New York, NY (2008)
170. Keinert, J., Streubühr, M., Schlichter, T., Falk, J., Gladigau, J., Haubelt, C., Teich, J., Meredith, M.: SystemCoDesigner—an automatic ESL synthesis approach by design space

exploration and behavioral synthesis for streaming applications. ACM Trans. Des. Autom. Electron. Syst. **14**(1), 1–23 (2009)
171. Keinert, J., Teich, J.: Data flow based system level design and analysis of image processing applications. Poster on the EDAA PhD forum at DATE 2009 (2009)
172. Kianzad, V., Bhattacharyya, S.S.: CHARMED: A multi-objective co-synthesis framework for multi-mode embedded systems. In: ASAP '04: Proceedings of the Application-Specific Systems, Architectures and Processors, 15th IEEE International Conference, pp. 28–40. IEEE Computer Society, Washington, DC (2004)
173. Kienhuis, B., Deprettere, E.F.: Modeling stream-based applications using the SBF model of computation. J. VLSI Signal Process. Syst. **34**(3), 291–300 (2003)
174. Kienhuis, B., Rijpkema, E., Deprettere, E.: Compaan: deriving process networks from MATLAB for embedded signal processing architectures. In: Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES '00), pp. 13–17. ACM Press, New York, NY (2000)
175. Kim, D.: A case study of system level specification and software synthesis of multi-mode multimedia terminal. ESTImedia **8**, 231–274 (2003)
176. Kjeldsberg, P., Catthoor, F., Aas, E.J.: Detection of partially simultaneously alive signals in storage requirement estimation for data intensive applications. In: DAC '01: Proceedings of the 38th conference on Design automation, pp. 365–370. ACM Press, New York, NY (2001)
177. Kjeldsberg, P.G., Catthoor, F., Aas, E.J.: Data dependency size estimation for use in memory optimization. IEEE Trans. CAD Integr. Circuits Syst. **22**(7), 908–921 (2003)
178. Kjeldsberg, P.G., Catthoor, F., Aas, E.J.: Storage requirement estimation for optimized design of data intensive applications. ACM Trans. Des. Autom. Electron. Syst. **9**(2), 133–158 (2004)
179. Ko, D.I.: System synthesis for image processing applications. Ph.D. thesis, University of Maryland (2006)
180. Ko, D.I., Bhattacharyya, S.S.: Modeling of block-based DSP systems. In: Proceedings of the IEEE Workshop on Signal Processing Systems, pp. 381–386. Seoul, Korea (2003)
181. Ko, D.I., Bhattacharyya, S.S.: Modeling of block-based DSP systems. J. VLSI Signal Process. Syst. **40**(3), 289–299 (2005)
182. Ko, M.Y., Murthy, P.K., Bhattacharyya, S.S.: Compact procedural implementation in DSP software synthesis through recursive graph decomposition. In: Proceedings of the International Workshop on Software and Compilers for Embedded Processors, pp. 47–61. Amsterdam, The Netherlands (2004)
183. Ku, D., Micheli, G.D.: HardwareC: A language for hardware design. Tech. Rep. CSTL-TR-90-419, Computer Systems Lab, Stanford Univ. (1990). Version 2.0
184. Kuzmanov, G., Gaydadjiev, G.N., Vassiliadis, S.: Multimedia rectangularly addressable memory. IEEE Trans. Multimedia **8**, 315–322 (2006)
185. Labbani, O.: Modélisation à haut niveau du contrôle dans des applications de traitement systématique à parallélisme massif. Ph.D. thesis, Université des Sciences et Technologies de Lille Laboratoire d'Informatique Fondamentale de Lille, 59655 Villeneuve (2006)
186. Labbani, O., Dekeyser, J.L., Boulet, P., Rutten, E.: Introduction of control into the Gaspard application UML metamodel: Synchronous approach. Tech. Rep. 5794, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille 59655 Villeneuve d'Ascq Cedex, France (2005)
