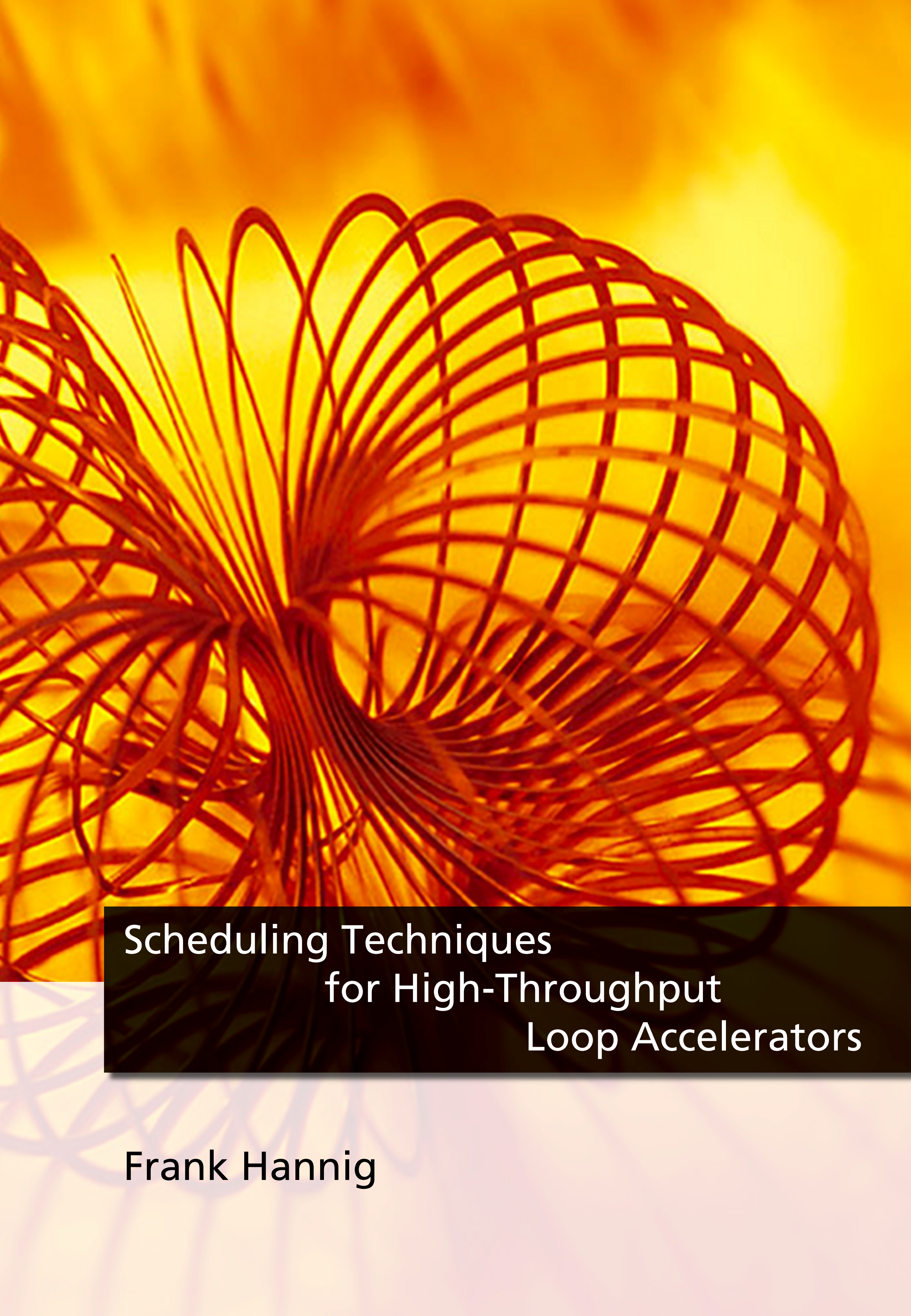


Scheduling Techniques for High-Throughput Loop Accelerators

Frank Hannig

Cite thesis as:

Frank Hannig. *Scheduling Techniques for High-Throughput Loop Accelerators*. Dissertation, University of Erlangen-Nuremberg, Germany, August 2009. Verlag Dr. Hut, Munich, Germany.

BibTeX:

```
@PHDTHESIS{Han09,
  AUTHOR    = {{Hannig}, {Frank}},
  MONTH     = aug,
  NOTE      = {Verlag Dr. Hut, Munich, Germany},
  SCHOOL    = {University of Erlangen-Nuremberg, Germany},
  TITLE     = {{Scheduling} {Techniques} for {High-Throughput} {Loop} {Accelerators}},
  TYPE      = {Dissertation},
  YEAR      = {2009},
  ISSN_ISBN = {978-3-86853-220-3}
}
```

Scheduling Techniques for High-Throughput Loop Accelerators

Der Technischen Fakultät der
Universität Erlangen-Nürnberg
zur Erlangung des Grades

D O K T O R - I N G E N I E U R

vorgelegt von

Frank Hannig

Erlangen – 2009

Als Dissertation genehmigt von der Technischen Fakultät
der Universität Erlangen-Nürnberg

Tag der Einreichung:	8. Juni 2009
Tag der Promotion:	11. August 2009
Dekan:	Professor Dr.-Ing. habil. Johannes Huber
Berichterstatter:	Professor Dr.-Ing. Jürgen Teich
	Professor Dr. Krzysztof Kuchcinski

This thesis is dedicated to my scintillating wife Ulrike
with love and gratitude.

Acknowledgments

I would like to thank first and foremost my doctoral adviser, Professor Dr.-Ing. Jürgen Teich, for all the support and advice, his confidence, and the numerous and varied responsibilities he has given over the last years to me. I would also like to thank Professor Krzysztof Kuchcinski for being the co-examiner of my thesis.

I would like to thank everyone in the architecture and compiler design (ACD) group for interest, advice, and support. In no particular order they are Hritam Dutta, Dmitrij Kissler, Vahid Lari, Farhadur Arifin, Richard Membarth, Moritz Schmid, Sunil Shukla, and the former members Alexej Kupriyanov and Holger Ruckdeschel. Especially, I would like to thank Hritam for all the scientific discussions and friendly chats. Moreover, I truly appreciate the work of all the students who designed and implemented parts of the PARO high-level synthesis tool, especially Jiali Teddy Zhai.

I would like particularly thank my previous office mate and office neighbor, respectively, Christian Haubelt for all the discussions concerning all issues of the chair of Hardware/Software Co-Design. I would like to thank Dirk Koch for his helpfulness, especially for his valuable knowledge in hardware design, the chats and sharing food with him at nights and weekends when finalizing this thesis.

I would like to thank Hritam and Moritz who took the time to read over my thesis and providing very valuable feedback.

Last, and by no means least, thanks to my family. I would like to thank my parents for my technical-oriented advancement, their constant support, and interest in my work as long as I got my first Lego bricks from them. Finally, I would like to thank my children Ricarda Sophie and Constantin Leander, and my wife Ulrike for their support, endless patience, and love.

Frank Hannig
Erlangen, August 2009

Contents

Chapter 1

Introduction

The desire for more mobility and the enthusiasm for ubiquitous electronic gadgets on the one hand side and the steady progress in semiconductor industry on the other hand are the driving forces in the market of embedded digital systems. The growing number of stream-based applications is eager for more computational power. Examples of such systems include handhelds for digital audio and video broadcasting and navigation, next generation game and entertainment consoles with high-definition television (HDTV) support and high-capacity storage media like HD DVD or Bluray Disc. Other demanding applications with real-time requirements can be found in the areas of medical image processing, radar technology, and robotics.

Feature sizes reaching nanoscopic scale allow implementations of complex systems on a single die. These designs are called *system-on-a-chip* (SoC) or in case multiple embedded processors are included, *multiprocessor system-on-a-chip* (MPSoC). The usual components of such a system are sketched in Figure 1.1. Typically, an SoC consists of one or more embedded processors, memories, I/O interfaces, accelerators for dedicated tasks, and a communication network for connecting the individual parts of the system. Note that the communication network as indicated by the graphic can be an arbitrary, maybe hierarchic interconnect structure, which might consist of several busses or point-to-point connections. The situation for the accelerators is similar. Here, an abundance of variations exists, ranging from custom hardware in form of application-specific integrated circuits (ASICs) to fine-grained reconfigurable devices (FPGAs) and finally to reconfigurable or programmable architectures consisting of an array of coarse-grained elements such as functional units of fixed word size.

In order to handle the complexity of such highly integrated systems, they are preferably designed in a *top-down strategy* starting at the so-called *electronic system-*

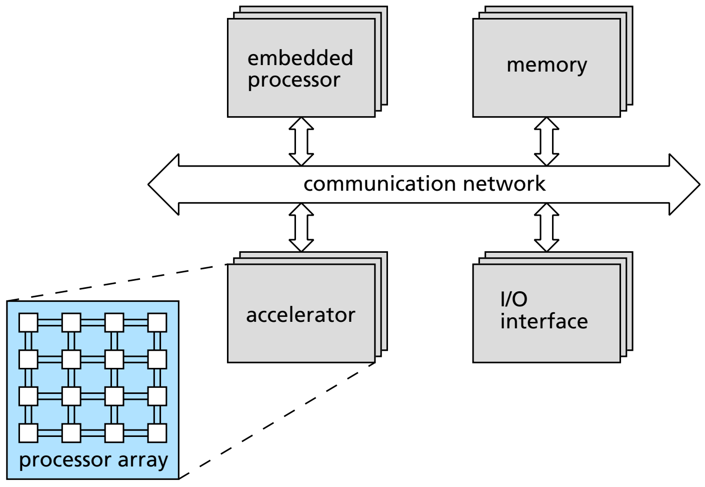


Figure 1.1: Schematic representation of a system-on-a-chip (SoC). One possible realization of an accelerator could be a processor array. These arrays, either implemented as dedicated hardware for a single algorithm or in form of a programmable architecture, are the field of investigation in this thesis.

level (ESL). In [BMP07], Bailey, Martin, and Piziali define ESL as: *"The use of appropriate abstractions to increase comprehension of a system and enhance the probability of successfully implementing its functionality in a cost-effective manner, while meeting necessary constraints."* Design decisions at system-level have the greatest influence on performance, cost, power consumption, and design time. Thus, ESL design is strongly related to the exploration of multiple possible solutions (*design space exploration*) [KSS+09]. Further, its integral view enables the early verification of a design specification. In order to reduce the time to market, mainly two approaches are pursued:

1. A *modular design principle* often referred to as *platform-based design* [KMN+00, SCDS04], where already designed components are systematically reused and assembled to an SoC.

2. *Synthesis* addresses (a), the automatic refinement of descriptions from one abstraction level to another; and (b), the translation of a behavioral description into a structural representation.

Both approaches seek expressive modeling and specification possibilities. Depending on the abstraction level (see Figure 1.2) and whether hardware or software is designed, different modeling and programming languages are used. On the hardware side, the description languages VHDL and Verilog are most widespread. For developing embedded software, common languages are assembler, C, C++, and Java. At

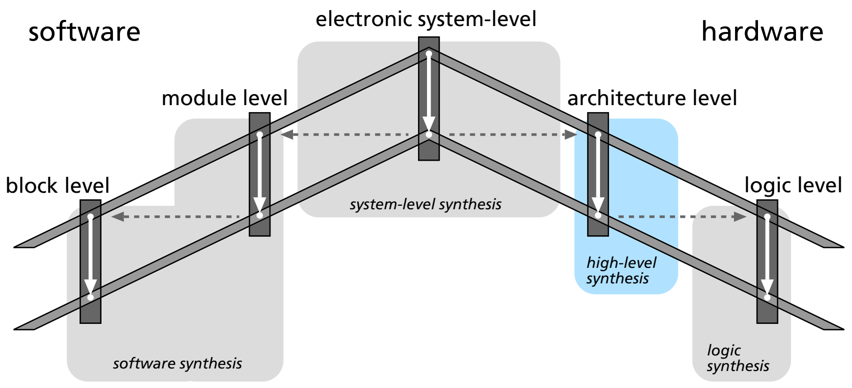


Figure 1.2: Abstraction levels for the design of embedded systems, adapted from Teich's *double roof model* [Tei97]. The upper roof denotes the behavioral view, the lower one the structural view. The present work focuses on high-level synthesis (marked in light blue).

the system-level, for the domain of digital signal and image processing, Matlab, C, C++, and SystemC are the languages of choice. Noteworthy about SystemC is that it tries to cover both sides, software and hardware [GLMS02]. Moreover, descriptions in SystemC can be successively refined for lower abstraction levels.

The synthesis at different abstraction levels is marked in Figure 1.2. The lower a synthesis type is placed on the *roof*, the older and more studied it is. For instance, logic synthesis can be traced back to the 19th century to the treatment of Boolean algebra, in addition the works in logic minimization by Shannon, Quine-McCluskey, or Karnaugh and Veitch are well-known. Thus, in the 1980s, logic synthesis was the first automated translation from Boolean equations to gate-level structures. Similar on the software side, the translation from assembly code to binary code is researched quite well. Some proponents of ESL design claim that the abstraction levels below system-level have been—similar to logic synthesis—mostly investigated. This is not true, since in academia as well as in industry there exists a great demand for *high-level synthesis* (also referred to as *behavioral synthesis*) methods that automatically transfer an algorithmic behavioral description into an efficient representation at *register transfer level* (RTL). High-level synthesis is a cutting-edge research topic with a multitude of challenges and open questions. This hypothesis is supported by a number of special sessions and workshops on this topic at recent premier electronic design automation conferences[1].

[1]For instance: Workshop *The New Wave of the High Level Synthesis* at Design, Automation and Test in Europe (DATE 2008); Workshop *High-level Synthesis: Back to the Future* at Design Automation Conference (DAC 2008); Workshop *High-Level Synthesis: Next Step to Efficient ESL Design* at Asia and South Pacific Design Automation Conference (ASP-DAC 2009).

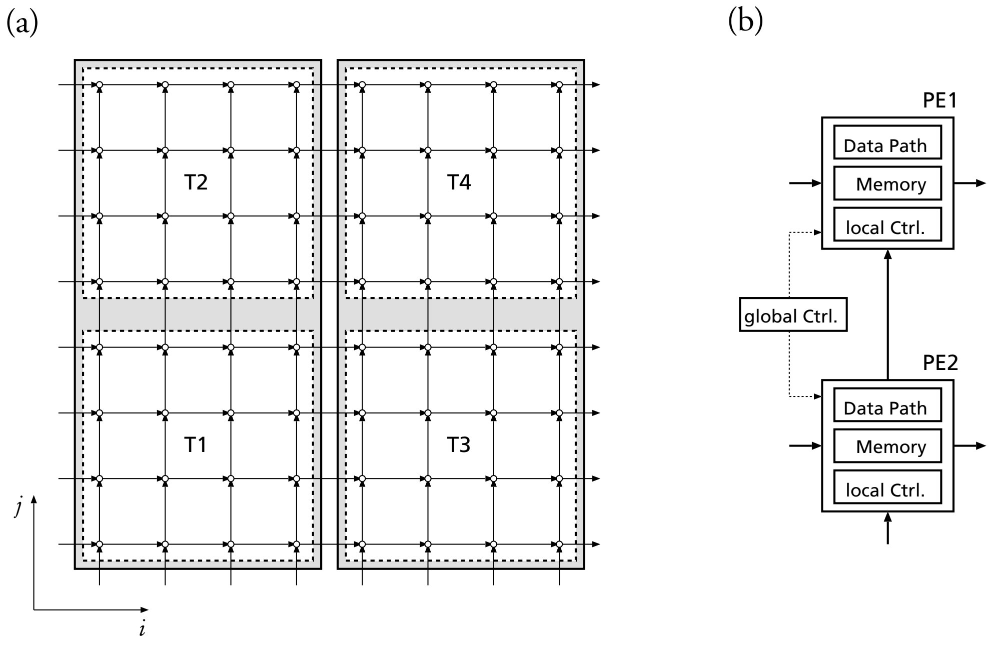


Figure 1.3: In (a), a two-dimensional iteration space consisting of 64 iterations and its hierarchical structuring into different subspaces, denoted by the white and gray tiles, is depicted. In (b), a corresponding dedicated hardware implementation with two processing elements is schematically shown.

The major challenges in high-level synthesis are:

- Domain-specific **models** and **languages** that preserve the parallelism of an initially given mathematical description.
- **Parallelization techniques** in order to exploit computationally intensive algorithms in the best possible way and **mapping methods**, which are able to target the hardware complexity of todays technology.
- **Scheduling** and **allocation** techniques that enable a holistic treatment of parallelism at different levels with simultaneous consideration of resource constraints and different requirements on performance, power consumption, and cost.

To overcome these very general challenges for arbitrary target architectures, we present a domain-specific methodology for so-called *processor arrays* in this thesis instead. The approach is focusing on computationally intensive algorithms with mostly regular and multi-dimensional data flow. The considered algorithms are expressed by nested loop programs. A wide variety of algorithms from the areas of digital image (e.g., median filter, 2-D convolution, edge detection), video (for instance, motion detection or motion compensation as in MPEG) and other signal processing (FIR, IIR, Kalman, and many more filters, linear predictive coding, discrete Fourier transform, etc.), linear algebra (matrix-vector multiplication, matrix multiplication, LU decomposition, matrix inversion, solving linear systems, etc.), combinatorial problems (e.g., shortest path problems or transitive closure problems), and many other scientific computing domains can be expressed by such loop programs. These processor arrays are typically tailored to the requirements of just one algorithm. This means, that the processors are not re-programmable for another application and thus, we are referring to them as *processing elements* (PE) in the course of the thesis. The regularity of the algorithms is reflected by the resulting processor arrays, whose structure corresponds to different levels of parallelism, memory, and control. An example is shown in Figure 1.3, where in (a) a 2-dimensional iteration space is visualized, which corresponds to a two nested loop. Each white point corresponds to the execution of a number of operations within the loop body. The arcs denote data dependencies between different iterations (also known as *loop-carried dependencies*) and inputs/outputs, respectively. Based on a given specification, a dedicated hardware accelerator should be implemented. Assuming that analysis of the performance requirements, memory and I/O cost leads to a *hierarchical partitioning* of the computations denoted by the two types of rectangles in Figure 1.3(a), then this partitioning corresponds to a hardware realization with two processing elements drafted in Figure 1.3(b). The iterations within the white tiles T1 and T2 are processed by the two elements PE1 and PE2. Once processing element PE1 has finished its execution of the 16 iterations within tile T1, it continues with tile T3; just as PE2 proceeds with tile T4 after finishing its work on T2. Such a hierarchical approach is not unusual since hereby communication cost can be traded for the cost of memory on several levels [BCD94, EM99, Eck01, HKV$^+$07, BKV$^+$08]. The assignment of iterations to processing elements can be seen as *global allocation*, but it scarcely says something about the schedule. Similarly, types and number of functional units inside the processing elements form a *local allocation*. Local and global control structures orchestrate the interplay of processing elements and functional units, respectively. For the considered class of algorithms, explictly defined execution orders and static resource allocation are used in order to increase performance and to reduce cost. Memory and interconnect structures can be generated according to the required demands. Consequently, a customized hardware, dedicated to the acceleration of the given loop program, is derived.

The term *processor array* is often associated equally with the term *systolic array* [KL78]. The example in Figure 1.3, which considers several levels of allocation and parallelism into account, is yet far beyond the concepts in systolic array design.

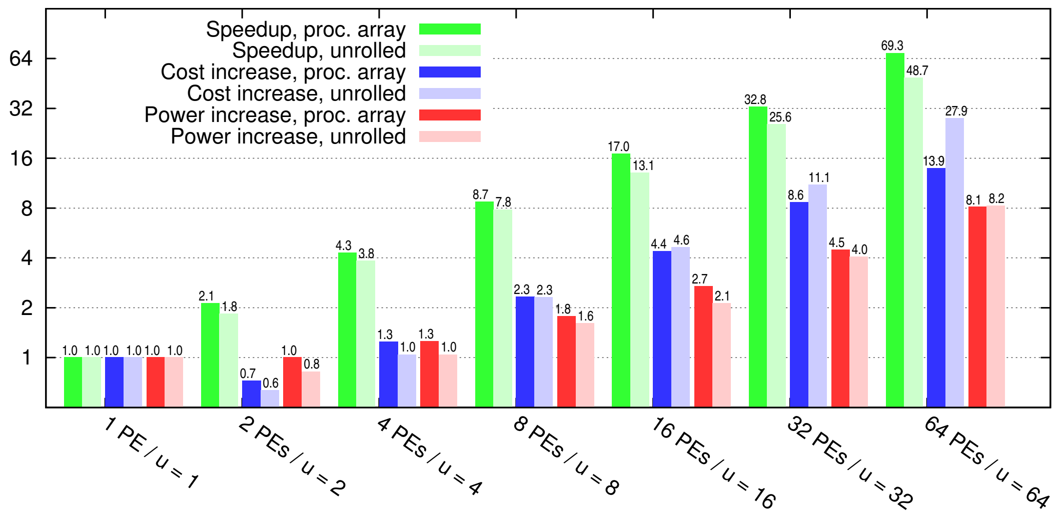
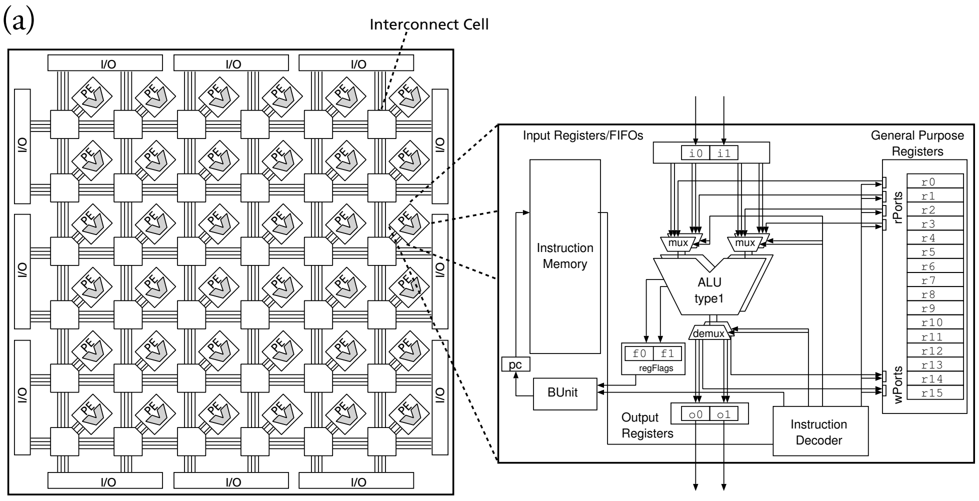


Figure 1.4: The speedup (with respect to throughput), cost and power increase for different 64-tap FIR filter implementations in relation to a sequentialized implementation, where only a single processing element with one multiplier and one adder is available, is shown. The x-axis shows the number of available resources (multipliers and adders), which is doubled from experiment to experiment. That means, in the case of the processor array approach, the values on the x-axis correspond to the number of PEs, and in the case of loop unrolling, the values on the abscissa match the unroll factor u.

As mentioned earlier, processor arrays are a powerful instrument to exploit parallelism on several levels. Whereas other high-level synthesis approaches often offer no parallelization at all, or they use *loop unrolling* in order to achieve a higher data throughput. The PARO synthesis tool [HRDT08], which has been developed within the scope of this thesis, is able to perform both loop unrolling and the processor array approach. Internally, it employs the same techniques for resource allocation and scheduling, hence a fair quantitative comparison between the two approaches can be performed. In Figure 1.4, such a comparison is shown for different 64-tap FIR filter implementations. Although at this point of the thesis only briefly explained, several experiments with different numbers of available resources were performed and implemented in FPGA technology. The speedup characterizes the performance gain with respect to the throughput for the FIR filter algorithm. The cost increase is related to the gate count of the designs. In the single PE solution of the FIR filter, the 64 iterations are executed sequentially within one PE. For this solution, both throughput and cost are normalized to 1.0. Partitioning the algorithm to 2, 4, . . . , 64 PEs theoretically also enables a higher throughput by the same factor. The super-

(a)



(b)

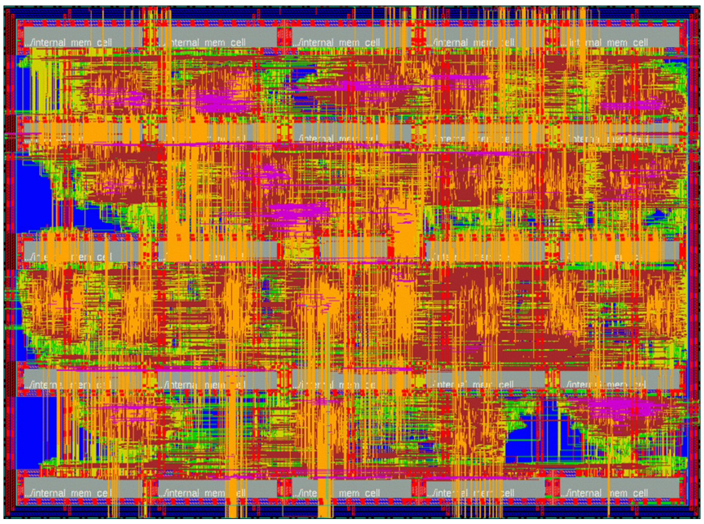


Figure 1.5: Processor array architecture of the class of so-called *weakly-programmable processor arrays* (WPPA) [HDK+05, KHKT06b]. In (a), the schematic array architecture and processor structure are depicted. A chip layout of a WPPA in 90 nm standard cell technology is shown in (b).

linear speedup in case of the processor array implementations is due to the increasing clock frequency for larger numbers of PEs. The moderate cost and power increase is caused by the decreasing amount of intermediate data, which have to be stored internally in the processor array. The results in terms of performance, area cost, and power of the two approaches clearly advocate the processor array approach.

Dedicated hardware accelerators can always outperform standard embedded processors in terms of performance and power consumption. Because of this, increasingly more and more unconventional processors are emerging, which try to combine the best of both worlds: The performance, cost and power efficiency of dedicated hardware with the flexibility of programmable processors. These architectures typically consist of an array of reconfigurable, or to some extent programmable, coarse-grained units for data processing, few distributed memory, and a tightly interconnected communication network. An example of such an architecture is shown in Figure 1.5(a). Shown is an array of simple VLIW processors, each having its own small instruction memory and register file, however, with no shared memory access. A reconfigurable, switched interconnect allows communication on cycle basis. The parallelism of the architecture is expressed on different levels: several parallel working processors (*loop-level parallelism*), multiple functional units within one processor (*instruction-level parallelism*), and *software pipelining*. From the structural point of view, these programmable architectures are very similar to the dedicated processor arrays derived by high-level synthesis. Thus, it would stand to reason that similar or even the same concepts of algorithm modeling, allocation, and scheduling might be used for both kinds of processor arrays (dedicated as well as programmable accelerators).

1.1 Contributions and Bibliographic Notes

The dissertation's primary contributions are in the fields of *modeling* and *scheduling and allocation* techniques for high-throughput loop accelerators. The major achievement is a unified mapping methodology [DHT06d, DHT06c, DHK+07, THR+07, HRDT08] for computationally intensive programs, that targets dedicated hardware accelerators [HT01, RDHT05, DHT05, DHT+06e, DHT06b, DHT06a, DHRT07, HRDT08, DHT08, HDT09, DHT09, KDH+09, DZHT09], special classes of coarse-grained, reconfigurable architectures [HDT04a, HDT04b, HDT06, ESO+08], and tightly-coupled, programmable processor arrays [HDK+05, KHK+06a, KHK+06b, KHKT06a, KHKT06b, KHKT06c, KHK+06c, KHK+07, KKHT07, KHT07] and [DKH+09, KHKT08, KSHT08, KSHT09]. The major contributations of this thesis are summarized briefly in the following. They, include advances in modeling, parallelization, and scheduling as well as high-level synthesis.

Modeling. In order to map computationally intensive programs onto tightly-coupled processor arrays in a systematic way, a profound mathematical model is essential. In this context, a new class of algorithms called *dynamic piecewise linear algorithms* [HT04b, HT04c, HRDT08] and a corresponding graph representation for modeling iterative, multi-dimensional data flow is presented in the thesis. The class of dynamic piecewise linear algorithms extends well-known models, that are based on systems of recurrence equations, defined over polyhedral iteration domains. Common to all existing approaches is that data dependencies are assumed to be static. Dynamic piecewise linear algorithms are able to model also a specific type of dynamic data dependencies, that is, dependencies becoming known only at run-time. By this enhancement, the range of applications with multi-dimensional data flow that can be parallelized and mapped onto massively parallel processor arrays is significantly increased. For instance, many computationally intensive applications for video and image processing consist of nested loop programs with only few and simple run-time dependent conditions.

On the basis of the class of dynamic piecewise linear algorithms, a novel programming language is presented in the thesis. The language is called PAULA [HRT08, HRDT08] and capable to model data flow dominated applications. It is intended for designing highly parallel applications, expressing parallelism at instruction, data, and loop level. The PAULA language allows very compact and efficient behavioral descriptions and serves as design entry when generating dedicated hardware accelerators [HRDT08], or might be used as high-level programming language for tightly-coupled multi-processor architectures [KHKT06b, THR+07]. The language covers a broad range of applications from the areas of digital image, video and other signal processing, linear algebra, cryptography, and many other scientific computing domains, where efficient parallelization techniques and hardware accelerators are indispensable.

Key features of the PAULA language are:

- A *functional programming language* dedicated to mapping computationally intensive algorithms onto parallel, tightly-coupled processor architectures. A full static single assignment form—also for *multi-dimensional arrays*—is provided.
- Powerful expressions to specify polyhedral and lattice iteration domains.
- Convenient usage of *big operators* such as $\sum$ or $\prod$.
- Can handle *run-time dependent control flow* (support of dynamic piecewise linear algorithms).
- Besides *behavioral description* possibilities, also *architectural modeling* can be considered.
- Its applicability has been demonstrated in real-world case studies.

Scheduling and Allocation. As mentioned earlier, allocation and scheduling are the basic problems of high-level synthesis. However, scheduling with resource constraints can quickly become a hard combinatorial problem. In that case, often heuristics are deployed or a hierarchical approach according to the different structural levels is used. A hierarchical scheduling approach could lead to suboptimal results. Consider for instance, that all operations of a loop body are scheduled first (local schedule) and afterwards, a schedule for the iterations of a loop is determined (global schedule). Even when overlapping the different iterations afterwards, this could be worse compared to considering the instruction-level and the loop-level simultaneously as for instance in *modulo scheduling* [RST92]. Whereas modulo scheduling deals only with single processors and one-dimensional data dependencies, in the processor array approach we have to deal with arrays of processors, multi-dimensional data flow, and maybe several partitioning levels (cf. Figure 1.3). Hence, if the execution of not all of these parallelism levels is interleaved, the performance loss might accumulate.

In this dissertation, exact and holistic scheduling techniques are developed that incorporate both, the local and the global allocation. By this close integration, performance optimal schedules may be derived. In order to obtain these schedules, an approach based on *mixed linear integer programming* (MIP) [NW99, HT01] is developed. In this thesis, the local schedule as well as the global schedule for several levels of partitioning is optimized for the first time, simultaneously. As depicted in Figure 1.3(a), the iterations within a white tile should be executed sequentially. In this context, a new *serialization constraint* for MIP is presented that leads to better schedules than existing approaches. The second main contribution in the area of scheduling is the formulation of resource constraints that take the mutual exclusivity of *iteration dependent conditions* as well as of *run-time dependent conditions* into account as shown in the following program fragment.

```
par (i >= 0 and i <= N-1)
{ // Equations with iteration dependent conditions
  c[i] = a[i] * b[i]           if (i == 0);
  c[i] = a[i] * b[i] + c[i-1]  if (i >= 1);
  d[i] = c[i] > 255;
  // Equation with run-time dependent condition
  e[i] = ifrt(d[i], c[i], 255);
}
```

Shown is a PAULA program describing a dynamic piecewise regular algorithm. The **ifrt** statement selects in dependence on the value of the Boolean variable `d[i]` if the value of `c[i]` or the constant (255) is assigned to variable `e[i]`.

In the context of the algorithm classes based on multi-dimensional recurrence equations, *conditional scheduling* [HT04a, HT04b, HT04d] has been applied for the first time within this work in order to schedule run-time dependent conditions.

In addition, the MIP-based techniques have been extended by further constraints to cope with the constraints induced by a given fixed size programmable processor array (see Figure 1.5). These constraints consider local *register constraints* within the processors such as different types and numbers of available registers, and *channel constraints* of the communication structure such as the number of I/O ports of a processor element.

A modular scheduling concept is presented that can also be combined with techniques for channel routing [SMHT06, WKTH08c, WKTH08b, WKTH08a] and *subword parallelism* [SMHT08].

In addition, a number of other results have been contributed, which are not exactly in the scope of this doctoral thesis. These works include the following research areas:

- Methods for distributed loop control for non-rectangular processor array architectures [BHT01, BHT02],
- Energy estimation and optimization for dedicated processor arrays, which are obtained by projection as algorithm mapping [HT02a, HT02b, HT04a],
- High-speed simulation at register transfer level [KHT04b, KHT04a],
- I/O serialization for processor arrays [HT05],
- Defragmentation of the module placement in partially reconfigurable devices [vFM+05],
- Symbolic feasibility testing during the design space exploration of heterogeneous multi-processor systems [SHHT05],
- Rapid and virtual prototyping of system-on-chip designs [KKH+06, LHT09],
- Acceleration of multiresolution imaging algorithms on graphics and Cell processors [MKD+09, MHDT09].

1.2 Organization of the Thesis

The thesis is organized as follows:

Chapter 2 reviews related work on modeling of structured computations and considered algorithm classes in the polytope model, which are essential in the course of the thesis. Subsequently, our achievement, a novel algorithm class that allows to model recurrence equations with run-time dependent conditions, is presented. Afterwards, an overview of languages for parallel programming is given. With this background, the novel functional programming language PAULA, which is based on the newly introduced algorithm class, is presented.

Chapter 3 presents contributations on scheduling and allocation of dynamic piecewise linear algorithms. It provides the major methodological results of the thesis. Here, related work in the fields of allocation, loop parallelization, and scheduling is presented first. Following, some basic definitions and concepts for mapping nested loop programs are presented. After these fundamentals, a modular concept for scheduling nested loop programs is conceived. The developed scheduling methods are based on mixed integer programming.

Chapter 4 discusses how the information obtained during the allocation and scheduling phases (Chapter 3) can be utilized in order to generate program code for a given target architecture. The range of considered target architectures include as well dedicated hardware accelerators as weakly-programmable processor arrays. In this context, the PARO design system, which has been developed in the course of the thesis, is briefly described. In addition, high-level synthesis approaches and tools with emphasis on array processors are reviewed.

After this presentation of new scheduling techniques and the description of the PARO design system, Chapter 5 provides several experimental results by applying the new methods to different examples and a number of algorithms chosen from different benchmarks. Furthermore, a comparison, in terms of performance, area and power cost, of loop unrolling with our proposed method of loop partitioning is given. A complex real world application for image processing is presented in this chapter.

Finally, concluding remarks and directions of future work are presented in Chapter 6.

Chapter 2

Modeling

Loops are the major source of parallelism in most programs. For instance, many scientific and digital signal processing programs spend a large fraction of the overall computing time in loops [Knu71,SMN$^+$03]. In order to accelerate these applications by use of parallel computation, either in software or hardware, loop parallelization techniques are required.

In this chapter, a formalism of recurrence equations, introduced by Karp, Miller, and Winograd [KMW67], is recapitulated, and related work on structured computations and algorithm classes in the polytope model [Len93, Fea96], that build upon this formalism, are discussed. All these models render important contributions in the area of loop parallelization, but have also limitations. Therefore, subsequently, our achievement, a novel algorithm class that allows recurrence equations with run-time dependent conditions, is presented.

Afterwards, an overview of languages for parallel programming is given. With this background, we present a new functional programming language based on the afore proposed algorithm class, which is tailored to our needs: The efficient specification and parallelization of loop programs for mapping them onto processor array architectures.

The main contributions of this chapter can be summarized as follows:

- Related work in the fields of structured computations, algorithm classes in the polytope model (Section 2.1.1), and parallel programming languages (Section 2.2.1) in general, are reviewed.
- The novel algorithm class of *dynamic piecewise linear algorithms* and a new model for *reduced dependence graphs* is presented in Section 2.1.2 and 2.1.4, respectively.

- The PAULA language that reflects the properties of the class of dynamic piecewise linear algorithms and many further concepts, such as reductions and full static single assignment form for multi-dimensional data flow, is presented in Section 2.2.2.

2.1 Structured Computations and Algorithm Classes in the Polytope Model

In 1967, Karp, Miller, and Winograd [KMW67] introduced in their seminal work on structured computations, the notion of a *system of uniform recurrence equations* and the concept of *dependence graphs*[1]. This concept provides a compact mathematical representation of the computations. In addition, the degree to which a computation can be processed in parallel, is characterized. Karp and others studied computation processes for solving partial differential equations by finite difference methods. Later, recurrence equations were often used for the specification of so-called *systolic* algorithms and arrays. The term *systolic array* dates back to Kung and Leiserson [KL78]. In the field of medicine, *systole* describes the contraction of the heart chambers, which presses the blood out of the chambers. Thus, a systolic array is an analogy to the regular beating of the heart and the pumping of blood. Kung and Leiserson themselves write in [KL78]:

> *A systolic system is a network of processors which rhythmically compute and pass data through the system. Physiologists use the work 'systole' to refer to the rhythmically recurrent contraction of the heart and arteries which pulses blood through the body. In a systolic computing system, the function of a processor is analogous to that of the heart. Every processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network.*

Since the whole parallelism is explicitly given, a system of uniform recurrence equations is in this feature similar to the *single assignment form* of programming languages, which are discussed in Section 2.2. The representation in single assignment form is particularly suitable for algorithmic transformations and has the following properties:

- On each right hand side of an equation, an expression is given, which defines the value of the variable standing on the left hand side of the equation (defi-

[1]Karp, Miller, and Winograd used the term *dependence graph*; but describe a folded graph that is commonly referred to as *reduced dependence graph*.

nition). Within the expression on the right hand side, other variables can be used (usage).

- Each variable exists at most once in the algorithm on the left hand side of an equation.
- For all specified equations in the algorithm, there exits a sequential execution order. That is, an order can be found where each equation, that defines a variable, appears before the usage of the variable.
- Variables that appear only on the right hand side of an equation, thus are only used but not defined, form the set of *input variables*. The other way around, variables, that are only defined but never used within the algorithm, form the set of *output variables*.

The following fragment of an algorithm is in single assignment form and consists of four statements. The variables a and b are input variables since they are only used. Similar, d and f are output variables since they are only defined.

$$\begin{aligned} c &= a-3 \\ d &= b+(c/10) \\ e &= b\cdot c \\ f &= 2e+(c-b) \end{aligned}$$

The formalism of systems of uniform recurrence equations has been used in many works. Over the years, this formalism evolved by concepts such as affine dependencies or piecewise definitions. In their original works, sometimes the authors precisely define a new structured algorithm class and give a name to it. At other times, the authors slightly modify or simplify the details of the structure. A detailed survey of different classes of structured computations and algorithms is given in [Zeh96]. In the following section of related work, classes of structured computations are discussed, that can be found in literature quite frequently or that are of interest for this thesis.

2.1.1 Related Work

Uniform Recurrence Equations. As stated before, in 1967, for modeling and for understanding regular and iterative processes, Karp and others [KMW67] introduced the concept of a *system of uniform recurrence equations* (SURE). Here, the authors consider problems associated with the evaluation of a system of k functions $a_1(I)$, $a_1(I)$, …, $a_k(I)$. Each function $a_i(I)$, $i \in [1..k]$ is quantified for all points

$I \in \mathcal{I}$, where $\mathcal{I}$ is an integral subset $\mathcal{I} \subseteq \mathbb{Z}^n$. The values of $a_1(I), a_1(I), \ldots, a_k(I)$ have to satisfy a system of k recurrence equations having *uniform dependencies*. First, we consider the case $k = 1$ in order to explain a uniform dependency. A single recurrence equation is of the form

$$a_1(I) = \mathcal{F}_1(a_1(I - d_1), a_1(I - d_2), \ldots, a_1(I - d_k)),$$

where $I \in \mathcal{I}$, d_j, $j \in [1..k]$ is an n-dimensional integer vector called *iteration vector*, and $\mathcal{F}_1$ is a single-valued function. Any difference $I - d_j$ is element of $\mathbb{Z}^n$ and each vector d_j is constant, thus the equation is said to have *uniform dependencies*. With this, a system of uniform recurrence equations is generally given by

$$a_i(I) = \mathcal{F}_i(a_{i_1}(I - d_{i_1}), a_{i_2}(I - d_{i_2}), \ldots, a_{i_k}(I - d_{i_k})).$$

An example for $I \in \mathbb{Z}$ is defined by the following system of equations:

$$\begin{aligned} a_1(I) &= \mathcal{F}_1(a_2(I), a_3(I-1)) \\ a_2(I) &= \mathcal{F}_2(a_1(I+1), a_1(I-2), a_3(I)) \\ a_3(I) &= \mathcal{F}_3(a_3(I-1)) \end{aligned}$$

Karp and others also studied the computability of SUREs [KMW67]. The computability of a SURE can be determined by whether there is a cycle of zero weight in the reduced dependence graph.

In the following and throughout the rest of the thesis, we use a slightly different terminology when considering recurrence equations. Namely, instead of functions $a_i(I)$, indexed variables $x_i[I]$ are considered. Equations are quantified, that is, an equation defines the indexed variable on its left hand side for each value of the iteration vector I. The set $\mathcal{I} \subseteq \mathbb{Z}^n$ of possible values of I is called *iteration space*. According to this, a *uniform algorithm* consists of a set of equations, that relate linearly indexed variables. In general, we write

$$x_i[I] = f_i(x_{i_1}[I - d_{i_1}], x_{i_2}[I - d_{i_2}], \ldots, x_{i_k}[I - d_{i_k}]) \qquad \forall I \in \mathcal{I}$$

with $\mathcal{I} \subseteq \mathbb{Z}^n$, $d_{i_j} \in \mathbb{Z}^n$, $j \in [1..k]$, any difference $I - d_{i_j} \in \mathbb{Z}^n$, and f_i is an arbitrary function. For example,

$$\begin{aligned} a[i,j] &= a[i,j-1] \\ b[i,j] &= \text{add}(a[i,j], b[i-1,j]) \\ c[i,j] &= \text{mul}(b[i,j], b[i,j]) \end{aligned}$$

where the system of equations is defined for all i and j in $\mathcal{I} = \{(i\ j)^{\mathrm{T}} \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \wedge 2 \leq j \leq 8\}$. Often, all data dependencies d_i, $1 \leq i \leq k$ are combined in one *dependence matrix* $D = (d_1\ d_2\ \dots\ d_k)$. For the above example, the following dependence matrix would result, if the variables on the right hand side of the equations are considered top down.

$$D = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The importance and versatile applicability of SUREs is studied in a multitude of works. Examples of these include the work of Quinton [Qui84], Delosme and Ipsen [DI85, DI86], Schwiegelshohn and Thiele [ST87a, ST87b, ST88], Djamegni [Dja04, Dja06], and many further works [XL91, KR06, OSY06, DQR+09] for the implementation of systolic arrays and asynchronous architectures [EJP92]. Some authors [SF91] even focus only on uniform algorithms that consist of a single equation. SUREs serve not only as basis for the synthesis of systolic arrays but are also used for other parallel models of computation. For instance, the authors in [GJM00] use SUREs for computing the minimal memory size in a PRAM (parallel random access memory) model. In [HOPS05], SUREs are used for the generation of parallel algorithms for cluster and grid computing. Schaffer et alii start from SUREs to exploit architectures with sub-word parallelism [SMC03, SSM06, SM06a, SMC06].

Systems of uniform recurrence equations also form the basis for the high-level synthesis tool PICO-NPA [SAM+02] (further details follow in Section 4.1.1.2).

Regular Iterative Algorithms. The class of *regular iterative algorithms* (RIA) is a continuation of the SUREs, shaped by Rao in [Rao85]. The first extension is that, each quantified equation might be assigned a further restriction of its iteration domain. This restriction is defined by one, or a set of inequalities (intersection of half spaces) and is referred to as *iteration dependent condition* in the following.

Definition 2.1 (Iteration dependent condition)**.** *A condition $\mathcal{C}^{\mathrm{I}}(I)$ is called* iteration dependent condition *of an equation and can be equivalently expressed by $I \in \mathcal{I}_{\mathcal{C}} \subseteq \mathbb{Z}^n$, where the space $\mathcal{I}_{\mathcal{C}}$ is an iteration space called* condition space.

In addition, the iteration variables on the left hand side of an equation can have an offset and the iteration space $\mathcal{I}$ is required to be convex. Then, each equation i is defined for all $I \in \mathcal{I}$ as follows.

$$x_i[I+f] = \mathcal{F}_i\left(x_1[I-d_{1i}], x_2[I-d_{2i}], \dots, x_k[I-d_{ki}]\right) \qquad \text{if } \mathcal{C}_i^{\mathrm{I}}(I)$$

In Example 2.1 the matrix multiplication of two square matrices ($C = A \cdot B$) is described by a regular iterative algorithm.

Example 2.1.

$$\begin{aligned}
a[i,j,k] &= a_{ik} && \text{if } j=1\\
b[i,j,k] &= b_{kj} && \text{if } i=1\\
a[i,j+1,k] &= a[i,j,k] && \text{if } j<N\\
b[i+1,j,k] &= b[i,j,k] && \text{if } i<N\\
c[i,j,k] &= a\,[i,j,k]\cdot b\,[i,j,k] && \text{if } k=1\\
c[i,j,k] &= c[i,j,k-1]+a\,[i,j,k]\cdot b\,[i,j,k] && \text{if } k>1\\
c_{ij} &= c[i,j,k] && \text{if } k=N
\end{aligned}$$

Many related work in the fields of loop parallelism [RK90, HH95] and data-path synthesis [AC94] is based upon this algorithm class. For further literature about the class of regular iterative algorithms, we refer to the overview in [RG06].

Piecewise Linear Algorithms and Piecewise Regular Algorithms. The class of *piecewise linear algorithms* (PLA) has been defined by Thiele and Roychowdhury in [TR91]. This class extends the notation of regular iterative algorithms. A piecewise linear algorithm consists of a set of N quantified equations, $S_1[I],\ldots,S_i[I],\ldots,S_N[I]$. Each equation $S_i[I]$ is defined for all $I \in \mathcal{I}_i$ and is of the following form.

$$x_i[P_iI+f_i]=\mathcal{F}_i(\ldots,x_j[Q_jI-d_{ji}],\ldots) \qquad \text{if } \mathcal{C}_i^{\mathrm{I}}(I)$$

where x_i, x_j are affinely indexed variables. The so-called *indexing functions* of the variables are defined by the constant rational indexing matrices P_i, Q_j and by the constant rational vectors f_i, d_{ji} of corresponding dimension. $\mathcal{F}_i$ denote arbitrary functions and the dots $(\ldots)$ denote similar arguments. $I \in \mathcal{I}_i \subseteq \mathbb{Z}^n$ is a linearly bounded lattice (definition follows), called iteration space of the quantified equation $S_i[I]$. The set of all points P_iI+f_i, $I \in \{\mathcal{I}_i \cap \mathcal{C}_i^{\mathrm{I}}(I)\}$ is called the index space of variable x_i.

A piecewise linear algorithm is called *piecewise regular algorithm* (PRA) [Thi88], if the matrices P_i and Q_j are the identity matrix.

An algorithm is in *output normal form* if P_i is the identity matrix and f_i is the zero vector.

These algorithm classes have been extensively studied by Teich and Thiele for the design of processor arrays [TT91, Thi92, Tei93]. A similar algorithm class, also able to handle affine data dependencies, is known as *system of affine recurrence equations* (SARE) [YC88, YC92, WS94, MP01].

Note that the transformation, maybe automatically, of affine data dependencies into uniform dependencies is called *localization.* Amongst others, localization techniques have been studied in the following works [TR91, Tei93, MMRR01, Eck01, MM04] and our work in [HT05].

It should be noticed, that many authors (for instance, [FM98,SMC00,MFMS03, GQR03, SM06b, SM06c, SM06d]) use the term SURE even if, according to the above taxonomy, RIAs or PRAs are meant.

Further Algorithm Classes. Roychowdhury and others introduced in [RTRK88] so-called *linearly indexed weak single assignment codes*. Here, the main novelty is the possibility to consider a generic operator on the right hand side of the equations. The operator is required to be associative and commutative, and is defined over several iterations. Examples of such operators are reductions, as for instance, summations or multiplications. This algorithm form is used for example by Eckhart in [EM99], although later he called this class *affine indexed algorithms* [Eck01].

Eventually, none of the aforementioned algorithm classes is powerful enough to specify *dynamic data dependencies*. This means, that branching of the program execution cannot depend on already computed variables. Only few results exist that incorporate specific run-time decisions. With regards to the loop bounds, the authors in [GGL99] propose an approach, which can handle while-loops. The method has been integrated in the parallelizing compiler LooPo [GL96]. In [Meg93] and [AR94], the authors study also a class of run-time dependencies. Their work studies dynamic programming for knapsack problems by consideration of indirect addressing. Furthermore, the authors show how optimal systolic array-like implementations can be derived. In [SD03,Ste04], the authors present a method to derive a so called *dynamic single assignment code* (dSAC), based on *fuzzy array data flow analysis* [CBF95]. The difference compared to single assignment code, where every left hand side variable is written exactly once, is, that in dSAC every variable is written at most once. That means, at compile-time, it is unclear whether a variable will be defined during the program execution.

2.1.2 Dynamic Piecewise Linear/Regular Algorithms

In order to consider and systematically map not only algorithms with iteration dependent conditions, which are static and known at compile time, in the following, we extend the algorithm class to allow for *run-time dependent conditions*.

Definition 2.2 (Run-time dependent condition). *A* run-time dependent condition *is a Boolean variable $C^{\mathrm{RT}}[I]$ of the form*

$$C^{\mathrm{RT}}[I] = \mathcal{F}_C(\ldots, x_j[Q_j I - d_j], \ldots)$$

where $\mathcal{F}_C$ denotes an arbitrary Boolean-valued function involving constants and linearly indexed variables only.

Typically, the function $\mathcal{F}_C$ of a run-time dependent condition describes a relational operator such as $=$, $>$, $\geq$, $<$, $\leq$, or, $\neq$.

Definition 2.3 (Dynamic Piecewise Linear/Regular Algorithms [HT04b])**.** *A dynamic piecewise linear algorithm (DPLA) is a PLA where the following extended type of equations expresses run-time dependent definitions of computations as follows:*

$$x_i[P_i I + f_i] = \begin{cases} \mathcal{F}_i^1(\ldots, x_j[Q_j I - d_{ji}], \ldots) & \text{if } (\mathcal{C}_i^{\mathrm{I}}(I) \wedge \mathcal{C}_i^{\mathrm{RT}}[I]) \\ \mathcal{F}_i^0(\ldots, x_k[Q_k I - d_{ki}], \ldots) & \text{if } (\mathcal{C}_i^{\mathrm{I}}(I) \wedge \neg\mathcal{C}_i^{\mathrm{RT}}[I]) \end{cases}$$

The notation $\neg\mathcal{C}_i^{RT}[I]$ denotes the negation of the run-time dependent condition $\mathcal{C}_i^{RT}[I]$, which is similar to the else-branch of an if-condition. We introduce intermediate variables $x_i^1[P_i I + f_i] = \mathcal{F}_i^1(\ldots)$ and $x_i^0[P_i I + f_i] = \mathcal{F}_i^0(\ldots)$ to express the conditional definition of $x_i[P_i I + f_i]$. A DPLA is called dynamic piecewise regular algorithm *(DPRA) if the matrices P_i, Q_j, and Q_k are the identity matrix.*

Note that by this definition we can strictly partition each condition into an iteration dependent condition and a run-time dependent condition (*separability*). Because of both, the run-time dependent condition ($\mathcal{C}_i^{\mathrm{RT}}$) and the negated run-time dependent condition ($\neg\mathcal{C}_i^{RT}$), the variable on the left hand side of an equation is defined whensoever $\mathcal{C}_i^{\mathrm{I}}(I)$ is fulfilled and thus, the computability property of a program remains satisfied. The definition of a variable in either case is the main difference compared to the approach presented in [SD03, Ste04] where not both cases have to be defined.

For illustration, two small examples are given. The first is a DPRA and the second describes a DPLA. The iteration space is omitted in each case.

Example 2.2.

$$x[i] = \begin{cases} 2 \cdot \frac{\cos(y[i-1])}{\sin(x[i-1])} & \text{if } (x[i-1] \neq 0) \\ \infty & \text{if } (x[i-1] = 0) \end{cases}$$

Example 2.3.

$$\begin{aligned} x[i,j] &= \begin{cases} x^1[i,j] & \text{if } (\mathcal{C}^{\mathrm{RT}}[i,j] \wedge (i > 0 \wedge j \leq 3)) \\ x^0[i,j] & \text{if } (\neg\mathcal{C}^{\mathrm{RT}}[i,j] \wedge (i > 0 \wedge j \leq 3)) \end{cases} \\ x^1[i,j] &= y[i,j] \cdot z[2i-1,j] \\ x^0[i,j] &= y[i,j] \\ \mathcal{C}^{\mathrm{RT}}[i,j] &= (z[2i-1,j] > 1) \end{aligned}$$

2.1.3 Iteration Spaces

So far, the possible domains of definition for the algorithms have not been formalized, except that we stated sometimes these iteration spaces have to be convex or an n-dimensional integral set. Formally, an iteration space is defined as follows.

Definition 2.4 (Iteration space). *An* iteration space $\mathcal{I}$ *is a set. Its elements are valid values for an index vector I. The iteration space is a discrete—not necessarily finite—set.*

Throughout this thesis, we assume that the iteration space $\mathcal{I}$ is an n-dimensional subset of integers, $\mathcal{I} \subset \mathbb{Z}^n$. Often, the terms *iteration space* and *index space* are used synonymously, however, we distinguish between the terms. The iteration space is the *domain* $\mathcal{I}$ whereas the index space denotes the *codomain* when transforming the iteration space by an index function.

In loop programs, the iteration space is defined by the loop bounds. The iteration variables are generally increased or decreased by a constant value (regularity of the iteration space). If the loop programs are *perfectly nested*[2], the iteration spaces are convex. Each loop bound defines a half space and the intersection of all half spaces describes a polyhedron or in case of boundedness a polytope. Loop parallelization in the polytope model is a powerful technique [Len93, Fea96], therefore in the following, the iteration spaces are formulated as polyhedra or even more general as so-called *linearly bounded lattices*.

Definition 2.5 (Linearly bounded lattice [Tei93]). *A* linearly bounded lattice *(LBL) denotes an iteration space of the form*

$$\mathcal{I} = \{I \in \mathbb{Z}^n \mid I = Mx + c \ \wedge \ Ax \geq b\}$$

where $x \in \mathbb{Z}^l$, $M \in \mathbb{Z}^{n \times l}$, $c \in \mathbb{Z}^n$, $A \in \mathbb{Z}^{m \times l}$ and $b \in \mathbb{Z}^m$. $\{x \in \mathbb{Z}^l \mid Ax \geq b\}$ denotes the set of integral points within a convex polyhedron or in case of boundedness within a polytope in $\mathbb{Z}^l$. This set is affinely mapped onto iteration vectors I using an affine transformation ($I = Mx + c$).

In [Tei93], it is shown that each set $\mathcal{I} \subseteq \mathbb{Z}^n$ can be described as an LBL. Although this is an interesting result, at the same time, this possible generality, to represent an arbitrary n-dimensional, integral point set, could make the treatment of a given algorithm very complicated and inefficient. Thus, throughout the thesis, we assume that the matrix M is square and of full rank. Then, each vector x is uniquely mapped to an iteration point I.

Often, when the lattice matrix M is the identity matrix and the vector c is zero, the iteration space coincides with the following definition of a polyhedron.

$$\mathcal{I} = \{I \in \mathbb{Z}^n \mid AI \geq b\}$$

[2]A loop nest is called *perfectly nested* if only the innermost loop contains statements.

2.1.4 Dependence Graphs

Another, geometric approach for modeling single assignment algorithms, is the representation by a *dependence graph*. For each variable instance at each iteration point $I \in \mathcal{I}$, there exists one node. Dependencies from one node (variable) to another are denoted by an edge in the graph. In detail, if there exists an edge from a node $a[J]$ to a node $b[K]$ with $J, K \in \mathcal{I}$, then $b[K]$ needs the result of $a[J]$ for its computation. Thus, a dependence graph expresses a partial order between the different operations of the algorithm. Since the iteration space of an algorithm can be arbitrarily large, also the graph can be arbitrarily large. If algorithms with uniform data dependencies are considered, the dependence graph is also regular and can be *folded* to a *reduced dependence graph* [Rao85, Thi88].

Definition 2.6 (Reduced dependence graph)**.** *Let a uniform algorithm in output normal form be given with $\mathcal{I} \subseteq \mathbb{Z}^n$. Then a corresponding* reduced dependence graph *(RDG) $G = (V, E, D)$ of dimension n has the following properties. There exists a set V of nodes and a set of edges $E \subseteq V \times V$. For each variable x_i of the algorithm exists one node $v_i \in V$ in the RDG. In addition there exists a mapping $E \mapsto D$ that assigns to each edge $e = (v_i, v_j) \in E$ an n-dimensional dependency vector $d_{ij} \in \mathbb{Z}^n$ if a variable x_j of the algorithm depends on a variable x_i of the algorithm.*

As example, an algorithm for IIR filtering is considered, that is used for speech analysis and synthesis with *linear predictive coding* (LPC) [PM06].

Example 2.4.

$$\begin{aligned}
z[i,j] &= a[i,j] \cdot x[i-1,j-1] \\
b[i,j] &= a[i,j] \cdot c[i,j-1] \\
c[i,j] &= c[i,j-1] + z[i,j] \\
x[i,j] &= x[i-1,j-1] + b[i,j]
\end{aligned}$$

where the iteration space is defined by $\mathcal{I} = \{(i\ j)^{\mathrm{T}} \in \mathbb{Z}^n \mid 1 \leq i \leq N \wedge 1 \leq j \leq M\}$.

The reduced dependence graph of the algorithm in Example 2.4 is shown in Figure 2.1. Variables that depend on the same iteration have a zero dependency vector and are not annotated to the edges.
The *unfolded* dependence graph is shown in Figure 2.2, where the upper bounds of the iteration space are chosen to be $N = 6$ and $M = 4$. The regularity of the algorithm is clearly visible. The concepts of RDGs and uniform algorithms are closely related to each other. However, with Definition 2.6 of an RDG, it is not possible do describe algorithms that go beyond the concept of uniform recurrence equations. Thus, a more general definition is necessary.

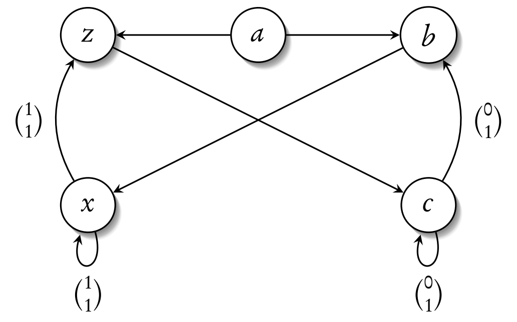


Figure 2.1: Reduced dependence graph of the LPC algorithm given in Example 2.4.

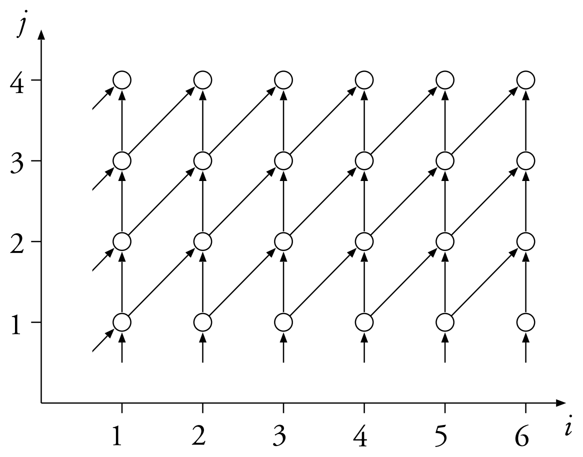


Figure 2.2: Dependence graph of the LPC algorithm for $N = 6$ and $M = 4$.

Definition 2.7 (Reduced dependence graph (extended)). *Let a dynamic piecewise linear algorithm be given. Then each node $v \in V$ of the corresponding* reduced dependence graph *(RDG) $G = (V, E)$ might have several*[3] *of the following types.*

- Normal*: For each indexed variable on the left hand side of the algorithm, one node of type* normal *exists in the graph.*
- Constant*: For each constant within the algorithm, one node of type* constant *exists in the graph.*
- Input*: Variables that are only used in the algorithm or rather nodes that are only used in the graph are of type* input.
- Output*: Similar to the input type. Nodes, that are only defined but not used in the graph are of type* output.

[3]For instance, a node might be a merge node as well as an output node at the same time.

- Propagation*: Similar to the normal type but no function or operation is associated to the node, only a copy operation/propagation.*
- Condition*: To denote a run-time dependent condition* C^{RT}.
- Merge*: A node to merge the data flow again after it was split up into different branches.*

Also different edge types exist.

- Normal*: An edge that denotes a data dependency from one node to another node.*
- Constant*: An edge that denotes the usage of a constant.*
- Conditional*: This type is used for outgoing edges of a node of type condition.*
- Serialization*: Denotes an artificial serialization between two nodes. That means, the serialization is not directly included in the originally given algorithm description but was added afterwards.*

To each node of type normal, input, output, or propagation further attributes are associated. Namely, the iteration domain it is defined for and the indexing function of the variable. Moreover, a unique identifier, the variable name, and its functionality are annotated.

To each edge of type normal, the corresponding indexing function is annotated. Compared to the traditional definition of an RDG (cf. Definition 2.6), this also allows to represent affine data dependencies.

For illustration, the following example of an algorithm for matrix multiplication, consisting of six equations (S_1 to S_6), is considered.

Example 2.5.

$$\begin{array}{lll}
S_1: & a[i,j,k] = a_{in}[i,k] & \text{if } j = 1 \\
S_2: & a[i,j,k] = a[i,j-1,k] & \text{if } j > 1 \\
S_3: & z[i,j,k] = a[i,j,k] \cdot b_{in}[k,j] & \\
S_4: & c[i,j,k] = z[i,j,k] & \text{if } k = 1 \\
S_5: & c[i,j,k] = c[i,j,k-1] + z[i,j,k] & \text{if } k > 1 \\
S_6: & c_{out}[i,j] = c[i,j,k] & \text{if } k = N
\end{array}$$

with $\mathcal{I} = \{(i\ j\ k)^T \in \mathbb{Z}^3 \mid 1 \leq i,j,k \leq N\}$ *as iteration space.*

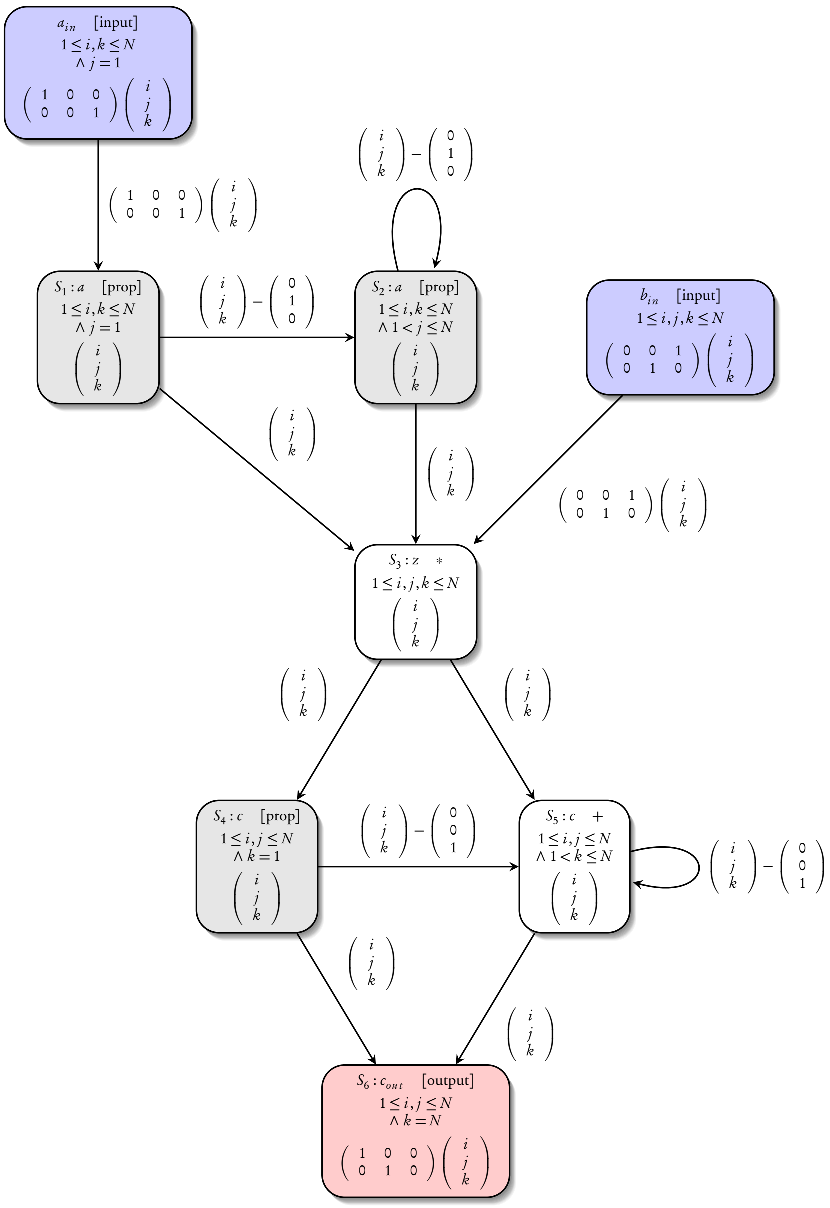

a_{in} [input]
$1 \le i,k \le N$
$\wedge\, j = 1$
$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$
$S_1 : a$ [prop]
$1 \le i,k \le N$
$\wedge\, j = 1$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$
$S_2 : a$ [prop]
$1 \le i,k \le N$
$\wedge\, 1 < j \le N$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
b_{in} [input]
$1 \le i,j,k \le N$
$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
$S_3 : z$ $*$
$1 \le i,j,k \le N$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
$S_4 : c$ [prop]
$1 \le i,j \le N$
$\wedge\, k = 1$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$
$S_5 : c$ $+$
$1 \le i,j \le N$
$\wedge\, 1 < k \le N$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
$\begin{pmatrix} i \\ j \\ k \end{pmatrix}$
$S_6 : c_{out}$ [output]
$1 \le i,j \le N$
$\wedge\, k = N$
$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} i \\ j \\ k \end{pmatrix}$

Figure 2.3: Reduced dependence graph for the matrix multiplication algorithm in Example 2.5

The corresponding RDG is shown in Figure 2.3. Here, different node types have different colors, input nodes are blue, output nodes are red, normal nodes are white, and propagations are colored gray. The distinction in terms of color between normal nodes and propagations only serves for better visibility in order to distinguish actual computations from propagations/assignments quickly. As can be seen, not all variables have to be embedded in the same dimension. For instance, the input variables

a_{in} and b_{in} as well as the output variable c_{out} are only two-dimensional, whereas all other variables are three-dimensional. Note that this characterization possibility also allows to model piecewise defined algorithms and even parallel/communicating loop programs defined over different iteration spaces. This is a substantial difference compared to traditional approaches where the algorithm has to be embedded in a common global iteration space, which is derived by the convex hull of the union of iteration subspaces.

The next program fragment, in Example 2.6, depicts a dynamic piecewise regular algorithm, consisting of seven equations (S_1 to S_7).

Example 2.6.

$$\begin{aligned}
S_1 &: \quad C^{\mathrm{RT}}[i] = (x[i] > 7)\\
S_2 &: \quad a[i] = \begin{cases} a^1[i] & \text{if } (C^{\mathrm{RT}}[i])\\ a^0[i] & \text{if } (\neg C^{\mathrm{RT}}[i]) \end{cases}\\
S_3 &: \quad a^1[i] = x[i] + 3\\
S_4 &: \quad a^0[i] = x[i] \cdot 5\\
S_5 &: \quad b[i] = \begin{cases} b^1[i] & \text{if } (C^{\mathrm{RT}}[i])\\ b^0[i] & \text{if } (\neg C^{\mathrm{RT}}[i]) \end{cases}\\
S_6 &: \quad b^1[i] = a[i] \cdot a[i]\\
S_7 &: \quad b^0[i] = a[i] + 4
\end{aligned}$$

with the LBL $\mathcal{I} = \{i \in \mathbb{Z} \mid i = 2x + 1 \wedge 1 \leq x \leq 10\}$ *as iteration space.*

The corresponding RDG is illustrated in Figure 2.4. In this figure, constant nodes are colored green, conditions are colored orange, and merge nodes are shown in yellow. The results of the condition (Boolean value) are drawn as red edges to the merge nodes in order to select the right input.

2.2 Languages for Parallel Programming

Languages for parallel programming have a long tradition in parallel and high performance computing, but also when considering digital signal processing and data flow computing. The following section about related work gives only an overview with an emphasis on functional languages for data flow intensive and streaming applications.

2.2.1 Related Work

As mentioned in the introduction, there is a continuous trend to higher abstraction levels and the usage of high-level languages for the design of embedded digital sys-

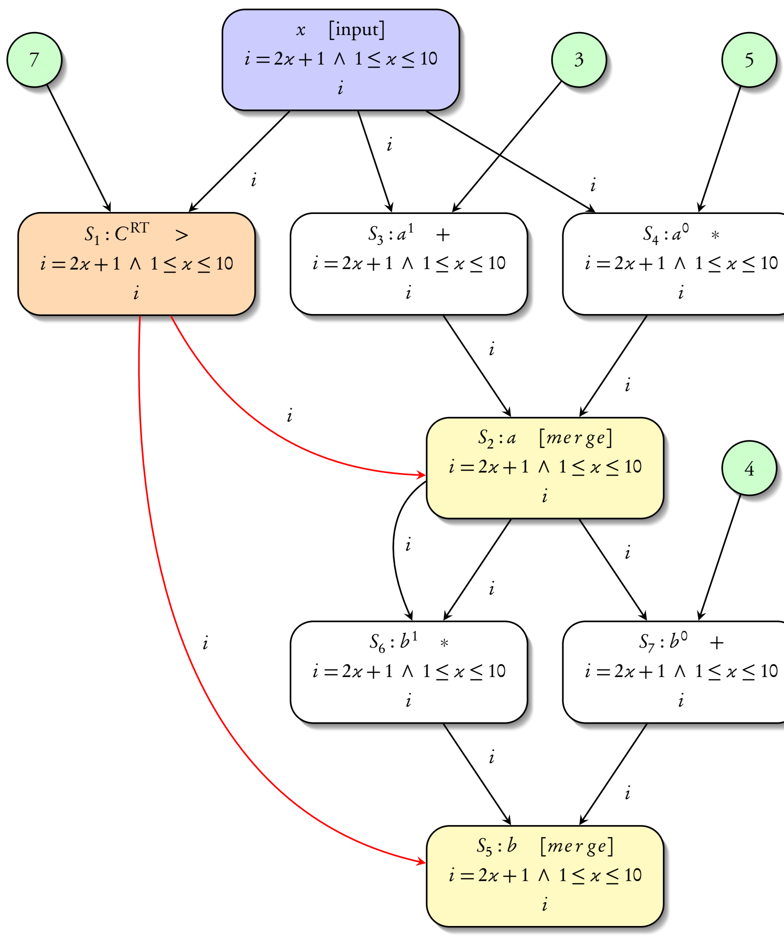


Figure 2.4: Reduced dependence graph of the DPRA given in Example 2.6.

tems. However, starting with a sequential language such as C, C++, or SystemC has the disadvantage that their semantics force a lot of restrictions on the execution order of the program. Most of the parallelism contained in the original mathematical model of the algorithm is lost during the transformation to sequential code. For instance, a simple summation $s = \sum_{i=0}^{7} a[i]$ is often written in C as a for loop in the following manner:

```
int s = 0;
for (i=0; i<=7; i++) { s += a[i]; }
```

By this, a sequential order is already predefined and a later parallelization can become a crucial task. In Figure 2.5, the difference between a sequential and a parallel implementation shall be demonstrated. Granted that enough resources (adders) are available, the latency might be reduced from linear to logarithmic run-time. The

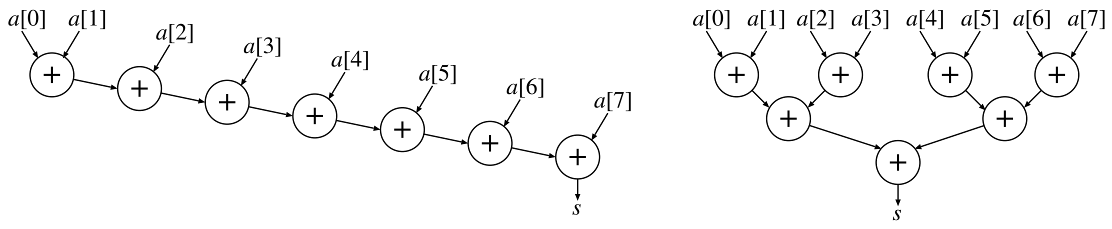


Figure 2.5: Sequential and parallel implementation of a sum consisting of eight summands.

mapping of such algorithms to massively parallel architectures requires data dependency analysis in order to make the inherent parallelism explicit. Still, this process is very complex, since sequential languages allow, that variables, once defined, can be overwritten arbitrarily. Another disadvantage of C-based hardware design is that most design tools support only a limited subset of the language. Porting existing, highly optimized C code to such an environment is a time consuming task and often ends in completely rewriting the code from scratch. In order to avoid this, modern software compilers like gcc [GCC09] as of version 4 or LLVM [LA04] use a so-called *static single assignment (SSA)* form as intermediate representation, where each variable is written exactly once. SSA allows the application of manifold compiler optimizations and transformations in a very efficient way. But since the SSA form is used only in the intermediate representation (basic block level), these compilers cannot solve the data dependence analysis problem for multi-dimensional arrays.

Thus, starting from a parallel programming language is advantageous. For the sake of completeness, the most popular languages for data parallelism and shared memory systems, namely High Performance Fortran (HPF) [Hig93] and C* [RS87] should be mentioned. HPF is based on Fortran 90 with extensions for parallel computing (for example, FORALL-statements) and data layout. The language C* is an extended version of C, which includes domains and reduction operators. Today, in most application domains, C* as well as HPF are being replaced by OpenMP [CJP07].

2.2.1.1 Languages for Functional Programming and Loop-Level Parallelism

Another option is to directly start from a functional language such as Id [Nik93], Haskell [Tho99], or Sisal [GDF$^+$01]. Sisal allows recursion and finite streams. It was derived from VAL [McG82], a function-based language designed for data flow computers. Sisal and Id can be classified as shared memory parallel programming languages. Because of many similarities between Id and Haskell, it was a foundation for pH, a parallel dialect of Haskell. However, Haskell also has only restricted abilities to handle true multi-dimensional arrays (that means, arrays in which every dimension is treated as equivalent) [Tho99]. All these languages have been mainly designed for programming shared memory systems.

2.2.1.2 Message Passing Languages

The basic concept of message passing languages requires the programmer explicitly to partition tasks and data. An example of a message passing language is CSP (*communicating sequential processes*), presented by Hoare in [Hoa78]. CSP is used for describing the interaction of processes in parallel systems. For this purpose, CSP programs are formulated as a parallel composition of several sequential processes, communicating with each other through synchronous message passing. Later, CSP has been mathematically refined in a process algebra. Today, several implementations in other languages such as Java, C++, and Haskell exist [Hoa85].

A concrete early adaption of CSP has been used in Occam [May83, Inm84], a concurrent programming language targeting transputer microprocessors [MS90].

Sometimes, MPI (message passing interface) [SO98] is also referred to as a message passing language, which is wrong since MPI is a language-independent communications protocol. It offers an application programmer interface for writing parallel programs for distributed memory systems.

2.2.1.3 Streaming Languages

In recent years, more and more streaming applications in many fields, such as multimedia, graphics, and other digital signal processing areas have emerged, and the number of new streaming languages is constantly growing. Streaming languages have the advantage, that they are closely related to data flow graphs, in which computations are represented by nodes and communications by edges. By this, concurrent implementations can be derived more easily, as opposed to starting from a sequential language and subsequent parallelization.

Examples of recent streaming languages, targeting graphics and other multi-core architectures include StreamIt, Spidle, Brook, Baker, SPUR, StreamC, CUDA, Cg, and SPEX, which are shortly discussed in the following.

Amarasinghe and others propose a language for streaming applications called StreamIt [TKA02], which is based on the model of synchronous data flow (SDF). StreamIt offers a structured model of streams, a messaging system for control, and re-initialization mechanisms [TKA02]. Remarkable, is the wide variety of studied target architectures starting form standard uniprocessor systems to multi-core architectures such as RAW [GTA06], the Cell Broadband Engine Architecture [Zha07], graphics processing units, and FPGAs [HKM$^+$08].

Spidle [CHR$^+$03] is a high-level language for declarative programming of streaming applications in the domain of digital audio processing.

Brook [BFH$^+$04] is a high-level language intended for programming graphics hardware both from NVIDIA and ATI. The language is imperative with explicit constructs for streaming structures, abstractions for the memory access, data-parallel operations executed on the graphics processors (GPU) as calls to parallel functions, and many-to-one reductions.

In [CLL$^+$05], the authors use Baker, a domain specific C-like high-level language, for the design of modular applications for Intel's IXP2400 network processor.

The authors in [ZLSL05] propose the SPUR programming model for a novel media processor with programmable vision coprocessor. The language is divided into a high-level and a low-level part. The low-level part describes sequences of operations that work on a stream, whereas the high-level part of the language is intended for describing the skeleton of a program, also known as the control flow.

StreamC [DDM06] is a C++ extension for defining high-level control and data flow in stream programs. The computation kernels themselves have to be defined in KernelC [DDM06], a language with limited C-like syntax.

CUDA [LNOM08] is a C-based parallel programming language for NVIDIA's graphics and Tesla architectures. A programmer has to write a sequential program that calls parallel kernels, which can be simple functions or programs. The compiler ensures that the sequential code is executed on the CPU of the host system and that parallel kernels are computed by a set of concurrent threads on the GPU. Another older approach, also from NVIDIA, is Cg [MGAK03] a C-like language, specifically for programming computer graphics and high-level shading. It offers support for both major 3D graphics APIs: OpenGL and DirectX.

OpenCL (Open Computing Language) [Khr08] is a recent parallel programming language with emphasis on computer graphics. OpenCL is based on the language C and is extended by functions and data types for the parallel execution of applications. The OpenCL execution model supports data and task parallel programming models, as well as hybrids of these two models [Khr08]. It is intended for various types of parallel systems, including high-performance compute servers, desktop computer systems, and handheld devices. In order to support a broad range of processors and heterogeneous architectures, such as multi-core CPUs, GPUs, Cell-type architectures, and DSPs, various memory types are available.

Index spaces in CUDA and OpenCL are limited to a maximum dimension of three. There exist only few streaming languages for arbitrary multi-dimensional data flow. One example is SPEX [LCM$^+$08], a language extension applied to C++. However, many features of the C++ language are not supported.

2.2.1.4 Visual Programming Languages

Visual programming languages allow the formulation of a program by entering program elements graphically rather than by specifying them textually. Most visual programming languages are based on graph models. Thus, their basic elements are nodes or boxes connected by arrows. Well-known examples of visual programming languages include Click [KMC+00] for packet processing applications, Simulink [DH01] for modeling and simulation of dynamic systems, and Ptolemy II [LN07], which allows to couple different models of computation. For an overview of older graphical data flow languages, we refer to [LP95] and [Hil92].

2.2.2 The PAULA Language

PAULA is a domain-specific functional programming language, dedicated for mapping computationally intensive algorithms onto parallel tightly-coupled processor architectures with local distributed memory. The class of algorithms that can be expressed by a PAULA program is based on the mathematical model of dynamic piecewise linear algorithms (see Section 2.1.2). That means, the language can handle run-time dependent data flow to a certain degree. In addition, the language contains constructs that go beyond the concepts of DPLAs, for instance, the convenient usage of big operators (reductions) such as $\sum$.

As described before, a DPLA consists of a set of recurrence equations. For instance, when modeling signal processing algorithms, a designer naturally considers mathematical equations. Hence, the PAULA language is very intuitive. A program is thus a system of quantified equations that implicitly defines a function of output variables in dependence of input variables.

The language provides a full SSA form, also for multi-dimensional arrays. Compared to other functional languages, PAULA has powerful expressions to specify polyhedral and lattice iteration domains. Finally, besides behavioral description possibilities, also architectural modeling and constraints can be considered. For this purpose, a program described in PAULA may be divided into two major parts: A part for the behavioral description and a part for the architectural description. Thus, at first, in Section 2.2.2.1 the syntax and semantics of the behavioral part are described. In Section 2.2.2.2, the architectural part of the language is discussed.

2.2.2.1 Behavioral Description

Several semantical properties have to be considered by the programmer when specifying an algorithm in the PAULA language. *Single assignment property*: Any instance of an indexed variable appears at most once on the left hand side of an equation. *Computability*: There exists a partial ordering of the equations such that any instance

Algorithm 1: Example PAULA program (FIR filter)

```
program FIR
{
  // Type alias definitions
  typealias coeff_t signed fixed<12,11>;
  typealias input_t signed fixed<16,15>;
  typealias product_t signed fixed<28,26>;
  typealias output_t signed fixed<36,26>;

  // Variable declarations
  variable A 1 in coeff_t;
  variable U 1 in input_t;
  variable Y 1 out output_t;
  variable a 2 coeff_t;
  variable u 2 input_t;
  variable x 2 product_t;

  // Parameter declarations
  parameter N;
  parameter T;

  // Program blocks
  par (i >= 0 and i <= T-1)
  { // Nested program blocks
    par (j >= 0 and j <= N-1)
    { // Equations
      a[i,j] = A[j];
      u[i,j] = U[i-j]      if (i-j >= 0);
      u[i,j] = 0           if (i-j <= -1);
      x[i,j] = a[i,j] * u[i,j];
    }
    // More equations
    Y[i] = SUM[j >= 0 and j <= N-1](cast<output_t>(x[i,j]));
  }
}
```

of a variable appearing on the right side of an equation appears earlier in the left hand side in the partial ordering. If the two aforementioned properties are respected, the following architecture independent *execution model* can be considered[4]. A program

[4]The PARO synthesis tool, which is described in Section 4.2, offers two methods for checking the single assignment property and computability, respectively. The first method investigates characteristics of the indexing functions and can be applied to regular algorithms. The second method can

may be executed as follows: (1) All instances of equations are ordered respecting the above defined partial ordering. (2) The indexed variables are determined by successive evaluation of equations.

In order to enable the synthesis of dedicated hardware accelerators or the mapping onto tightly-coupled multi-processor architectures, the mathematical model of DPLAs have been extended by several constructs. Algorithm 1 shows an example program that describes an FIR filter. The individual language elements are described in the subsequent sections.

Programs. The key element of the PAULA language is the *program*. Every program starts with a program header that contains the program name. The header is followed by a declarative part that includes type alias definitions and declarations of variables and parameters in an arbitrary order[5]. After the declarations, there can be one or more so called *program blocks*, which are described in detail in Section 2.2.2.1.

Declarations. PAULA supports many different, parameterized data types, for example `signed integer<32>`, which is a signed integer with a width of 32 bits. Several other integer, Boolean, fixed, and float data types are also supported. Moreover, the language could easily be extended by other data types. In order to keep a program readable and to reduce the effort when types of several related variables need to be changed, one may define aliases for existing data types. For example, the above mentioned FIR filter program uses type aliases for the data types of the filter coefficients, input, and output samples. This obviously increases readability and also helps to avoid mistakes when changing data types. If, for example, the width of the output data should be changed, only the **`typealias`** statements have to be modified, and there is no danger of forgetting to also update the type cast in the last equation of the program (`Y[i] = SUM...cast<output_t>`). The syntax of the **`typealias`** statement is the following:

```
typealias alias_type existing_type;
```

Example:

```
typealias input_t signed fixed<16,15>;
```

also be applied to algorithms with affine dependencies. However, in order to validate the properties, the entire algorithm has to be simulated in this case.

[5]In contrast to C, there are different name spaces for each kind of identifiers. So it is no problem to have a variable and a function that both have the name ”foo”. It is always clear from the context whether an identifier denotes an indexed variable, an iteration variable, a function or a data type. However, in order to improve readability, it is strongly recommended to use distinct identifiers for variables, iteration variables, and so on.