187. Lauwereins, R., Engels, M., Ade, M., Peperstraete, J.: Grape-II: a system-level prototyping environment for DSP applications. Computer **28**(2), 35–43 (1995)
188. Lawal, N., O'Nils, M.: Embedded FPGA memory requirements for real-time video processing applications. In: 23rd NORCHIP Conference, pp. 206–209. Oulu, Finland (2005)
189. Lawal, N., O'Nils, M., Thörnberg, B.: C++ based system synthesis of real-time video processing systems targeting FPGA implementation. In: IPDPS, pp. 1–7 (2007)
190. Lawal, N., Thörnberg, B., O'Nils, M.: Address generation for FPGA RAMS for efficient implementation of real-time video processing systems. In: International Conference on Field Programmable Logic and Applications, pp. 136–141 (2005)
191. Lawal, N., Thörnberg, B., O'Nils, M.: Power-aware automatic constraint generation for FPGA based real-time video processing systems. In: NORCHIP, 2007, pp. 1–5. Aalborg, Denmark (2007)

192. Lee, E., Neuendorffer, S.: Concurrent models of computation for embedded software. IEE Proc. Comput. Digital Tech. **152**(2), 239–250 (2005)
193. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **C-36**(1), 24–35 (1987)
194. Lefebvre, V.: Restructuration automatique des variables d'un programme en vue de sa parallélilsation. Ph.D. thesis, Université de Versailles (1998)
195. Lefebvre, V., Feautrier, P.: Storage management in parallel programs. In: 5th Euromicro Workshop on Parallel and Distributed Processing, pp. 181–188. London, Great Britian (1997)
196. Lefebvre, V., Feautrier, P.: Automatic storage management for parallel programs. Parallel Comput. **24**(3-4), 649–671 (1998)
197. Lewis B. Baumstark, J., Wills, L.M.: Multidimensional dataflow-based parallelization for multimedia instruction set extensions. In: ICPPW '06: Proceedings of the 2006 International Conference Workshops on Parallel Processing, pp. 319–326. IEEE Computer Society, Washington, DC (2006)
198. Li, J.: Image compression: The mathematics of JPEG2000. Modern Signal Process. MSRI Publ. **46**, 185–221 (2003)
199. Liang, X., Jean, J., Tomko, K.: Data buffering and allocation in mapping generalized template matching on reconfigurable systems. J. Supercomput. **19**(1), 77–91 (2001)
200. Liao, H., Mandal, M., Cockburn, B.: Efficient architectures for 1-D and 2-D lifting-based wavelet transforms. IEEE Trans. Signal Process. **52**(5), 1315–1326 (2004)
201. Liu, T.M., Chung, C.C., Lee, C.Y., Lin, T.A., Wang, S.Z.: Design of a 125μw, fully-scalable MPEG-2 and H.264/AVC video decoder for mobile applications. In: Proceedings of the 43rd Annual Conference on Design Automation (DAC '06), pp. 288–289. ACM Press, New York, NY (2006)
202. LLVM: LLVM. http://llvm.org/ (2010)
203. Mallat, S.: A Wavelet Tour of Signal Processing, 2nd edition (Wavelet Analysis & Its Applications). San Diego, CA (1999)
204. Manolache, S., Eles, P., Peng, Z.: Buffer space optimisation with communication synthesis and traffic shaping for NoCs. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE '06), pp. 718–723. European Design and Automation Association, 3001 Leuven, Belgium, Belgium (2006)
205. Martinez, F.: Reduce FPGA design time with PICO express FPGA. Xcell J. **64**, 75–77 (2008)
206. Massachusetts Institute of Technology (MIT): Streamit. http://groups.csail.mit.edu/cag/streamit/index.shtml (2010)
207. MathWorks, T.: Simulink. www.mathworks.com/products/simulink/ (2010)
208. Mencer, O.: ASC: a stream compiler for computing with FPGAs. IEEE Trans. CAD Integr. Circuits Syst. **25**(9), 1603–1617 (2006)
209. Mentor Graphics: Catapult C. http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/ (2010)