Like in many other programming languages, the PAULA language requires variables to be declared before they can be used[6]. The syntax of such a declaration is the following:

```
variable name dimension [in|out] datatype;
```

Variables can either be declared as input variables using the **`in`** keyword, output variables when the **`out`** keyword is present, or normal (internal) variables when neither **`in`** nor **`out`** is given. The *`name`* of a variable is an arbitrary identifier. The *`dimension`* of a variable must be a positive integer, *`datatype`* can either be one of the built-in types or a type alias. Example:

```
variable x 2 in integer<32>;
```

This declares a 2-dimensional input variable `x` of data type **`integer`**`<32>`.

Further declarations are parameter and constant declarations. In a wide range of applications, it is desired to have parameterized iteration spaces. For example, consider the afore given FIR filter, which has a certain number of taps N. This number appears several times in the program. A good way to describe this is to declare N as a parameter[7]. As most other languages, PAULA allows to define constants. These *named* constants can be used everywhere in an expression.

Program Blocks. A *program block* contains a set of equations (see Section 2.2.2.1) and/or recursively nested program blocks. Furthermore, each program block must somehow define an iteration space, that is, a set of iterations in which all equations and nested program blocks are defined.

There are two kinds of program blocks, which differ only in the way their iteration space is defined: *par statements* and *for loops*. The par statement is the basic kind of program block and has the following syntax:

```
[label:]  par (iteration space)
{ /* ...  equations and nested program blocks ...  */}
```

The optional label is a name that might be used to identify and reference a program block. Labels are useful if program transformations (for instance, loop unrolling) should be applied only to a specific part of a program. If par statements are nested, the iteration variables of the parent program block(s) become parameters for the iteration space of the inner statement and can be used in the indexing functions of indexed variables in the inner program block. Example:

[6]In this section, only indexed variables are discussed. For iteration variables, different rules apply, which are described later.

[7]Parameters only apply to iteration spaces. If parameterized data types are needed, type aliases can be used.

```
par (i >= 0 and i <= 10)
{ y[i] = ...         // Here, only i is visible
  pb1:  par (j >= i and j <= 2*i)
  { x[i,j] = ...   // Here, i and j are visible
  }
}
```

For a detailed description of how iteration spaces are defined, it is referred to the corresponding paragraph. Also note that in this example, the inner program block is labeled 'pb1', while the outer block has no name assigned.

If the iteration space of a program block is a simple, 1-dimensional range like $1 \dots 10$, *for loops* can be used as a more compact alternative to par statements. The syntax is the following:

```
[label:]  for (it_var = l_bound to u_bound [step stepsize])
{ /* ...  equations and nested program blocks ...  */}
```

If the step size is omitted, it is assumed as 1. This loop is similar to the "for"-loop that can be found in many classic programming languages. Note, the "for"-loop is only used to generate a set of iteration points. It implies no execution order of the loop body, that is, if there are no restrictions by data dependencies, all iterations might be executed in parallel. For example, the par statement

```
par (i >= 0 and i <= 10) { ...  }
```

is equivalent to the following for loop:

```
for (i = 0 to 10) { ...  }
```

Similarly, the par statement

```
par (i=2*x:  x >= 3 and x <= 8) { ...  }
```

which uses a lattice iteration space, could also be described in the following way:

```
for (i = 6 to 16 step 2) { ...  }
```

Iteration Spaces. The type of an iteration space that is used in the PAULA language is the type of *parameterized linearly bounded lattices*, an extension of LBLs, as introduced in Definition 2.5, which is defined as follows:

Definition 2.8. *(Parameterized Linearly Bounded Lattice). A* parameterized linearly bounded lattice *denotes an iteration space of the form*

$$\mathcal{I} = \{I \in \mathbb{Z}^n \mid I = Mx + c \ \wedge \ Ax + Cp + b \geq 0\} \tag{2.1}$$

where $x \in \mathbb{Z}^l$, $M \in \mathbb{Z}^{n\times l}$, $c \in \mathbb{Z}^n$, $A \in \mathbb{Z}^{m\times l}$, $C \in \mathbb{Z}^{m\times q}$, $b \in \mathbb{Z}^m$, and $p \in \mathbb{Z}^q$ is a symbolic vector of parameters. $\{x \in \mathbb{Z}^l \mid Ax + Cp + b \geq 0\}$ denotes the set of integral points within a convex polyhedron or in case of boundedness within a polytope in $\mathbb{Z}^l$. This set is affinely mapped onto iteration vectors I using an affine transformation ($I = Mx + c$). This transformation is referred to as lattice definition *in the following.*

The syntax to define iteration spaces in the PAULA language is very intuitive as the following examples show.

```
[lattice_definition:  ]  polyhedron_definition
```

The following iteration space is considered as example.

$$\mathcal{I} = \{(i,j) \in \mathbb{Z}^2 \mid i = 2x + 3 \wedge j = 4y + x - 1 \wedge 0 \leq x \leq 10 \wedge 2 \leq y \leq 8\}$$

In the PAULA language, this iteration space is written as follows:

```
i=2*x+3, j=4*y+x-1 :  x >= 0 and x <= 10 and
                      y >= 2 and y <= 8
```

The lattice definition implicitly declares the *iteration variables i* and j, and the so-called *internal iteration variables x* and y. The latter are only available in the polyhedron definition that follows the lattice definition. They cannot be referenced inside the program block to which the iteration space belongs, neither in the indexing functions of indexed variables nor by nested program blocks' iteration spaces. If L is the identity matrix and m is the zero vector, Equation (2.1) can be simplified to $\mathcal{I} = \{I \in \mathbb{Z}^n \mid AI + Cp + b \geq 0\}$. In such situations, the PAULA language allows an equally compact notation, in which the lattice definition is omitted. For example, the iteration space $\mathcal{I} = \{0 \leq i \leq 3 \wedge -2 \leq j \leq 9i\}$ can be defined in the following way:

```
i >= 0 and i <= 3 and j >= -2 and j <= 9*i
```

If this notation is used, there are obviously no internal iteration variables, and both i and j are normal iteration variables. The next example describes a 2-dimensional parameterized iteration space.

$$\mathcal{I} = \{i = x, j = 2 \cdot y + 1 \wedge -N \leq x \leq 2 \cdot N \wedge 0 \leq y \leq 5\}$$

where N is a parameter. The PAULA syntax for this iteration space is straight forward.

```
par (i=x, j=2*y+1:  x>=-N and x<=2*N and y>=0 and y<=5)
{ ... }
```

The lattice definition declares the *iteration variables i* and j, and the so-called *internal iteration variables x* and y. The latter are used only to generate the iteration space but cannot be referenced anywhere else.

Equations and Indexed Variables. Actual calculations (the data flow) of a program are indicated by *Equations*. The basic syntax of an equation is:

```
[label:]  indexed_variable = expression;
```

An example of an equation in a PAULA program could be:

```
x[i,j] = x[i-1,j] + y[2*i,j-1];
```

Similar to program blocks, equations may have a label, which gives them a unique name and allows referencing.

```
eq3:  x[i,j] = x[i-1,j+1] + y[2*i,j-1];
```

The data type of the expression on the right hand side of the equation must be equal to, or automatically convertible to the data type of the indexed variable on the left hand side. See Appendix B.2 for more details about data types and type conversions.

In a DPLA, variables that are used for data flow computations are called *indexed variables*. As stated earlier, these variables must be declared at the beginning of a program, and need a data type, and a certain dimension. The content of an indexed variable is accessed by an indexing function. For instance, the indexing function of variable y of the above example can also be written as $\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix}$. The general form of an indexing function is given by $QI + d$ with $Q \in \mathbb{Z}^{k \times n}$ and $d \in \mathbb{Z}^k$, where k is the dimension of the indexed variable and $I \in \mathcal{I}$ is an iteration point inside the iteration space $\mathcal{I} \subset \mathbb{Z}^n$.

For each equation, all iteration variables of the iteration spaces of all enclosing program blocks are available as iteration variables in the indexing function.

```
par (i >= 0 and i <= 10 and j >= 0 and j <= 10)
{ // Here, i and j are valid iteration variables
  c[i,j,0] = 0;
  par (k >= 0 and k <= 10)
  { // Here, i, j, k are available as iteration variables
    c[i,j,k] = a[i,j,k] * b[i,j,k] + c[i,j,k-1]
  }
}
```

Equations with Iteration Dependent Conditions. In order to allow irregularities in a program, an equation may have iteration dependent conditions. That is, a given equation is valid only for a subset of iterations inside the current program block. The syntax is

```
indexed_variable = expression if (iteration_space);
```

Iteration dependent conditions can only use iteration variables that are defined by iteration spaces of enclosing program blocks. An example, including iteration dependent conditions, is given in Algorithm 2 for a matrix multiplication algorithm ($C = A \times B$).

Algorithm 2: Algorithm for matrix multiplication.

```
  ⋮
  par (i >= 1 and i <= N and
       j >= 1 and j <= N and
       k >= 1 and k <= N)
  { // Equations with iteration dependent conditions
    c[i,j,k] = 0                                if (k == 1);
    c[i,j,k] = a[i,j,k]*b[i,j,k] + c[i,j,k-1]  if (k > 1);
    C[i,j]   = c[i,j,k]                         if (k == N);
    // More equations (without iteration dependent cond.)
    a[i,j,k] = A[i,k];   // Input matrix A
    b[i,j,k] = B[k,j];   // Input matrix B
  }
  ⋮
```

Again, note that the order of equations does not matter and thus the equations, reading the input matrices, can appear after the computation and output equations of the program.

Equations with Run-time Dependent Conditions. Besides iteration dependent conditions, a PAULA equation can also have a run-time dependent condition. The syntax is

```
indexed_variable = ifrt(cond_exp, then_exp, else_exp);
```

where `cond_exp` is the actual condition, an expression, which must be of data type Boolean. Furthermore, the expressions `then_exp` and `else_exp` must each be of the same type as the indexed variable on the left hand side of the equation or must be convertible to that type. The value of the right hand side of the equation, that is, the value "returned" by the **`ifrt`** statement, is the value of `then_exp` if `cond_exp` evaluates to "true", or the value of `else_exp` otherwise. In order to satisfy the single assignment property, both branches must be defined. An example equation with a run-time dependent condition to catch a division by zero is given as follows.

```
x[i,j] = ifrt(b[i,j] != 0, a[i,j]/b[i,j], 65535);
```

The next example might be part of an image processing algorithm. It ensures that the value of a pixel does not exceed a certain limit. Of course, if the condition is more complex, it can be separated into an equation of its own.

```
c[i,j] = a[i,j] > 255;
v[i,j] = ifrt(c[i,j], a[i,j], 255);
```

Expressions. The syntax and semantics of expressions in the PAULA language are not much different from those found in C and similar languages. Table 2.1 lists all available operators and their precedence. During the design flow, operators are replaced by functions. This is done because of binding possibilities (see Section 2.2.2.2) are only defined for functions, not operators. Table 2.2 lists the mapping of operators to functions. Of course, the user may define (give binding possibilities for) arbitrary additional functions. Note that the semantics of only the operators is defined by the language. The design system may transform expressions that contain operators by applying common mathematical rules. For instance, `a*b+a*c` can be transformed to `a*(b+c)` and thus saving one multiplication. Such transformations may not be performed on functions.

For the sake of brevity, we describe only the operators *function* and *reduction* in detail because their semantics differ or do not exist in conventional languages such as C.

Functions. In the PAULA language, functions are real mathematical functions, which means their return values solely depend on the arguments. Functions cannot have a state and there is also no global state. For every function, a binding possibility has to be defined (see Section 2.2.2.2), which describes a hardware implementation and/or a simulation model, and thus brings the semantics of the function. Examples of functions are:

```
add(x[i], x[i-1])        or        sin(y[i,j])
```

Big Operators. Big operators implement mathematical operators such as $\sum$ or $\prod$. These operators are also often called reductions. The syntax is the following

```
operator [ iteration_space ] (expression)
```

where *`operator`* is one of **SUM**, **PRODUCT**, **MIN**, **MAX**. For instance, $\sum_{j=0}^{10} b\,[i,j]$ can be written in PAULA as

```
SUM[j >= 0 and j <= 10](b[i,j])
```

However, in contrast to the common mathematical notation, the iteration space is not required to be 1-dimensional. For example, we can write

```
SUM[j >= 0 and j <= 10 and k >= 0 and k <= j](c[i,j,k])
```

Table 2.1: In the table, the operators in PAULA and their precedence is shown.

Precedence	Operator	Description	Associativity
1	`()`	grouping operator	—
	`42`	constant	
	`a[]`	indexed variable access	
	`f()`	function call	
	`SUM[]()`	big operator	
	`cast<>()`	type cast	
2	+	unary plus	right to left
	−	unary minus	
	!	logical negation	
	~	bitwise complement	
3	*	multiplication	left to right
	/	division	
	%	modulus	
4	+	addition	left to right
	−	subtraction	
5	<< >>	bitwise shift operators	left to right
6	==, !=, < >, <=, >=	relational operators	only binary relations allowed
7	&	bitwise and	left to right
8	^	bitwise xor	left to right
9	|	bitwise or	left to right
10	&&	logical and	left to right
11	||	logical or	left to right

to describe the following sum:

$$\sum_{(j\ k)^{\mathrm{T}} \in \mathcal{I}} c\,[i, j, k]$$

with $\mathcal{I} = \left\{ (j\ k)^{\mathrm{T}} \in \mathbb{Z}^2 : 0 \leq j \leq 10 \wedge 0 \leq k \leq j \right\}$. This is equivalent to the double sum:

$$\sum_{j=0}^{10} \sum_{k=0}^{j} c\,[i, j, k]$$

Table 2.2: The table depicts the mapping of operators to functions.

Operator	Function
+	add
−	sub
*	mul
/	div
%	mod
==	eq
!=	neq
>	gt
<	lt
>=	geq
<=	leq
&	band
|	bor
^	bxor
~	bnot
<<	shl
>>	shr
&&	land
||	lor
!	lnot

Similar to the nesting concept of program blocks, iteration variables from the program block that contains the expression are available as parameters in the iteration space of the big operator:

```
par (i >= 0 and i <= 15)
{ a[i] = SUM[j >= 0 and j <= i](b[i,j]);
}
```

The usage of big operators allows a compact and intuitive description style as the image processing code fragment of an approximated 2-D Gaussian window filter demonstrates in Algorithm 3.

2.2.2.2 Architectural Description

One of the major purposes of the PARO design system (for further readings, we refer to Chapter 4) is to generate synthesizable hardware descriptions (for instance in VHDL) for parallel algorithms or to generate code for tightly-coupled processor arrays. The design flow involves, among others, the steps of allocation, binding,

Algorithm 3: Approximated 2-D Gaussian window filter

```
⋮
w[0,0] = 1; w[0,1] = 2; w[0,2] = 1;
w[1,0] = 2; w[1,1] = 4; w[1,2] = 2;
w[2,0] = 1; w[2,1] = 2; w[2,2] = 1;
h[x,y] = SUM[i>=0 and i<=2 and
             j>=0 and j<=2](pic_in[x+i,y+j] * w[i,j]);
pic_out[x,y] = h[x,y] » 4;  // divided by 16
⋮
```

scheduling, and synthesis and code generation, respectively. Obviously, during hardware or code generation, detailed knowledge about the target hardware architecture is required. The definition of this architecture model and constraints such as number and types of available processing units can also be done in the PAULA language. This section is necessary to describe resource constraints for scheduling and synthesis as well as for code generation. An architecture model can be subdivided into three parts: (1) *resource type definitions*, (2) *resource allocation*, (3) *binding possibility definitions*. Since the architectural definitions are completely independent of the algorithm, it is recommended to put them into separate source files and include them in the main file, for example:

```
include("alu.arch.paro")
include("multiplier.arch.paro")

program example { ...  }
```

Two files are included in this example, which contain architectural data for an ALU and a dedicated multiplier.

Resource Type Definitions. A *resource type definition* describes which resource types are available in an architecture model. The following code fragment shows an example definition for a resource type that is called 'alu16'.

```
resourcetype alu16
{ ops 16;
  input a integer<16>;
  input b integer<16>;
  output c integer<16>;
  output d boolean;
  component alu;
  parameter WORDSIZE = 16;
}
```

This resource type offers up to 16 different operations ('**ops** 16'). For resource types that support only one operation, the '**ops**' statement may be omitted. Next, the data I/O ports of the corresponding hardware component are defined. Our example ALU has two inputs 'a' and 'b', and one output 'c', all of type 16 bit integer, and an additional Boolean output 'd'. Depending on which operation the ALU currently performs, the result of the computation is available on either port 'c' (for operations that have a result of 16 bit integer, for instance, addition, subtraction, shift, and so forth), or on output 'd' (for operations that have a Boolean result, for instance, comparisons). Afterwards, the resource type is mapped to a hardware component 'alu', which may be a VHDL entity name. This mapping together with the following '**parameter**' statement allows to use an existing, generic HDL implementation (for example, a generic VHDL entity) as basis for many different resource types. It should be mentioned that there exist further statements, as for instance, one to annotate different cost metrics to a resource type in order to allow early cost analysis and to enable design space exploration.

Resource Allocation. The *resource allocation* determines how many instances of a given resource type shall be available within one processor element[8]. The syntax is the following:

```
allocation resourcetype_name integer;
```

For example, "**allocation** alu16 3;" would allocate up to three instances of the resource type 'alu16'. It is also possible to allocate an infinite number of instances per resource type, using the keyword '**infinite**' instead of a number.

```
allocation alu16 infinite;
```

Definition of Binding Possibilities. By defining *binding possibilities*, information about which operations are supported by each resource type is added to the architecture model. In PAULA, only functions can be mapped to hardware. Operators (like '+' or '-') are replaced by functions as a first step during synthesis. The syntax of a binding possibility definition is the following:

```
bindingpossibility function
     name (types_of_params) ret_type on resource { body }
```

Where `types_of_params` is a comma-separated list of data types, which implicitly defines the number of parameters of the function. For example:

```
function mul(integer<32>,integer<32>) integer<64>
```

[8]The number of processors is not specified in the architectural part, it is defined in the tiling and mapping phases during synthesis.

This declares a function called "mul", which gets two 32-bit integers and returns a 64-bit integer. Two sample binding possibilities for the resource type 'alu16' (cf. Section 2.2.2.2) may look as follows.

```
bindingpossibility function
    mul(integer<16>, integer<16>) integer<16> on alu16
{ op 0;
  input a,b;
  output c;
  cycles 2;
  pipelinerate 1;
  simulatorplugin "sim_alu16.so", "mul_int16";
}

bindingpossibility function
    sub(integer<16>, integer<16>) integer<16> on alu16
{ op 1;
  input a,b;
  output c;
  cycles 1;
  pipelinerate 1;
  simulatorplugin "sim_alu16.so", "sub_int16";
}
```

The body of a binding possibility consists of three parts, (1) the operation and (2) port mapping, and (3) the scheduling parameters. The *operation mapping* ('**op**' statement) defines which operation number (opcode) must be selected by the generated hardware control unit so that the resource performs the desired operation. Or, in case of programmable architectures, this operation number corresponds to the appropriate assembler instruction. The *port mapping* ('**input**' and '**output**' statements) assigns function parameters and the result to input/output ports of the hardware component that implements the resource type, the function is bound to. The port names that appear here, must be declared in the respective resource type definition, and their data types must match. The order of the input ports corresponds to the order of the function parameters. The *scheduling parameters* ('`cycles`' and '`pipelinerate`' statements) define the execution time and the pipeline rate of the operation on the given resource type. The statement `simulatorplugin` defines where the simulator can get a simulation model for the considered function. The two arguments are the plugin file (for example, a dynamically loadable C library) and the name of the (C-)function in the plugin.

2.3 Summary and Conclusions

We have introduced the class of dynamic piecewise linear algorithms, which is a consistent extension of recurrence equation-based algorithm classes in the polytope by run-time dependent conditions. This extension significantly increases and eases the modeling of a number of algorithms that also have few run-time dependent conditions beside the data flow. Next, a reduced dependence graph model that incorporates the newly introduced conditional data flow has been developed. Here, in addition to existing models, piecewise definitions and affine indexing functions of left-hand side as well as of right-hand side variables of an equation are allowed. That is, non-perfect nested loop programs with affine data dependencies and iteration and run-time dependent conditions can be equivalently expressed by this graph model. Third, a functional language, named PAULA, that reflects the properties of DPLAs has been presented. Here, the intention was not to propagate yet another language for parallel programming but to specify an input language that is exactly tailored for our needs when mapping nested loop programs to dedicated hardware accelerators or tightly-coupled arrays consisting of programmable and lightweight processors. The language is based on the class of dynamic piecewise linear algorithms, therefore it offers a full static single assignment form for multi-dimensional data flow. Moreover, it contains elements such as big operators (reductions) and parts for the architectural description of a target architecture.

It should be noted that the proposed features of PAULA may also be expressed in a ’C-like’ or any other high-level programming language. But, in order to not mislead a programmer, the catalog of restrictions and modifications would be as long as the description of the PAULA language itself.

Another option would be, to start from a sequential program description, for instance in C, and to parallelize the code afterwards. But, as already discussed in Section 2.2.1, the necessary data dependency analysis might quickly become a very complex task. In this context, we experimented with C-code parallelization [Bey02] by using an exact method for data dependency analysis [Kie00], which is based on *parametric integer programming* by Paul Feautrier [Fea88]. The method is even for small examples very computational as well as memory intensive. Furthermore, a large number of intermediate variables is introduced, which unnecessarily pad the program out.

A further indication for not starting from C, is the notable number of new recently emerging parallel programming languages for multi-core and graphics architectures. Thus, in the future, it would be reasonable to consider a combination of PAULA with one of these novel domain-specific languages, or to directly extend such a language, as for instance OpenCL [Khr08].

Chapter 3

Scheduling and Allocation

Scheduling and allocation are the key tasks in high-level synthesis and algorithm-to-processor mapping. Because of the concepts and their application domains are highly multifaceted, this chapter is focused on data flow dominated types of algorithms, which are representable by DPLAs as already introduced in Chapter 2. As target architectures, accelerators are considered that might be integrated as *intellectual property core* [BMP07] in a system-on-a-chip. These accelerators can be custom-tailored for just a single application or they can be domain-specific. This means, that they can either occur in the form of dedicated hardware architectures or as tightly-coupled arrays, consisting of lightweight processors with limited programming, communication, and memory capacity.

First, related work in the areas of allocation, loop parallelization, and scheduling are presented. Subsequently, basic definitions and concepts when mapping nested loop programs onto the aforementioned accelerators, and synthesizing accelerators from these programs, are presented. Moreover, different methods for the allocation of a given loop program to several processors are introduced. After these fundamentals, a modular concept for scheduling nested loop programs with run-time dependent conditions is developed, where the conceived scheduling methods are based on *mixed integer programming* (MIP) [PS82, NW99].

The major contributions of this chapter can be summarized as follows:

- An introduction to and differentiation from related approaches is given in Section 3.1.

- A formulation of modular and exact scheduling concepts that simultaneously consider scheduling on different levels (processor level and array level) is presented. Here, in particular new concepts for the scheduling of partitioned algorithms are developed (see Section 3.5).

- For the first time, an exact resource constrained scheduling method is derived that regards software pipelining for programs with multi-dimensional data flow in consideration of iteration dependent as well as run-time dependent conditions (see Section 3.6).
- The modularity of the proposed concept is utilized in order to incorporate further constraints (register, channel) that are necessary to target a special class of tightly-coupled, programmable processor arrays (see Section 3.8).

3.1 Related Work

In the following, we summarize previous work in the areas of allocation and scheduling that are related to this thesis. Afterwards, a differentiation from similar approaches is given.

3.1.1 Allocation

In general, *allocation* [GDWL92, Tei97] denotes the type and quantity of components that are used in an implementation, for example, different types and number of processors, memories, communication resources, or functional units such as adders or multipliers. Often, pre-estimated metrics (latency, area and power cost, etc.) are annotated to the components, which allow an early examination of cost and performance characteristics of the system. The granularity of the allocated components can vary depending on the different abstraction levels during the design of a system (cf. Figure 1.2 in Chapter 1).

In this work, parallelism is exploited at several levels and the considered processor architectures are hierarchically composed. Thus, in the following, we differentiate between a *local allocation* and a *global allocation*.

Local allocation denotes the resource allocation of a single processing element or processor. This definition corresponds to the concept of allocation in high-level synthesis (see above). For instance, the number and types of functional units, number of registers, or number of I/O ports that are available for one processing element and processor, respectively. Local allocation should be complete in the sense that for each functionality to be implemented, at least one corresponding component should exist. This relation is called *binding possibility* and the actual assignment of one operation to a component is called *binding* [Tei97] or *module selection* [JPP88, GDWL92].

Commonly, the relation between operations with binding possibilities and the allocation is expressed by graph models, namely problem or data flow graphs and a resource graph.

Global allocation represents a macroscopic view and reflects a number of processors. Often, this allocation is accompanied by the static assignment[1] of work to the processors. More precisely, the assignment, called *space mapping*, denotes which iterations of a nested loop program are executed on which processor. Most of the time, linear transformations [Mol83, Len93] are used as space mappings because they are leading to regular and simple designs, and therefore ease implementability. One of the first and simplest allocation methods is to consider a *projection* of the iteration space along a vector [Kuh80]. All iteration points that are crossed by the same vector, are assigned to the same processor. Though it is possible to define the projection in such a way, that a certain number of processors is not exceeded, data distribution and possible dependencies between the iterations might be unfavorably arranged. Hence, in most works, for instance in [Mol83, MW84, Qui84, LW85, Rao85, HL87, Omt88, Len89, QRW00], the projection vector is rather selected in order to maximize the throughput with respect to the data dependencies of the algorithm. Unfortunately, by following this strategy, the number of processors depends on the problem size (size of the iteration space). In addition, projections reduce the dimension of the iteration space by one. Hence, in practice, they are only suitable to a certain dimension. This problem led to the idea to project the iteration space several times. Works that studied these so-called *multi-projections* include [LK90, KH03, DSV05].

Beside projection, *partitioning*[2] [IT88] is the other well known transformation, which covers the iteration space of computation using congruent tiles as for example, hyperplanes [MF86, WPS96], hyperquaders, or parallelotopes. Other common terms for partitioning in literature [WS91, Wol96a, Xue00, CDK$^+$02] are *loop tiling*, *blocking*, or *strip mining*. The transformation has been studied in detail for compilers and its use has led to program acceleration through better cache reuse on sequential processors [CM95, XH98, KK05, KNB$^+$08]. Here, the idea is to divide the iteration space into chunks such that the corresponding data fits into the cache. Furthermore, partitioning is used for the implementation of algorithms on given parallel architectures ranging from supercomputers to multi-DSP solutions, processor arrays, and FPGA architectures [DRS00]. It is carried out in order to match a loop nest implementation to resource constraints in terms of available number of processors, local memory, and communication bandwidth. In the following, we focus only on techniques that are used for algorithm mapping onto processor array

[1] In this work, only the static workload distribution is studied. That is, no operating system that distributes the work to several processors or a run-time library for thread dispatching (like in modern graphics processors [LNOM08] or in the Cell processor [IBM07]) is considered.

[2] In mathematics, similar problems are even studied since a longer time [Haj42].

architectures. Here, well known partitioning techniques are *LSGP* (*locally sequential, globally parallel*) [Jai86], which is also often referred to as *clustering* or blocking [Wol96a] and *LPGS* (*locally parallel, globally sequential*) [Jai86, MF86], which is also referred to as tiling. In their original definition, these partitioning techniques are not applied to the iteration space but to the *full size array*. These arrays are decomposed into groups of processors. For instance, in the LSGP approach, the array is divided into clusters of n processors. Then, each cluster is mapped onto a single processor that executes the workload of these n processors sequentially. In the LPGS method, the array is decomposed into subarrays, which are then executed by a target array in a serial manner (an illustration follows in Section 3.3.2). These approaches preserve the regularity and locality of the original algorithm. Moreover, control overhead remains small and manageable. LPGS partitioning [TT92, TT93, Zim97, KLY00, ZA01, KMAC06, BRS07] and LSGP partitioning [BDD90, DD90, TT92, TT93, Dar91, MC95, TTZ97, Zim97, Fim00, ZA01] have been widely studied. Both the LSGP and the LPGS partitioning scheme can be used in order to derive architectures with a fixed number of processors. In the LSGP approach, local memories within the processing elements are necessary, which depend on the size of the iteration space. Due to local processing, communication within the array and to the peripherals is moderate. On the contrary, in the LPGS approach, less memory within the processor is necessary. Thus, data reuse is only possible to a limited extent and thus, the communication to external memory increases.

The pros and cons of the LPGS and LSGP approaches led to the idea of *co-partitioning* [EM97a, Eck01]. Co-partitioning uses both LSGP and LPGS methods in a combined way in order to balance local memory requirements with the I/O bandwidth and, simultaneously, has the advantage of problem size independence. Since the approach partitions an already partitioned iteration space, it can be seen as a *hierarchical partitioning* method [ML90, EM99, DHT06c, RHDR07]. Hierarchical partitioning schemes are well suited in order to adapt the algorithm to a memory hierarchy [CDK$^+$02].

Other works [AKN95, RR02, PL06] systematically study the derivation of optimal tile shapes in form of n-dimensional parallelotopes. Here, the estimation of the so-called *footprint* [ST86] is often used during optimization. A footprint denotes the number of data accesses within a tile. Ideally, the entire footprint fits into a local memory, thus I/O accesses are reduced.

Most approaches are focused on finding optimal tile shapes but do not present a solution to the problem of partitioning possible data dependencies across the iterations. The work of Teich and Thiele [TT93, TT02] as well as our work in [DHT06c], give insight into this problem.

All of the aforementioned partitioning schemes consider congruent tiles with an equal number of iteration points. For the sake of completeness, we would like to mention that there exist some works, which also consider irregular partitionings[3] of a loop nest [KNBP05, KNS$^+$06].

3.1.2 Scheduling

Scheduling exists since humans plan different activities over time. It can be easily imagined that scheduling problems are omnipresent in our daily life. Most prominent examples include the scheduling of computational, manufacturing, or logistic processes. In practice, these processes[4] compete for resources, which can be of very different nature, for instance processors, machines, energy, tools, money, or manpower. Often, the individual tasks depend on each other or have to fulfill deadlines. Usually, scheduling is not only concerned with finding a valid order and assignment of processes to resource but also include optimization. Examples for optimization goals are cost (area, power, monetary), performance (latency, throughput, turnaround, waiting time), or fairness.

In order to classify the later presented related work, some distinctive features are discussed in the following.

Offline and Online Scheduling. A schedule is called *offline* if it is determined in advance at compile-time. In contrast, methods that determine a schedule at run-time are called *online*. Online scheduling occurs for instance in operating systems where tasks can be created dynamically and have to be assigned to available resources. For control intensive applications, these methods often have to incorporate real-time constraints. In offline methods, often the execution times of the processes are known in advance in order to determine the start times of the different processes and not only the execution order. Whereas, in online scheduling methods, the execution time of a process is commonly known only at run-time. Hence, methods and models for offline and online scheduling are greatly different [But02].

Preemptive and Non-Preemptive Scheduling. In *non-preemptive* scheduling techniques, the resources that are assigned to a running process are occupied until the process has completely finished its execution. On the contrary, in *preemptive* scheduling methods, the execution of a process can be suspended in order to allow the execution of other processes. Preemptive methods are most common in software systems

[3]Here, methods are meant that can be written again as a higher dimensional loop nest, not general graph partitioning, multilevel approaches [WC00], or other methods used for load balancing.

[4]When generally speaking about scheduling the term process might be synonymous with terms such as activity, task, operation, or job.

where relatively long tasks are executed in order to give an impression of *quasi-parallel* processing [SGG04].

Data Dependencies and Resource Constraints. A further attribute of a scheduling problem is whether data dependencies between different processes can be considered. Also it can be distinguished between scheduling problems with or without resource constraints. With regard to complexity, problems not considering data dependencies and resource constraints are trivial. This is also the case if all tasks are independent of each other and have the same execution time. This case is exploited for instance in modern graphics hardware, where the same pixel operations are executed millions of times, distributed over dozens of processing units. In case of data dependencies and no resource constraints, scheduling can be performed in *polynomial time* (which will be discussed later). Whereas, if resource constraints are considered, scheduling can become a hard combinatorial problem. Often, these problems can be formulated as packing problems and thus are *NP-hard* [CLR97] in general.

Iterative Scheduling. Eventually, a distinction can be made between a set of processes that has to be executed only once or if this set has to be executed repeatedly. Methods that consider the latter behavior are called *periodic* or *iterative scheduling*. As the name suggests, they are closely related to iterative algorithms in form of loop specifications. Among the execution of loop programs [Wol96a, Dar99, DRV00], iterative scheduling is an important topic in real-time systems [But02] and digital signal processing [Swa87, SB00].

The overview of related work in the next section is focused on scheduling problems for computers and in particular scheduling operations under resource constraints. Mainly static and non-preemptive methods are discussed.

First, some methods for scheduling acyclic data flow graphs without and with resource constraints are presented. Afterwards, methods that incorporate control flow constraints are discussed. Subsequently, an overview of scheduling methods for processors that consider register constraints is given and finally, scheduling methods for one-dimensional and multi-dimensional data flow are discussed.

3.1.2.1 Scheduling of Data Flow Graphs

In high-level synthesis, data flow is often expressed by directed acyclic graphs, where operations are denoted by nodes and data dependencies between the operations are expressed by directed edges.

Absence of Resource Constraints. Even though in practice resource constraints always exist, scheduling methods without resource constraints are of interest, for instance, in order to determine lower bounds of the execution time. A valid execution order can be found by *topologically sorting* a given data flow graph $G = (V, E)$, which can be done in $\mathcal{O}(|V| + |E|)$ time [CLR97]. This graph sorting algorithm is also adapted in the *ASAP* (*as soon as possible*) method [TS86] for finding a schedule with minimal *latency*[5]. Sometimes this method is also referred to as *forward scheduling* [Bal88]. Similarly, *backward scheduling* plans the execution of operations for a given latency bound *as late as possible* (*ALAP*). The difference of a node's start time obtained by the ALAP and ASAP methods is called *mobility*.

Scheduling with Resource Constraints. A method that is alternating between the ALAP and ASAP method is called *double headed scheduling* [Bal88]. In this method, a higher priority is given to operations with small mobility. If the number of available resources are exceeded, the operations are moved to a later start time.

Other most popular scheduling heuristics are the *critical path method* proposed in the 1950s, *urgency scheduling*, which plans operations first with higher urgency, *list scheduling* [Gra66, DLSM81], *force-directed scheduling* [PK89, CT90], or methods [DN89] that are based on *simulated annealing* [KGV83]. List scheduling is the prevalent method, where each operation is prioritized in advance and many different prioritization criteria have been proposed [ACD74]. For instance, the mobility of nodes, the execution time of a node, or the critical path length can be considered.

Aside from heuristics, there exist exact methods based on *integer linear programming* to obtain optimal schedules. The main idea, to use integer linear programming with binary variables for the synthesis at register transfer level, dates back to the work of Hafer and Parker [HP81]. Many works are based on their approach, for instance [KKT90, GE93]. Particularly worthy of mention is the work by Hwang and others [HHL90], who presented an integer linear programming model for the scheduling problem in high-level synthesis, taking different types of resource constraints (multi-cycle operations, functional pipelining, etc.) into account. Other exact approaches are based on solving dis-equations[6] [CDF06] and graph coloring [CF07].

3.1.2.2 Scheduling of Conditional Branches

Heuristics. Several heuristics have been proposed in literature [TWR$^+$88, Cam91, CGR93, KYLL94, EKP$^+$98, Gra00, KWL01] that consider the scheduling of condi-

[5]The *latency* of an algorithm or graph denotes its execution time.

[6]In contrast to inequalities, denoted by $a > b$, a *dis-equation* is denoted by $a \neq b$. This terminology has been proposed by the authors is [GM86].

tional branches within data flow graphs. Other approaches consider the scheduling of *synchronous data flow* (SDF) graphs using token models [BL93]. The application of token models for the execution of different branches is especially beneficial when the different execution paths are relatively long [AS06, GK08].

Exact Methods. However, due to the complexity of enumerating different paths, there exist only few exact methods [HHL90, KW01]. Hwang and others integrate conditional resource sharing in their proposed integer linear program by the definition of a so called *relation tree* [HHL90]. By the relation tree, it is denoted whether operations might be executed in parallel or if they are mutually exclusive. Kuchcinski and Wolinski propose a scheduling method [KW01], which is based on so-called *hierarchical conditional dependency graphs* [KW98] and *constraint logic programming* [Kuc98, Dec03].

3.1.2.3 Scheduling of Loop Programs and Loop Parallelization

Scheduling of loop programs is of great importance since many scientific and digital signal processing programs spend a large fraction of the overall computing time in loops [Knu71, SMN+03]. Existing works for scheduling loop programs can be mainly grouped into two areas. In the first area, the iteration space of a loop program and data dependencies across the iterations are considered but not the operations in the loop body itself. Often, no resource constraints are accounted for in these methods. First goal is to obtain a maximum degree of parallelism and to find a valid iteration order with minimal latency. Thus these methods are referred to as *loop parallelization*. Such techniques are of great interest in high-performance computing and parallel processor arrays. The other area covers the *instruction level parallelism* of the loop body in an iterative fashion. Here, the challenge is to minimize the intervals between subsequent iterations or even better to overlap the execution of operations that belong to different iterations.

Loop Parallelization. Most of the time, the analysis is restricted to well defined algorithm classes as loop nests with static bounds and uniform data dependencies (cf. Section 2.1.1). But even these simple classes are very important in practice and thus a lot of research has been performed in this area. The regularity of the considered algorithms is the key to efficient parallelization since it becomes independent of the size of the iteration space.

Lamport [Lam74] studied the parallel execution of loops by introducing the so called *hyperplane method*. In his seminal work, the entire loop body is considered as one block. The start times of different iterations are defined by a linear function that minimizes the overall execution time. Iterations that lie on the same hyperplane can be executed in parallel. Based on Lamport's work, many other works have been proposed, for instance [SF91, HS93, DV95]. In [Rao85], Rao presented a scheduling method based on linear programming for the design of regular processor arrays. Darte and Robert [DR92] reduced the scheduling of uniform loop nests to the solution of a single linear program. Darte et alii [DKR92] also proved that linear scheduling is asymptotically as good as free scheduling for a uniform loop nest when considering the entire loop body as one block. Consequently, in [Fea92a, DR94a] affine scheduling methods have been developed that consider a same linear part plus an offset for each statement of the loop body. Methods that assign an affine function to each statement of a loop body are presented in [DR95] and called *affine-by-statement* scheduling.

Since scheduling of loop programs might lead to large linear programs, *structured scheduling* was proposed in [QR02, Fea06] where a large program is divided into smaller parts, which can be scheduled separately.

Another way to structure computations is to partition the iteration space into several spaces (cf. Section 3.1.1). The assignment of different affine scheduling functions to these spaces can be interpreted as *multi-dimensional time* within the original iteration space [Fea92b, BL01, BRS07]. In [ML90], the authors propose a multi-mesh method for matrix computations. The authors in [TTZ96] and [ZA97] propose sequentialization constraints in order to determine schedules for partitioned algorithms. Eckhart and Merker are concerned with so-called *co-partitioned* algorithms [EM97a] and their scheduling [EM97b]. In [AZ00, ZA01], Zimmermann and Achtziger propose an optimal scheduling method based on quadratic programming for LSGP and LPGS partitioned processor arrays.

Other approaches emphasize on the scheduling of the communications and data reuse in processor arrays [CK93, DRR95, SM04, SM06b, SM06c].

Instruction Scheduling with Software Pipelining. Since the body of a loop program might be traversed an enormous[7] number of times, particular attention has been spent for optimizing the execution time of the loop by exploiting *instruction-level parallelism* (ILP) [RF93]. The exploitation of instruction-level parallelism is of vital importance in high-level synthesis since, thanks to the parallel nature of hardware, a large number of modules could work in parallel, maybe processing several iterations at the same time. This technique is called *software pipelining* [AJLA95].

The counterpart are programmable systems such as VLIW processor architectures [Fis83, CNO$^+$88, RYYT89] and *explicitly parallel instruction computing* (EPIC) [SR00] that offer several functional units, which execute a fixed schedule determined

[7]For instance, the processing of a 10 megapixel image by a 5×5 window filter results in 250 million loop iterations.

at compile-time of the program. Examples of contemporary VLIW and EPIC CPUs include the general purpose architecture of Intel's Itanium IA-64 [Int04, GCC+05] and a number of embedded processors for digital signal and media processing such as the the TriMedia family from NXP Semiconductors [ESV+99], the TigerSHARC processors by Analog Devices [FG00], the TMS320C6000 family of DSPs by Texas Instruments [Tex08], the ST200 family from STMicroelectronics [FBF+00], and the HiveFlex family by Silicon Hive [PBSF06].

Most compilers for the aforementioned VLIW architectures use heuristics for loop scheduling [Lam88, RF93, SCD+97]. Well known methods are *trace scheduling* [Fis81, SDJ84], which is used in a slightly modified version for the ST200 processors, and *iterative modulo scheduling* [Rau94]. Based on these methods, there exists a multitude of variations, for instance [LGAV96, SL99]. For a comparative study, we refer to [CLG02].

In [SL96, SM98, PJ03], software pipelining for loops that contain conditional branches is considered.

Modern processors (for instance the Itanium processor [Int04] or the Cell Broadband Engine Architecture [KDH+05]) include instructions for so-called *predicated execution* [MHB+94]. The idea is to replace if-statements by straight line scalar operations prior to scheduling. Thus, both branches might be executed in parallel and finally only the result in dependence of the condition has to be selected. Example:

```
if (b>0) then                c  = compare(b>0);
  a = x+b;                   a1 = x+b;
else                         a2 = y+7;
  a = y+7;                   a  = select(c, a1, a2);
endif
```

The removal of branches is called *if-conversion* [AKPW83, PS91]. This technique is used in many scheduling approaches [WHSB92, MLC+92, AHM97, SG04, Fer07] in order to handle conditional branches.

Often, for high performance computing or embedded computing with stringent constraints, the best solutions are needed. Hence, a number of exact approaches most of them based on linear programming formulations have been proposed [Fea94, EDA95, GAG96, DSRV02, TE02, YKM03, MFM04, FKPM05].

All the aforementioned works consider only the innermost loop of a given loop nest. This leads to an exploration problem of the loop order. Few other methods such as [RTG+07, QSL+08] consider the entire loop nest. These methods are referred to as *multi-dimensional loop* or *multi-dimensional data flow* [PS96].