210. Moore, M.S.: Model integrated program synthesis for real time image processing. Ph.D. thesis, Vanderbilt University (1997)
211. Murthy, P.K.: Scheduling techniques for synchronous and multidimensional synchronous dataflow. Ph.D. thesis, University of California at Berkeley (1996)
212. Murthy, P.K., Bhattacharyya, S.S.: Shared buffer implementations of signal processing systems using lifetime analysis techniques. IEEE Trans. CAD Integr. Circuits Syst. **20**(2), 177–198 (2001)
213. Murthy, P.K., Bhattacharyya, S.S.: Buffer merging—a powerful technique for reducing memory requirements of synchronous dataflow specifications. ACM Trans. Des. Autom. Electron. Syst. **9**(2), 212–237 (2004)
214. Murthy, P.K., Lee, E.A.: On the optimal blocking factor for blocked, non-overlapped schedules. In: 28th Asilomar Conference on Signals, Systems, and Computers, vol. 2, pp. 1052–1057. IEEE CS Press. Pacific Grove, CA (1994)

215. Murthy, P.K., Lee, E.A.: Multidimensional synchronous dataflow. IEEE Trans. Signal Process. **50**(7), 2064–2079 (2002)
216. Najjar, W.A., Böhm, W., Draper, B.A., Hammes, J., Rinker, R., Beveridge, J.R., Chawathe, M., Ross, C.: High-level language abstraction for reconfigurable computing. Computer **36**(8), 63–69 (2003)
217. Natarajan, S., Levine, B., Tan, C., Newport, D., Bouldin, D.: Automatic mapping of Khoros-based applications to adaptive computing systems. In: Proceedings of 1999 Military and Aerospace Applications of Programmable Devices and Technologies International Conference (MAPLD), pp. 101–107. Laurel, MD (1999)
218. National Instruments: LabVIEW FPGA. http://www.ni.com/fpga/ (2010)
219. Neema, S., Bapty, T., Scott, J.: Development environment for dynamically reconfigurable embedded systems. In: Proceedings of the International Conference on Signal Processing Applications and Technology. Orlando, FL (1999)
220. Nichols, J., Moore, M.: An adaptable, cost effective image processing system. In: The 10th JANNAF Non-destructive Evaluation Sub Committee, pp. 1–5. Salt Lake City, UT (1998)
221. Nichols, J., Neema, S.: Dynamically reconfigurable embedded image processing system. In: Proceedings of the International Conference on Signal Processing Applications and Technology. Orlando, FL (1999)
222. Nikolov, H., Stefanov, T., Deprettere, E.: Multi-processor system design with ESPAM. In: Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06), pp. 211–216. ACM Press, New York, NY (2006)
223. Nikolov, H., Stefanov, T., Deprettere, E.: Systematic and automated multiprocessor system design, programming, and implementation. IEEE Trans. CAD Integr. Circuits Syst. **27**(3), 542–555 (2008)
224. Ning, Q., Gao, G.R.: A novel framework of register allocation for software pipelining. In: Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 29–42. Charleston, SC (1993)
225. Norell, H., Lawal, N., O'Nils, M.: Automatic generation of spatial and temporal memory architectures for embedded video processing systems. EURASIP J. Embedded Syst. **2007**, Article ID 75,368, 10 pages (2007)
226. Object Management Group: Unified Modeling Language. http://www.uml.org (2010)
227. Oh, H., Ha, S.: Fractional rate dataflow model and efficient code synthesis for multimedia applications. In: Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES/SCOPES '02), pp. 12–17. ACM Press, New York, NY (2002)
228. Oh, H., Ha, S.: Memory-optimized software synthesis from dataflow program graphs with large size data samples. EURASIP J. Appl. Signal Process. **2003**(1), 514–529 (2003)