Thus, it seems obvious that some of these VLIW methods have been adapted for the design of dedicated hardware. The basic idea is to instantiate a number of functional units and registers that are necessary for the execution of one dedicated loop program and to use VLIW compilation techniques in order to synthesize the control path of the circuit. The most prominent example of such a method is PICO [SAR+00a, KAS+02], developed at the HP labs and commercialized by Synfora [Syn09]. Other recent VLIW-based approaches for the design of custom hardware are presented in [Moh06] and [KLB08]. A disadvantage of VLIW methods is that with an increasing number of functional units the cost for multiplexers and registers grows dramatically. Other high-level synthesis approaches [CD04, KSP07] that rely on the scheduling of data flow graphs perform *loop unrolling* in order to increase the number of operations within the loop body or rather the number of nodes in the data flow graph. Subsequently, thanks to the increased number of nodes, a higher throughput and better resource utilization might be achieved.

Combined Methods. In [Thi95], Thiele combined for the first time the work of [HHL90] (scheduling with resource constraints in high-level synthesis) with the works of Darte and Robert on scheduling of uniform loop nests [DR92]. For loops with uniform dependencies, he formulated a single integer linear program, which simultaneously optimizes the schedule between different processing elements and the local execution order within the processing elements. The authors in [FM97, HT01, DRK01] presented similar approaches. However, the number of processing elements in all these approaches is dependent on the problem size of the algorithm. Methods for a fixed number of processing elements derived by so-called LSGP partitioning (see Section 3.1.1) have been proposed by Teich and others [TTZ96, TTZ97] and Fimmel [Fim00].

3.1.3 Differentiation

Most closely related to the scheduling methods presented in this thesis are the works of Eckhart, Fimmel, Müller, and Siegel. Thus, we differentiate in more detail from these works in the following.

Eckhart [EM97a, EM97b, Eck01] developed and extensively studied co-partitioning with respect to I/O, communication, and memory demands. He proposed a scheduling method [EM97b] for the execution order of iteration points, but did not consider the local allocation and operation scheduling. Further, the proposed scheduling technique is limited to rectangular tiles.

In [FM97, Fim02], Fimmel presented integer linear formulations for obtaining suitable allocations (projections of the iteration space) as well as for scheduling these projected algorithms. But even in his work [Fim00] on LSGP partitioning, the processor allocation is derived by projection. Hence, the derived linear schedule corresponds to the hyperplane method [Lam74]. In order to not exceeded a given number n of processors, the hyperplane is chosen in such a manner that the number of iteration points on the plane is less or equal to n.

Müller [MFM04, Mül04, Mül06] developed optimal scheduling methods for software pipelining taking several parallel functional units and register constraints into account. The work considers single processor scheduling and one-dimensional loops.

Siegel [SM04, SM06b, SM06c, SM06d] studied integer linear program formulations that minimize data reuse and communication within processor arrays. His works are orthogonal to ours. In [SMHT06], we studied the combination of both approaches.

As another distinction from previously published results, all aforementioned works do not consider conditional program execution. Finally, in case of partitioning, only rectangular and higher dimensional orthogonal shaped tiles have been considered so far.

3.2 Preliminaries

In the following, perfectly nested loop programs with a linearly bounded lattice as iteration space $\mathcal{I}$ and uniform data dependencies are considered as defined in Section 2.1.1. For methods, how to transform non-perfectly loop nests into perfectly ones and how to handle affine data dependencies, we refer to Chapter 4.

Affine transformations are used in order to assign each iteration point $I \in \mathcal{I} \subset \mathbb{Z}^n$ a processor index $p \in \mathcal{P}$ (global allocation) and a time index $t \in \mathcal{T}$ (scheduling). Both affine transformations together define the so-called *space-time mapping*[8] as given in Equation (3.1).

$$\begin{pmatrix} p \\ t \end{pmatrix} = \underbrace{\begin{pmatrix} \Phi \\ \Lambda \end{pmatrix}}_{\Theta} I + \begin{pmatrix} \phi \\ \lambda \end{pmatrix} \tag{3.1}$$

where $\Phi \in \mathbb{Z}^{s \times n}$, $\Lambda \in \mathbb{Z}^{1 \times n}$, $\phi \in \mathbb{Z}^s$, $s < n \in \mathbb{N}$, and $\lambda \in \mathbb{Z}$. The set $\mathcal{T} = \{t \mid t = \Lambda I + \lambda \ \wedge \ I \in \mathcal{I}\} \subset \mathbb{Z}$ is called *time space*, that is the set of all time steps where an execution takes place. The set $\mathcal{P} = \{p \mid p = \Phi I + \phi \ \wedge \ I \in \mathcal{I}\} \subset \mathbb{Z}^s$ is called *processor space*. Its cardinality $|\mathcal{P}|$ denotes the number of processors. Φ is called *allocation matrix* and determines the processor (maybe offset by ϕ) that executes an iteration point I. Λ is called *schedule vector* and provides the start time (optionally shifted by

[8]Moldovan [Mol83] uses already a similar linear transformation but he requires the transformation to be bijective, that is, the transformation matrix Θ must have full rank. The same definition is used by Lengauer [Len93] and referred to as *space-time mapping*.

the offset λ) of each iteration point I. Together, Φ and Λ build the *transformation matrix* Θ.

Due to technological and physical constraints, the dimension s is usually limited to two or in case of *3D stacking* [LDG$^+$02, BAB$^+$06] to three. Otherwise, the higher dimensional processor array has to be embedded in a lower dimensional one.

The space mappings, considered within the thesis as processor allocation (global allocation) as well as the allocation of functional units are described more precisely in the following section. Afterwards, in Sections 3.4 to 3.8, different resource constrained scheduling methods are developed in order to derive suitable scheduling vectors Λ.

3.3 Allocation and Space Mapping

Let an extended reduced dependence graph (cf. Definition 2.7) for a DPRA (uniform data dependencies are represented by a set D) be given, thus we can write $G = (V,E,D)$. In order to have a relation between the individual operations denoted by the nodes $v_i \in V$ and the actual available resources (adders, multipliers, etc.), a so-called *resource graph* [Thi95] is introduced.

Definition 3.1 (Resource graph). *A* resource graph $G_R = (V_R, E_R, W, \Delta)$ *is a bipartite graph. The set of nodes* $V_R = V \cup V_T$ *includes, as the first partition of nodes, the set of nodes* V *of the reduced dependence graph* $G = (V,E,D)$. *The second partition is defined by the* resource types *(for instance adders or multipliers)* $r_k \in V_T$. *An edge* $(v_i, r_k) \in E_R$ *with* $v_i \in V$ *and* $r_k \in V_T$ *denotes a* binding possibility, *that is,* v_i *might be implemented on an instance of resource type* r_k. *There exist a function* $w : E_R \to \mathbb{N}_0$, *which assigns a time* $w(v_i, r_k) \in W$ *to each edge* $(v_i, r_k) \in E_R$. $w(v_i, r_k)$ *denotes the execution time of node* v_i *processed on* r_k. *Furthermore, there exist a function* $\delta : V_T \to \mathbb{N}$, *which assigns to each resource type a* pipeline rate $\delta(r_k) \in \Delta$. *In case of multi-cycle operations, the pipeline rate denotes the number of clock cycles that a resource is occupied before the next operation can start.*

Similar to the execution times and pipeline rates, Definition 3.1 can be easily extended by further attributes such as area or power cost, which should not be formalized in greater detail at this point.

In order to express the quantity of available resource types, the local allocation (cf. Section 3.1.1) is defined as follows.

Definition 3.2 (Local allocation). *Let a specification defined by a reduced dependence graph* $G = (V,E,D)$ *and a resource graph* $G_R = (V_R, E_R, W, \Delta)$ *be given. Then, the* allocation *is a function* $\alpha : V_T \to \mathbb{N}_0$ *that assigns to each resource type* $r_k \in V_T$ *a number* $\alpha(r_k)$ *of available instances.*

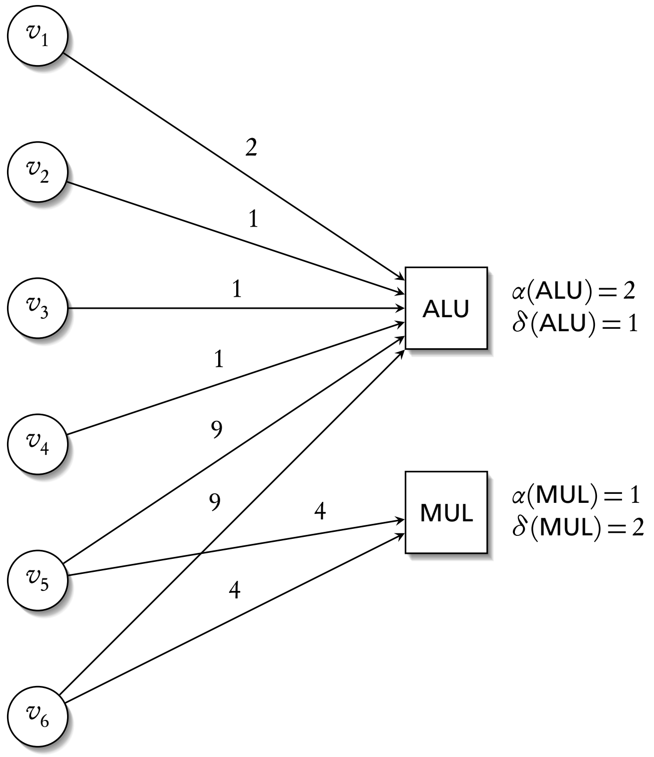


Figure 3.1: Resource graph and allocation.

In Figure 3.1, the concepts of a resource graph and allocation are shown. Here, two instances of an ALU and one dedicated multiplier are available. The ALUs have a pipeline rate of one, the multiplier of two. For instance, the node v_6, representing a multiplication operation, has two binding possibilities. As the first one, the operation can be bound to the ALU with an execution time of nine clock cycles. The second binding possibility is on the dedicated multiplier with four cycles latency.

Next, different variants of space mappings are presented that define the global allocation (cf. Section 3.1.1). For illustration purposes, the following simple example is used.

Example 3.1 (FIR filter)**.** *An FIR (finite impulse response) filter is described by the difference equation*

$$Y(i) = \sum_{j=0}^{N-1} A(j) \cdot U(i-j) \qquad \forall i \, : \, 0 \leq i < M \tag{3.2}$$

Where N denotes the number of filter taps, $A(j)$ the filter coefficients, $U(i)$ the filter inputs, and $Y(i)$ the filter result. After parallelization *and* embedding *in a common it-*

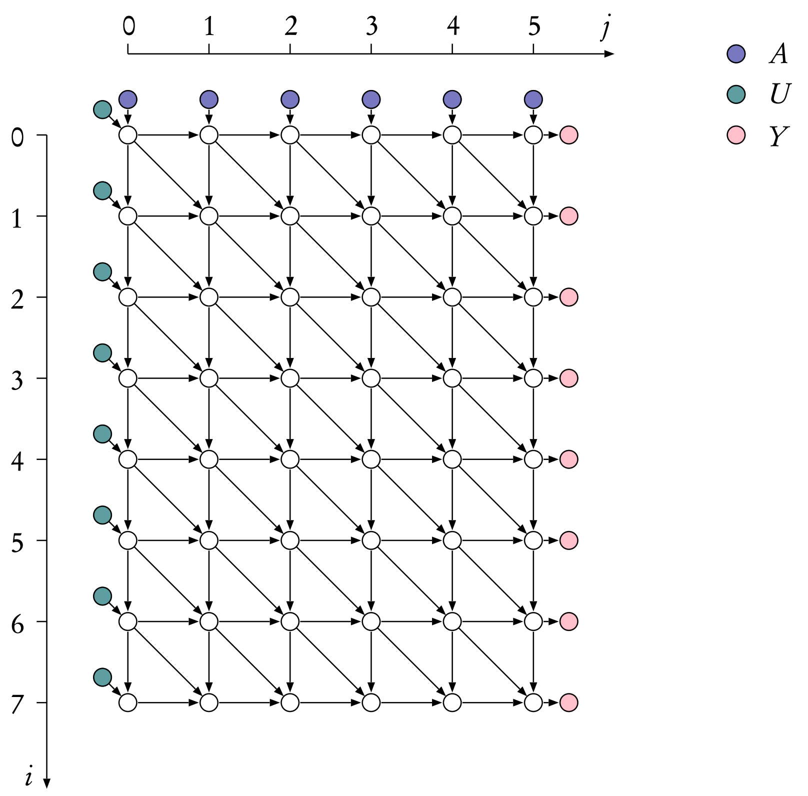


Figure 3.2: Dependence graph of the FIR filter in Equation (3.2) for $N = 6$ and $M = 8$. Note that each instance of the input and output variables (A, U, and Y) belongs to its nearest integral point. They are only shifted for visibility reasons.

eration space, the difference equation can be expressed by the following equivalent uniform algorithm.

$$
\begin{array}{rll}
a[i,j] &= A[j] & \text{if } i=0 \\
a[i,j] &= a[i-1,j] & \text{if } i>0 \\
u[i,j] &= U[i] & \text{if } j=0 \\
u[i,j] &= 0 & \text{if } i=0 \wedge j>0 \\
u[i,j] &= u[i-1,j-1] & \text{if } i>0 \wedge j>0 \\
z[i,j] &= a[i,j]\cdot u[i,j] & \\
y[i,j] &= z[i,j] & \text{if } j=0 \\
y[i,j] &= y[i,j-1]+z[i,j] & \text{if } j>0 \\
Y[i] &= y[i,j] & \text{if } j=N-1
\end{array}
$$

The iteration space is defined by $\mathcal{I} = \{(i\ j)^T \in \mathbb{Z}^2 \mid 0 \le i \le M-1 \wedge 0 \le j \le N-1\}$.

The dependence graph of the FIR filter for $N = 6$ and $M = 8$ is shown in Figure 3.2.

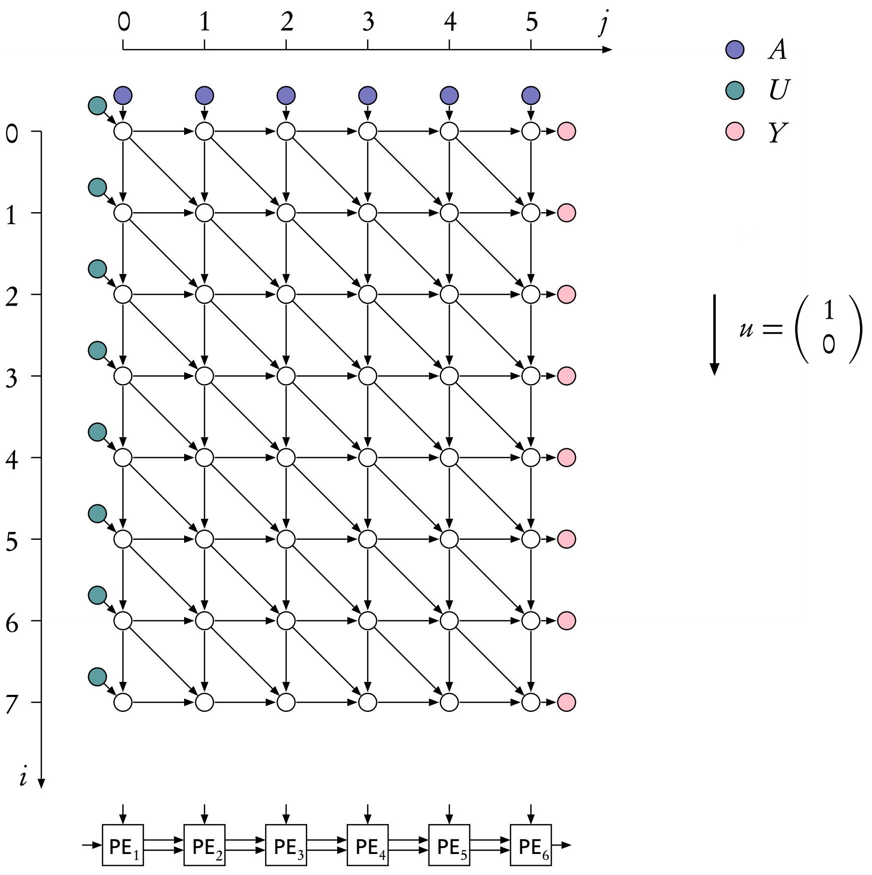


Figure 3.3: A projection in direction $u = (1\ 0)^T$ of the iteration space leads to an implementation with six processing elements.

3.3.1 Projection

In order not to synthesize one dedicated processing element for each iteration point, a projection of the iteration space along a vector u is considered. Projection of the iteration space of the FIR filter example in direction $u = (1\ 0)^T$ results in six processing elements (see Figure 3.3). Each of the six elements PE_j has to process the eight iteration points that are projected on each other. Since an iteration space $\mathcal{I} \subset \mathbb{Z}^n$ is reduced by a projection vector $u \in \mathbb{Z}^n$ by one, the corresponding allocation matrix Φ is an $(n-1) \times n$ matrix. The matrix is an equivalent description of the allocation, if it satisfies $\Phi u = 0$ [Kuh80]. If the projection vector is given by $u = (u_1\ u_2\ \dots\ u_n)^{\mathrm{T}} \in \mathbb{Z}^n$, the allocation matrix Φ can be constructed as follows:

$$\Phi = \begin{pmatrix} u_i & 0 & \dots & 0 & -u_1 & 0 & \dots & 0 \\ 0 & u_i & \dots & 0 & -u_2 & \vdots & & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & & & \\ 0 & 0 & \dots & u_i & -u_{i-1} & 0 & \dots & 0 \\ 0 & \dots & & 0 & -u_{i+1} & u_i & \dots & 0 \\ \vdots & & & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & & 0 & -u_n & 0 & \dots & u_i \end{pmatrix}$$

where $u_i \neq 0,\ 1 \leq i \leq n$.

3.3.2 Partitioning

Formally, partitioning divides the iteration space $\mathcal{I} \subset \mathbb{Z}^n$ into congruent tiles (see Section 3.1.1), such that it is decomposed into spaces $\mathcal{I}_1$ and $\mathcal{I}_2$ to fulfill $\mathcal{I} \subseteq \mathcal{I}_1 \oplus \mathcal{I}_2$. In case of parallelotope-shaped tiles, the decomposition is defined as follows.

$$\mathcal{I}_1 \oplus \mathcal{I}_2 = \{ I = I_1 + T I_2 \mid I_1 \in \mathcal{I}_1 \wedge I_2 \in \mathcal{I}_2 \wedge T \in \mathbb{Z}^{n \times n} \} \tag{3.3}$$

Here, $\mathcal{I}_1 \in \mathbb{Z}^n$ represents the points within the tile and $\mathcal{I}_2 \in \mathbb{Z}^n$ accounts for the regular repetition of the tiles, meaning, the origin of each tile. The tile shape and its size is defined by a tiling matrix T. When partitioning a given algorithm by a tiling matrix T, mainly two things have to be carried out. The iteration space of the algorithm has to be decomposed according to Equation (3.3). Furthermore, since the dimension of the iteration space is increased (two times n), all variables have to be embedded in the higher dimensional iteration space such that all data dependencies are preserved, additional equations may have to be added that define the inter-tile dependencies. For further details we refer to [TT92, TT93, Tei93] and our work in [DHT06c].

For exemplification, the FIR filter example is partitioned by the following tiling matrix T, which defines a rectangular tile.

$$T = \begin{pmatrix} M_1 & 0 \\ 0 & N_1 \end{pmatrix} \qquad \text{with } M_1 < M \wedge N_1 < N$$

$$
\begin{array}{ll}
a[i_1,j_1,i_2,j_2] = A[i_1,j_1,i_2,j_2] & \text{if } i_1 = 0 \wedge i_2 = 0 \\
a[i_1,j_1,i_2,j_2] = a[i_1-1,j_1,i_2,j_2] & \text{if } i_1 > 0 \\
a[i_1,j_1,i_2,j_2] = a[i_1+M_1-1,j_1,i_2-1,j_2] & \text{if } i_1 = 0 \wedge i_2 > 0 \\
u[i_1,j_1,i_2,j_2] = U[i_1,j_1,i_2,j_2] & \text{if } j_1 = 0 \wedge j_2 = 0 \\
u[i_1,j_1,i_2,j_2] = 0 & \text{if } i_1 = 0 \wedge i_2 = 0 \\
 & \quad \wedge\ j_1 + j_2 > 0 \\
u[i_1,j_1,i_2,j_2] = u[i_1-1,j_1-1,i_2,j_2] & \text{if } i_1 > 0 \wedge j_1 > 0 \\
u[i_1,j_1,i_2,j_2] = u[i_1+M_1-1,j_1-1,i_2-1,j_2] & \text{if } i_1 = 0 \wedge j_1 > 0 \\
 & \quad \wedge\ i_2 > 0 \\
u[i_1,j_1,i_2,j_2] = u[i_1-1,j_1+N_1-1,i_2,j_2-1] & \text{if } i_1 > 0 \wedge j_1 = 0 \\
 & \quad \wedge\ j_2 > 0 \\
u[i_1,j_1,i_2,j_2] = u[i_1+M_1-1,j_1+N_1-1,i_2-1,j_2-1] & \text{if } i_1 = 0 \wedge j_1 = 0 \\
 & \quad \wedge\ i_2 > 0 \wedge j_2 > 0 \\
z[i_1,j_1,i_2,j_2] = a[i_1,j_1,i_2,j_2] \cdot u[i_1,j_1,i_2,j_2] & \\
y[i_1,j_1,i_2,j_2] = z[i_1,j_1,i_2,j_2] & \text{if } j_1 = 0 \wedge j_2 = 0 \\
y[i_1,j_1,i_2,j_2] = y[i_1,j_1-1,i_2,j_2] + z[i_1,j_1,i_2,j_2] & \text{if } j_1 > 0 \\
y[i_1,j_1,i_2,j_2] = y[i_1,j_1+N_1-1,i_2,j_2-1] + z[i_1,j_1,i_2,j_2] & \text{if } j_1 = 0 \wedge j_2 > 0 \\
Y[i_1,j_1,i_2,j_2] = y[i_1,j_1,i_2,j_2] & \text{if } j_1 = N_1 - 1 \wedge j_2 = N_2 - 1
\end{array}
$$

Where the two iteration spaces are defined by

$$
\begin{aligned}
\mathcal{I}_1 &= \left\{ (i_1\ j_1)^{\mathrm{T}} \in \mathbb{Z}^2 \mid 0 \le i_1 \le M_1 - 1 \wedge 0 \le j_1 \le N_1 - 1 \right\} \\
\mathcal{I}_2 &= \left\{ (i_2\ j_2)^{\mathrm{T}} \in \mathbb{Z}^2 \mid 0 \le i_2 \le M_2 - 1 \wedge 0 \le j_2 \le N_2 - 1 \right\}
\end{aligned}
$$

Where $M_2 = \left\lceil \frac{M}{M_1} \right\rceil$ and $N_2 = \left\lceil \frac{N}{N_1} \right\rceil$. Note, if the fractions are integral (that is, if it is not necessary to round up), a partitioning is called *perfect*. Partitioned by $T = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$, the FIR filter example is shown in Figure 3.4. According to the tiling matrix, the iteration spaces are defined by

$$
\begin{aligned}
\mathcal{I}_1 &= \left\{ (i_1\ j_1)^{\mathrm{T}} \in \mathbb{Z}^2 \mid 0 \le i_1 \le 1 \wedge 0 \le j_1 \le 2 \right\} \\
\mathcal{I}_2 &= \left\{ (i_2\ j_2)^{\mathrm{T}} \in \mathbb{Z}^2 \mid 0 \le i_2 \le 3 \wedge 0 \le j_2 \le 1 \right\}
\end{aligned}
$$

Until now, the performed partitioning is only a decomposition of the iteration space and clustering of iterations. Only by interpretation, the partitioning becomes a LSGP or LPGS partitioning and therefore the processor allocation. For instance,

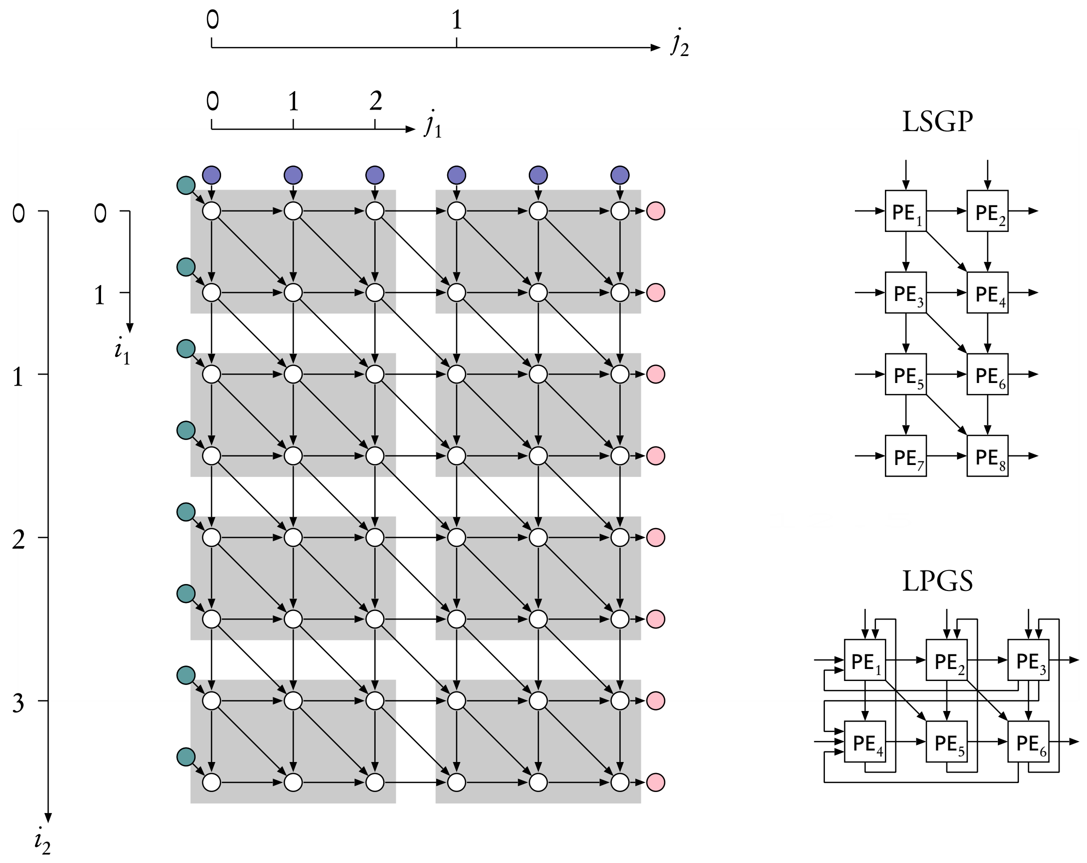


Figure 3.4: Partitioned iteration space of the FIR filter algorithm as introduced in Example 3.1. On the right, different interpretations of the processor allocation (LSGP and LPGS) are shown.

if the iteration points within the tile (iteration space $\mathcal{I}_1$) should be executed in a sequential manner and each tile can work concurrently, the LSGP approach is at hand. This corresponds to a total of eight processors running in parallel and each executing six iterations sequentially. The other way around, if six processor are working in parallel on the points of one tile and afterwards on the next tile, and so on, the partitioning corresponds to the LPGS approach. In order to distinguish between the methods, the iteration spaces are named by the following convention.

- $\mathcal{I}_{LS}$ and $\mathcal{I}_{GS}$ denote iteration spaces where the iterations should be executed one after the other.

- $\mathcal{I}_{par}$ denotes an iteration space where the iterations might be executed in parallel.

For the LSGP approach, the allocation (number of processors $|\mathcal{I}_{par}|$) depends on the size of original iteration space $\mathcal{I}$. In case of the LPGS method, it is defined by the size of the tile $|\det(T)|$ and is thus independent of the original size of the iteration space.

If an n-dimensional iteration space is partitioned by the LSGP approach, the space mapping is defined as follows.

$$p = \Phi \begin{pmatrix} I_{LS} \\ I_{par} \end{pmatrix} + \phi = (Z\ E) \begin{pmatrix} I_{LS} \\ I_{par} \end{pmatrix} + \phi \qquad \forall I_{LS} \in \mathcal{I}_{LS} \wedge I_{par} \in \mathcal{I}_{par}$$

Where I_{LS} and I_{par} are each n-dimensional iteration vectors and Z and E are an $n \times n$ zero and identity matrix, respectively.

In the case that an n-dimensional iteration space is partitioned by the LPGS method, the processor allocation is defined as follows.

$$p = \Phi \begin{pmatrix} I_{par} \\ I_{GS} \end{pmatrix} + \phi = (E\ Z) \begin{pmatrix} I_{par} \\ I_{GS} \end{pmatrix} + \phi \qquad \forall I_{par} \in \mathcal{I}_{par} \wedge I_{GS} \in \mathcal{I}_{GS}$$

In conclusion, remark that, if an iteration space $\mathcal{I} \subset \mathbb{Z}^n$ is partitioned, the resulting subspaces could also be of lower dimension than the original. This case occurs when one facet of the tile entirely covers all iteration points within one direction of the original iteration space. An example is given in the next section.

3.3.3 Hierarchical Partitioning

Hierarchical partitioning methods use different tiling matrices to divide the iteration space on several levels. For instance, if an n-dimensional iteration space $\mathcal{I}$ is partitioned twice, it is decomposed into the spaces $\mathcal{I}_1, \mathcal{I}_2$, and $\mathcal{I}_3$ such that $\mathcal{I} \subseteq \mathcal{I}_1 \oplus \mathcal{I}_2 \oplus \mathcal{I}_3$. The decomposition is defined as follows.

$$\mathcal{I}_1 \oplus \mathcal{I}_2 \oplus \mathcal{I}_3 = \left\{ I = T_1 I_1 + T_2 I_2 + I_3 \ \middle|\ \begin{array}{l} I_1 \in \mathcal{I}_1 \wedge I_2 \in \mathcal{I}_2 \wedge I_3 \in \mathcal{I}_3 \\ \wedge\ T_1, T_2 \in \mathbb{Z}^{n \times n} \end{array} \right\} \tag{3.4}$$

Similarly an n-hierarchical partitioning method decomposes the iteration space $\mathcal{I}$ into $n + 1$ spaces.

As mentioned earlier, co-partitioning is one such example of a 2-level hierarchical partitioning [EM97a]. It uses both LSGP and LPGS methods in order to balance local memory requirements with the I/O bandwidth and having simultaneously the advantage of problem size independence. In detail, the iteration space is first partitioned into LS (locally sequential) tiles. This tiled iteration space is tiled once more

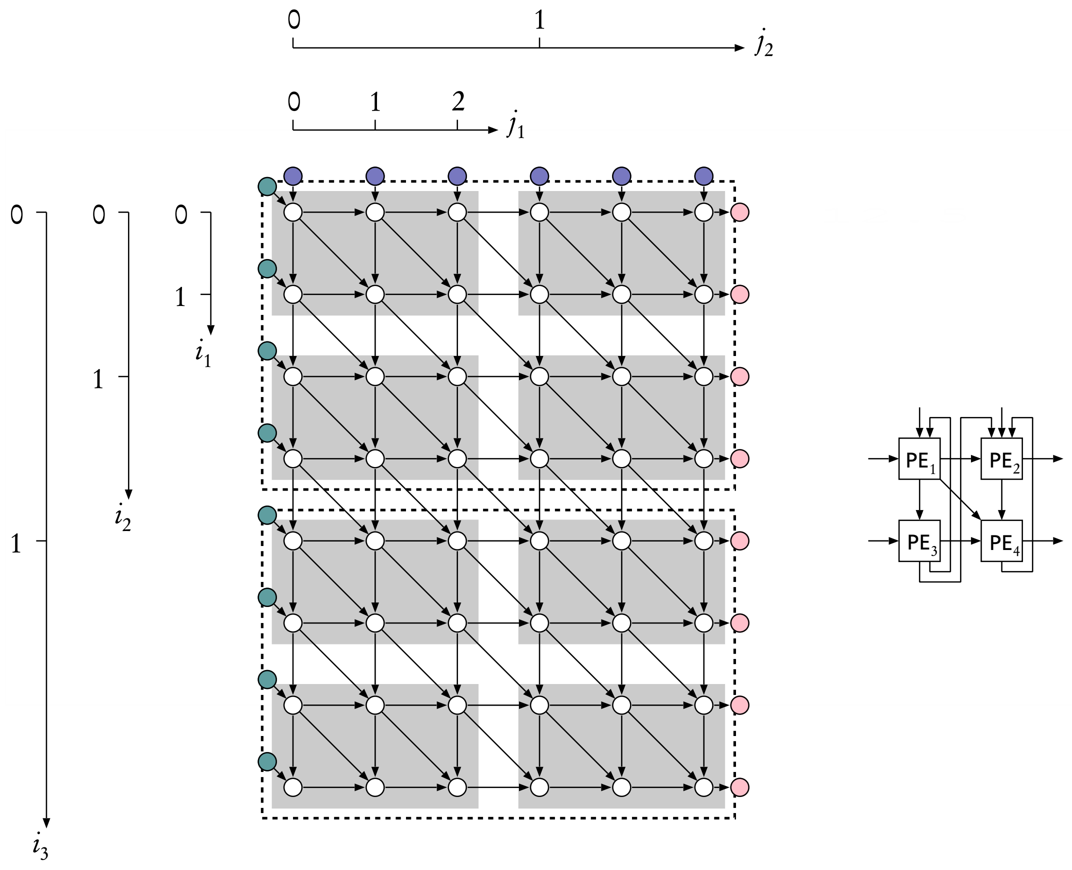


Figure 3.5: Co-partitioned iteration space of the FIR filter algorithm of Example 3.1. The processor array on the right side is defined by the number of LS tiles (gray tiles) within a GS tile (dashed rectangle).

using GS (globally sequential) tiles as shown in Figure 3.5. Formally, by rewriting Equation (3.4), the decomposition is given by

(3.5)

$$\mathcal{I}_{par} \oplus \mathcal{I}_{LS} \oplus \mathcal{I}_{GS} = \left\{ I = I_{LS} + T_{LS} I_{par} + T_{GS} I_{GS} \;\middle|\; \begin{array}{l} I_{par} \in \mathcal{I}_{par} \;\wedge\; I_{LS} \in \mathcal{I}_{LS} \\ \wedge\; I_{GS} \in \mathcal{I}_{GS} \\ \wedge\; T^{LS}, T^{GS} \in \mathbb{Z}^{n \times n} \end{array} \right\}$$

The two congruent tile types are defined by the tiling matrices T_{LS} and T_{GS}. $\mathcal{I}_{par} \subset \mathbb{Z}^n$ represents the origins of the LS tiles and $\mathcal{I}_{LS} \subset \mathbb{Z}^n$ represents the points within the LS tiles (that is, the smaller gray tiles in Figure 3.5). $\mathcal{I}_{GS} \subset \mathbb{Z}^n$ accounts for the regular repetition of the GS tiles (the bigger tiles marked with dashed lines in Figure 3.5). Note that the number of LS tiles within a GS tile defines the number of processors.

In case that an n-dimensional iteration space is co-partitioned, the processor allocation is defined as follows.

$$(3.6)\quad p = \Phi \begin{pmatrix} I_{LS} \\ I_{par} \\ I_{GS} \end{pmatrix} + \phi = (Z\ E\ Z) \begin{pmatrix} I_{LS} \\ I_{par} \\ I_{GS} \end{pmatrix} + \phi \quad \forall I_{LS} \in \mathcal{I}_{LS} \wedge I_{par} \in \mathcal{I}_{par} \wedge I_{GS} \in \mathcal{I}_{GS}$$

Here, I_{LS}, I_{par}, and I_{GS} are each n-dimensional iteration vectors and Z and E denote the $n \times n$ zero and identity matrix, respectively.

Example 3.2 (Co-partitioned FIR filter). *Let the FIR filter from Example 3.1 with* $N = 6$ *and* $M = 8$ *being co-partitioned by the following tiling matrices (cf. Figure 3.5).*

$$T_{LS} = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix} \qquad T_{GS} = \begin{pmatrix} 4 & 0 \\ 0 & 6 \end{pmatrix}$$

Normally, since the original iteration space is 2-dimensional, the co-partitioned space would have $3 \cdot 2 = 6$ *dimensions, but since the* T_{GS} *tile is as wide as the iteration space, five dimensions are sufficient in order to describe the partitioned algorithm.*

$$\begin{array}{ll}
a[i_1,j_1,i_2,j_2,i_3] = A[i_1,j_1,i_2,j_2,i_3] & \text{if } i_1 = 0 \wedge i_2 = 0 \wedge i_3 = 0 \\
a[i_1,j_1,i_2,j_2,i_3] = a[i_1-1,j_1,i_2,j_2,i_3] & \text{if } i_1 > 0 \\
a[i_1,j_1,i_2,j_2,i_3] = a[i_1+1,j_1,i_2-1,j_2,i_3] & \text{if } i_1 = 0 \wedge i_2 > 0 \\
a[i_1,j_1,i_2,j_2,i_3] = a[i_1+1,j_1,i_2+1,j_2,i_3-1] & \text{if } i_1 = 0 \wedge i_2 = 0 \wedge i_3 > 0 \\
u[i_1,j_1,i_2,j_2,i_3] = U[i_1,j_1,i_2,j_2,i_3] & \text{if } j_1 = 0 \wedge j_2 = 0 \\
u[i_1,j_1,i_2,j_2,i_3] = 0 & \text{if } i_1 = 0 \wedge i_2 = 0 \wedge i_3 = 0 \\
 & \quad \wedge\ j_1 + j_2 > 0 \\
u[i_1,j_1,i_2,j_2,i_3] = u[i_1-1,j_1-1,i_2,j_2,i_3] & \text{if } i_1 > 0 \wedge j_1 > 0 \\
u[i_1,j_1,i_2,j_2,i_3] = u[i_1+1,j_1-1,i_2-1,j_2,i_3] & \text{if } i_1 = 0 \wedge j_1 > 0 \wedge i_2 > 0 \\
u[i_1,j_1,i_2,j_2,i_3] = u[i_1-1,j_1+2,i_2,j_2-1,i_3] & \text{if } i_1 > 0 \wedge j_1 = 0 \wedge j_2 > 0 \\
u[i_1,j_1,i_2,j_2,i_3] = u[i_1+1,j_1+2,i_2-1,j_2-1,i_3] & \text{if } i_1 = 0 \wedge j_1 = 0 \wedge i_2 > 0 \\
 & \quad \wedge\ j_2 > 0 \\
u[i_1,j_1,i_2,j_2,i_3] = u[i_1+1,j_1-1,i_2+1,j_2,i_3-1] & \text{if } i_1 = 0 \wedge j_1 > 0 \wedge i_2 = 0 \\
 & \quad \wedge\ i_3 > 0 \\
u[i_1,j_1,i_2,j_2,i_3] = u[i_1+1,j_1+2,i_2+1,j_2-1,i_3-1] & \text{if } i_1 = 0 \wedge j_1 = 0 \wedge i_2 = 0 \\
 & \quad \wedge\ j_2 > 0 \wedge i_3 > 0 \\
z[i_1,j_1,i_2,j_2,i_3] = a[i_1,j_1,i_2,j_2,i_3] \cdot u[i_1,j_1,i_2,j_2,i_3] & \\
y[i_1,j_1,i_2,j_2,i_3] = z[i_1,j_1,i_2,j_2,i_3] & \text{if } j_1 = 0 \wedge j_2 = 0 \\
y[i_1,j_1,i_2,j_2,i_3] = y[i_1,j_1-1,i_2,j_2,i_3] + z[i_1,j_1,i_2,j_2,i_3] & \text{if } j_1 > 0 \\
y[i_1,j_1,i_2,j_2,i_3] = y[i_1,j_1+2,i_2,j_2-1,i_3] + z[i_1,j_1,i_2,j_2,i_3] & \text{if } j_1 = 0 \wedge j_2 > 0 \\
Y[i_1,j_1,i_2,j_2,i_3] = y[i_1,j_1,i_2,j_2,i_3] & \text{if } j_1 = 2 \wedge j_2 = 1
\end{array}$$

The three iteration spaces are defined by

$$\begin{aligned}
\mathcal{I}_{LS} &= \left\{(i_1\ j_1)^T \in \mathbb{Z}^2 \mid 0 \leq i_1 \leq 1 \wedge 0 \leq j_1 \leq 2\right\} \\
\mathcal{I}_{par} &= \left\{(i_2\ j_2)^T \in \mathbb{Z}^2 \mid 0 \leq i_2 \leq 1 \wedge 0 \leq j_2 \leq 1\right\} \\
\mathcal{I}_{GS} &= \{i_3 \in \mathbb{Z} \mid 0 \leq i_3 \leq 1\}
\end{aligned}$$

3.4 Linear Scheduling

In this section, several important results by Darte and others [DKR92] are recapitulated, which are important in the course of the thesis.

Let an algorithm with uniform data dependencies $D = (d_1, d_2, \dots, d_m)$ and a convex iteration space $\mathcal{I}$ be given. Consider two iteration points $I_1, I_2 \in \mathcal{I}$ where I_2 depends on I_1, that is, $I_2 = I_1 + d_i$ for some data dependence $d_i \in D$. In this case, we write $I_2 \succ I_1$.

Definition 3.3 (Schedule)**.** *A* schedule *for an algorithm with uniform data dependencies with a corresponding convex iteration space $\mathcal{I}$ is a function $t : \mathcal{I} \rightarrow \mathbb{Z}$ so that for any iteration points $I_1, I_2 \in \mathcal{I}\ :\ t(I_2) > t(I_1)$ if $I_2 \succ I_1$.*

Definition 3.4 (Free schedule)**.** *A schedule for an algorithm with uniform data dependencies with a corresponding convex iteration space $\mathcal{I}$ is called* free *or* greedy *if*

$$t(I) = \begin{cases} 0 & \text{if there exists no } J\ :\ I \succ J, \text{ i.e., } J \notin \mathcal{I} \\ \max(t(J)\ :\ I, J \in \mathcal{I} \wedge I \succ J) + 1 & \text{else} \end{cases}$$

If a free schedule exist, it defines the fastest possible execution order of a given algorithm.

Definition 3.5 (Linear schedule)**.** *Let an algorithm with uniform data dependencies $D = (d_1, d_2, \dots, d_n)$ with a corresponding iteration space $\mathcal{I}$ be given. Then, a schedule is called* linear *if*

$$t(I) = \lfloor \Lambda I \rfloor \qquad\qquad \forall I \in \mathcal{I}$$

where the linear schedule vector $\Lambda \in \mathbb{Q}^{1\times n}$ is such that $\Lambda d_i \geq 1$ for all $d_i \in D$. This condition [Lam74] ensures that all data dependencies are preserved.

Note that $t(I)$ is not necessarily a hyperplane since it can consist of several successive parallel fronts if Λ is not integral. The use of floor functions with a rational scheduling vector Λ turns out to be more powerful than restricting the search to integer scheduling vectors [DR95].

Various studies and experimental evaluations, for instance, the works in [FP84, Pol88, SOF94], have shown that linear schedules allow the parallel computation of nested loop programs with only low overhead in execution time. Fortes and Parisi-Presiccein [FP84] already presumed that the difference between the execution time of the linear and the corresponding free schedule is only a constant and that it is invariant up to the size of the iteration space. This presumption has been proven by Darte and others in [DKR92]. The authors have shown that for an arbitrary convex iteration space that is sufficiently *fat*[9], the difference between the total execution time of the best linear schedule and that of the free schedule is bound by a constant independent of the size of the iteration space.

The closeness of linear schedules to optimality is of great relevance since linear schedules have many benefits, such as they are very simple and thus easy to use in practice. Further, their simplicity results in a low implementation overhead for control.

Darte et alii [DKR92] derived the following linear program for the determination of an optimal schedule vector Λ for a given algorithm with uniform data dependencies.

Linear scheduling

Input:

- Algorithm with data dependencies defined by matrix D
- Iteration space $\mathcal{I}$ is defined by a polyhedron $\mathcal{I} = \{I \in \mathbb{Z}^n \mid AI \geq b\}$ where $A \in \mathbb{Z}^{m \times n}$

Output:

- Schedule vector $\Lambda \in \mathbb{Q}^{1 \times n}$

Program:

$$\begin{array}{rll} \min & -(y_1 + y_2)b & \\ \text{subject to} & y_1 A = \Lambda & y_1 \in \mathbb{Q}^{1 \times m} \\ & y_2 A = -\Lambda & y_2 \in \mathbb{Q}^{1 \times m} \\ & y_1 \geq 0 & \\ & y_2 \geq 0 & \\ & \Lambda d_i \geq 1 & \forall d_i \in D \end{array}$$

[9]An iteration space is called *fat* if it contains the zero vector, all data dependence vectors of a corresponding algorithm, and all canonical basis vectors [DKR92].

The derivation of the above linear program formulation as well as the meaning of constraints and variables is described in the next section in the context of affine scheduling.

3.5 Affine Scheduling

The concept of linear scheduling introduced in the last section is rather "coarse-grained" since each iteration is considered as a whole. That means, just an optimal execution order of the iterations with respect to the data dependencies is determined. The derived schedules are only optimal if the iteration is atomic. However, most of the time, one iteration (the loop body) consists of several operations. This leads to the idea to have an offset for each operation of the iteration besides the linear part of the schedule. We refer to this type of schedules as *affine schedules*[10].

In the following algorithms in *output normal form* (see Chapter 4) with regular data dependencies (cf. Section 2.1.1), defined over a convex iteration space $\mathcal{I}$, are considered. An elaboration of the codomain of the iteration space follows from case to case later. These algorithms consist of one or more equations of the following form.

$$x_i[I] = \mathcal{F}(\ldots, x_j[I + d_e], \ldots) \qquad I \in \mathcal{I}_i \subseteq \mathcal{I} \tag{3.7}$$

When regarding scheduling, an equivalent representation in form of the reduced dependence graph $G = (V, E, D, W)$ is considered. Hence, the terms variable x_i of an algorithm and node $v_i \in V$ are used synonymously.

Formally, when an affine schedule is applied to the reduced dependence graph G with the iteration space $\mathcal{I}$, the calculation of a variable and the execution of an operation $x_i[I]$, respectively, for $I \in \mathcal{I}$ starts at time step

$$t_i(I) = \lfloor \Lambda I + \tau(v_i) \rfloor \tag{3.8}$$

where $t_i(I) \in \mathbb{Z}$, $\Lambda \in \mathbb{Q}^{1\times n}$, and $\tau(v_i) \in \mathbb{Q}$.
The execution is finished at

$$t_i(I) + w_i = \lfloor \Lambda I + \tau(v_i) + w_i \rfloor \tag{3.9}$$

where $w_i \in \mathbb{Z}$.

In our case, the primary objective of scheduling is to minimize the total execution time for a given algorithm. This latency L is given by:

$$L = \max_{I_2 \in \mathcal{I}} \left(\left\lfloor \Lambda I_2 + \max_{v_i \in V} \left(\tau(v_i) + w_i \right) \right\rfloor \right) - \min_{I_1 \in \mathcal{I}} \left(\left\lfloor \Lambda I_1 + \min_{v_i \in V} \left(\tau(v_i) \right) \right\rfloor \right) \tag{3.10}$$

[10]The authors in [DR95] call this type of schedules *affine with same linear part.*

In Equation (3.10), the first max-term denotes the latest execution time (ΛI_2) of any iteration point $I_2 \in \mathcal{I}$ plus the finishing time of the last operation that is executed at this iteration point. The subtrahend denotes the earliest execution time (ΛI_1) of any iteration iteration point $I_1 \in \mathcal{I}$ plus the starting time of any operation that is executed at this iteration point. It can be easily ensured that $\min_{v_i \in V} \left(\tau(v_i)\right) = 0$ by adding an offset. Hereby, Equation (3.10) can be simplified as follows.

$$L = \max_{I_2 \in \mathcal{I}} \left(\left\lfloor \Lambda I_2 + \max_{v_i \in V} \left(\tau(v_i) + w_i\right) \right\rfloor \right) - \min_{I_1 \in \mathcal{I}} (\lfloor \Lambda I_1 \rfloor) \tag{3.11}$$

The total evaluation time is approximated [DR92, Thi95] by

$$L = \max_{I_1, I_2 \in \mathcal{I}} (\Lambda(I_2 - I_1)) + \max_{v_i \in V} \left(\lfloor \tau(v_i) + w_i \rfloor\right) = L_g + L_l \tag{3.12}$$

The *global latency* L_g denotes the time needed for the execution of the entire iteration space and L_l is the *local latency* for computing a single iteration point. If the iteration space $\mathcal{I}$ is defined by a polyhedron $\mathcal{I} = \{I \in \mathbb{Z}^n \mid AI \geq b\}$, where $A \in \mathbb{Z}^{m \times n}$ and $b \in \mathbb{Z}^m$, the global latency L_g is given by the following formulation.

$$\begin{array}{rll} \max & \Lambda(I_2 - I_1) & \Lambda \in \mathbb{Q}^{1 \times n} \\ \text{subject to} & AI_1 \geq b & \\ & AI_2 \geq b & \end{array}$$

Since the schedule vector Λ as well as the iteration points I_1, I_2 are unknown, the objective function is non-linear and thus not directly determinable. Using the duality theorem of linear programming [Sch86, DR92, DKR92], the problem can be rewritten as follows.

$$\begin{array}{rll} \min & -(y_1 + y_2)b & \\ \text{subject to} & y_1 A = \Lambda & y_1 \in \mathbb{Q}^{1 \times m} \\ & y_2 A = -\Lambda & y_2 \in \mathbb{Q}^{1 \times m} \\ & y_1 \geq 0 & \\ & y_2 \geq 0 & \end{array}$$

The implied data dependencies given as edge annotations of a reduced dependence graph $G = (V, E)$ have to be guaranteed. That is, for all edges $(v_i, v_j) \in E$, the computation of node v_j cannot begin until the execution of node v_i has finished. Therefore, the starting time of v_j must be at least w_i greater than the starting time of node v_i. The corresponding precedence constraint

$$t_j(I) \; - \; t_i(I - d_e) \geq w_i \qquad \forall \, (v_i, v_j) = e \in E$$

together with Equation (3.8) directly leads to

$$\Lambda d_e + \tau(v_j) - \tau(v_i) \geq w_i \qquad \forall\, (v_i, v_j) = e \in E \tag{3.13}$$

Note that if not the start times $\tau(v_i)$ for each individual node are considered separately but the entire iteration I with an execution time of one itself, Lamport's [Lam74] constraint $\Lambda d_e \geq 1$ is obtained for each dependence vector d_e.

In the following, the entire mixed integer program for affine scheduling is given at a glance.

Affine scheduling

Input:

- Reduced dependence graph $G = (V, E, D)$
- Execution time $w_i \in W$ of each node $v_i \in V$, $W : V \to \mathbb{Z}$
- Iteration space $\mathcal{I}$ is defined by a polyhedron $\mathcal{I} = \{I \in \mathbb{Z}^n \mid AI \geq b\}$ where $A \in \mathbb{Z}^{m \times n}$

Output:

- Schedule vector $\Lambda \in \mathbb{Q}^{1 \times n}$
- Start times $\tau(v_i) \in \mathbb{Q}$ of all nodes $v_i \in V$

Program:

$$\begin{array}{lll}
\text{min} & -(y_1 + y_2)b & \\
\text{subject to} & y_1 A = \Lambda & y_1 \in \mathbb{Q}^{1 \times m} \\
 & y_2 A = -\Lambda & y_2 \in \mathbb{Q}^{1 \times m} \\
 & y_1 \geq 0 & \\
 & y_2 \geq 0 & \\
 & \Lambda d_e + \tau(v_j) - \tau(v_i) \geq w_i & \forall e = (v_i, v_j) \in E
\end{array}$$

The above MIP defines only an execution order of iterations I and nodes $v_i \in V$, but does not take any resource constraints into account. If there are no data dependencies ($\forall e \in E : d_e = 0$) in a given program, this leads to a high degree of parallelism since all iteration points can be executed concurrently. Therefore, in the next section, projection is considered as an allocation to assign iterations to processors.

3.5.1 Affine Scheduling with Projection as Allocation

As described in Section 3.3.1, allocation of iterations to processors by a given projection direction u, leads to the execution of all iteration points, satisfying $QI = \text{const}$ on the same processor. Since all iterations, satisfying the aforementioned condition, cannot be executed on the same processor simultaneously, they have to be serialized in an appropriate manner. Using an affine function for scheduling results in a constant interval for the successive execution of different iteration points on the same processor. This interval is called *iteration interval*, *initiation interval* [RST92], or *period*.

Definition 3.6 (Iteration interval). *The* iteration interval $P \in \mathbb{Z}$ *of an allocated and scheduled algorithm with regular data dependencies is the number of time instances between the evaluation of two successive instances of a variable within one processor.*

Theorem 3.1 (Iteration interval for projection [Thi95]). *The iteration interval P of an allocated and scheduled algorithm with regular data dependencies is given by*

$$P = |\Lambda u|. \tag{3.14}$$

Proof. The computations of variables $x_i[I_0 + \alpha u]$ assigned to one processing element are finished at times $\Lambda I + \tau(v_i) + w_i = \Lambda I_0 + \alpha \Lambda u + \tau(v_i) + w_i$. Two neighboring instances, that is $x_i[I_0 + \alpha u]$ and $x_i[I_0 + (\alpha \pm 1)u]$, have a distance in time of $\Delta t = \pm \Lambda u$. As the iteration interval is positive, we have $P = |\Lambda u|$. □

3.5.1.1 Determination of the Iteration Interval

In order to satisfy the condition in Equation (3.14), we have to formulate the absolute value of Λu and consider two different cases.

Case 1: Iteration interval P is given as a constant.
This case typically applies if scheduling shall be performed to obtain a given throughput. However, since the projection vector u is also given, it can happen that no valid schedule can be found and P has to be relaxed. This procedure of successive variation of the iteration interval leads to another typical approach, where a latency-optimal schedule shall be derived for a minimal P and the throughput is maximized, respectively.

A minimal iteration interval can be derived by starting with the value of one and successively increasing it until a solution is found. In order to accelerate this procedure, a bisectioning algorithm for $P \in [1..P_{max}]$ is applied. The upper bound P_{max} can be selected as the sum of worst case execution times of all nodes of the RDG. Since for larger P, more variables and constraints in the MIP are generated and therefore, an asymmetric bisection method is recommended.

Theorem 3.2 (MIP constraints for a constant iteration interval)**.** *Given a projection direction $u \in \mathbb{Z}^n$ and an iteration interval $P \in \mathbb{Z}$, $P > 0$, the projection vector u according to Definition 3.6 must satisfy*

$$\Lambda u \geq P - 2\beta_u P \tag{3.15}$$

$$\Lambda u \leq P \tag{3.16}$$

$$-\Lambda u \geq P - 2(1 - \beta_u)P \tag{3.17}$$

$$-\Lambda u \leq P \tag{3.18}$$

where $\beta_u \in \{0, 1\}$.

Proof. Since the iteration interval P is a positive integral number, two cases have to be considered.

- Assumption, $\Lambda u > 0$:
  If $\Lambda u > 0$, β_u is forced to be 0. The inequalities in Equation (3.15) and Equation (3.16) lead to $\Lambda u = P$. Otherwise, if $\beta_u = 1$, the inequality in Equation (3.17) is not satisfied ($-\Lambda u \geq 0$), which is a contradiction to the assumption.

- Assumption, $\Lambda u < 0$:
  If $\Lambda u < 0$, β_u is forced to be 1. The inequalities in Equation (3.17) and in Equation (3.18) lead to $-\Lambda u = P$. Otherwise, if $\beta_u = 0$, the inequality in Equation (3.16) is not satisfied ($\Lambda u \geq 0$), which is a contradiction to the assumption. □

Case 2: Iteration interval P is a variable of the MIP.
In this case, the iteration interval is not a fixed parameter. Thus, it has to be considered as a variable P in the linear program. Let P_{max} be an upper bound of the iteration interval, then the absolute value $|\Lambda u|$ can be determined by the set of inequalities given in Theorem 3.3.

Theorem 3.3 (MIP constraints for a variable iteration interval). *Given a projection direction $u \in \mathbb{Z}^n$, an iteration interval $P \in \mathbb{Z}$, $P > 0$, and an upper bound $P_{max} \in \mathbb{Z}$ for P. Then the projection vector u according to Definition 3.6 satisfies*

$$\begin{aligned} P &\leq P_{max} \\ \Lambda u &\geq P - 2\beta_u P_{max} \\ \Lambda u &\leq P \\ -\Lambda u &\geq P - 2(1-\beta_u)P_{max} \\ -\Lambda u &\leq P \end{aligned}$$

$$P = \sum_{l=1}^{P_{max}} l\, p_l \tag{3.19}$$

$$\sum_{l=1}^{P_{max}} p_l = 1 \tag{3.20}$$

where $\beta_u, p_l \in \{0, 1\}$.

Proof. Equation (3.19) encodes the iteration interval P by binary variables p_l. If a binary variable p_l is 1, P equals l. The constraint in Equation (3.20) ensures that exactly one value for $P \in [1, P_{max}]$ of Equation (3.19) is selected. The rest of the proof is analogous to the one of Theorem 3.2. □

In summary, the entire MIP for affine scheduling taking a given projection direction u into account is given in the following.

Affine scheduling with projection as allocation

Input:

- Reduced dependence graph $G = (V, E, D)$
- Execution time $w_i \in W$ of each node $v_i \in V$, $W : V \to \mathbb{Z}$
- Iteration space $\mathcal{I}$ is defined by a polyhedron $\mathcal{I} = \{I \in \mathbb{Z}^n \mid AI \geq b\}$ where $A \in \mathbb{Z}^{m \times n}$
- Projection vector $u \in \mathbb{Z}^n$
- The iteration interval P is given or not, as the case may be

⇝

⇝

Output:

- Schedule vector $\Lambda \in \mathbb{Q}^{1\times n}$
- Start times $\tau(v_i) \in \mathbb{Q}$ of all nodes $v_i \in V$
- Iteration interval P in case it is not given

Program:

$$
\begin{array}{lrl}
\min & & -(y_1+y_2)b \\
\text{subject to} & y_1 A = \Lambda & \qquad y_1 \in \mathbb{Q}^{1\times m} \\
& y_2 A = -\Lambda & \qquad y_2 \in \mathbb{Q}^{1\times m} \\
& y_1 \geq 0 & \\
& y_2 \geq 0 & \\
\Lambda d_e \; + \; \tau(v_j) - \tau(v_i) \geq w_i & & \qquad \forall\, e=(v_i,v_j)\in E \\
& \langle P \leq P_{max} & \qquad P_{max} \in \mathbb{Z}\rangle \\
& \Lambda u \geq P - 2\beta_u P_{max} & \qquad \beta_u \in \{0,1\} \\
& \Lambda u \leq P & \\
& -\Lambda u \geq P - 2(1-\beta_u)P_{max} & \\
& -\Lambda u \leq P & \\
& \left\langle P = \sum_{l=1}^{P_{max}} l\, p_l \right. & \qquad \left. p_l \in \{0,1\} \right\rangle \\
& \left\langle \sum_{l=1}^{P_{max}} p_l = 1 \right\rangle &
\end{array}
$$

If P is given, P_{max} becomes P and the optional constraints denoted by $\langle\cdot\rangle$ can be omitted.

3.5.1.2 Linearly Bounded Lattice as Iteration Space

All of the aforementioned scheduling methods consider an n-dimensional polyhedron $\mathcal{P}$, intersected by $\mathbb{Z}^n$, as iteration space $\mathcal{I} = \mathcal{P} \cap \mathbb{Z}^n$. For the sake of completeness, the scheduling methods are extended to more general iteration domains, in the form of linearly bounded lattices given by $\mathcal{I} = \{I \in \mathbb{Z}^n \mid I = M\kappa + c \,\wedge\, A\kappa \geq b\}$.

In case the matrix M is square and invertible, Thiele proposed in [Thi95] an appropriate MIP formulation.

Affine scheduling for LBLs with projection as allocation

Input:

- Reduced dependence graph $G = (V, E, D)$
- Execution time $w_i \in W$ of each node $v_i \in V$, $W : V \to \mathbb{Z}$
- Iteration space $\mathcal{I}$ is defined by a linearly bounded lattice $\mathcal{I} = \{I \in \mathbb{Z}^n \mid I = M\kappa + c \ \wedge \ A\kappa \geq b\}$ where $M \in \mathbb{Z}^{n \times n}$ and $A \in \mathbb{Z}^{m \times n}$
- Projection vector $u \in \mathbb{Z}^n$
- The iteration interval P is given or not, as the case may be

Output:

- Schedule vector $\Lambda \in \mathbb{Q}^{1 \times n}$
- Start times $\tau(v_i) \in \mathbb{Q}$ of all nodes $v_i \in V$
- Iteration interval P in case it is not given

Program:

$$\begin{array}{llr}
\min & -(y_1 + y_2)b & \\
\text{subject to} & \Lambda = \Lambda' M^{-1} & \\
& y_1 A = \Lambda' & y_1 \in \mathbb{Q}^{1 \times m} \\
& y_2 A = -\Lambda' & y_2 \in \mathbb{Q}^{1 \times m} \\
& y_1 \geq 0 & \\
& y_2 \geq 0 & \\
& \Lambda' \mathrm{adj}(M) d_e \geq |\det(M)|(\tau(v_i) - \tau(v_j)) + w_i & \forall\, e = (v_i, v_j) \in E \\
& \langle P \leq P_{max} & P_{max} \in \mathbb{Z} \rangle \\
& \Lambda' \mathrm{adj}(M) u \geq |\det(M)|(P - 2\beta_u P_{max}) & \beta_u \in \{0, 1\} \\
& \Lambda' \mathrm{adj}(M) u \leq |\det(M)| P & \\
& -\Lambda' \mathrm{adj}(M) u \geq |\det(M)|(P - 2(1 - \beta_u) P_{max}) & \\
& -\Lambda' \mathrm{adj}(M) u \leq |\det(M)| P & \\
& \left\langle P = \sum_{l=1}^{P_{max}} l\, p_l \right. & \left. p_l \in \{0, 1\} \right\rangle \\
& \left\langle \sum_{l=1}^{P_{max}} p_l = 1 \right\rangle &
\end{array}$$

If P is given, P_{max} becomes P and the optional constraints denoted by $\langle \cdot \rangle$ can be omitted.

3.5.2 Affine Scheduling with Partitioning as Allocation

In this section, constraints for the mixed integer programs are developed in order to cope with single and hierarchically partitioned algorithms. The main contribution are novel serialization constraints to ensure the right execution of LS and GS tiles (cf. Section 3.3.2).

Before formulating the appropriate serialization constraints and MIPs for several different partitioning methods, the concept of *strides* is introduced.

3.5.2.1 Determination of Strides and Path Strides

Definition 3.7 (Stride)**.** *The* stride *of a loop denotes an incremental or decremental step of an iteration variable. Sometimes, the stride is also referred to as* step size *or* increment. *A stride of size one is denoted as* unit stride. *If several loops are nested, each iteration variable has its own stride independent of outer loops. The sequential ordering of an iteration space in form of an n-dimensional parallelotope can be represented by a stride matrix $S \in \mathbb{Z}^{n \times n}$, containing n strides $s_i \in \mathbb{Z}^n$.*

$$S = (s_1\ s_2\ \ldots\ s_n) \tag{3.21}$$

Here, operations corresponding to iteration points in direction of s_1 are executed one after another, operations in direction of s_2 are separated in time by blocks of operations in direction s_1 and so on.

The sequential proceeding of iteration points is named *scanning*.

Definition 3.8 (Path stride)**.** *The* path strides *of an n-dimensional parallelotope are represented by a matrix $\vec{S} \in \mathbb{Z}^{n \times n}$ containing n vectors $\vec{s}_i \in \mathbb{Z}^n$.*

$$\vec{S} = (\vec{s}_1\ \vec{s}_2\ \ldots\ \vec{s}_n) \tag{3.22}$$

Here, in contrast to Definition 3.7, the strides depend on each other and on the size of the considered parallelotope so that a connected path is represented. That is, after several—depending on the size of the iteration space—steps $\vec{s}_1$ to an iteration point I, $\vec{s}_2$ denotes a vector from I to the next block of iterations. Remark: The first stride of matrix S is identical to the first path stride of matrix $\vec{S}$, $s_1 \equiv \vec{s}_1$.

Definition 3.7 and Definition 3.8 are illustrated by two examples in the following. Consider the following nested loop.

```
for (j=0 to 6 step 2)
{ for (i=0 to 7)
  { ...
  }
}
```

The inner loop of the program has unit stride. The outer loop has a stride of two. The combination of both strides in a stride matrix S results in:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$$

The path strides are given by

$$\vec{S} = \begin{pmatrix} 1 & -7 \\ 0 & 2 \end{pmatrix}$$

which can be verified by the illustration in Figure 3.6.

In case of arbitrary parallelotopes, in particular non-rectangular parallelotopes, the derivation of the path strides might not be intuitive since the path can abandon the bounds of the iteration domain. This behavior is shown in Figure 3.7 where the parallelogram defined by

$$\mathcal{I} = \left\{ (i\ j)^{\mathrm{T}} \in \mathbb{Z}^2 \mid 0 \leq 2i + 4j < 20 \land 0 \leq 2i - 6j < 20 \right\} \tag{3.23}$$

is scanned in two different ways.

It raises the question of how the path strides can be calculated for a given parallelotope. For this purpose, we introduce the alternative concept of *loop matrices*[11] [TTZ97, DHT06c] that expresses the iteration domain as well as the scanning directions.

Definition 3.9 (Loop matrix). *A nonsingular*[12] loop matrix $R = (r_1\ r_2 \ldots\ r_n) \in \mathbb{Z}^{n \times n}$ *consists of n* loop vectors *that determine the ordering of iteration points within a parallelotope-shaped tile. Iteration points in direction of r_1 are mapped side by side. Iteration points in direction r_2 are separated by blocks of points in direction r_1 and so on. The ordering is similar to a sequential nested loop program where the loop index i_k corresponds to iteration in direction of r_k. The inner loop index is i_1 and the the outermost loop index is i_n.*

The length of the loop vectors is chosen in such a way that the iteration space $\mathcal{I}_{seq}$ of the tile is spanned.

$$\mathcal{I}_{seq} = \{ I \in \mathbb{Z}^n \mid I = R\kappa \land z \leq \kappa < o \} \qquad \kappa \in \mathbb{Q}^n,\ z = (0\ \ldots\ 0)^{\mathrm{T}} \in \mathbb{Z}^n \tag{3.24}$$
$$o = (1\ \ldots\ 1)^{\mathrm{T}} \in \mathbb{Z}^n$$

According to the theorem of Minkowski [Sch86], we call the representation in Equation (3.24) Minkowski characterization.

[11]A similar definition is also used by Agarwal and others [AKN95], which is homely named the L matrix. However, their concept defines only the tile shape but not the iteration directions.

[12]A square matrix A that is not singular. That means, A has a matrix inverse A^{-1}.

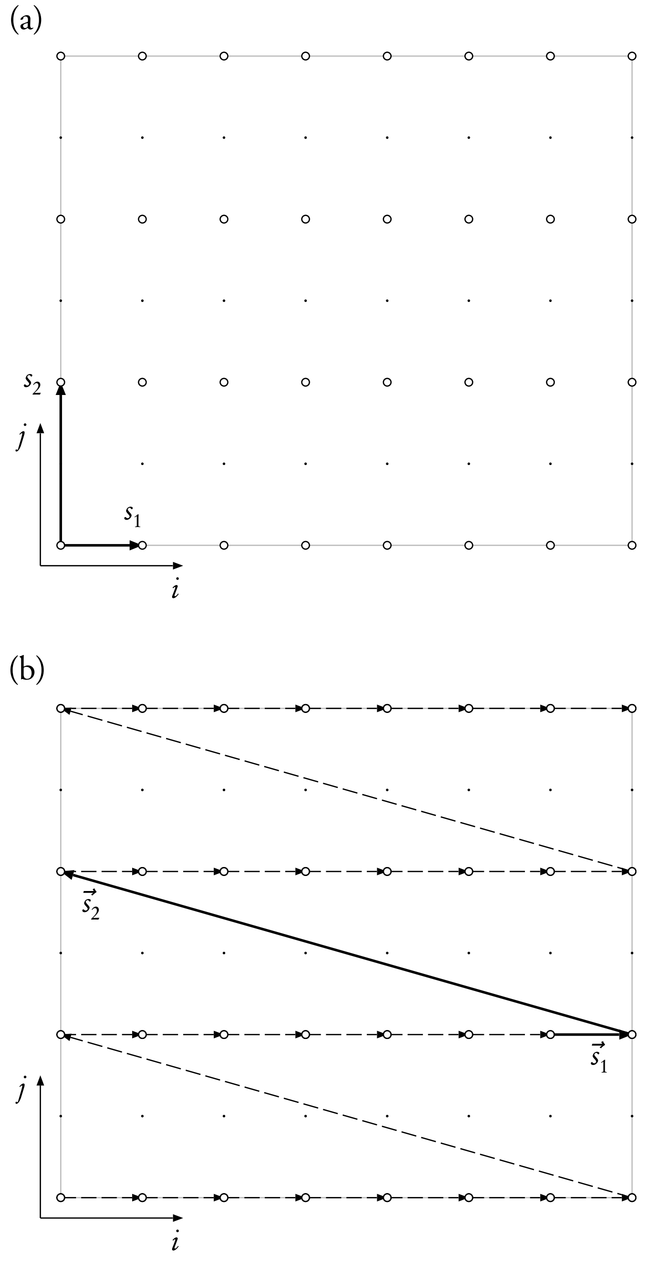


Figure 3.6: Sequential processing of a rectangular domain. In (a), the strides are depicted. (b) shows the corresponding path and the path strides (bold arrows), respectively.

The loop vectors of the parallelogram introduced in Figure 3.7(a) are given by

$$R = (r_1\ r_2) = \begin{pmatrix} 6 & 4 \\ 2 & -2 \end{pmatrix} \tag{3.25}$$

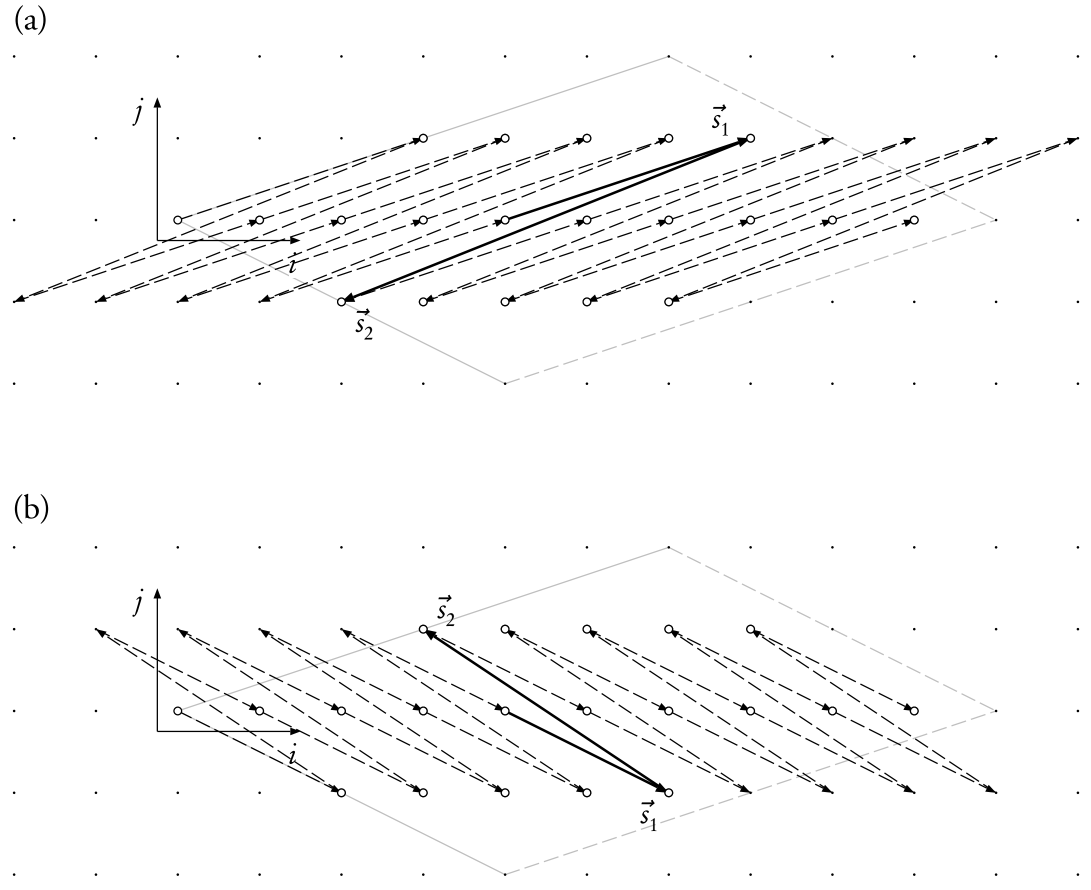
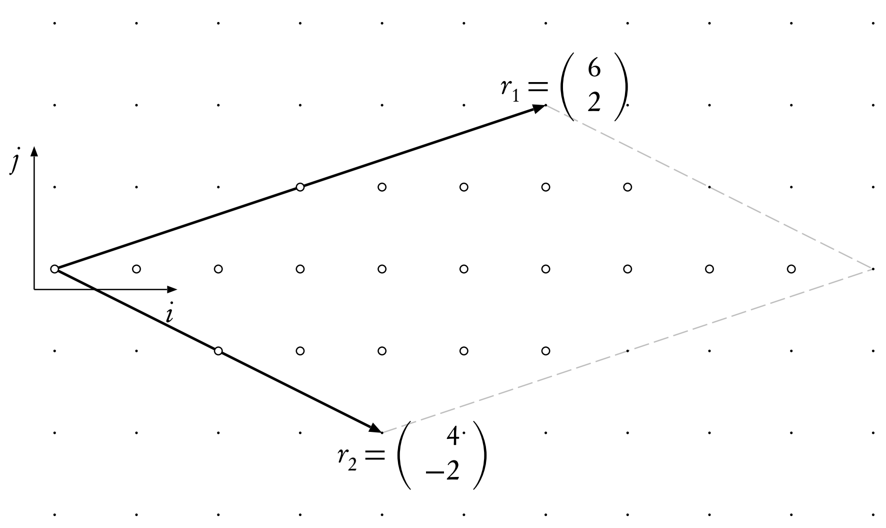


Figure 3.7: Sequential processing of an iteration space defined by a parallelogram. Two different execution orders and the path strides (bold arrows) are shown in (a) and (b), respectively. Integral points on the dashed borders of the polytope do not belong to it.

and are visualized in Figure 3.8.

Let a loop matrix R be given, then, in addition to the Minkowski characterization in Equation (3.24), the dual equivalent *implicit definition* of the parallelotope can be determined as follows.



$r_1 = \begin{pmatrix} 6 \\ 2 \end{pmatrix}$
j
i
$r_2 = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$

Figure 3.8: Tile shape with loop vectors r_1 and r_2. Integral points on the dashed borders of the polytope do not belong to it.

Theorem 3.4 (Implicit definition of a tile). *Let a tile be represented by its Minkowski characterization as defined in Equation (3.24). Then, the following implicit definition is equivalent.*

$$\begin{aligned} \mathcal{I}_{seq} &= \{I \in \mathbb{Z}^n \mid AI \geq b\} \\ &= \left\{ I \in \mathbb{Z}^n \mid \begin{pmatrix} \sigma \operatorname{adj}(R) \\ -\sigma \operatorname{adj}(R) \end{pmatrix} I \geq \begin{pmatrix} z \\ w - \sigma \det(R) o \end{pmatrix} \right\} \end{aligned} \tag{3.26}$$

where

$$\sigma = \frac{\det(R)}{|\det(R)|} \qquad z = (0 \; \ldots \; 0)^{\mathrm{T}} \in \mathbb{Z}^n$$

$$o = (1 \; \ldots \; 1)^{\mathrm{T}} \in \mathbb{Z}^n$$

$$\begin{aligned} w &= (w_1 \; w_2 \; \ldots \; w_n)^{\mathrm{T}} \in \mathbb{Z}^n \\ \text{with } w_i &= \frac{1}{g_i} \prod_{j=1}^{n} g_j \quad \forall 1 \leq i \leq n \\ \text{and } g_k &= \underset{\forall l=\{1,2,\ldots,n\}}{\gcd} (r_{l,k}) \quad \forall k \in \{i,j\} \end{aligned}$$

A variable g_k denotes the greatest common denominator of all elements of loop vector r_k.

Proof. We start with the characterization as given in Equation (3.24). Since R is nonsingular, we can write:

$$\begin{aligned}\mathcal{I}_{seq} &= \{I \in \mathbb{Z}^n \mid I = R\kappa \wedge z \leq \kappa < o\}\\ &= \left\{I \in \mathbb{Z}^n \mid z \leq R^{-1} I < o\right\}\\ &= \left\{I \in \mathbb{Z}^n \mid z \leq \frac{\operatorname{adj}(R)}{\det(R)} I < o\right\}\\ &= \{I \in \mathbb{Z}^n \mid z \leq \sigma \operatorname{adj}(R) I < \sigma \det(R) o\}\end{aligned}$$

The extraction of the greatest common denominator from each column r_i of R leads to:

$$R = (r_1\ r_2\ \dots\ r_n) = (r_1'\ r_2'\ \dots\ r_n')G \qquad \text{where } G = \begin{pmatrix} g_1 & 0 & \dots & 0\\ 0 & g_2 & \ddots & \vdots\\ \vdots & \ddots & \ddots & 0\\ 0 & \dots & 0 & g_n\end{pmatrix}$$

$$\begin{aligned}\operatorname{adj}(R) &= \operatorname{adj}\left((r_1'\ r_2'\ \dots\ r_n')G\right)\\ &= \operatorname{adj}(G)\operatorname{adj}\left((r_1'\ r_2'\ \dots\ r_n')\right)\\ &= \det(G)G^{-1}\operatorname{adj}\left((r_1'\ r_2'\ \dots\ r_n')\right)\\ &= \left(\prod_{i=1}^{n} g_i\right)\begin{pmatrix} 1/g_1 & 0 & \dots & 0\\ 0 & 1/g_2 & \ddots & \vdots\\ \vdots & \ddots & \ddots & 0\\ 0 & \dots & 0 & 1/g_n\end{pmatrix}\operatorname{adj}\left((r_1'\ r_2'\ \dots\ r_n')\right)\\ &= \begin{pmatrix} w_1 & 0 & \dots & 0\\ 0 & w_2 & \ddots & \vdots\\ \vdots & \ddots & \ddots & 0\\ 0 & \dots & 0 & w_n\end{pmatrix}\operatorname{adj}\left((r_1'\ r_2'\ \dots\ r_n')\right)\end{aligned}$$

$\Rightarrow$

$$\mathcal{I}_{seq} = \left\{ I \in \mathbb{Z}^n \mid z \leq \sigma \begin{pmatrix} w_1 & 0 & \dots & 0\\ 0 & w_2 & \ddots & \vdots\\ \vdots & \ddots & \ddots & 0\\ 0 & \dots & 0 & w_n\end{pmatrix}\operatorname{adj}\left((r_1'\ r_2'\ \dots\ r_n')\right) I < \sigma \det(R) o \right\}$$

Since $\det(R)$ is divisible by w_i without remainder as the following intermediate step proves, $\det(R) = \det\left((r'_1\ r'_2\ \dots\ r'_n)\right)\det(G) = \det\left((r'_1\ r'_2\ \cdots\ r'_n)\right)\left(\prod_{j=1}^{n} g_j\right) = \det\left((r'_1\ r'_2\ \cdots\ r'_n)\right) w_i g_i$ for all $1 \le i \le n$, we can finally write:

$$\begin{aligned}\mathcal{I}_{seq} &= \left\{ I \in \mathbb{Z}^n \mid z \le \sigma \mathrm{adj}(R) I \le \begin{pmatrix} \sigma \det(R) - w_1 \\ \sigma \det(R) - w_2 \\ \vdots \\ \sigma \det(R) - w_n \end{pmatrix} \right\} \\ &= \left\{ I \in \mathbb{Z}^n \mid \begin{pmatrix} \sigma \mathrm{adj}(R) \\ -\sigma \mathrm{adj}(R) \end{pmatrix} I \ge \begin{pmatrix} z \\ w - \sigma \det(R) o \end{pmatrix} \right\} \end{aligned} \qquad \square$$

One remark, the authors in [TT93, DHT06c] propose a construction method similar to the method presented in Theorem 3.4 but with a crucial difference: They use $\sigma \mathrm{adj}(R) I \le (\sigma \det(R) - 1) o$ as upper bounds of the polytope, whereas we use $\sigma \mathrm{adj}(R) I \le \sigma \det(R) o - w$. That means, instead of subtracting 1 from each right hand side of the inequalities, we subtract w_i. This novel construction method leads to considerably tighter bounds of the polytope, avoids rational weighted half spaces, and is essential in the course of the thesis.

Coming back to the initial question how can the path strides be obtained? In summary, we propose the following approach.

1. The tile space $\mathcal{I}_{seq}$, given by an n-dimensional loop matrix R, will be transformed to an integral space that is spanned by *n canonical vectors*[13]. In the following, we call this space also *orthogonal*.
2. Determine the path strides in the orthogonal space.
3. Transform the path strides back to the original space.

The consideration of the orthogonal space has the advantage that therein scanning can be performed in a monotonically increasing manner with respect to the lexicographic order.

Before we systematically describe the proposed approach, we demonstrate the procedure by means of our running example (the parallelogram in Equation (3.23)). We assume the following transformation is appropriate for the first step of our proposed approach.

$$\mathcal{I}_{seq} = \{ I \in \mathbb{Z}^n \mid AI \ge b \} \qquad \Rightarrow \qquad \mathcal{I}'_{seq} = \{ I' \in \mathbb{Z}^n \mid AT^{-1} I' \ge b \}$$

[13] A canonical vector $e = (e_1\ \dots\ e_i\ \dots\ e_n)^{\mathrm{T}} \in \mathbb{Z}^n$, $1 \le i \le n$ is a vector with a one in the i^{th} coordinate and zero elsewhere.

where

$$T = \begin{pmatrix} 1 & 2 \\ 1 & -3 \end{pmatrix} \qquad T^{-1} = \frac{1}{5}\begin{pmatrix} 3 & 2 \\ 1 & -1 \end{pmatrix} \tag{3.27}$$

The transformation to the orthogonal domain defined by i', j' is done using the transformation matrix T, $(i'\ j')^{\mathrm{T}} = T(i\ j)^{\mathrm{T}}$. The transformed parallelogram is defined by

$$\begin{aligned}
\mathcal{I}'_{seq} &= \left\{ \begin{pmatrix} i' \\ j' \end{pmatrix} \in \mathbb{Z}^2 \mid \begin{pmatrix} 2 & 4 \\ 2 & -6 \\ -2 & -4 \\ -2 & 6 \end{pmatrix} \begin{pmatrix} \frac{3}{5} & \frac{2}{5} \\ \frac{1}{5} & \frac{-1}{5} \end{pmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ -19 \\ -19 \end{pmatrix} \right\} \\
&= \left\{ \begin{pmatrix} i' \\ j' \end{pmatrix} \in \mathbb{Z}^2 \mid \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -2 & 0 \\ 0 & -2 \end{pmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ -19 \\ -19 \end{pmatrix} \right\} \\
&= \left\{ \begin{pmatrix} i' \\ j' \end{pmatrix} \in \mathbb{Z}^2 \mid \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ -9 \\ -9 \end{pmatrix} \right\}
\end{aligned}$$

The transformation of the iteration space is depicted in Figure 3.9. The gray iteration points in Figure 3.9(b) have no images in the original, which can be easily verified from $(i\ j)^{\mathrm{T}} = T^{-1}(i'\ j')^{\mathrm{T}}$ and the fact that the matrix T^{-1} has rational elements. Hence, the gray iteration points have rational images in the original domain. These points are also known as *holes* [Ram95, TKD02, Bas03, Bas04, TKD05]. The problem of scanning the transformed domain is to avoid the holes by using a suitable step size (stride) for each iteration variable and adapting the bounds of the loop nest. In many papers, the authors propose to solve the problem of avoiding holes by using the *Hermite normal form* [Sch86]. For instance, Li and Pingali [LP92, LP94], Xue [Xue94], Ramanujam [Ram92, Ram95], and Fernández and others [FLV95] discuss such a method for non-unimodular loop transformations. Darte and Robert [DR94b] apply the Hermite normal form when using projection as processor allocation. Other authors employ the form for code generation of partitioned iteration spaces [GAK03], as well as for efficient control generation [Bas03, Bas04, VBC06].

In linear algebra, the Hermite normal form is a triangular matrix H that is derived when transforming a given matrix A by a *unimodular*[14] matrix U. In addition, a certain set of conditions on the elements of H is specified, which makes H,

[14]A real square matrix $A \in \mathbb{R}^{n\times n}$ is said to be *unimodular* if its determinant $\det(A) = \pm 1$.