229. O'Nils, M., Thörnberg, B., Norell, H.: A comparison between local and global memory allocation for FPGA implementation of real-time video processing systems. In: Proceedings of the International Conference on Signals and Electronic Systems, pp. 429–432. Poznán, Poland (2004)
230. OSCI: Functional Specification for SystemC 2.0. Open SystemC Initiative. www.systemc.org (2002)
231. Ouaiss, I., Vemuri, R.: Global memory mapping for FPGA-based reconfigurable systems. In: Proceedings of 15th International Symposium on Parallel and Distributed Processing, pp. 1473–1480. San Francisco, CA (2001)
232. Ouaiss, I., Vemuri, R.: Hierarchical memory mapping during synthesis in FPGA-based reconfigurable computers. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE '01), pp. 650–657. IEEE Press, Piscataway, NJ (2001)
233. Ouaiss, I.E.: Hierarchical memory synthesis in reconfigurable computers. Ph.D. thesis, University of Cincinnati (2002)
234. Park, C., Chung, J., Ha, S.: Extended synchronous dataflow for efficient DSP system prototyping. IEEE International Workshop on Rapid System Prototyping pp. 196–201. Clear water, FL (1999)

235. Park, C., Ha, S.: Hardware synthesis from SPDF representation for multimedia applications. In: Proceedings of the 13th International Symposium on System Synthesis (ISSS '00), pp. 215–220. IEEE Computer Society, Washington, DC (2000)
236. Park, C., Kim, S., Ha, S.: A dataflow specification for system level synthesis of 3D graphics applications. In: ASP-DAC, pp. 78–84. Yokohama, Japan (2001)
237. Park, J., Diniz, P.C.: Synthesis of pipelined memory access controllers for streamed data applications on FPGA-based computing engines. In: Proceedings of the 14th International Symposium on Systems Synthesis (ISSS '01), pp. 221–226. ACM Press, New York, NY (2001)
238. Park, J., Diniz, P.C.: Partial data reuse for windowing computations: Performance modeling for FPGA implementations. In: Reconfigurable Computing: Architectures, Tools and Applications (ARC), Lecture Notes in Computer Science, vol. 4419, pp. 97–109. Mangaratiba, Brazil (2007)
239. Park, J.W.: An efficient buffer memory system for subarray access. IEEE Trans. Parallel Distrib. Syst. **12**(3), 316–335 (2001)
240. Parks, T.M., Pino, J.L., Lee, E.A.: A comparison of synchronous and cycle-static dataflow. In: ASILOMAR '95: Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set), p. 204. IEEE Computer Society, Washington, DC (1995)
241. Petri, C.A.: Communication with automata. Supplement 1 to technical report RADC-TR-65-337, Rome Air Develop. Cent. (1962)
242. Piel Éric, Attitalah, R.B., Marquet, P., Meftali, S., Niar, S., Etien, A., Dekeyser, J.L., Boulet, P.: Gaspard2: from MARTE to SystemC simulation. In: Design, Automation and Test in Europe (DATE 08). Munich, Germany (2008)
243. Pimentel, A.D., Thompson, M., Polstra, S., Erbas, C.: On the calibration of abstract performance models for system-level design space exploration. In: International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2006), pp. 71–77. Samos, Greece (2006)
244. Porter, R.B.: Image processing algorithms and architectures for reconfigurable computers. In: Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays. Springer, New York, NY (2005)
245. Ramanujam, J., Hong, J., Kandemir, M., Narayan, A.: Reducing memory requirements of nested loops for embedded systems. In: Proceedings of the 38th Conference on Design Automation (DAC '01), pp. 359–364. ACM Press, New York, NY (2001)
246. Reimann, F., Glaß, M., Lukasiewycz, M., Haubelt, C., Keinert, J., Teich, J.: Symbolic voter placement for dependability-aware system synthesis. In: Proceedings of the 6th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 237–242. Atlanta GA (2008)