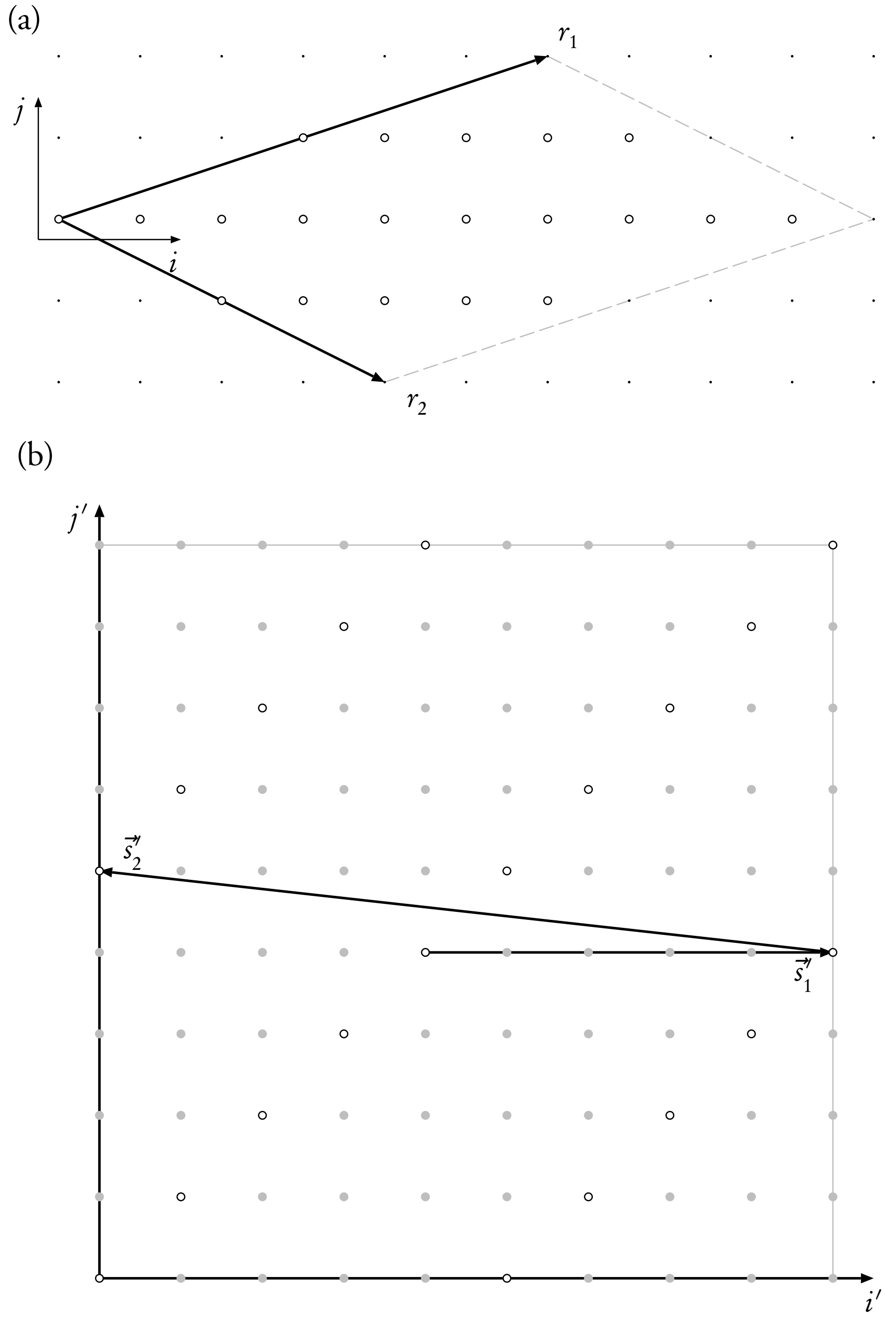


Figure 3.9: In (a), the original tile and in (b) the transformed domain is shown. The white iteration points denote the original and transformed integral points, respectively.

and therefore also U, unique. The transformation goes back to the mathematician

Charles Hermite [Her51]. There exist several slightly different definitions of the Hermite normal form, for instance, if H is an upper right triangular or a lower left triangular matrix (see for instance Domich et al. [DKT87]). In the following, we use as definition an upper right triangular matrix for the Hermite normal form H.

Definition 3.10 (Hermite normal form). *Given a square nonsingular integer matrix $A \in \mathbb{Z}^{n\times n}$, there exists an unimodular matrix $U \in \mathbb{Z}^{n\times n}$ and a matrix $H \in \mathbb{Z}^{n\times n}$—known as the* Hermite normal form (HNF) *of A—such that*

$$H = AU$$

The entries of H satisfy:

1. *H is a upper right triangular, that is, $h_{i,j} = 0$ for all $i > j$,*
2. *$h_{i,i} > 0$ for all i, and*
3. *$h_{i,i} > h_{i,j} \geq 0$ for all $i < j$.*

The right multiplication of A by an unimodular matrix U corresponds to a sequence of the following *elementary column operations*.

1. Interchange two columns.
2. Multiply a column by -1.
3. Add an integral multiple of one column to another.

The Hermite normal form can be found in polynomial time by using a sequence of the above defined elementary column operations [Sch86].

Application of the Hermite normal form to our example or rather, to the transformation matrix T, results in the following decomposition.

$$H = TU$$

$$H = \begin{pmatrix} 5 & 1 \\ 0 & 1 \end{pmatrix} \qquad T = \begin{pmatrix} 1 & 2 \\ 1 & -3 \end{pmatrix} \qquad U = \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix}$$

Since the columns of T and H generate the same lattice, the diagonal elements of matrix H directly represent the strides of i' and j'. Namely, 5 in direction of i' and unit stride in direction of j'.

$$S' = \begin{pmatrix} s'_1 & s'_2 \end{pmatrix} = \begin{pmatrix} 5 & 0 \\ 0 & 1 \end{pmatrix}$$

The corresponding path strides are $\vec{s}_1' = s_1'$ and $\vec{s}_2' = (-9\ 0)^{\mathrm{T}} + s_2'$, which can be verified in Figure 3.9.

$$\vec{S}' = \begin{pmatrix} \vec{s}_1' & \vec{s}_2' \end{pmatrix} = \begin{pmatrix} 5 & -9 \\ 0 & 1 \end{pmatrix}$$

Subsequently, the sought after path strides $\vec{S}$ of the original parallelogram can be obtained by inverse transformation (cf. Figure3.7(a)).

$$\vec{S} = T^{-1}\vec{S}' = \begin{pmatrix} \frac{3}{5} & \frac{2}{5} \\ \frac{1}{5} & -\frac{1}{5} \end{pmatrix} \begin{pmatrix} 5 & -9 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 3 & -5 \\ 1 & -2 \end{pmatrix}$$

Before we formally present a theorem (Theorem 3.5) and prove how to transform a given parallelotope to an equivalent *orthotope*[15], we formulate two general lemmas.

Lemma 3.1 (Determinant property). *Let a matrix $A \in \mathbb{R}^{n\times n}$ and a scalar $c \in \mathbb{R}$ be given. Then, the property* $\det(cA) = c^n \det(A)$ *holds.*

Proof. In general, it is easy to see that $\det(cE) = c^n$, where E is the $n \times n$ identity matrix. From this follows that $\det(cA) = \det(cEA) = \det(cE)\det(A) = c^n \det(A)$ which concludes the proof. □

Lemma 3.2 (Adjugate/determinant relationship). *Let a matrix $A \in \mathbb{R}^{n\times n}$ be given. The adjugate of A has the property:*

$$\det(\mathrm{adj}(A)) = \det(A)^{n-1}$$

Proof. Since $A\mathrm{adj}(A) = \det(A)E$ [Str03], then taking the determinant of both sides and consideration of E is an $n \times n$ matrix, we have

$$\det(A)\det(\mathrm{adj}(A)) = \det(A)^n$$

If $\det(A) \neq 0$, then the relation $\det(\mathrm{adj}(A)) = \det(A)^{n-1}$ follows at once. If $\det(A) = 0$, then A and its adjugate matrix are singular, and consequently also $\det(\mathrm{adj}(A)) = 0$. Thus $\det(\mathrm{adj}(A)) = \det(A)^{n-1} = 0$. □

Theorem 3.5 (Orthogonal transformation). *Let a loop matrix $R \in \mathbb{Z}^{n\times n}$, which is equivalently represented by a parallelotope $\mathcal{I}_{seq} = \{I \in \mathbb{Z}^n \mid AI \geq b\}$, be given. If a matrix $T \in \mathbb{Z}^{n\times n}$ is chosen to be*

$$T = \sigma W^{-1}\mathrm{adj}(R) \qquad \text{with } W = \begin{pmatrix} w_1 & 0 & \dots & 0 \\ 0 & w_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & w_n \end{pmatrix} \tag{3.28}$$

[15]An *orthotope* is a parallelotope whose edges are all mutually orthogonal to each other. The orthotope is a generalization of the rectangle and rectangular parallelepiped.

where σ and w_i, $1 \leq i \leq n$ are defined as in Theorem 3.4, then the polytope $\mathcal{I}'_{seq} = \{I' \in \mathbb{Z}^n \mid AT^{-1}I' \geq b\}$ is an orthogonal transformation *of $\mathcal{I}_{seq}$. That is, the basis of $\mathcal{I}'_{seq}$ is spanned by n weighted canonical vectors.*

Proof. The transformation of Equation (3.28) into $WT = \sigma \operatorname{adj}(R)$ and its insertion in the construction rule leads to

$$
\begin{aligned}
\mathcal{I}_{seq} &= \left\{ I \in \mathbb{Z}^n \mid \begin{pmatrix} WT \\ -WT \end{pmatrix} I \geq \begin{pmatrix} z \\ w - \sigma \det(R) o \end{pmatrix} \right\} \\
\Rightarrow \quad \mathcal{I}'_{seq} &= \left\{ I' \in \mathbb{Z}^n \mid \begin{pmatrix} WTT^{-1} \\ -WTT^{-1} \end{pmatrix} I' \geq \begin{pmatrix} z \\ w - \sigma \det(R) o \end{pmatrix} \right\} \\
&= \left\{ I' \in \mathbb{Z}^n \mid \underbrace{\begin{pmatrix} E \\ -W \end{pmatrix}}_{A'} I' \geq \begin{pmatrix} z \\ w - \sigma \det(R) o \end{pmatrix} \right\}
\end{aligned}
$$

Hence, it is shown that the basis of $\mathcal{I}'_{seq}$ is spanned only by orthogonal vectors since matrix A' has only canonical row vectors weighted by a constant.

It remains to show that the *volume*[16] is related to the original parallelotope $\mathcal{I}_{seq}$. Since $\mathcal{I}_{seq}$ is *dense*, its volume $V(\mathcal{I}_{seq})$ equals $|\det(R)|$. The number of *valid* integral points (points that are no holes) is given by transforming $\mathcal{I}'_{seq}$ by the nonsingular stride matrix $S' \in \mathbb{Z}^{n \times n}$.

$$
S' = \begin{pmatrix} h_{1,1} & 0 & \dots & 0 \\ 0 & h_{2,2} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & h_{n,n} \end{pmatrix} \tag{3.29}
$$

where $h_{i,i}$, $1 \leq i \leq n$ are the diagonal elements of the Hermite normal form of the transformation matrix T ($H = TU$). From this follows a dense orthogonal iteration space $\mathcal{I}''_{seq}$.

$$
\mathcal{I}''_{seq} = \left\{ I'' \in \mathbb{Z}^n \mid \begin{pmatrix} E \\ -WS' \end{pmatrix} I'' \geq \begin{pmatrix} z \\ w - \sigma \det(R) o \end{pmatrix} \right\}
$$

$\mathcal{I}''_{seq}$ is an orthotope, thus its volume can be calculated as follows.

$$
V(\mathcal{I}''_{seq}) = \left\lfloor \frac{\sigma \det(R) - w_1}{s'_{1,1} w_1} + 1 \right\rfloor \cdot \left\lfloor \frac{\sigma \det(R) - w_2}{s'_{2,2} w_2} + 1 \right\rfloor \cdot \ldots \cdot \left\lfloor \frac{\sigma \det(R) - w_n}{s'_{n,n} w_n} + 1 \right\rfloor
$$

[16]The *volume* of an n-dimensional polytope $\mathcal{I} \subset \mathbb{Z}^n$ is the number of all integral points inside $\mathcal{I}$.

$\det(R)$ is divisible by $s'_{i,i}w_i$, $1 \le i \le n$ without remainder, thus we can write

$$V(\mathcal{I}''_{seq}) = \left(\frac{\sigma \det(R)}{s'_{1,1}w_1} + \left\lfloor 1 - \frac{1}{s'_{1,1}} \right\rfloor\right) \cdot \ldots \cdot \left(\frac{\sigma \det(R)}{s'_{n,n}w_n} + \left\lfloor 1 - \frac{1}{s'_{n,n}} \right\rfloor\right)$$

The *floor functions*[17] can be removed by taking into account that $s'_{i,i} > 0$, $1 \le i \le n$.

$$\begin{aligned} V(\mathcal{I}''_{seq}) &= \frac{\sigma \det(R)}{s'_{1,1}w_1} \cdot \frac{\sigma \det(R)}{s'_{2,2}w_2} \cdot \ldots \cdot \frac{\sigma \det(R)}{s'_{n,n}w_n} \\ &= (\sigma \det(R))^n \prod_{i=1}^{n} \frac{1}{s'_{i,i}w_i} = \sigma^n \frac{\det(R)^n}{\det(H)\det(W)} \end{aligned}$$

We have to show the equivalence of the volumes of $\mathcal{I}_{seq}$ and $\mathcal{I}''_{seq}$:

$$\begin{aligned} && V(\mathcal{I}_{seq}) &= V(\mathcal{I}''_{seq}) \\ \Leftrightarrow && |\det(R)| &= \sigma^n \frac{\det(R)^n}{\det(H)\det(W)} \\ \Leftrightarrow && \sigma \det(R) &= \sigma^n \frac{\det(R)^n}{\det(H)\det(W)} \\ \Leftrightarrow && \sigma \det(TU)\det(W) &= \sigma^n \det(R)^{n-1} \end{aligned}$$

By application of Equation (3.28), we obtain:

$$\sigma \det\left(\sigma W^{-1}\mathrm{adj}(R)U\right)\det(W) = \sigma^n \det(R)^{n-1}$$

Lemma 3.1 leads to:

$$\begin{aligned} && \sigma\sigma^n \det(\mathrm{adj}(R))\det(U) &= \sigma^n \det(R)^{n-1} \\ \Leftrightarrow && \det(\mathrm{adj}(R)) &= \det(R)^{n-1} \end{aligned}$$

Finally, we conclude the proof by Lemma 3.2. □

The application of Theorem 3.5 to the parallelogram defined in Equation (3.23) with the loop matrix given in Equation (3.25) results in

$$T = \begin{pmatrix} 1 & 2 \\ 1 & -3 \end{pmatrix} \qquad T^{-1} = \frac{1}{5}\begin{pmatrix} 3 & 2 \\ 1 & -1 \end{pmatrix}$$

which is the same transformation matrix as in Equation (3.27) on page 86. Recapitulate the transformed iteration space: $\mathcal{I}'_{seq} = \left\{(i'\ j')^{\mathrm{T}} \in \mathbb{Z}^2 \mid 0 \le i', j' \le 9\right\}$

Once the orthogonal space $\mathcal{I}'_{seq}$ has been derived, the path strides of it and the original space can be determined by the following construction rule.

[17] The *floor function* is defined by $\lfloor x \rfloor = \max\{z \in \mathbb{Z} \mid z \le x\}$.

Corollary 3.1 (Path strides). *From Theorem 3.5 and its proof, it follows how the path strides $\vec{S}$ can be obtained:*

$$\vec{S} = T^{-1}\vec{S'} = T^{-1}\left(S' + \begin{pmatrix} 0 & \bar{h}_{1,2} & \dots & \bar{h}_{1,n} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \bar{h}_{n-1,n} \\ 0 & \dots & \dots & 0 \end{pmatrix} \right) \tag{3.30}$$

The coefficients $\bar{h}_{i,j}$ for all $1 \leq i \leq n-1$ and $i < j \leq n$ are calculated as follows.

$$\bar{h}_{i,j} = h_{i,j} - \left\lfloor \frac{h_{i,j} + l_i}{h_{i,i}} \right\rfloor h_{i,i}$$

Where $l_i = \frac{\sigma \det(R) - w_i}{w_i}$ denotes the side length of the orthotope $\mathcal{I}'_{seq}$ in direction i.

Proof. Considering, for instance, Figure 3.9(b), the idea of determination of the path strides is very intuitive. By Definition 3.10, the Hermite normal form H has only non-negative elements. Therefore, at first, several steps (say x steps) of length s'_1 in the first direction are made until the bound of the orthotope is reached. Then, it is necessary to *jump back* these steps and make one step of length s'_2 in the second direction, once again x steps in the first direction, jump back, and so on. Hence, to scan all valid iteration points (lattice points) in one direction i, $x = l_i / s'_i$ steps in that direction and a jump back by $-xs'_i$ have to be made. Let the length of the jump back be $-b_i = \| - xs'_i \|$. Consequently, it follows that $-b_i \leq l_i$ and $b_i = h_{i,i}x$. In order to *catch* all valid iteration points, $l_i + b_i$ has to be minimized, which leads to $\left\lfloor \frac{h_{i,j}+l_i}{h_{i,i}} \right\rfloor h_{i,i}$. Nota bene, to obtain appropriate path strides, they have to be a linear combination of the lattice defined by H. Thus, taking the position i, j in matrix H into account, we obtain $h_{i,j} - \left\lfloor \frac{h_{i,j}+l_i}{h_{i,i}} \right\rfloor h_{i,i}$. □

Our proposed method for the computation of the path strides is most likely to be associated with the work for scanning polyhedra without do-loops by Boulet and Feautrier [BF98]. However, the two approaches are totally different, since our method is entirely based on linear algebra, whereas their method heavily relies on parametric integer programming [Fea88].

Once the path strides are calculated, the sequentialization constraints of the mixed integer program can be formulated.

3.5.2.2 Sequentialization Constraint

We have to require that the operations corresponding to a node v_i are executed sequentially within a tile $\mathcal{I}_{seq}$.

$$\text{(3.31)} \qquad \Lambda_{seq}(I_1 - I_2) \neq 0 \qquad\qquad \forall I_1, I_2 \in \mathcal{I}_{seq} \wedge I_1 \neq I_2$$

Since the condition in Equation (3.31) is a quadratic term, it cannot be directly incorporated into a MIP formulation, but has to be linearized. In case that $\mathcal{I}_{seq}$ is a parallelotope, the condition in Equation (3.31) is satisfied by the following theorem.

Theorem 3.6 (Sequentialization constraint)**.** *Let a loop matrix R be given, which denotes the shape and the execution order of a parallelotope-shaped tile. Then, the following constraints ensure that all iteration points within the tile are executed at different time steps.*

$$\Lambda_{seq}\vec{S} \geq o$$

where Λ_{seq} denotes the schedule vector (sequential execution order) of the tile, $\vec{S}$ the matrix of the path strides (cf. Corollary 3.1) according to the given loop matrix R, and $o = (1 \ldots 1) \in \mathbb{Z}^{1\times n}$.

Proof. In terms of execution order, let an iteration point I_2 be the direct successor of an iteration point I_1, where $\vec{s}_i = I_2 - I_1$. That is, I_1 must be executed before I_2, which means $\Lambda_{seq}(I_2 - I_1) = \Lambda_{seq}\vec{s}_i > 0$. This condition has to be satisfied for all path strides $\vec{s}_i \in \vec{S}$. □

3.5.2.3 Mixed Integer Program for LSGP and LPGS Partitioned Algorithms

Consider a given DPRA, which is locally sequential, globally parallel partitioned such that its iteration space $\mathcal{I}$ is decomposed into $\mathcal{I} \subseteq \mathcal{I}_{LS} \oplus \mathcal{I}_{par}$. The locally sequential part of the iteration space is denoted by $\mathcal{I}_{LS} \subset \mathbb{Z}^n$ and a corresponding loop matrix R_{LS}. The parallel part of the iteration space is denoted by $\mathcal{I}_{par} \subset \mathbb{Z}^n$. A scheduling function for a node v_i according to the decomposition is given by

$$t_i(I_{LS}, I_{par}) = (\Lambda_{LS}\ \Lambda_{par}) \begin{pmatrix} I_{LS} \\ I_{par} \end{pmatrix} + \tau(v_i)$$

Furthermore, let the iteration interval P be given without loss of generality[18]. Since a periodic execution within one tile with a corresponding period P is required, we have to formulate the following constraint.

$$\Lambda_{seq} = P\Lambda'_{seq} \qquad\qquad \Lambda'_{seq} \in \mathbb{Z}^{1\times n}$$

[18]Note, this is no restriction since the same concepts as discussed in Section 3.5.1.1 can be applied.

Then, a latency optimal solution for affine scheduling, which takes LSGP into account, can be obtained by solving the following MIP.

Affine scheduling with LSGP partitioning as allocation

Input:

- Reduced dependence graph $G = (V, E, D)$
- Execution time $w_i \in W$ of each node $v_i \in V$, $W : V \rightarrow \mathbb{Z}$
- Iteration space $\mathcal{I} \subseteq \mathcal{I}_{LS} \oplus \mathcal{I}_{par}$ defined by
  - $\mathcal{I}_{LS} = \{I \in \mathbb{Z}^n \mid A_{LS} I \geq b_{LS}\}$ where $A_{LS} \in \mathbb{Z}^{2n \times n}$
  - $\mathcal{I}_{par} = \{I \in \mathbb{Z}^n \mid A_{par} I \geq b_{par}\}$ where $A_{par} \in \mathbb{Z}^{m \times n}$
- Loop matrix R_{LS} and a matrix $\vec{S}_{LS}$ with the corresponding path strides
- Iteration interval P

Output:

- Schedule vector $(\Lambda_{LS}\ \Lambda_{par}) \in \mathbb{Z}^{1 \times 2n}$, start times $\tau(v_i)$ of all nodes $v_i \in V$

Program:

$$
\begin{array}{lrll}
\text{min} & & -(y_1 + y_2)b_{LS} - (y_3 + y_4)b_{par} & \\
\text{subject to} & & y_1 A_{LS} = \Lambda_{LS} & y_1 \in \mathbb{Q}^{1 \times 2n} \\
 & & y_2 A_{LS} = -\Lambda_{LS} & y_2 \in \mathbb{Q}^{1 \times 2n} \\
 & & y_3 A_{par} = \Lambda_{par} & y_3 \in \mathbb{Q}^{1 \times m} \\
 & & y_4 A_{par} = -\Lambda_{par} & y_4 \in \mathbb{Q}^{1 \times m} \\
 & & y_1, y_2, y_3, y_4 \geq 0 & \\
(\Lambda_{LS}\ \Lambda_{par})d_e & + & \tau(v_j) - \tau(v_i) \geq w_i & \forall\, e = (v_i, v_j) \in E \\
 & & \Lambda_{LS} = P\Lambda'_{LS} & \Lambda'_{LS} \in \mathbb{Z}^{1 \times n} \\
 & & \Lambda_{LS}\vec{S}_{LS} \geq o & o = (1\ \dots\ 1) \in \mathbb{Z}^{1 \times n}
\end{array}
$$

Similar to LSGP, we can consider an algorithm, which is locally parallel, globally sequential partitioned such that its iteration space $\mathcal{I}$ is decomposed into $\mathcal{I} \subseteq \mathcal{I}_{GS} \oplus \mathcal{I}_{par}$. The globally sequential part of the iteration space is denoted by $\mathcal{I}_{GS} \subset \mathbb{Z}^n$ and the corresponding loop matrix R_{GS}, whereas the parallel part is denoted by $\mathcal{I}_{par} \subset \mathbb{Z}^n$. A scheduling function for a node v_i according to the decomposition is given by

$$t_i(I_{GS}, I_{par}) = (\Lambda_{GS}\ \Lambda_{par}) \begin{pmatrix} I_{GS} \\ I_{par} \end{pmatrix} + \tau(v_i)$$

A latency optimal solution for affine scheduling, which takes LPGS into account, is given by the solution of the following MIP.

Affine scheduling with LPGS partitioning as allocation

Input:

- Reduced dependence graph $G = (V, E, D)$
- Execution time $w_i \in W$ of each node $v_i \in V$, $W : V \to \mathbb{Z}$
- Iteration space $\mathcal{I} \subseteq \mathcal{I}_{GS} \oplus \mathcal{I}_{par}$ defined by
  - $\mathcal{I}_{GS} = \{I \in \mathbb{Z}^n \mid A_{GS} I \geq b_{GS}\}$ where $A_{GS} \in \mathbb{Z}^{2n \times n}$
  - $\mathcal{I}_{par} = \{I \in \mathbb{Z}^n \mid A_{par} I \geq b_{par}\}$ where $A_{par} \in \mathbb{Z}^{m \times n}$
- Loop matrix R_{GS} and a matrix $\vec{S}_{GS}$ with the corresponding path strides
- Iteration interval P

Output:

- Schedule vector $(\Lambda_{GS}\ \Lambda_{par}) \in \mathbb{Z}^{1 \times 2n}$, start times $\tau(v_i)$ of all nodes $v_i \in V$

Program:

$$\begin{array}{lrl}
\text{min} & -(y_1 + y_2) b_{GS} - (y_3 + y_4) b_{par} & \\
\text{subject to} & y_1 A_{GS} = \Lambda_{GS} & y_1 \in \mathbb{Q}^{1 \times 2n} \\
& y_2 A_{GS} = -\Lambda_{GS} & y_2 \in \mathbb{Q}^{1 \times 2n} \\
& y_3 A_{par} = \Lambda_{par} & y_3 \in \mathbb{Q}^{1 \times m} \\
& y_4 A_{par} = -\Lambda_{par} & y_4 \in \mathbb{Q}^{1 \times m} \\
& y_1, y_2, y_3, y_4 \geq 0 & \\
& (\Lambda_{GS}\ \Lambda_{par}) d_e + \tau(v_j) - \tau(v_i) \geq w_i & \forall\, e = (v_i, v_j) \in E \\
& \Lambda_{GS} = P \Lambda'_{GS} & \Lambda'_{GS} \in \mathbb{Z}^{1 \times n} \\
& \Lambda_{GS} \vec{S}_{GS} \geq o & o = (1\ \ldots\ 1) \in \mathbb{Z}^{1 \times n}
\end{array}$$

3.5.2.4 Mixed Integer Program for Hierarchically Partitioned Algorithms

As described already in Section 3.3.3, an algorithm which is co-partitioned is decomposed into $\mathcal{I} \subseteq \mathcal{I}_{LS} \oplus \mathcal{I}_{par} \oplus \mathcal{I}_{GS}$. Where, the elements in $\mathcal{I}_{LS} \subset \mathbb{Z}^n$ denote the iteration points within an LS tile. The elements in $\mathcal{I}_{par} \subset \mathbb{Z}^n$ and $\mathcal{I}_{GS} \subset \mathbb{Z}^n$ are the origins of the LS and GS tiles, respectively. A scheduling function for a node v_i according to the decomposition is given by

$$t_i(I_{LS}, I_{par}, I_{GS}) = (\Lambda_{LS}\ \Lambda_{par}\ \Lambda_{GS}) \begin{pmatrix} I_{LS} \\ I_{par} \\ I_{GS} \end{pmatrix} + \tau(v_i)$$

The execution order of the locally sequential and the globally sequential tile are defined by the loop matrices R_{LS} and R_{GS}. Respectively, a latency optimal solution for affine scheduling taking co-partitioning into account can be obtained by solving the following MIP.

Affine scheduling with co-partitioning as allocation

Input:

- Reduced dependence graph $G = (V, E, D)$
- Execution time $w_i \in W$ of each node $v_i \in V$, $W : V \to \mathbb{Z}$
- Iteration space $\mathcal{I} \subseteq \mathcal{I}_{LS} \oplus \mathcal{I}_{par} \oplus \mathcal{I}_{GS}$ defined by
  - $\mathcal{I}_{LS} = \{I \in \mathbb{Z}^n \mid A_{LS} I \geq b_{LS}\}$ where $A_{LS} \in \mathbb{Z}^{2n \times n}$
  - $\mathcal{I}_{par} = \{I \in \mathbb{Z}^n \mid A_{par} I \geq b_{par}\}$ where $A_{par} \in \mathbb{Z}^{m \times n}$
  - $\mathcal{I}_{GS} = \{I \in \mathbb{Z}^n \mid A_{GS} I \geq b_{GS}\}$ where $A_{GS} \in \mathbb{Z}^{2n \times n}$
- Loop matrices R_{LS}, R_{GS}, and the corresponding path strides $\vec{S}_{LS}$ and $\vec{S}_{GS}$
- Iteration interval P

Output:

- Schedule vector $(\Lambda_{LS}\ \Lambda_{par}\ \Lambda_{GS}) \in \mathbb{Z}^{1 \times 3n}$
- Start times $\tau(v_i)$ of all nodes $v_i \in V$

⇝

$\rightsquigarrow$

Program:

$$\begin{array}{rll}
\min & -(y_1+y_2)b_{LS}-(y_3+y_4)b_{par}-(y_5+y_6)b_{GS} & \\
\text{subject to} & y_1A_{LS}=\Lambda_{LS} & y_1\in\mathbb{Q}^{1\times 2n} \\
& y_2A_{LS}=-\Lambda_{LS} & y_2\in\mathbb{Q}^{1\times 2n} \\
& y_3A_{par}=\Lambda_{par} & y_3\in\mathbb{Q}^{1\times m} \\
& y_4A_{par}=-\Lambda_{par} & y_4\in\mathbb{Q}^{1\times m} \\
& y_5A_{GS}=\Lambda_{GS} & y_5\in\mathbb{Q}^{1\times 2n} \\
& y_6A_{GS}=-\Lambda_{GS} & y_6\in\mathbb{Q}^{1\times 2n} \\
& y_1,y_2,y_3,y_4,y_5,y_6\geq 0 & \\
& (\Lambda_{LS}\ \Lambda_{par}\ \Lambda_{GS})d_e\ +\ \tau(v_j)-\tau(v_i)\geq w_i & \forall\ e=(v_i,v_j)\in E \\
& \Lambda_{LS}=P\Lambda'_{LS} & \Lambda'_{LS}\in\mathbb{Z}^{1\times n} \\
& \Lambda_{LS}\vec{S}_{LS}\geq o & o=(1\ \dots\ 1)\in\mathbb{Z}^{1\times n} \\
& \Lambda_{GS}\vec{S}_{GS}\geq o &
\end{array}$$

Since the proposed MIPs for LSGP and LPGS, as well as for co-partitioning are very similar, the method can be easily extended to further hierarchically partitioning schemes, such as proposed, for instance in [ML90, EM97a, EM99, Eck01, DHT06c].

Eckhart and Merker [EM97b] present a scheduling method for co-partitioned array architectures. In contrast to our proposed approach, they can handle only partitionings defined by a diagonal matrix ($\mathrm{diag}(\vartheta_1,\dots,\vartheta_n)$), that means the tiles are in form of an orthotope.

3.5.2.5 On the Number of Sequentialization Orders

One or more loop matrices are arguments of the afore presented MIPs, that is, the sequentialization directions are given by the columns (loop vectors) of the loop matrices. Thus, in order to derive an overall latency minimal solution for all possible combinations of the loop vectors, a MIP has to be generated and solved. In Figure 3.10, all possible combinations for a 2-dimensional example are visualized.

Let us consider hierarchical partitioning with m levels, that means m loop matrices describe the partitioning scheme and sequentialization, respectively. Let each loop matrix be an $n \times n$ matrix, where n is the dimension of the iteration subspace

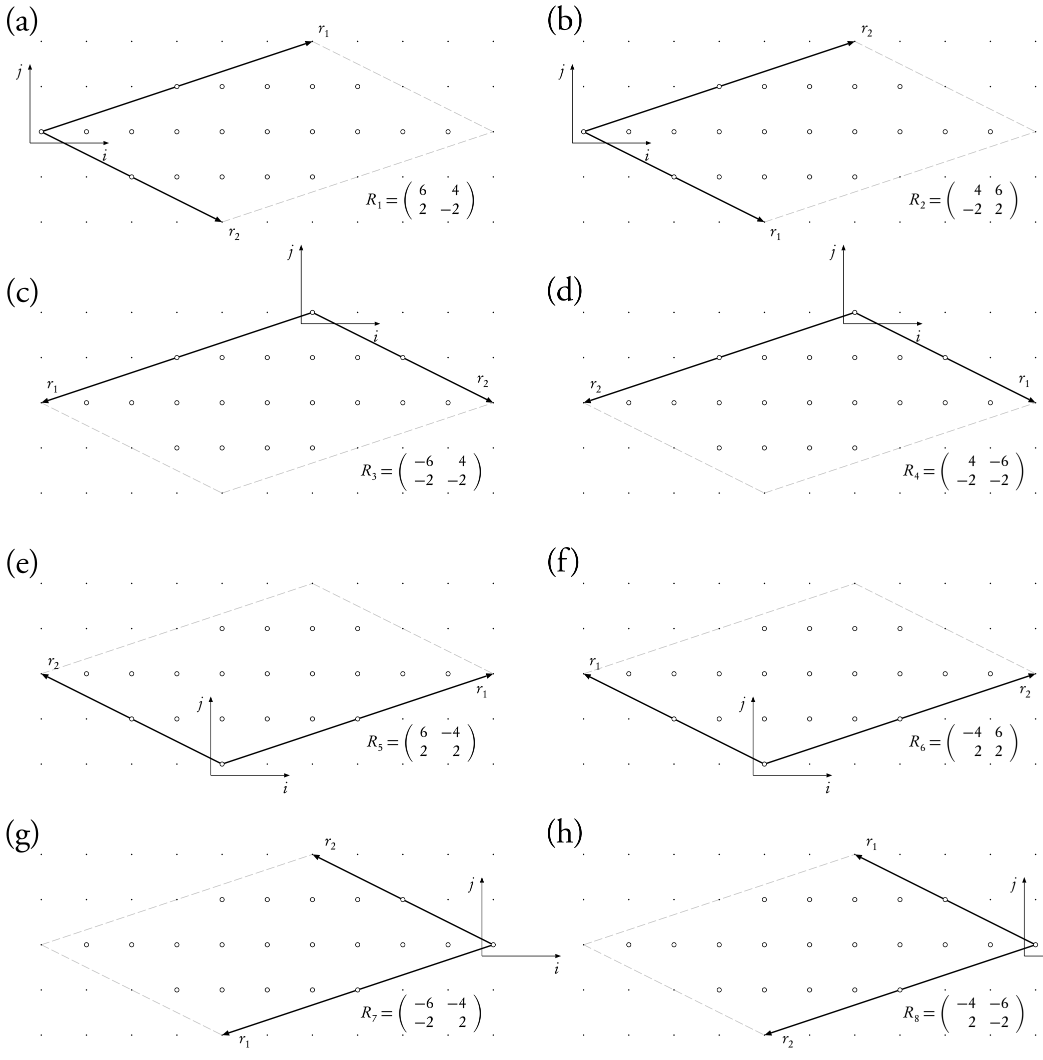

(a)
$R_1 = \begin{pmatrix} 6 & 4 \\ 2 & -2 \end{pmatrix}$
(b)
$R_2 = \begin{pmatrix} 4 & 6 \\ -2 & 2 \end{pmatrix}$
(c)
$R_3 = \begin{pmatrix} -6 & 4 \\ -2 & -2 \end{pmatrix}$
(d)
$R_4 = \begin{pmatrix} 4 & -6 \\ -2 & -2 \end{pmatrix}$
(e)
$R_5 = \begin{pmatrix} 6 & -4 \\ 2 & 2 \end{pmatrix}$
(f)
$R_6 = \begin{pmatrix} -4 & 6 \\ 2 & 2 \end{pmatrix}$
(g)
$R_7 = \begin{pmatrix} -6 & -4 \\ -2 & 2 \end{pmatrix}$
(h)
$R_8 = \begin{pmatrix} -4 & -6 \\ 2 & -2 \end{pmatrix}$
r_1
r_2
i
j

Figure 3.10: For a 2-dimensional tile in shape of a parallelogram, all of its eight different possible loop directions are visualized in (a)-(h).

(iteration space of the tile). Then the number k of loop matrix *candidates* can be computed as follows.

$$k = m^{2^n n!} \tag{3.32}$$

First let us consider the exponent $2^n n!$, where the 2^n denotes, that each loop vector might also be negated. The $n!$ is the number of permutations of the n loop vectors. Finally, when m loop matrices are given the number of combinations ($2^n n!$) is exponentiated with base m.

From the number-theoretic point of view, the growth of the function in Equation (3.32) is extraordinarily large. In practice, where tile dimensions from one to

Table 3.1: Number k of loop matrix candidates for m matrices of dimension n.

no. m of matrices	**dimension n**	**no. k of candidates**
1	1	2
1	2	8
1	3	48
1	4	384
1	5	3 840
1	6	46 080
2	1	4
2	2	256
2	3	$> 2.81 \cdot 10^{14}$
2	4	$> 3.94 \cdot 10^{115}$
3	1	9
3	2	6 561
3	3	$> 7.97 \cdot 10^{22}$
4	1	16
4	2	65 536

four are typically and one-level partitioning is considered, the growth is manageable. For two to four levels of partitioning, the use of the method is practicable up to a dimension of $n = 2$, as Table 3.1 shows.

It should not go unnoticed, that the number of candidates can be further reduced and MIPs need not be generated for all k combinations. For instance, if the considered algorithm has no data dependencies at all, the *direction* (negation of loop vectors) is irrelevant. Hence, the number of candidates becomes $k(n, m) = m^{n!}$. In case, that an algorithm has data dependencies, the following theorem must hold in order to have a *legal partitioning* [RS92].

Theorem 3.7 (Legal partitioning)**.** *Let an n-dimensional loop matrix R and the corresponding path strides $\vec{S}$ be given. In addition, l data dependencies $D = (d_1 \ \dots \ d_l) \in \mathbb{Z}^{n \times l}$ have to be satisfied. Then, R and D define a* legal partitioning *if and only if*

$$\sigma \operatorname{adj}(\vec{S}) D \leq 0 \tag{3.33}$$

where $\sigma = \det(\vec{S}) / |\det(\vec{S})|$.

Proof. We consider the *cone* [Sch86] generated by the path strides $\vec{s}_i \in \vec{S}$:

$$C = \left\{ \sum_{i=1}^{n} \theta_i \vec{s}_i \mid \theta_i \in \mathbb{R} \wedge \theta_i \geq 0 \wedge 1 \leq i \leq n \right\}$$

It can easily be seen that a data dependency d_i can be satisfied if and only if it lies inside the cone C. Similar as in the proof of Theorem 3.4, the equivalent implicit definition of the cone is given by

$$C = \left\{ x \in \mathbb{R}^n \mid \sigma \operatorname{adj}(\vec{S}) x \geq 0 \right\}$$

Since, by definition, the data dependence vectors are represented with opposite sign, we finally get the condition in Equation (3.33). □

Note that the condition in Theorem 3.7 is much more stringent than the conditions proposed in [IT88, RS92] since it considers not only the tile shape but also the scanning directions.

Consequently, Theorem 3.7 can be used as an early verification if a loop matrix is legal and thus the generation of *unnecessary*[19] MIPs may be avoided. This is especially useful if internal resources (see Section 3.5.3) are also modeled within the MIP, which could lead to a large number of variables and constraints.

3.5.3 Allocation of Resources within Processors

Until now, we have only considered a *global allocation* of iteration points to processors. This was realized by projecting or partitioning the iteration space. With regard to the fact that the data dependencies of a given algorithm were considered, not only the start times of the different iterations but also the relative start times $\tau(v_i)$ of each node v_i of the reduced dependence graph were determined. However, this method assumes that *enough* resources, such as functional units, are available to enable the derived schedule. Generally, this means that for one distinct time step an arbitrary large number of resources must be available—of course this is not feasible in practice. Hence, in the following, a number of resource constraints are presented that allow a *local allocation* within the processors. The ideas of [HHL90, Thi95] (introduction of binary variables within the MIP, cf. Section 3.1.2.3) are revamped in the next section. In summary, these concepts allow:

- Module selection, that is, having regard to different binding possibilities,
- Consideration of a fixed amount of resources for each type (allocation),
- Functional pipelining, and
- Software pipelining of multi-dimensional data flow.

Moreover, we demonstrate how this local resource constraints can be combined with the afore developed methods in Sections 3.5.1 and 3.5.2.

[19]Mixed integer programs with no solution.

3.5.3.1 Resource Constraints

In order to take resource constraints within a processor into account, an additional representation is introduced. For this purpose, integer variables that are required to be 0 or 1 are used. These variables are referred to as *binary variables* or *0-1 variables*.

Let a reduced dependence graph $G = (V, E, D)$ (see Section 2.1.4) and a resource graph $G_R = (V_R, E_R, W, \Delta)$ with $V_R = V \cup V_T$ (see Definition 3.1) be given, which represent a nested loop program as well as the resource model of a considered target architecture. Then, the scheduling of an operation $v_i \in V$ is characterized by a binary variable $x_{i,k,t}$, where $x_{i,k,t} = 1$ denotes that at relative time instance $t \in \mathbb{Z}$ the operation v_i starts its operation on resource type $r_k \in V_T$, otherwise variable $x_{i,k,t}$ equals 0. The relative start time $\tau(v_i)$ of each operation in G has a lower and an upper bound. These bounds can be obtained from schedules by either following the well known *as soon as possible* (ASAP) or *as late as possible* (ALAP) principles. Through this, for each operation $v_i \in V$ an interval of possible starting times is determined, $\tau(v_i) \in [l_i, h_i]$, where l_i denotes the earliest and h_i the latest possible starting time, respectively. Some remarks on the approach: Within the processors, unlimited resources are assumed. Also, since ASAP and ALAP are improper for scheduling cyclic graphs, all edges $e \in E$, having nonzero data dependencies ($d_e \neq 0$), are not examined for the determination of the bounds. This means that only local dependencies at one distinct iteration are considered but no loop-carried ones. The required input for the ALAP algorithm, the *latency bound* [Tei97], is set to the latest time, derived by the ASAP algorithm, plus the duration of the considered iteration interval P.

The binary variables $x_{i,k,t} \in \{0, 1\}$ encode not only the start times of the nodes but also may realize the selection from different binding possibilities, represented by the resource graph G_R. Thus, *module selection* and *binding* can be incorporated in the MIP. As each operation can be executed on one resource type only, we need to add the following *binding constraint*.

$$\sum_{\forall k \,:\, (v_i, r_k) \in E_R} \sum_{t=l_i}^{h_i} x_{i,k,t} = 1 \qquad \forall v_i \in V \tag{3.34}$$

The first sum in the constraint in Equation (3.34) denotes that a node v_i should be executed every time on the same resource type, that is, its resource type binding is static. The second sum makes sure that a node is started exactly once per iteration.