247. Reiter, R.: Scheduling parallel computations. J. ACM **15**(4), 590–599 (1968)
248. Richardson, I.E.G.: H.264 and MPEG-4 Video Compression – Video Coding for Next-generation Multimedia. Wiley, West Sussex, England (2003)
249. Rijpkema, E., Deprettere, E., Kienhuis, B.: Compilation from MATLAB to process networks. In: Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99), pp. 388–395. Washington, DC (1999)
250. Rinker, R., Carter, M., Patel, A., Chawathe, M., Ross, C., Hammes, J., Najjar, W., Bohm, W.: An automated process for compiling dataflow graphs into reconfigurable hardware. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **9**(1), 130–139 (2001)
251. Schlichter, T., Lukasiewycz, M., Haubelt, C., Teich, J.: Improving system level design space exploration by incorporating SAT-solvers into multi-objective evolutionary algorithms. In: Proceedings of Annual Symposium on VLSI, pp. 309–314. IEEE Computer Society, Karlsruhe, Germany (2006)
252. Schreiber, R., Aditya, S., Mahlke, S., Kathail, V., Rau, B.R., Cronquist, D., Sivaraman, M.: PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. J. VLSI Signal Process. Syst. **31**(2), 127–142 (2002)
253. Séméria, L., Sato, K., Micheli, G.D.: Synthesis of hardware models in C with pointers and complex data structures. IEEE Trans. Very Large Scale Integr. Syst. **9**(6), 743–756 (2001)

254. Sen, M.: Model-based hardware design for image processing systems. Ph.D. thesis, Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies (2006)
255. Sen, M., Bhattacharyya, S., Lv, T., Wolf, W.: Modeling image processing systems with homogeneous parameterized dataflow graphs. Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '05) **5**, 133–136. Philadelphia, PA (2005)
256. Sen, M., Corretjer, I., Haim, F., Saha, S., Bhattacharyya, S.S., Schlessman, J., Wolf, W.: Computer vision on FPGAs: Design methodology and its application to gesture recognition. In: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) Workshops, p. 133. IEEE Computer Society, Washington, DC (2005)
257. Sen, M., Corretjer, I., Haim, F., Saha, S., Schlessman, J., Lv, T., Bhattacharyya, S.S., Wolf, W.: Dataflow-based mapping of computer vision algorithms onto FPGAs. EURASIP J. Embedded Syst. **2007**(49236), 1–12 (2007)
258. So, B., Hall, M.W.: Increasing the applicability of scalar replacement. In: Proceedings of 13th International Conference on Compiler Construction, Lecture Notes in Computer Science, vol. 2985, pp. 185–201. Barcelona, Spain (2004)
259. So, B., Hall, M.W., Ziegler, H.E.: Custom data layout for memory parallelism. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO '04), pp. 291–302. IEEE Computer Society, Washington, DC (2004)
260. Sovani, C., Edwards, S.A.: FIFO sizing for high-performance pipelines. In: Proceedings of the International Workshop on Logic and Synthesis. San Diego, CA (2007)
261. Stefanov, T., Kienhuis, B., Deprettere, E.: Algorithmic transformation techniques for efficient exploration of alternative application instances. In: Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES), pp. 7–12. Estes Park, CO (2002)
262. Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B., Deprettere, E.F.: System design using Kahn Process Networks: The Compaan/Laura approach. In: Proceedings of Design Automation & Test in Europe (DATE), pp. 340–345. Paris, France (2004)
263. Stichling, D., Kleinjohann, B.: CV-SDF - a model for real-time computer vision applications. In: WACV 2002: IEEE Workshop on Applications of Computer Vision. Orlando, FL (2002)