The relationship between the relative start times $\tau(v_i)$ and binary variables $x_{i,k,t}$ is represented by the following weighted sum of the binary variables.

$$\tau(v_i) = \sum_{\forall k \,:\, (v_i, r_k) \in E_R} \sum_{t=l_i}^{h_i} t \; x_{i,k,t} \qquad \forall v_i \in V \tag{3.35}$$

Since by the concept of the resource graph an operation v_i may have several binding possibilities (cf. Figure 3.1 in Section 3.3), its execution time w_i depends on the selected resource. Hence, the evaluation time $w(v_i, r_k)$, corresponding to a node $v_i \in V$, executed on a resource type $r_k \in V_T$, can be determined as follows.

$$w_i = \sum_{\forall k \,:\, (v_i, r_k) \in E_R} \sum_{t=l_i}^{h_i} w(v_i, r_k)\, x_{i,k,t} \qquad \forall v_i \in V \tag{3.36}$$

Finally, it has to be ensured that at no time instance t the number of available resources of one type is exceeded. By the inequality in Equation (3.37), it is guaranteed for all resource types r_k that, at no time step, the number of operation being executed, outruns the number $\alpha(r_k)$ of available instances. The outer summation considers all operations v_i that are potentially executed on the same resource type r_k. The middle summation takes into account, how long a resource is occupied by an operation. Here, the possible functional pipelining of functional units has to be considered. As specified by the resource graph, to each resource type r_k belongs a *pipeline rate* $\delta(r_k)$, which denotes how long the resource is occupied by an operation before the next one can be issued. Even though the execution time $w(v_i, r_k)$ of different $v_i \in V' = \left\{\forall j \mid (v_j, r_k) \in E_R\right\}$ on the same resource type r_k might be different, we assume that $\delta(r_k)$ is constant and $\delta(r_k) \leq w(v_i, r_k)$ for all $v_i \in V'$. From this, it follows that $\delta(r_k)$ becomes $w(v_i, r_k)$ if a functional unit supports no pipelining. Thus, a resource is occupied for the interval $\left[0, \delta(r_k) - 1\right]$. That is, it starts its execution at time $x_{i,k,t}$ but it also occupies the resource at time steps $x_{i,k,t-1}, \ldots, x_{i,k,t-(\delta(r_k)-1)} = \sum_{d=0}^{\delta(r_k)-1} x_{i,k,t-d}$. Since the operations' execution might cross the bounds of the iteration interval P, possibly integral multiples thereof have to be considered, νP. This is incorporated by the inner summation.

$$\sum_{\forall i \,:\, (v_i, r_k) \in E_R} \sum_{d=0}^{\delta(r_k)-1} \sum_{\forall \nu \in \mathbb{Z} \,:\, l_i \leq t-d-\nu P \leq h_i} x_{i,k,t-d-\nu P} \leq \alpha(r_k) \quad \forall r_k \in V_T \quad \forall t \,:\, 0 \leq t \leq P-1 \tag{3.37}$$

The constraint in Equation (3.37) can be formulated within a MIP only for a fixed iteration interval P. From this the same question as discussed in Section 3.5.1.1 arises if the iteration interval is given as a constant or if it should be a variable of the MIP. Whereas the first case is covered by the constraint in Equation (3.37), the latter can be solved by limiting P to a fixed upper bound $P_{max} \in \mathbb{Z}$ and the introduction of further binary variables p_l.

$$P = \sum_{l=1}^{P_{max}} l\, p_l \qquad p_l \in \{0, 1\} \tag{3.38}$$

(3.39)
$$\sum_{l=1}^{P_{max}} p_l = 1$$

Equation (3.39) ensures that exactly one fixed value is selected for $P \in [1, P_{max}]$ from Equation (3.38). Hereby, Equation (3.37) can be rewritten as follows.

(3.40)
$$\sum_{\forall i\,:\,(v_i,r_k)\in E_R} \sum_{d=0}^{\delta(r_k)-1} \sum_{\forall v\in\mathbb{Z}\,:\,l_i\le t-d-vl\le h_i} p_l\ x_{i,k,t-d-vl} \le \alpha(r_k) \quad \forall r_k \in V_T$$
$$\forall l\ :\ 1 \le l \le P_{max}$$
$$\forall t\ :\ 0 \le t \le l-1$$

The product of the two binary variables p_l and $x_{i,k,t-d-vl}$ is non-linear and thus does not fit into a linear program. However, it can be linearized by introducing a new binary variable $\gamma_{l,i,k,t-d-vl} \in \{0,1\}$, which depends on four indices.

$$\gamma_{l,i,k,s} = p_l x_{i,k,s} \qquad \text{where } s = t-d-vl$$

(3.41)
$$\gamma_{l,i,k,s} \le p_l$$

(3.42)
$$\gamma_{l,i,k,s} \le x_{i,k,s}$$

Using the inequalities in Equation (3.41) and Equation (3.42), the following table illustrates that for $\gamma_{l,i,k,s}$, the right values of the product $p_l x_{i,k,s}$ are obtained.

p_l	$x_{i,k,s}$	$\gamma_{l,i,k,s}$
0	0	0
0	1	0
1	0	0
1	1	$\{0,1\}$

Since $\gamma_{l,i,k,s}$ can be either zero or one, in the last row of the table, we can add $-\gamma_{l,i,k,s}$ to the objective function of the MIP in order to force it to one. The new objective function is defined as follows, where f denotes the original objective function.

(3.43)
$$\min \left(f - \sum_{\forall \text{ possible } \gamma_{l,i,k,s}} \gamma_{l,i,k,s} \right)$$

The formulation of the resource constraints becomes:

(3.44)
$$\sum_{\forall i\,:\,(v_i,r_k)\in E_R} \sum_{d=0}^{\delta(r_k)-1} \sum_{\forall v\in\mathbb{Z}\,:\,l_i\le s\le h_i} \gamma_{l,i,k,s} \le \alpha(r_k) \qquad \forall r_k \in V_T$$
$$\forall l\ :\ 1 \le l \le P_{max}$$
$$\forall t\ :\ 0 \le t < l$$

Additionally, the following binding constraint has to be satisfied.

$$\sum_{\forall k\,:\,(v_i,v_k)\in E_R}\ \sum_{s=l_i}^{h_i}\sum_{l=1}^{P_{max}} \gamma_{l,i,k,s} = 1 \qquad \forall v_i \in V \tag{3.45}$$

From the binding constraint in Equation (3.45) it follows that the summation over all possible variables $\gamma_{l,i,k,s}$ equals $|V|$ and thus is constant. Therefore, the modification of the objective function, as proposed in Equation (3.43), does not matter and can be neglected.

Without further modifications, the afore formulated constraints can be easily added directly to the MIPs presented in Section 3.5.1 and Section 3.5.2. However, it should be noticed that because of the discretization (introduction of binary variables), the start times of the nodes are enforced to be integral.

In a nutshell, the mixed integer programs have to be augmented by the following constraints.

Local allocation constraints if iteration interval is given

Additional input:

- Resource graph $G_R = (V_R, E_R, W, \Delta)$, where $V_R = V \cup V_T$
- Allocation $\alpha(r_k)$ for all $r_k \in V_T$

Additional output:

- Bindings of nodes $v_i \in V$ to resource types $r_k \in V_T$

Additional constraints:

$$\sum_{\forall k\,:\,(v_i,r_k)\in E_R}\ \sum_{t=l_i}^{h_i} x_{i,k,t} = 1 \qquad \forall v_i \in V$$

$$\sum_{\forall k\,:\,(v_i,r_k)\in E_R}\ \sum_{t=l_i}^{h_i} t\ x_{i,k,t} = \tau(v_i) \qquad \forall v_i \in V$$

$$\sum_{\forall k\,:\,(v_i,r_k)\in E_R}\ \sum_{t=l_i}^{h_i} w(v_i, r_k)\ x_{i,k,t} = w_i \qquad \forall v_i \in V$$

$$\sum_{\forall i\,:\,(v_i,r_k)\in E_R}\ \sum_{d=0}^{\delta(r_k)-1}\ \sum_{\forall \nu\in\mathbb{Z}\,:\,l_i\le t-d-\nu P\le h_i} x_{i,k,t-d-\nu P} \le \alpha(r_k) \qquad \forall r_k \in V_T \quad \forall t\,:\,0\le t\le P-1$$

where $x_{i,k,t} \in \{0,1\}$.

If the iteration interval P should be a variable of the MIP, the mixed integer program with projection as allocation in Section 3.5.1 can be augmented by the following constraints.

Local allocation constraints if iteration interval is a variable

Additional input:

- Resource graph $G_R = (V_R, E_R, W, \Delta)$, where $V_R = V \cup V_T$
- Allocation $\alpha(r_k)$ for all $r_k \in V_T$

Additional output:

- Bindings of nodes $v_i \in V$ to resource types $r_k \in V_T$

Additional constraints:

$$
\begin{aligned}
&\sum_{\forall k\,:\,(v_i,r_k)\in E_R} \sum_{t=l_i}^{h_i} x_{i,k,t} = 1 && \forall v_i \in V\\
&\sum_{\forall k\,:\,(v_i,v_k)\in E_R} \sum_{s=l_i}^{h_i} \sum_{l=1}^{P_{max}} \gamma_{l,i,k,s} = 1 && \forall v_i \in V\\
&\sum_{\forall k\,:\,(v_i,r_k)\in E_R} \sum_{t=l_i}^{h_i} t\ x_{i,k,t} = \tau(v_i) && \forall v_i \in V\\
&\sum_{\forall k\,:\,(v_i,r_k)\in E_R} \sum_{t=l_i}^{h_i} w(v_i, r_k)\ x_{i,k,t} = w_i && \forall v_i \in V\\
&\sum_{\forall i\,:\,(v_i,r_k)\in E_R} \sum_{d=0}^{\delta(r_k)-1} \sum_{\forall \nu\in\mathbb{Z}\,:\,l_i\le s\le h_i} \gamma_{l,i,k,s} \le \alpha(r_k) && \forall r_k \in V_T\\
& && \forall l\ :\ 1 \le l \le P_{max}\\
& && \forall t\ :\ 0 \le t < l\\
&\gamma_{l,i,k,s} \le p_l && \forall \text{ possible } \gamma_{l,i,k,s}\\
&\gamma_{l,i,k,s} \le x_{i,k,s} && \forall \text{ possible } \gamma_{l,i,k,s}
\end{aligned}
$$

where $x_{i,k,t}, \gamma_{l,i,k,s} \in \{0,1\}$ and $s = t - d - \nu l$.

3.5.3.2 Remarks

It should be noted, that if the iteration interval is considered as a variable of the MIP, the number of variables in the MIP grows by a factor of P. Thus, the entire program can become relatively large and might not be solvable anymore. Therefore, in practice, a bisection method that successively generates and solves several MIPs for different fixed values of P is better applicable.

Programs and reduced dependence graphs with run-time dependent conditions can be handled by the same mixed integer programs since conditions can be modeled by comparators and the merging of the program path can be considered as multiplexers in the resource model. However, this approach may have unnecessarily high computation cost since both branches have to be evaluated—possibly in sequential order—before the *right* result is selected. Anyhow, in some VLIW processor architectures and especially in EPIC [SR00], this so-called *predicated execution* [MHB+94] is the agent of choice in order to decrease the occurrence of branches and to increase the exploitation of instruction level parallelism. The removal of branches by *if-conversion* [AKPW83] is beneficial, especially, when considering deeply pipelined architectures such as in modern microprocessors. For instance, branches might cause an overhead of 18 to 19 clock cycles [BWSF06, IBM07] at the co-processors (called *synergistic processing elements*) of the Cell Broadband Engine Architecture [KDH+05]. In turn, the evaluation of a condition can be realized within one clock cycle in high-level synthesis. Thus, in the next section, the mutual exclusivity of different execution paths is studied with respect to resource constraints.

3.6 Conditional Scheduling

As mentioned earlier, if an algorithm has run-time dependent conditions (DPRA), it can be scheduled by the techniques introduced in Section 3.5.3. However, the scheduling of all possible execution paths may unnecessarily increase the length of the schedule as well as energy cost since some equations may have to be evaluated, which do not affect the data flow at all. This motivates to develop new resource constraints, which take advantage of mutually exclusive cases such as iteration and run-time dependent conditions.

As introduction, some simple examples are discussed. At first, a fragment in PAULA notation of an algorithm with iteration dependent conditions is considered (Example 3.3).

Example 3.3.

```
S1: c[i,j] = 0                if (i==0 and j==0);
S2: c[i,j] = a[i-1,j] * 3     if (i>=1 and j==0);
S3: c[i,j] = a[i,j-1] * 5     if (j>=1);
S4: d[i,j] = c[i,j] + b[i,j];
```

Assume that iteration dependent conditions can be evaluated in parallel to the data flow, with zero overhead. Without information about a global allocation (space map-

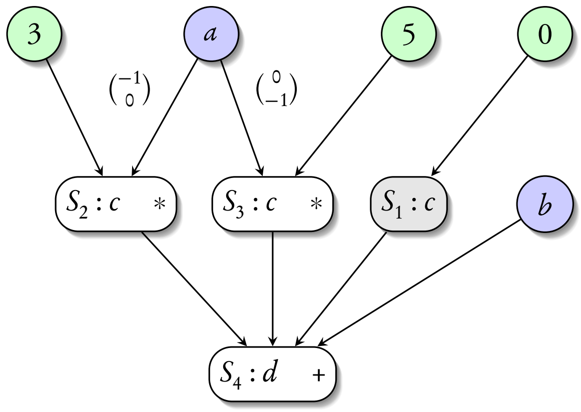


Figure 3.11: Reduced dependence graph (representation simplified) of Example 3.3. Corresponding to the extended RDG Definition 2.7, a separate node is spend for each variable c.

ping), it is not known whether these conditions can be resolved spatially or temporally.

According to Definition 2.7, the reduced dependence graph of the code fragment in Example 3.3 is depicted in Figure 3.11.
Regarding the resource constraints formulated in Section 3.5.3.1, both nodes[20] v_2 and v_3 have to be scheduled. Hence, either two multipliers are necessary or the two nodes have to be scheduled one after the other, even though the statements S_2 and S_3 are mutually exclusive, see for illustration Figure 3.12.

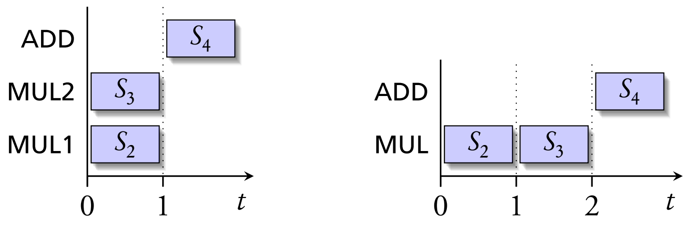


Figure 3.12: Possible Gantt charts, representing the execution order of the RDG shown in Figure 3.11, in the case of two available multipliers (left) and one (right), respectively.

It is possible to argue that this unfavorable behavior is due to the definition of the RDG because other definitions (cf. Definition 2.6 or for instance [TTZ97]) define only one node per variable name, which would lead to the RDG shown in Figure 3.13 on the left.

[20]Note that we use statement S_i and node v_i synonymously.

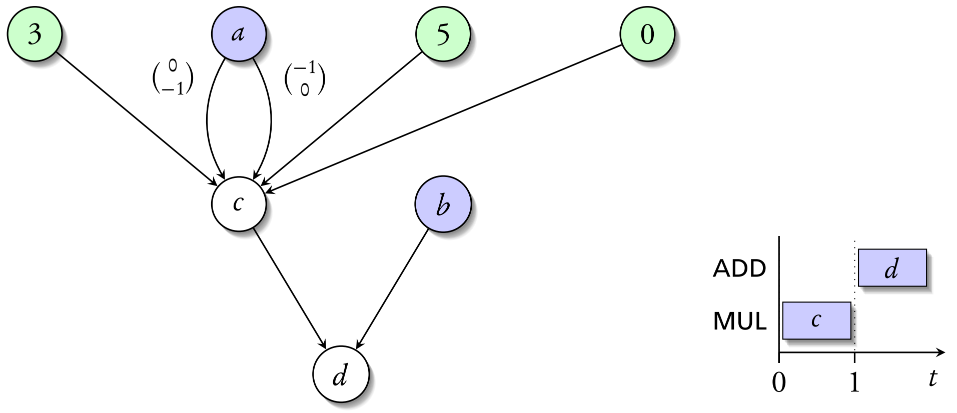


Figure 3.13: Reduced dependence graph of the code fragment of Example 3.3. According to Definition 2.6 of an RDG, only one node represents variable c.

With this graph, the problem does not exist since only one node "c" has to be scheduled as shown in Figure 3.13 on the right. This argument holds as long as several statements, that are combined to one RDG node, can be bound to the same resource type, which is not the case in the next example.

Example 3.4.

```
S1: c[i] = a[i] * b[i]       if (i==0);
S2: c[i] = a[i] + b[i]       if (i>=1);
S3: d[i] = b[i] + 4          if (i==0);
S4: d[i] = a[i] * 5          if (i>=1);
```

For Example 3.4, the reduced dependence graph and a possible resource-constrained schedule are visualized in Figure 3.14.

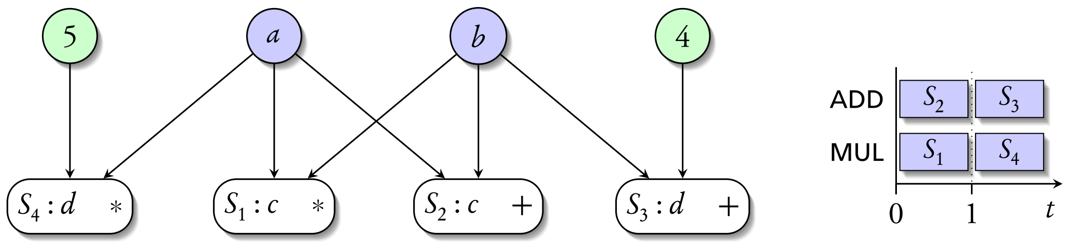


Figure 3.14: On the left, RDG of the code fragment of Example 3.4. On the right, possible schedule in case of one available adder and multiplier.

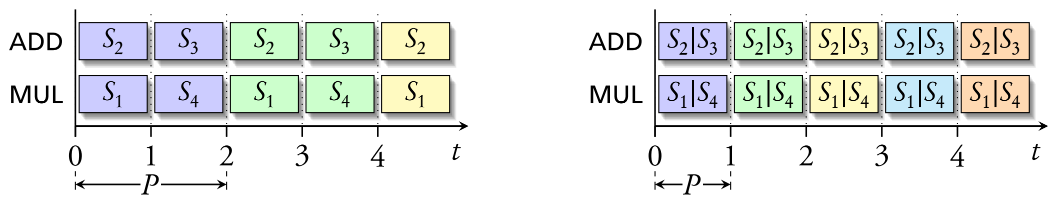


Figure 3.15: Different periodic schedules of the code fragment specified in Example 3.4 are shown. On the left, all statements are evaluated, leading to an iteration interval of two. That is, in every second cycle a new iteration can be started, which is denoted by a different color in the Gantt chart. On the right, the mutual exclusivity is incorporated, leading to a doubled throughput ($P = 1$).

If mutual exclusivity is considered in Example 3.4, the throughput can be doubled at the same number of resources. In the following, a "|" in the bars of the Gantt charts denotes that only one statement/node of a given set is executed in the interval. The iterative execution of the above example, in case of parallel and conditional execution, is shown in Figure 3.15. Diverse colors denote that operations belong to different iterations.

The throughput associated with a schedule can be even worse if more than two disjoint execution paths exist, which is not unusual, for instance, if an algorithm has been partitioned.

Some remarks on run-time dependent conditions are given in the following Sobel edge detection algorithm.

Example 3.5 (Sobel edge detection algorithm)**.**

```
par (x>=0 and x<=N-1 and y>=0 and y<=M-1)
{
 S1:  p[x,y]  = pi[x,y];
 S2:  q[x,y]  = 2 * p[x-1,y-1]                    if (x>=1 and y>=1);
 S3:  h1[x,y] = p[x-1,y-2] - p[x-1,y]             if (x>=1 and y>=2);
 S4:  h2[x,y] = h1[x,y] + q[x,y]                  if (x>=1 and y>=2);
 S5:  v1[x,y] = p[x-2,y-1] + p[x,y-1]             if (x>=2 and y>=1);
 S6:  v2[x,y] = v1[x,y] + q[x,y]                  if (x>=2 and y>=1);
 S7:  h3[x,y] = h2[x-2,y-1] - h2[x,y-1]           if (x>=3 and y>=3);
 S8:  h4[x,y] = ifrt(h3[x,y]<0, -h3[x,y], h3[x,y]) if (x>=3 and y>=3);
 S9:  v3[x,y] = v2[x-1,y-2] - v2[x-1,y]           if (x>=3 and y>=3);
 S10: v4[x,y] = ifrt(v3[x,y]<0, -v3[x,y], v3[x,y]) if (x>=3 and y>=3);
 S11: s[x,y]  = h4[x,y] + v4[x,y]                 if (x>=3 and y>=3);
 S12: po[x,y] = ifrt(s[x,y]>255, 255, s[x,y])     if (x>=3 and y>=3);
}
```

Two out of the three run-time dependent conditions in the algorithm (Example 3.5) are used for the calculation of absolute values, whereas the third is used for a threshold function. Once again, the semantics, for instance, of statement S8 is equivalent to the following pseudo code fragment.

```
if (h3[x,y]<0) then
  h4[x,y] = -h3[x,y];
else
  h4[x,y] = h3[x,y];
endif
```

These simple cases can be implemented by dedicated functions, as, for instance, statement S_8 could be replaced by the following

```
S8:  h4[x,y] = abs(h3[x,y])    if (x>=3 and y>=3);
```

which might be true in case of high-level synthesis at the cost of defining new basic functions. But in case of generating assembly code for a programmable architecture, only the available instructions can be used. The situation becomes even more interesting if a run-time dependent condition is used that has influence on more than one statement as in the following program [HT04c], which is introduced as pseudo code at first.

Example 3.6 (Nested run-time dependent conditions)**.** *The following pseudo code is used to illustrate the nesting of run-time dependent conditions. Similar to the PAULA notation, the order of statements does not matter and the* FORALL *keyword denotes a parallel loop construct.*

```
FORALL ($i = 1$; $i \leq N$; $i++$)
  FORALL ($j = 1$; $j \leq M$; $j++$)
    $b[i,j] = b[i,j-1] - c[i-1,j]$;
    $a[i,j] = a[i-1,j] + b[i,j]$;
    IF ($a[i,j] > 10$)
      $c[i,j] = b[i,j] * b[i,j]$;
      IF ($b[i,j] > 8$)
        $d[i,j] = b[i,j] + a[i,j]$;
        $e[i,j] = a[i,j] + 10$;
      ELSE
        $d[i,j] = 2 * b[i,j]$;
        $e[i,j] = a[i,j] + 3$;
      ENDIF
    ELSE
      $c[i,j] = a[i,j] + 8$;
```

```
        $d[i,j] = b[i,j]+3$;
        $e[i,j] = 2*a[i,j]$;
      ENDIF
    ENDFOR
  ENDFOR
```

The corresponding program in PAULA notation is defined as follows.

```
par (i>=1 and i<=N and j>=1 and j<=M)
{S1:  a[i,j]   = a[i-1,j] + b[i,j];
 S2:  b[i,j]   = b[i,j-1] - c[i-1,j];
 S3:  c[i,j]   = ifrt(C1[i,j], c1[i,j], c0[i,j]);
 S4:  c1[i,j]  = b[i,j] * b[i,j];
 S5:  c0[i,j]  = a[i,j] + 8;
 S6:  d[i,j]   = ifrt(C1[i,j], d1[i,j], d0[i,j]);
 S7:  d1[i,j]  = ifrt(C2[i,j], d11[i,j], d10[i,j]);
 S8:  d0[i,j]  = b[i,j] + 3;
 S9:  d11[i,j] = b[i,j] + a[i,j];
 S10: d10[i,j] = 2 * b[i,j];
 S11: e[i,j]   = ifrt(C1[i,j], e1[i,j], e0[i,j]);
 S12: e1[i,j]  = ifrt(C2[i,j], e11[i,j], e10[i,j]);
 S13: e0[i,j]  = 2 * a[i,j];
 S14: e11[i,j] = a[i,j] + 10;
 S15: e10[i,j] = a[i,j] + 3;
 S16: C1[i,j]  = a[i,j] > 10;
 S17: Ch[i,j]  = b[i,j] > 8;
 S18: C2[i,j]  = ifrt(C1[i,j], Ch[i,j], false);
}
```

In Example 3.6, run-time dependent condition C_1 is used in four statements (S_3, S_6, S_{11}, and S_{18}) and condition C_2 in two statements (S_7 and S_{12}). If a projection in direction $(1\ 0)^T$ is used as global allocation, the schedules in Figure 3.16 and Figure 3.17 depict the case that all statements are considered and scheduled ($P = 6$) and the case of conditional execution ($P = 4$), respectively. In this example, the throughput is increased by 33 % when exploiting the conditional execution in comparison to the version without conditional execution. The simple examples for both, iteration dependent and run-time dependent conditions, demonstrate the benefits of a conditional execution. The question to be answered now is, whether and how the conditional behavior can be integrated into the resource constraints of the afore introduced MIPs. For this, the concept of so-called *AND-XOR-trees* is introduced.

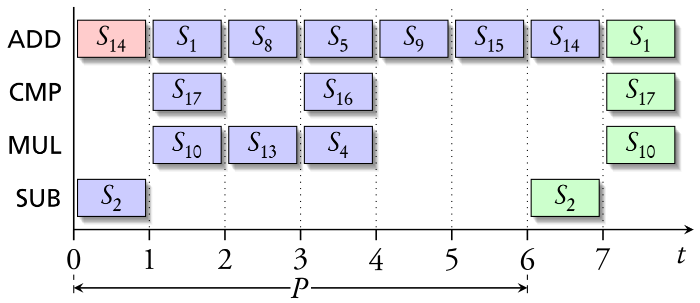


Figure 3.16: Gantt chart for Example 3.6 when all statements have to be scheduled.

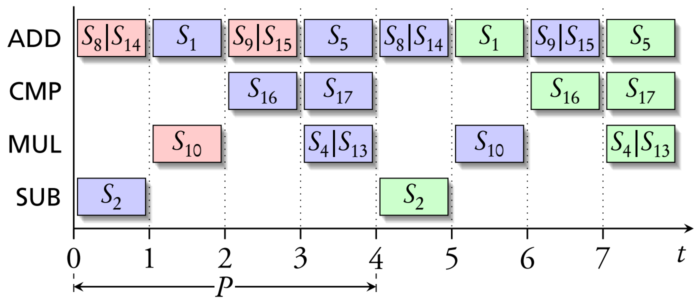


Figure 3.17: Gantt chart for Example 3.6 when conditional execution is considered.

3.6.1 AND-XOR-Tree

If the number of available resources (functional units) within a processor is limited, several operations may compete for the same resource. It has to be prevented that more than $\alpha(r_k)$ operations are being simultaneously executed by the same resource type $r_k \in V_T$. Here, we have to distinguish between two different cases:

1. Concurrent operations/statements, we also say the statements are in AND-relation.
2. If there are different execution branches in an algorithm, such as iteration or run-time dependent conditions, the statements are mutually exclusive, we say in XOR-relation.

The relationships among a set of statements can be represented as a tree [HHL90].

Definition 3.11 (AND-XOR-tree)**.** *An* AND-XOR-tree $X = (V^x, E^x)$ *or shortened* AXT *represents the relationship among a set of statements, where the internal nodes* $v_i^x \in V^x$ *are of the introduced types, XOR (*$\oplus$*) and AND (*$\odot$*), and the leaves* v_i^x *correspond to statements* S_i *and left-hand side variables* v_i *of the statement, respectively. The root of the*

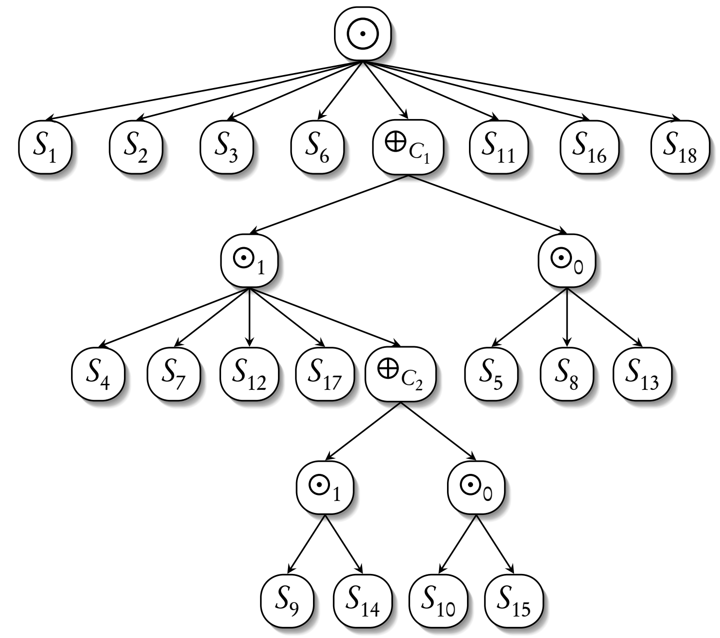


Figure 3.18: AND-XOR-tree for the PAULA program in Example 3.6.

tree is represented by $\odot$. The affiliation of a node to an XOR or AND node is denoted by a directed edge $e^x \in E^x$.

The AXT of the previous program is shown in Figure 3.18. The subscripts of the XOR nodes denote the corresponding condition C_i. The subscripts of the AND nodes are either 1, denoting the "if"-branch of the condition, or 0, denoting the "else"-branch of the condition. The nesting of the two conditions C_1 and C_2 is represented by the different levels of the tree. Iteration dependent conditions cannot be nested since each statement is assigned to exactly one iteration space. A program fragment which contains both types of conditions is given in Example 3.7.

Example 3.7.

```
S0:  b[i]  = a[i] * 2                    if (i==1);
S1:  C1[i] = a[i] > 10                   if (i==1);
S2:  c[i]  = ifrt(C1[i], c0[i], c1[i])  if (i==1);
S3:  c0[i] = a[i] * 3                    if (i==1);
S4:  c1[i] = a[i] * 4                    if (i==1);
S5:  b[i]  = a[i] * 5                    if (i>1);
```

The AXT according to Example 3.7 is depicted in Figure 3.19, where the decomposition of the iteration space $\mathcal{I}$ is denoted by $\oplus_{\mathcal{I}}$ and the corresponding mutual exclusive cases are expressed by the subscripts of the child nodes (AND nodes).

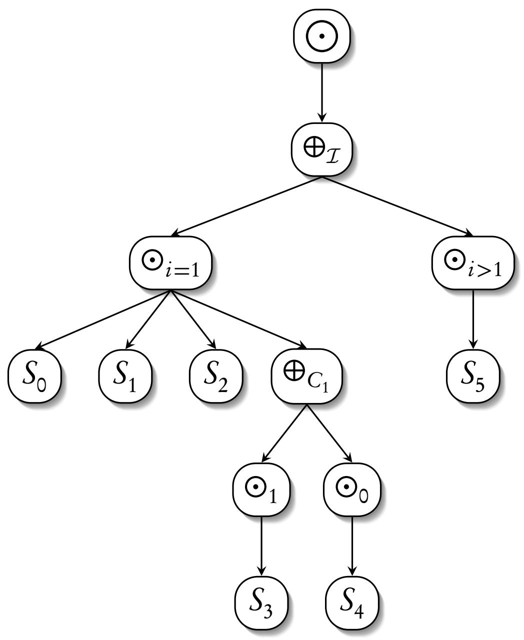


Figure 3.19: AND-XOR-tree with both, run-time and iteration dependent conditions.

Since the number of mutually exclusive iteration subspaces of a given iteration space can be arbitrary, two possibilities of how to decompose the iteration space are given. Consider, for instance, the following example.

Example 3.8.

```
S0:  b[i,j] = a[i,j] * 2    if (i==1 and j==1);
S1:  b[i,j] = a[i,j] * 5    if (i>1 and j==1);
S2:  b[i,j] = a[i,j] * 3    if (j>1);
S3:  c[i,j] = a[i,j] * 4;
S4:  d[i,j] = a[i,j] * 7    if (j>1);
S5:  e[i,j] = a[i,j] * 8    if (i==1);
S6:  e[i,j] = a[i,j] * 9    if (i>1);
```

As a first variant, we consider a decomposition into disjoint iteration spaces per variable as shown in Figure 3.20.
Afterwards (see Figure 3.21), the statements with equal iteration dependent conditions can be clustered; here, statements S_2 and S_4.
In the second variant, a decomposition of the entire iteration space into disjoint subspaces is performed and the individual statements are assigned to these subspaces. If a variable is present in all subspaces, it has no iteration dependent condition and can

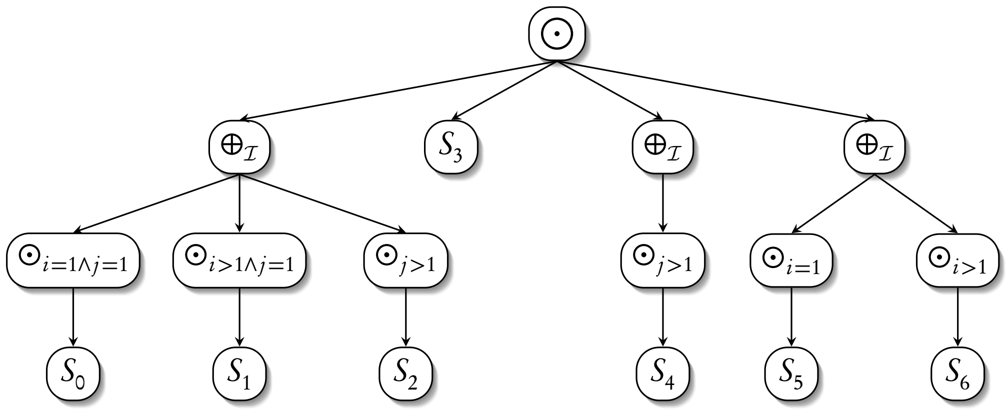


Figure 3.20: AND-XOR-tree of Example 3.8, first decomposition variant (after decomposition into disjoint iteration spaces per variable).

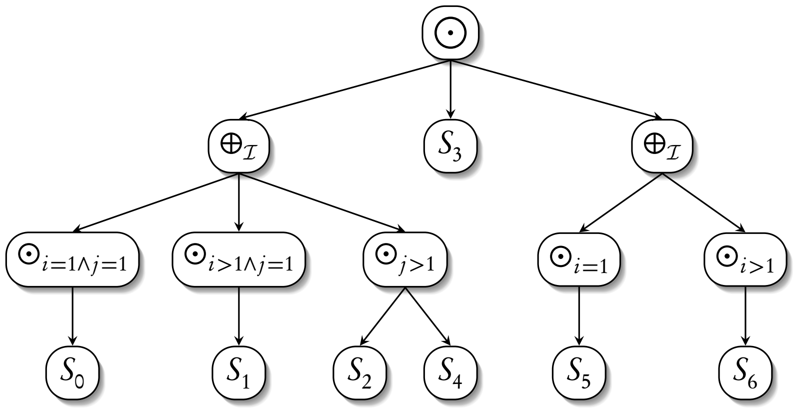


Figure 3.21: AXT of Example 3.8, first decomposition variant (after clustering).

therefore be connected directly to the root node. The decomposition of Example 3.8 leads to four disjoint subspaces.

$$\{i = 1 \wedge j = 1\} \qquad \{i > 1 \wedge j = 1\} \qquad \{i = 1 \wedge j > 1\} \qquad \{i > 1 \wedge j > 1\}$$

An allocation of statements to the four spaces results in the AND-XOR-tree depicted in Figure 3.22.
Whether the first or second variant should be preferred, has to be studied from case to case, depending on the complexity and control requirements.

The concept of relationship trees allows us to formulate conditional resource constraints in the next section.

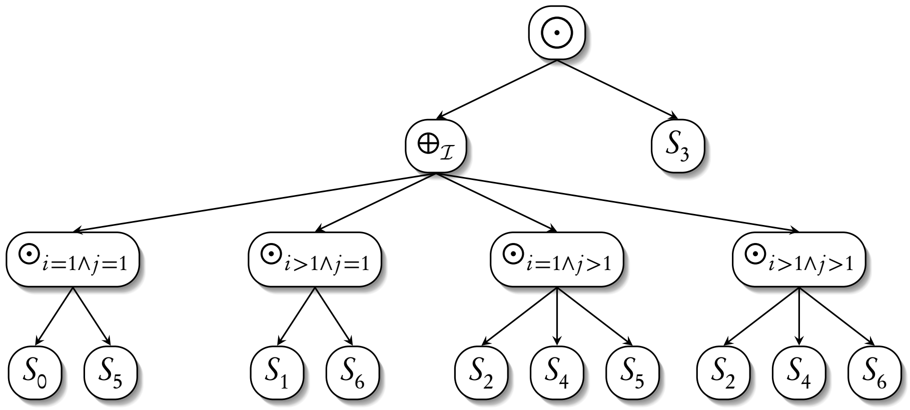


Figure 3.22: AND-XOR-tree of Example 3.8, second decomposition variant.

3.6.2 Resource Constraints for Conditional Execution

Let the relationship between the nodes of a given RDG be expressed by a corresponding AXT $X = (V^x, E^x)$ as defined in Section 3.6.1. Further, let function f_i be associated to each node $v_i^x \in V^x$. Since resource constraints (cf. Section 3.5.3.1) depend on the resource type r_k and the time step t, we write $f_i(r_k, t)$. The function $f_i(r_k, t)$ denotes the number of occupied instances of resource type r_k at time step t. The resource constraints can be formulated as follows.

$$f_{\odot}(r_k, t) \leq \alpha(r_k) \qquad \forall r_k \in V_T \quad \forall t \,:\, 0 \leq t \leq P-1 \tag{3.46}$$

Here, $f_{\odot}$ denotes the function f_i of the root node ($i = \odot$). Function f_i is recursively defined as follows.

$$f_i(r_k, t) = \begin{cases} \displaystyle\sum_{j \in \left\{j \,\mid\, (v_i^x, v_j^x) \in E^x\right\}} f_j(r_k, t) & \text{if node } v_i^x \text{ is root node} \\ \displaystyle\sum_{t' \in T_i^{all}(r_k, t)} \max_{j \in \left\{j \,\mid\, (v_i^x, v_j^x) \in E^x\right\}} f_j'(r_k, t, t') & \text{if node } v_i^x \text{ is XOR node} \\ \displaystyle\sum_{d=0}^{\delta(r_k)-1} \sum_{\forall \nu : l_i \leq t-d-\nu P \leq h_i} x_{i,k,t-d-\nu P} & \text{if node } v_i^x \text{ is leaf} \,:\, (v_i, r_k) \in E_R \\ 0 & \text{if node } v_i^x \text{ is leaf} \,:\, (v_i, r_k) \notin E_R \end{cases} \tag{3.47}$$

with

(3.48)

$$f_i'(r_k,t,t') = \begin{cases} \sum\limits_{j \in \left\{j \mid (v_i^x, v_j^x) \in E^x\right\}} f_j'(r_k,t,t') & \text{if node } v_i^x \text{ is AND node} \\ \max\limits_{j \in \left\{j \mid (v_i^x, v_j^x) \in E^x\right\}} f_j'(r_k,t,t') & \text{if node } v_i^x \text{ is XOR node} \\ x_{i,k,t'} & \begin{array}{r}\text{if node } v_i^x \text{ is leaf} : (v_i, r_k) \in E_R \\ \wedge\ t' \in T_i(r_k,t)\end{array} \\ 0 & \begin{array}{r}\text{if node } v_i^x \text{ is leaf} : (v_i, r_k) \notin E_R \\ \vee\ t' \notin T_i(r_k,t)\end{array} \end{cases}$$

The sets T_i^{all}, depending on r_k and t, are recursively derived as follows.

(3.49)

$$T_i^{all}(r_k,t) = \begin{cases} \bigcup\limits_{j \in \left\{j \mid (v_i^x, v_j^x) \in E^x\right\}} T_j^{all}(r_k,t) & \text{if node } v_i^x \text{ is AND or XOR node} \\ T_i(r_k,t) & \text{if node } v_i^x \text{ is leaf} : (v_i, r_k) \in E_R \\ \emptyset & \text{if node } v_i^x \text{ is leaf} : (v_i, r_k) \notin E_R \end{cases}$$

Finally, the sets $T_i(r_k,t)$ are defined as follows.

(3.50) $$T_i(r_k,t) = \left\{t' \mid l_i \le t' \le h_i \ \wedge\ t' = t - d - \nu P \ \wedge\ 0 \le d \le \delta(r_k) - 1\right\}$$

In Equations (3.47) and (3.48), the maximum functions reflect the mutual exclusivity of different branches. For instance, if two binary variables $x_{i,k,t}$ and $x_{j,k,t}$ belong to nodes v_i and v_j, respectively, that are mutually exclusive, the values of both binary variables can be one since $\max(x_{i,k,t}, x_{j,k,t})$ will be at most one and thus only one resource r_k is occupied at time t.

In the following, the specific constraints are explained in detail. The first case of Equation (3.47) is considered when the given node is the root node—this is only the case when the function is called in Equation (3.46). It sums up all its direct child nodes, that is, all its child nodes are in AND-relation. The second case of Equation (3.47) is considered if the nodes are in exclusive disjunction. Here, it is not only sufficient to consider the node at time step t but also at all time steps $t' \in T_i^{all}$ at which operations possibly compete for the same resource. This is important if the latency of a branch is longer than the iteration interval. That is, the set $T_i^{all}(r_k,t)$ represents all possible start times of competing operations within the same subtree rooted at node $v_i^x \in V^x$. The third case of Equation (3.47) is the same as in the normal resource constraint (see Equation (3.37)), when the nodes are in AND-relation.

The last case of Equation (3.47) returns zero if the corresponding node v_i has no binding possibility on resource type r_k.

The different cases of Equation (3.48) are similar to that of Equation (3.47), except that the possible time steps t' are explicitly given by the calling functions. Thus, in the third case of the constraint, it has to be verified if the given t' belongs to the set of possible start times $T_i(r_k, t)$ of node v_i.

The resource constraint is illustrated by the following example.

Example 3.9.

```
par (i>=1 and i<=N)
{ S0:  C1[i] = x[i] > 7;
  S1:  a[i]  = ifrt(C1[i], a1[i], a0[i]);
  A:   a1[i] = x[i] + 3;
  B:   a0[i] = x[i] * 3;
  S2:  b[i]  = ifrt(C1[i], b1[i], b0[i]);
  C:   b1[i] = a[i] * 4;
  D:   b0[i] = a[i] + 4;
}
```

In Example 3.9, it is assumed that the resource model contains one multiplier, one adder, and one comparator, which all complete their operations within one clock-cycle. The corresponding AND-XOR-tree is depicted in Figure 3.23 and the associated reduced dependence graph is shown in Figure 3.24.