264. Strehl, K., Thiele, L., Gries, M., Ziegenbein, D., Ernst, R., Teich, J.: Funstate—an internal design representation for codesign. IEEE Trans. Very Large Scale Integr. Syst. **9**(4), 524–544 (2001)
265. Strehl, K., Thiele, L., Ziegenbein, D., Ernst, R.: Scheduling hardware/software systems using symbolic techniques. Tech. Rep. 67, Swiss Federal Institute of Technology (ETH) and Technical University of Braunschweig (1999)
266. Streubühr, M., Falk, J., Haubelt, C., Teich, J., Dorsch, R., Schlipf, T.: Task-accurate performance modeling in SystemC for real-time multi-processor architectures. In: Proceedings of Design, Automation and Test in Europe, pp. 480–481. IEEE Computer Society, Munich, Germany (2006)
267. Stuijk, S., Geilen, M., Basten, T.: Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In: Proceedings of the 43rd Annual Conference on Design Automation (DAC '06), pp. 899–904. ACM Press, New York, NY (2006)
268. Sutherland, S., Davidmann, S., Flake, P., Moorby, P.: SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling, vol. 2. Kluwer, Norwell, MA (2006)
269. Synopsys: Synopsys system studio. http://www.synopsys.com/systemstudio (2010). Accessed 19 Sep 2010
270. Taha, S., Radermache, A., Gerard, S., Dekeyser, J.L.: An open framework for detailed hardware modeling. In: International Symposium on Industrial Embedded Systems (SIES '07), pp. 118–125. Lisbon, Portugal (2007)

271. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
272. Teich, J., Zitzler, E., Bhattacharyya, S.: Optimized software synthesis for digital signal processing algorithms: an evolutionary approach. In: IEEE Workshop on Signal Processing Systems, pp. 589–598. Boston, MA (1998)
273. Teich, J., Zitzler, E., Bhattacharyya, S.S.: Buffer memory optimization in DSP applications — an evolutionary approach. In: Fifth International Conference on Parallel Problem Solving from Nature (PPSN-V), pp. 885–894. Springer, Berlin, Germany (1998)
274. Tessier, R., Betz, V., Neto, D., Egier, A., Gopalsamy, T.: Power-efficient RAM mapping algorithms for FPGA embedded memory blocks. IEEE Trans. CAD Integr. Circuits Syst. **26**(2), 278–290 (2007)
275. The SUIF Group: The SUIF compiler system. http://suif.stanford.edu/. Accessed 19 Sep 2010
276. Thies, W., Vivien, F., Sheldon, J., Amarasinghe, S.: A unified framework for schedule and storage optimization. Proc. ACM SIGPLAN 2001 Conf. Programming Lang. Des. Implementation **36**(5), 232–242 (2001)
277. Thomas, D.R., Moorby, P.: The Verilog Hardware Description Language, vol. 5. Kluwer, Boston, MA (2002)
278. Thompson, M., Nikolov, H., Stefanov, T., Pimentel, A.D., Erbas, C., Polstra, S., Deprettere, E.F.: A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In: Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '07), pp. 9–14. ACM Press, New York, NY (2007)
279. Thörnberg, B., Hu, Q., Palkovic, M., O'Nils, M., Kjeldsberg, P.G.: Polyhedral space generation and memory estimation from interface and memory models of real-time video systems. J. Syst. Softw. **79**(2), 231–245 (2006)
280. Thörnberg, B., Norell, H., O'Nils, M.: Conceptual interface and memory-modeling for real-time image processing systems. In: IEEE Workshop on Multimedia Signal Processing, pp. 138–141. Paris, France (2002)
281. Thörnberg, B., Olsson, L., O'Nils, M.: Optimization of memory allocation for real-time video processing on FPGA. In: The 16th IEEE International Workshop on Rapid System Prototyping (RSP2005), pp. 141–147. Montreal, Canada (2005)
282. Thörnberg, B., Palkovic, M., Hu, Q., Olsson, L., Kjeldsberg, P.G., O'Nils, M., Catthoor, F.: Bit-width constrained memory hierarchy optimization for real-time video systems. IEEE Trans. CAD Integr. Circuits Syst. **26**(4), 781–800 (2007)
283. Tomasi, C., Manduchi, R.: Bilateral filtering for gray and color images. In: ICCV, pp. 839–846. Bombay, India (1998)
284. Tronçon, R., Bruynooghe, M., Janssens, G., Catthoor, F.: Storage size reduction by in-place mapping of arrays. In: VMCAI '02: Revised Papers from the 3rd International Workshop on Verification, Model Checking, and Abstract Interpretation, pp. 167–181. Springer, London, UK (2002)
285. Turjan, A., Kienhuis, B., Deprettere, E.: Translating affine nested-loop programs to process networks. In: Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '04), pp. 220–229. ACM Press, New York, NY (2004)
286. Turjan, A., Kienhuis, B., Deprettere, E.: Solving out-of-order communication in Kahn Process Networks. J. VLSI Signal Process. **40**, 7–18 (2005)
287. Turjan, A., Kienhuis, B., Deprettere, E.F.: Realizations of the extended linearization model. In: 2nd Int. Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS 2002). Samos, Greece (2002)
288. Vasudevan, N., Edwards, S.: Static deadlock detection for the SHIM concurrent language. In: 6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2008), pp. 49–58. Anaheim, CA (2008)

289. Vasudevan, N., Edwards, S.A.: A JPEG decoder in SHIM. Computer Science Tech. Rep. CUCS-048-06, Columbia University (2006)
290. Venkataramani, G., Najjar, W., Kurdahi, F., Bagherzadeh, N., Bohm, W., Hammes, J.: Automatic compilation to a coarse-grained reconfigurable system-on-chip. ACM Trans. Embedded Comput. Syst. (TECS) **2**, 560–589 (2003)
291. Verdoolaege, S., Bruynooghe, M., Janssens, G., Catthoor, F.: Multi-dimensional incremental loop fusion for data locality. In: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors, pp. 17–27. The Hague, The Netherlands (2003)
292. Verdoolaege, S., Nikolov, H., Stefanov, T.: PN: a tool for improved derivation of process networks. EURASIP J. Embedded Syst. **2007**(1), 19–19 (2007)
293. Verhaegh, W.F.J., Aarts, E.H.L., van Gorp, P.C.N., Lippens, P.E.R.: A two-stage solution approach to multidimensional periodic scheduling. IEEE Trans. CAD Integr. Circuits Syst. **20**(10), 1185–1199 (2001)
294. Vincent, L.: Morphological grayscale reconstruction in image analysis: applications and efficient algorithms. IEEE Trans. Image Process. **2**(2), 176–201 (1993)
295. Vitkovski, A., Kuzmanov, G., Gaydadjiev, G.N.: Memory organization with multi-pattern parallel accesses. In: Proceedings of DATE, pp. 1420–1425. New York, NY (2008)
296. Wakabayashi, K.: CyberWorkBench: integrated design environment based on C-based behavior synthesis and verification. In: IEEE VLSI-TSA International Symposium on VLSI Design, Automation and Test (VLSI-TSA-DAT), pp. 173–176. Hsinchu, Taiwan (2005)
297. Wakabayashi, K., Okamoto, T.: C-based SoC design flow and EDA tools: An ASIC and system vendor perspective. IEEE Trans. CAD Integr. Circuits Syst. **19**(12), 1507–1522 (2000)
298. Wauters, P., Engels, M., Lauwereins, R., Peperstraete, J.: Cyclo-dynamic dataflow. Internal Report ESAT/ACCA/95/2, Katholieke Universiteit Leuven, ESAT Department, Kard. Mercierlann 94, B-3001 Heverlee, Belgium (1996)
299. Wauters, P., Engels, M., Lauwereins, R., Peperstraete, J.: Cyclo-dynamic dataflow. In: Proceedings of the 4th Euromicro Workshop on Parallel and Distributed Processing (PDP '96), pp. 319–326. Braga, Portugal (1996)
300. Weinhardt, M., Luk, W.: Memory access optimization for reconfigurable systems. IEEE Proc. Comput. Digital Tech. **148**, 105–112 (2001)
301. Wiggers, M., Bekooij, M., Jansen, P., Smit, G.: 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), pp. 10–15. ACM Press, New York, NY (2006)
302. Wiggers, M., Bekooij, M., Smit, G.: Efficient computation of buffer capacities for cyclo-static dataflow graphs. Tech. Rep., Centre for Telematics and Information Technology, University of Twente, Enschede (2006)
303. William Thies, J.L., Amarasinghe, S.: Phased computation graphs in the polyhedral model. Tech. Rep., MIT Laboratory for Computer Science Cambridge, MA 02139 (2002)
304. Wilt, A.J.: Progressive segmented frame. In: DV Magazine. http://www.adamwilt.com/TechDiffs/FieldsAndFrames.html (2000)
305. Wuytack, S., Catthoor, F., Nachtergaele, L., De Man, H.: Power exploration for data dominated video applications. Int. Symp. Low Power Electron. Des. pp. 359–364 (1996). DOI 10.1109/LPE.1996.547539
306. Xilinx: CORE Generator. http://www.xilinx.com/tools/coregen.htm. Accessed 19 Sep 2010
307. XILINX: Embedded SystemTools Reference Manual - Embedded Development Kit EDK 8.1i. URL http://www.xilinx.com/support/documentation/sw_manuals/edk81i_est_rm.pdf (2005)
308. Xilinx: Virtex-4 Family Overview, 3.0 edn. (2007)
309. Yang, H., Jung, H., Ha, S.: Buffer minimization in RTL synthesis from coarse-grained dataflow specification. In: SASMI. Nagoya, Japan (2006)
310. Yu, H., Leeser, M.: Optimizing data intensive window-based image processing on reconfigurable hardware boards. In: IEEE Workshop on Signal Processing Systems Design and Implementation, pp. 491–496. Athens, Greece (2005). DOI 10.1109/SIPS.2005.1579918

311. Yu, H., Leeser, M.: Automatic sliding window operation optimization for FPGA-based computing boards. In: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06), pp. 76–88. IEEE Computer Society, Washington, DC (2006)
312. Zebelein, C., Falk, J., Haubelt, C., Teich, J.: Classification of general data flow actors into known models of computation. In: Proc. 6th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2008), pp. 119–128. Anaheim, CA (2008)
313. Zhang, F., Yoo, Y.M., Koh, L.M., Kim, Y.: Nonlinear diffusion in Laplacian pyramid domain for ultrasonic speckle reduction. IEEE Trans. Med. Imaging **26**(2), 200–211 (2007)
314. Zhao, Y., Malik, S.: Exact memory size estimation for array computations. IEEE Trans. Very Large Scale Integr. Syst. **8**(5), 517–521 (2000)
315. Zhu, H., Luican, I.I., Balasa, F.: Exact computation of storage requirements for multi-dimensional signal processing applications. Tech. Rep., University of Illinois at Chicago (2005)
316. Ziegler, H., Hall, M.: Evaluating heuristics in automatically mapping multi-loop applications to FPGAs. In: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays (FPGA '05), pp. 184–195. ACM Press, New York, NY (2005)
317. Ziegler, H.E., Hall, M.W., Diniz, P.C.: Compiler-generated communication for pipelined FPGA applications. In: DAC '03: Proceedings of the 40th Conference on Design Automation, pp. 610–615. ACM Press, New York, NY (2003)
318. Zissulescu, C., Kienhuis, B., Deprettere, E.: Communication synthesis in a multiprocessor environment. In: International Conference on Field Programmable Logic and Applications, pp. 360–365. Tampere, Finland (2005)
319. Zissulescu, C., Kienhuis, B., Deprettere, E.F.: Increasing pipelined IP core utilization in process networks using exploration. In: FPL, pp. 690–699. Belgium (2004)
320. Zissulescu, C., Turjan, A., Kienhuis, B., Deprettere, E.: Solving out of order communication using CAM memory; an implementation. In: 13th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2002). Veldhoven, Netherlands (2002)

Index