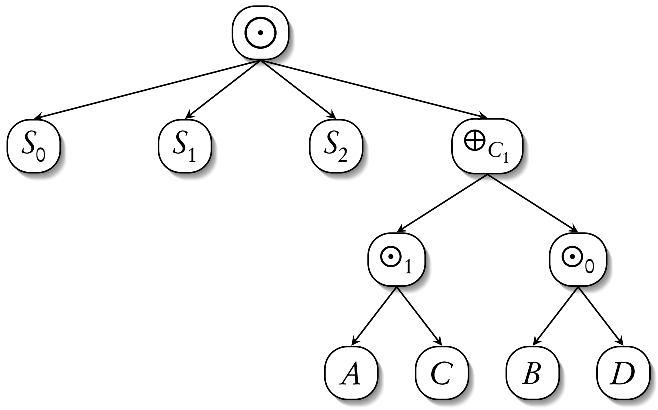


Figure 3.23: AND-XOR-tree of the program in Example 3.9.

Since only one addition and one multiplication are performed per iteration, one might guess that an iteration interval of $P = 1$ is possible.

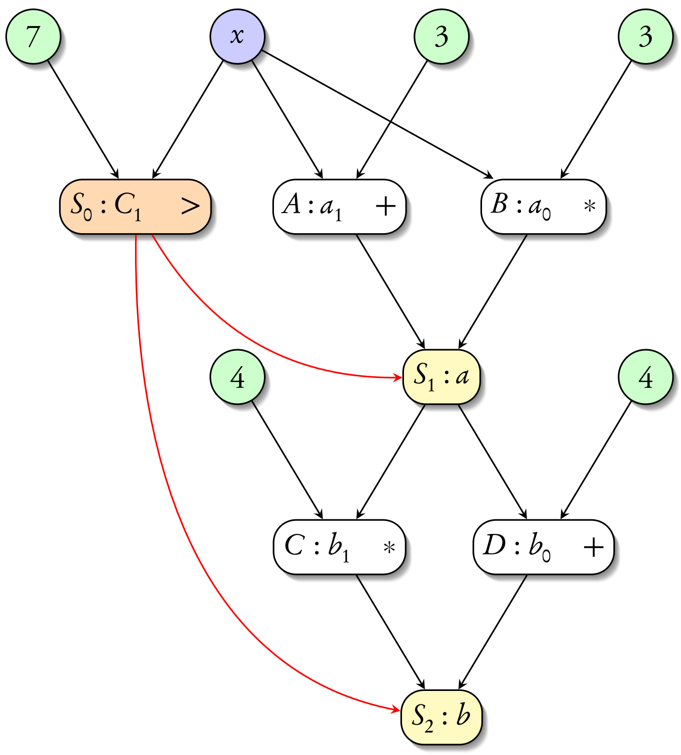


Figure 3.24: Reduced dependence graph of the program in Example 3.9.

$$P = 1 \qquad r_k = \text{ADD} \qquad l_A = h_A = 0 \qquad l_D = h_D = 1$$

$$t = 0: \qquad f_{\odot}(\text{ADD}, 0) = \sum_{t' \in T^{all}_{\oplus_{C_1}}(\text{ADD},0)} \max(f'_{\odot_1}(\text{ADD}, 0, t'), f'_{\odot_0}(\text{ADD}, 0, t'))$$

$$T^{all}_{\oplus_{C_1}}(\text{ADD}, 0) = \{0, 1\}$$

$$\Rightarrow \quad f_{\odot}(\text{ADD}, 0) = \max(x_{A,\text{ADD},0}) + \max(x_{D,\text{ADD},1}) = x_{A,\text{ADD},0} + x_{D,\text{ADD},1}$$

The resource constraint $f_{\odot}(\text{ADD}, 0) \leq \alpha(\text{ADD}) = 1$ cannot be satisfied since the statements A and D have no more mobility, that is, A has to be started at time step 0 and D at time step 1. An iteration interval of $P = 2$ leads to more flexibility in starting the operations (mobility).

$$P = 2 \qquad r_k = \text{ADD} \qquad l_A = 0 \qquad h_A = 1 \qquad l_D = 1 \qquad h_D = 2$$

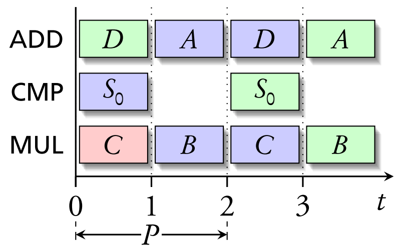


Figure 3.25: Schedule for Example 3.9 with the minimal iteration interval of $P = 2$.

$$t = 0: \quad f_{\odot}(\text{ADD}, 0) = \sum_{t' \in T^{all}_{\oplus_{C_1}}(\text{ADD},0)} \max(f'_{\odot_1}(\text{ADD}, 0, t'), f'_{\odot_0}(\text{ADD}, 0, t'))$$

$$T^{all}_{\oplus_{C_1}}(\text{ADD}, 0) = \{0, 2\}$$

$$\Rightarrow \quad f_{\odot}(\text{ADD}, 0) = \max(x_{A,\text{ADD},0}) + \max(x_{D,\text{ADD},2}) = x_{A,\text{ADD},0} + x_{D,\text{ADD},2}$$

$$t = 1: \quad f_{\odot}(\text{ADD}, 1) = \sum_{t' \in T^{all}_{\oplus_{C_1}}(\text{ADD},1)} \max(f'_{\odot_1}(\text{ADD}, 1, t'), f'_{\odot_0}(\text{ADD}, 1, t'))$$

$$T^{all}_{\oplus_{C_1}}(\text{ADD}, 1) = \{1\}$$

$$\Rightarrow \quad f_{\odot}(\text{ADD}, 1) = \max(x_{A,\text{ADD},1}) + \max(x_{D,\text{ADD},1}) = x_{A,\text{ADD},1} + x_{D,\text{ADD},1}$$

A solution that satisfies both constraints $f_{\odot}(\text{ADD}, 0) \leq 1$ and $f_{\odot}(\text{ADD}, 1) \leq 1$ is given by the schedule shown in Figure 3.25. The merge nodes (statements S_1 and S_2) are not shown since it is assumed that they are implemented as a multiplexer and, hence, do not require extra execution time units.

The proposed resource constraints for conditional execution can be integrated into the previously presented MIPs fairly easily. The only question is, how the maximum functions in Equations (3.47) and (3.48) should be formulated. Here, several possibilities exist, which are briefly described in the following (we refer to Appendix A.2 for a more detailed description).

Let $a_1, a_2, \ldots, a_n$ be variables of a mixed integer program. Then, the maximum $b = \max(a_1, a_2, \ldots, a_n)$ can be determined as follows.

1. By adding n inequalities $b \geq a_i$, $i = 1 \ldots n$ as constraints to the MIP and by adding b to the objective function, $f(x) + b$.

2. By adding constraints that contain binary variables and *big-M constants* (see Appendix A.2).

In summary, the MIPs have to be augmented by the following constraints in order to handle resource constraints for the conditional execution of statements.

Local allocation constraints for conditional execution

Additional input:

- Resource graph $G_R = (V_R, E_R, W, \Delta)$, where $V_R = V \cup V_T$
- AND-XOR-tree $X = (V^x, E^x)$
- Allocation $\alpha(r_k)$ for all $r_k \in V_T$

Additional output:

- Bindings of nodes $v_i \in V$ to resource types $r_k \in V_T$

Additional constraints:

$$f_{\odot}(r_k, t) \le \alpha(r_k) \qquad \forall r_k \in V_T \qquad \forall t\ :\ 0 \le t \le P-1$$

starting with $i = \odot$ and

$$f_i(r_k,t) = \begin{cases} \sum\limits_{j \in \left\{ j \mid (v_i^x, v_j^x) \in E^x \right\}} f_j(r_k,t) & \text{if node } v_i^x \text{ is root node} \\ \sum\limits_{t' \in T_i^{all}(r_k,t)} \max\limits_{j \in \left\{ j \mid (v_i^x, v_j^x) \in E^x \right\}} f'_j(r_k,t,t') & \text{if node } v_i^x \text{ is XOR node} \\ \sum\limits_{d=0}^{\delta(r_k)-1} \sum\limits_{\forall \nu : l_i \le t-d-\nu P \le h_i} x_{i,k,t-d-\nu P} & \text{if node } v_i^x \text{ is leaf : } (v_i, r_k) \in E_R \\ 0 & \text{if node } v_i^x \text{ is leaf : } (v_i, r_k) \notin E_R \end{cases}$$

$$f'_i(r_k,t,t') = \begin{cases} \sum\limits_{j \in \left\{ j \mid (v_i^x, v_j^x) \in E^x \right\}} f'_j(r_k,t,t') & \text{if node } v_i^x \text{ is AND node} \\ \max\limits_{j \in \left\{ j \mid (v_i^x, v_j^x) \in E^x \right\}} f'_j(r_k,t,t') & \text{if node } v_i^x \text{ is XOR node} \\ x_{i,k,t'} & \begin{array}{l}\text{if node } v_i^x \text{ is leaf : } (v_i, r_k) \in E_R \\ \qquad \wedge\ t' \in T_i(r_k,t)\end{array} \\ 0 & \begin{array}{l}\text{if node } v_i^x \text{ is leaf : } (v_i, r_k) \notin E_R \\ \qquad \vee\ t' \notin T_i(r_k,t)\end{array} \end{cases}$$

$$T_i^{all}(r_k,t) = \begin{cases} \bigcup\limits_{j \in \left\{ j \mid (v_i^x, v_j^x) \in E^x \right\}} T_j^{all}(r_k,t) & \text{if node } v_i^x \text{ is AND or XOR node} \\ T_i(r_k,t) & \text{if node } v_i^x \text{ is leaf : } (v_i, r_k) \in E_R \\ \emptyset & \text{if node } v_i^x \text{ is leaf : } (v_i, r_k) \notin E_R \end{cases}$$

$$T_i(r_k,t) = \left\{ t' \mid l_i \le t' \le h_i\ \wedge\ t' = t - d - \nu P\ \wedge\ 0 \le d \le \delta(r_k) - 1 \right\}$$

where $x_{i,k,t} \in \{0,1\}$.

3.7 Lifetime of Variables

In order to compute the number of necessary memory or to incorporate further resource constraints into the MIPs, the lifetime of a variable is of interest.

Definition 3.12 (Lifetime). *The* lifetime interval *of a variable v is the time interval beginning at the definition (production) of the variable and ending at the time where no more operations use (consume) the variable. The lifetime interval is denoted by $\underline{\rho}(v) = [\rho_a(v), \rho_b(v)]$. The duration, defined by $\rho(v) = \rho_b(v) - \rho_a(v)$ is called* lifetime.

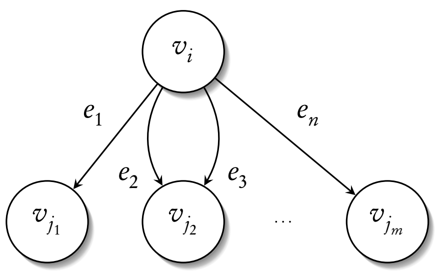


Figure 3.26: Illustration for the lifetime of a variable v_i.

The lifetime of a variable, or rather the lifetime of its corresponding node $v_i \in V$ of the reduced dependence graph $G = (V, E)$ is illustrated in Figure 3.26. The node v_i has a set $F(v_i)$ of outgoing edges, which is defined by $F(v_i) = \{e \in E \mid e = (v_i, v_k) \in E\}$. Let n be the number of edges in this set $F(v_i)$ and the edges themselves be enumerated from e_1, e_2 to e_n. Then, the dependencies can be expressed by the set of the following equations.

$$\begin{aligned}
x_i[I] &= \ldots \\
x_{j_1}[I] &= \mathcal{F}_{j_1}(\ldots, x_i[I + d_{e_1}], \ldots) \\
x_{j_2}[I] &= \mathcal{F}_{j_2}(\ldots, x_i[I + d_{e_2}], x_i[I + d_{e_3}], \ldots) \\
&\vdots \\
x_{j_m}[I] &= \mathcal{F}_{j_m}(\ldots, x_i[I + d_{e_n}], \ldots)
\end{aligned}$$

It follows, that the beginning ρ_a and the end ρ_b of the lifetime interval of node v_i are given by:

$$\begin{aligned}
\rho_a(v_i) &= t_i(I) + w_i = \Lambda I + \lambda + \tau(v_i) + w_i \\
\rho_b(v_i) &= \max_{\forall v_j \in V \,:\, e=(v_i,v_j) \in F(v_i)} \left(t_j(I + d_e) \right) \\
&= \max_{\forall v_j \in V \,:\, e=(v_i,v_j) \in F(v_i)} \left(\Lambda(I + d_e) + \lambda + \tau(v_j) \right)
\end{aligned}$$

Consequently, the lifetime interval $\varrho(v_i)$ and the lifetime $\rho(v_i)$ of node v_i are given by:

$$\begin{aligned}
\varrho(v_i) &= [\rho_a(v_i), \rho_b(v_i)] \\
&= \left[\Lambda I + \lambda + \tau(v_i) + w_i, \max_{\forall v_j \in V \,:\, e=(v_i,v_j)\in F(v_i)} \big(\Lambda(I + d_e) + \lambda + \tau(v_j)\big)\right] \\
\rho(v_i) &= \rho_b(v_i) - \rho_a(v_i) \\
&= \max_{\forall v_j \in V \,:\, e=(v_i,v_j)\in F(v_i)} \big(\Lambda(I + d_e) + \lambda + \tau(v_j)\big) - \big(\Lambda I + \lambda + \tau(v_i) + w_i\big) \\
&= \max_{\forall v_j \in V \,:\, e=(v_i,v_j)\in F(v_i)} \big(\Lambda d_e + \tau(v_j) - \tau(v_i) - w_i\big)
\end{aligned}$$

Let $e \in E$ be an edge of the given reduced dependence graph $G = (V, E)$ and d_e be the associated data dependence vector. Hence, in dependence on the space-mapping Φ and the value of d_e, three different types of edges and lifetimes can be identified. The set of outgoing edges $F(v_i)$ of a node v_i can be decomposed into three disjoint sets, $F(v_i) = F_\circ(v_i) \cup F_\triangleleft(v_i) \cup F_\triangleright(v_i)$. That is, edge e can be classified to exactly one of the three sets $F_\circ(v_i)$, $F_\triangleleft(v_i)$, and $F_\triangleright(v_i)$ as follows.

Type 1

If the data dependency vector d_e satisfies $\Phi d_e = 0 \wedge d_e = 0$, e belongs to set $F_\circ(v_i)$. From the hardware perspective, the set $F_\circ(v_i)$ represents local memory within a processor to store intermediate results within one iteration period. The corresponding lifetime $\rho_\circ$ of the node v_i is defined as follows.

(3.51) $$\rho_\circ(v_i) = \max_{\forall v_j \in V \,:\, e=(v_i,v_j)\in F_\circ(v_i)} \big(\tau(v_j) - \tau(v_i) - w_i\big)$$

Type 2

If the data dependency vector d_e satisfies $\Phi d_e = 0 \wedge d_e \neq 0$, e belongs to set $F_\triangleleft(v_i)$. Set $F_\triangleleft(v_i)$ also denotes the internal storage within a processor but since $d_e \neq 0$, the corresponding data need to be stored for more than one iteration. The corresponding lifetime $\rho_\triangleleft$ of the node v_i is defined as follows.

(3.52) $$\rho_\triangleleft(v_i) = \max_{\forall v_j \in V \,:\, e=(v_i,v_j)\in F_\triangleleft(v_i)} \big(\Lambda d_e + \tau(v_j) - \tau(v_i) - w_i\big)$$

Type 3

If the data dependency vector d_e satisfies $\Phi d_e \neq 0$, e belongs to set $F_\triangleright(v_i)$. From the hardware perspective, this set denotes delay memory between different processors. The corresponding lifetime $\rho_\triangleright$ of the node v_i is defined as follows.

(3.53) $$\rho_\triangleright(v_i) = \max_{\forall v_j \in V \,:\, e=(v_i,v_j)\in F_\triangleright(v_i)} \big(\Lambda d_e + \tau(v_j) - \tau(v_i) - w_i\big)$$

Since we consider iterative algorithms with a period of P, the amount of memory $m(v_i)$ needed to store all life instances of a variable v_i is proportional to the lifetime of the variable.

$$m(v_i) = \left\lceil \frac{\rho(v_i)}{P} \right\rceil$$

If we want to distinguish between the afore defined types, we have to add the individual memories.

$$m(v_i) = \left\lceil \frac{\rho_\circ(v_i)}{P} \right\rceil + \left\lceil \frac{\rho_\triangleleft(v_i)}{P} \right\rceil + \left\lceil \frac{\rho_\triangleright(v_i)}{P} \right\rceil \tag{3.54}$$

3.7.1 Lifetime Constraints

The afore proposed MIPs can be augmented by the following constraints in order to determine the different lifetime types of the variables.

Lifetime constraints

Additional input:

- None

Additional output:

- Lifetimes $\rho_\circ(v_i)$, $\rho_\triangleleft(v_i)$, and $\rho_\triangleright(v_i)$ of each node $v_i \in V$

Additional constraints:

$$\begin{aligned}
\rho_\circ(v_i) &= \max_{\forall v_j \in V \,:\, e=(v_i,v_j) \in F_\circ(v_i)} \big(\tau(v_j) - \tau(v_i) - w_i\big) && \forall v_i \in V \\
\rho_\triangleleft(v_i) &= \max_{\forall v_j \in V \,:\, e=(v_i,v_j) \in F_\triangleleft(v_i)} \big(\Lambda d_e + \tau(v_j) - \tau(v_i) - w_i\big) && \forall v_i \in V \\
\rho_\triangleright(v_i) &= \max_{\forall v_j \in V \,:\, e=(v_i,v_j) \in F_\triangleright(v_i)} \big(\Lambda d_e + \tau(v_j) - \tau(v_i) - w_i\big) && \forall v_i \in V
\end{aligned}$$

where

$$\begin{aligned}
F_\circ(v_i) &= \{e \in E \mid e = (v_i, v_k) \in E \wedge \Phi d_e = 0 \wedge d_e = 0\} \\
F_\triangleleft(v_i) &= \{e \in E \mid e = (v_i, v_k) \in E \wedge \Phi d_e = 0 \wedge d_e \neq 0\} \\
F_\triangleright(v_i) &= \{e \in E \mid e = (v_i, v_k) \in E \wedge \Phi d_e \neq 0\}
\end{aligned}$$

3.8 Scheduling for Weakly-Programmable Processor Arrays

In the previously introduced scheduling methods, we have developed concepts for global resource allocation in terms of the number of considered processing elements. Simultaneously, local resource constraints, in terms of available functional units within a processing element have been incorporated. Due to the freedom of high-level synthesis, we were able to assume that all other necessary resources, such as a sufficient number of data links between the processing elements or enough (shift) registers, in order to store intermediate data, are available. However, these assumptions do not hold if a fixed processor architecture is considered. Then, not only the number of available processors and functional units is constrained but also the amount of registers and communication channels is limited. In the following, we want to show that almost the same scheduling methods may be applied for such a fixed class of tightly-coupled, programmable processor arrays called *weakly-programmable processor arrays*.

In the next section, we briefly introduce the class of weakly-programmable processor arrays with emphasis on resources that have not been considered during scheduling yet. Afterwards, in Section 3.8.2 and Section 3.8.3 appropriate register and channel constraints are formulated.

3.8.1 Weakly-Programmable Processor Arrays

In [HDK+05, KHKT06b], a new class of tightly-coupled, programmable multiprocessor architectures called *weakly-programmable processor arrays* (WPPA) has been introduced. Such architectures consist of an array of weakly programmable processing elements (WPPE) that may contain subword processing units with a small local memory and a regular interconnect structure. In order to efficiently implement a certain algorithm, each PE may only implement a dedicated functionality. Also, the instruction set is limited and may only be configured at synthesis-time. The PEs are called weakly-programmable because of the control overhead of each PE is optimized to a minimum. For example, there is no support for interrupts and exceptions. An example of such an architecture is shown in Figure 3.27. The massive parallelism might be expressed on different levels: (a) several parallel working processing elements, (b) multiple functional units within one PE, and finally (c) subword parallelism within the PEs. WPPAs can be seen as a compromise between programmability and specialization by exploiting architectures, to realize the full synergy of programmable processor elements and dedicated processing units.

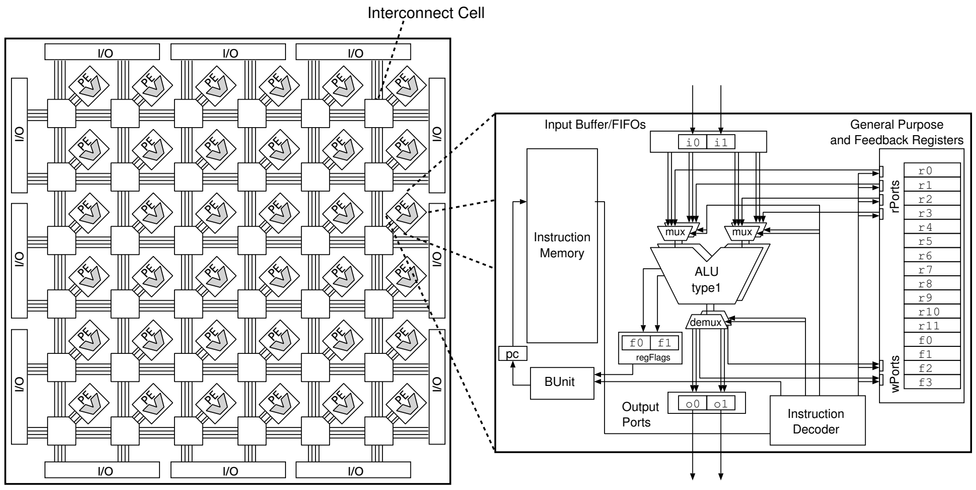


Figure 3.27: Example of a WPPA with parameterizable processing elements (WPPEs). A WPPE consists of several functional units. Few registers and a special data memory exists to store temporary computation results. An instruction sequencer exists as part of the control path, which executes a set of control instructions from a small local program memory. The WPPEs are interconnected via channels among each other.

Apart from the instruction memory within a processor, each processor has three different types of memory elements for data storage. These memory types directly correspond to the different types of lifetimes defined in Section 3.7.

Input buffer: Each input of a processor is buffered by a FIFO. We call this resource type *input buffer* denoted by $r_{\triangleright}$. The maximum length of the buffers $l_{\triangleright}^{max}$ is defined at synthesis-time. The overall number of available input buffers of the same type is denoted by $\alpha(r_{\triangleright})$. In the example of a WPPA in Figure 3.27, two input buffers `i0` and `i1` are depicted.

Register: Each processor has several data registers $r_{\circ}$. These are part of a register file of size $\alpha(r_{\circ})$. (Registers `r0` to `r11` in Figure 3.27.)

Feedback shift register: A *feedback shift register* (FSR) $r_{\triangleleft}$ is an internal buffer for data reuse purposes. It is implemented as a shift register where its output is fed back to its input. The maximum length of the FSRs $l_{\triangleleft}^{max}$ is defined at synthesis-time. At run-time, these registers can have an arbitrary integral length $l_{\triangleleft}$, ranging from one register up to the defined maximum ($1 \le l_{\triangleleft} \le l_{\triangleleft}^{max}$). The overall number of available feedback shift registers is denoted by $\alpha(r_{\triangleleft})$. In Figure 3.27, the FSRs are labeled from `f0` to `f3`. The exact semantics of a feedback shift register is as follows. Let $(a_1, a_2, \ldots, a_{l_\triangleleft})$ denote the values of a feedback shift register of length $l_\triangleleft$. In case of a write operation (write value a_{in}) to the shift register, its contents change as follows.

$$a_{in} \rightarrow (a_1, a_2, \ldots, a_{l_\triangleleft}) \qquad \Rightarrow \qquad (a_{in}, a_1, \ldots, a_{l_\triangleleft - 1})$$

In case of a read operation (read value a_{out}) from the shift register, its contents change as follows.

$$(a_1, a_2, \ldots, a_{l_\triangleleft}) \qquad \Rightarrow \qquad (a_{l_\triangleleft}, a_1, \ldots, a_{l_\triangleleft - 1}) \rightarrow a_{out} = a_{l_\triangleleft}$$

Note, the output ports (o0 and o1 in Figure 3.27) are direct outputs. That is, if data is written to an output port, it is immediately transferred to the input port of a connected target processor. But, in addition, it is stored in an output register and can be accessed just like a general purpose register until another instruction writes data to the same output port. This concept is advantageous, for instance, when propagating variables through the array and simultaneously holding them for a certain time within a WPPE.

3.8.2 Register Constraints

As stated in the last section, only the lifetime of type $\rho_\circ$ variables corresponds to registers that are necessary within one iteration. Hence, it seems obvious that a similar resource constraint as in the case of the functional units can be used. Therefore, new binary variables $y_{i,k,t} \in \{0,1\}$ are defined. Let $g_{k'} \in V_T$ denote a register type[21]. A binary variable $y_{i,k',t}$ equal to the value one denotes that a node (variable) $v_i \in V$ of a given RDG $G = (V, E, D)$ is stored at time step t in a register of type $g_{k'}$. The following constraint ensures that each variable/node v_i is stored exactly once per iteration.

$$\sum_{\forall k' \,:\, (v_i, g_{k'}) \in E_R} \sum_{t=l'_i}^{h'_i} y_{i,k',t} = 1 \qquad \forall v_i \,:\, v_i \in V \wedge F_\circ(v_i) \neq \emptyset \tag{3.55}$$

where

$$l'_i = l_i + \min_{\forall i \,:\, (v_i, r_k) \in E_R} \big(w(v_i, r_k)\big) \qquad \text{and} \qquad h'_i = h_i + \max_{\forall i \,:\, (v_i, r_k) \in E_R} \big(w(v_i, r_k)\big)$$

[21]The breakdown into several types of registers can be useful for *clustered registers banks* [ZLAV03].

Just as a reminder, the resource constraints of the functional units have been formulated as follows.

$$\sum_{\forall i\,:\,(v_i,r_k)\in E_R} \sum_{d=0}^{\delta(r_k)-1} \sum_{\forall \nu\in\mathbb{Z}\,:\,l_i\le t-d-\nu P\le h_i} x_{i,k,t-d-\nu P} \le \alpha(r_k) \qquad \forall r_k\in V_T \quad \forall t\,:\,0\le t\le P-1$$

The register allocation can be stated in a similar fashion

(3.56)

$$\sum_{\substack{\forall i\,:\,(v_i,g_{k'})\in E_R\\ \wedge\ F_\circ(v_i)\neq\emptyset}} \sum_{d=0}^{\rho_\circ(v_i)-1} \sum_{\forall \nu\in\mathbb{Z}\,:\,l'_i\le t-d-\nu P\le h'_i} y_{i,k',t-d-\nu P} \le \alpha(g_{k'}) \qquad \forall g_{k'}\in V_T \quad \forall t\,:\,0\le t\le P-1$$

Since the upper bound of the second sum, the lifetime, is not known a priori, new binary variables $\rho_{i,j}\in\{0,1\}$ are introduced in order to encode the lifetime $\rho_\circ(v_i)\le \rho_{max}(v_i)$.

$$\rho_\circ(v_i)=\sum_{j=1}^{\rho_{max}(v_i)} j\rho_{i,j} \qquad\qquad \sum_{j=0}^{\rho_{max}(v_i)} \rho_{i,j}=1$$

Afterwards, Equation (3.56) can be decomposed as follows.

(3.57)
$$\sum_{\substack{\forall i\,:\,(v_i,g_{k'})\in E_R\\ \wedge\ F_\circ(v_i)\neq\emptyset}} Y_{i,k',t}\le \alpha(g_{k'}) \qquad \forall g_{k'}\in V_T \quad \forall t\,:\,0\le t\le P-1$$

where

(3.58)
$$Y_{i,k',t}=\sum_{d=0}^{\rho_\circ(v_i)-1} \sum_{\forall \nu\in\mathbb{Z}\,:\,l'_i\le t-d-\nu P\le h'_i} y_{i,k',t-d-\nu P}$$

(3.59)
$$Y_{i,k',t}=\begin{cases} \sum\limits_{\forall \nu\in\mathbb{Z}\,:\,l'_i\le t-\nu P\le h'_i} y_{i,k',t-\nu P} & \text{if } \rho_\circ(v_i)=1\\ \sum\limits_{d=0}^{1} \sum\limits_{\forall \nu\in\mathbb{Z}\,:\,l'_i\le t-d-\nu P\le h'_i} y_{i,k',t-d-\nu P} & \text{if } \rho_\circ(v_i)=2\\ \vdots & \vdots\\ \sum\limits_{d=0}^{\rho_{max}(v_i)-1} \sum\limits_{\forall \nu\in\mathbb{Z}\,:\,l'_i\le t-d-\nu P\le h'_i} y_{i,k',t-d-\nu P} & \text{if } \rho_\circ(v_i)=\rho_{max}(v_i) \end{cases}$$

(3.60)

$$Y_{i,k',t} = \begin{cases} Y_{i,k',t,1} = \sum\limits_{\forall v \in \mathbb{Z} \,:\, l_i' \le t - vP \le h_i'} y_{i,k',t-vP} & \text{if } \rho_{i,1} = 1 \\ Y_{i,k',t,2} = \sum\limits_{d=0}^{1} \sum\limits_{\forall v \in \mathbb{Z} \,:\, l_i' \le t-d-vP \le h_i'} y_{i,k',t-d-vP} & \text{if } \rho_{i,2} = 1 \\ \vdots & \vdots \\ Y_{i,k',t,\rho_{max}(v_i)-1} = \sum\limits_{d=0}^{\rho_{max}(v_i)-1} \sum\limits_{\forall v \in \mathbb{Z} \,:\, l_i' \le t-d-vP \le h_i'} y_{i,k',t-d-vP} & \text{if } \rho_{i,\rho_{max}(v_i)} = 1 \end{cases}$$

Since it is assumed that $\rho_{max}(v_i)$ is sufficiently large in Equation (3.60), all of the cases on the right-hand side have fixed bounds and thus these constraints could be generated in a MIP. It remains to figure out, how the appropriate case on the right-hand side is selected and assigned to variable $Y_{i,k',t}$. Equation (3.60) is equal to a *one-hot multiplexer*[22].

Theorem 3.8 (One-hot multiplexer). *A* one-hot multiplexer *is defined by the following equation*

$$y = \begin{cases} a_1 & \text{if } \beta_1 = 1 \\ a_2 & \text{if } \beta_2 = 1 \\ & \vdots \\ a_n & \text{if } \beta_n = 1 \end{cases}$$

where $\beta_1, \beta_2, \ldots, \beta_n \in \{0,1\}$ *and* $\sum\limits_{i=1}^{n} \beta_i = 1$ *is equivalent to the following set of inequalities.*

$$\begin{aligned} y &\le a_1 + M_1^a(1-\beta_1) \\ y &\ge a_1 - M_1^b(1-\beta_1) \\ y &\le a_2 + M_2^a(1-\beta_2) \\ y &\ge a_2 - M_2^b(1-\beta_2) \\ &\vdots \\ y &\le a_n + M_n^a(1-\beta_n) \\ y &\ge a_n - M_n^b(1-\beta_n) \end{aligned}$$

[22]A *one-hot multiplexer* is an n-to-1 multiplexer, which has n binary select signals, where only one select signal stays at value 1 at any time.

Here, M_i^a and M_i^b, $i \in [1..n]$ are big-M constants[23] *that should be large enough to suffice to $y \le a_i + M_i^a$ and $y \ge a_i - M_i^b$.*

Proof. Let us consider one $\beta_j = 1$. Then, all other binary variables have to be zero since $\sum_{i=1}^{n} \beta_i = 1$ must hold. That is, all β_i with $i \neq j$ have to satisfy

$$y \le a_i + M_i^a \qquad \wedge \qquad y \ge a_i - M_i^b$$

which is equal to the definition of the Big-M constants in the theorem. It remains to consider the two inequalities for β_j.

$$y \le a_j + M_j^a(1 - \beta_j)$$
$$y \ge a_j - M_j^b(1 - \beta_j)$$

Since $\beta_j = 1$, it follows $y = a_j$. □

The constants M_i^a and M_i^b, $i \in [1..n]$, should be chosen as small as possible in order to tighten the limits of the search space of the MIP.

Lemma 3.3. *The Big-M constants M_i^a and M_i^b in Theorem 3.8 are sufficiently large if they are chosen to be*

$$M_i^a = \max_{\forall j \in [1..n]}(a_j) - a_i$$
$$M_i^b = a_i - \min_{\forall j \in [1..n]}(a_j)$$

Proof. Let y_{min} and y_{max} be the minimal and maximal values of y. Then each "less than" inequality can be bound by y_{max}

$$y \le a_i + M_i^a(1 - \beta_i) \le y_{max} \qquad \Rightarrow \qquad a_i + M_i^a = y_{max}$$

and each "greater than" inequality has the lower bound y_{min}

$$y \ge a_i - M_i^b(1 - \beta_i) \ge y_{min} \qquad \Rightarrow \qquad a_i - M_i^b = y_{min}$$

With $y_{min} = \min_{\forall j \in [1..n]}(a_j)$ and $y_{max} = \max_{\forall j \in [1..n]}(a_j)$ the proof is concluded. □

[23]*Big-M constants* are used to reformulate a logic or nonconvex constraint in a MIP to a set of constraints describing the same feasible set, using additional constraints and auxiliary binary variables.

If Lemma 3.3 is applied to the afore presented constraints, the following Big-M constants are obtained.

$$M^a_{i,k',t,j} = \max_{\forall l \in [1..\rho_{max}(v_i)]}(Y_{i,k',t,l}) - Y_{i,k',t,j} \qquad \forall j \;:\; 1 \leq j \leq \rho_{max}(v_i)$$
$$M^b_{i,k',t,j} = Y_{i,k',t,j} - \min_{\forall l \in [1..\rho_{max}(v_i)]}(Y_{i,k',t,l}) \qquad \forall j \;:\; 1 \leq j \leq \rho_{max}(v_i)$$

Since $Y_{i,k',t,j}$ and $Y_{i,k',t,l}$ are variables of the MIP, the Big-M constants cannot be computed in advance, with the exception, that upper bounds can be determined for $Y_{i,k',t,j}$ and $Y_{i,k',t,l}$.

Theorem 3.9. *Let j, t, P, l'_i, and h'_i be given integral numbers. Further, let $j \geq 1$, $0 \leq t < P$, and $y_{i,k',t-d-\nu P} \in \{0,1\}$. Then, a variable $Y_{i,k',t,j}$ that is defined by*

$$Y_{i,k',t,j} = \sum_{d=0}^{j-1} \sum_{\forall \nu \in \mathbb{Z} \;:\; l'_i \leq t-d-\nu P \leq h'_i} y_{i,k',t-d-\nu P}$$

has

$$Y_{i,k',t,j} \leq j$$

as upper bound.

Proof. Since Equation (3.55) must be satisfied, the inner summation can be at most one. Thus, overall the double sum can be at most j. □

The upper bound $\rho_{max}(v_i)$ of the lifetime should be chosen as small as possible and calculated for each node v_i individually in order to reduce the number of inputs of the one-hot multiplexers and the number of variables in the MIP. A large enough $\rho_{max}(v_i)$ can be estimated by considering the worst case in Equation (3.51).

$$\rho_{max}(v_i) = \max_{\forall v_j \in V \;:\; e=(v_i,v_j) \in F_\circ(v_i)} \Big(\max(\tau(v_j)) - \min(\tau(v_i)) - \min(w_i)\Big) \tag{3.61}$$
$$= \max_{\forall v_j \in V \;:\; e=(v_i,v_j) \in F_\circ(v_i)} \Big(h_j - l_i - \min(w_i)\Big)$$

In summary, the register constraints can be incorporated into the MIP as follows.

Register constraints

Additional input:

- The set of resource types V_T of the resource graph G_R is extended by register types $g_{k'}$
- Allocation $\alpha(g_{k'})$ for all $g_{k'} \in V_T$

Additional output:

- Register type binding if a valid schedule can be obtained for the given register allocation

Additional constraints:

$$\sum_{\forall k' \,:\, (v_i, g_{k'}) \in E_R} \sum_{t=l'_i}^{h'_i} y_{i,k',t} = 1 \qquad \forall v_i \,:\, v_i \in V \wedge F_\circ(v_i) \neq \emptyset$$

$$\rho_\circ(v_i) = \sum_{j=1}^{\rho_{max}(v_i)} j\rho_{i,j} \qquad \forall v_i \in V$$

$$\sum_{j=0}^{\rho_{max}(v_i)} \rho_{i,j} = 1 \qquad \forall v_i \in V$$

$$\rho_\circ(v_i) = \sum_{j=1}^{\rho_{max}(v_i)} j\rho_{i,j} \qquad \forall v_i \in V$$

$$\sum_{\substack{\forall i \,:\, (v_i, g_{k'}) \in E_R \\ \wedge\, F_\circ(v_i) \neq \emptyset}} Y_{i,k',t} \leq \alpha(g_{k'}) \qquad \begin{array}{l} \forall g_{k'} \in V_T \\ \forall t \,:\, 0 \leq t \leq P-1 \end{array}$$

$$Y_{i,k',t} \leq Y_{i,k',t,j} + M^a_{i,k',t,j}(1-\rho_{i,j}) \qquad \begin{array}{l} \forall g_{k'} \in V_T \\ \forall t \,:\, 0 \leq t \leq P-1 \\ \forall i \,:\, (v_i, g_{k'}) \in E_R \\ \qquad \wedge\, F_\circ(v_i) \neq \emptyset \\ \forall j \,:\, 1 \leq j \leq \rho_{max}(v_i) \end{array}$$

$$Y_{i,k',t} \geq Y_{i,k',t,j} - M^b_{i,k',t,j}(1-\rho_{i,j}) \qquad \begin{array}{l} \forall g_{k'} \in V_T \\ \forall t \,:\, 0 \leq t \leq P-1 \\ \forall i \,:\, (v_i, g_{k'}) \in E_R \\ \qquad \wedge\, F_\circ(v_i) \neq \emptyset \\ \forall j \,:\, 1 \leq j \leq \rho_{max}(v_i) \end{array}$$

⇝

$\rightsquigarrow$

$$Y_{i,k',t,j} = \sum_{d=0}^{j-1} \sum_{\forall \nu \in \mathbb{Z} \,:\, l'_i \le t-d-\nu P \le h'_i} y_{i,k',t-d-\nu P} \qquad \begin{array}{l} \forall g_{k'} \in V_T \\ \forall t \,:\, 0 \le t \le P-1 \\ \forall i \,:\, (v_i, g_{k'}) \in E_R \\ \quad\;\; \wedge\; F_\circ(v_i) \neq \emptyset \\ \forall j \,:\, 1 \le j \le \rho_{max}(v_i) \end{array}$$

where $\rho_{i,j} \in \{0,1\}$ and

$$\begin{aligned} \rho_{max}(v_i) &= \max_{\forall v_j \in V \,:\, e=(v_i,v_j) \in F_\circ(v_i)} \left(h_j - l_i - \min(w_i) \right) \\ F_\circ(v_i) &= \{ e \in E \mid e = (v_i, v_k) \in E \;\wedge\; \Phi d_e = 0 \;\wedge\; d_e = 0 \} \\ M^a_{i,k',t,j} &= \max_{\forall l \in [1..\rho_{max}(v_i)]} (Y_{i,k',t,l}) - Y_{i,k',t,j} \\ M^b_{i,k',t,j} &= Y_{i,k',t,j} - \min_{\forall l \in [1..\rho_{max}(v_i)]} (Y_{i,k',t,l}) \\ l'_i &= l_i + \min_{\forall i \,:\, (v_i,r_k) \in E_R} \left(w(v_i, r_k) \right) \\ h'_i &= h_i + \max_{\forall i \,:\, (v_i,r_k) \in E_R} \left(w(v_i, r_k) \right) \end{aligned}$$

3.8.3 Channel Constraints

The channels or rather the I/O ports of a processor are considered as just another resource type, which is shared among the operations it can be accessed by. Channels or rather the input buffers of a WPPA are configured as FIFOs (cf. Figure 3.27). Consider, an algorithm with a corresponding space mapping. Let d_e be a data dependency that results in a *data link* between two processors P_a and P_b. If a value is written by processor P_a to the output port associated with the data link to P_b, the value is instantaneously written to the channel so that it might be read from the input port (buffer) of P_b in the next cycle. In Figure 3.28, the channel is depicted by the horizontal line between the processors P_a and P_b, the tail of the arrow denotes an output port of processor P_a, and the arrowhead an input buffer of processor P_b.

Definition 3.13 (Data link). *Denote a DPRA with RDG $G = (V, E, D)$ and space mapping, defined by matrix Φ. A data dependency d_e of the algorithm is called* data link *if it satisfies $\Phi d_e \neq 0$. That means, the data dependency results in a connection between* *two different processors. To each data link $l = (p, c)$ a source node $p \in V$ and a target node $c \in V$ is associated. A data-link is also called* inter-processor data dependency.

In the following it is assumed that all data links are *unit data links*.

Definition 3.14 (Unit data link)**.** *Let the processor space $\mathcal{P}$ be a dense space in $\mathbb{Z}^n$. Then, a* unit data link *expresses an inter-processor data dependency between two processors $P_a \in \mathcal{P}$ and $P_b \in \mathcal{P}$ such that $P_a - P_b$ is a unit vector.*

The assumption of unit data links is no restriction since several techniques exist to obtain unit data links. For instance, at algorithmic level there exist localization methods [TR91] to decompose data dependencies into a sequence of shorter ones. At architecture level, similar methods for routing have been developed [HT05, SM06d, WKTH08c]. In Figure 3.29(a), a schematic diagram of a 4×4 processor array with a dependency in direction $(2\ 1)^{\mathrm{T}}$ is shown. This dependency is decomposed into three unit data links. The possible variants are depicted in Figure 3.29(b)-(d). It can be seen that the number of necessary vertical and horizontal channels is always the same. The basis of this behavior is the regularity of the considered algorithms. From the hardware perspective, the advantages of unit data links is their regularity and their shortness, which may lead to high clock frequencies. From the programming perspective, unit data dependencies between processors have the advantage that input and output variables directly correspond to one processor that produces the data and one processor that consumes the data (see Figure 3.29(a)). Hence, they can be considered during the allocation and scheduling phase. If the data is routed through several other processors on the way to their destination, the allocation of processor ports can only be indirectly obtained from the decomposition of the data dependencies. However, this "routing freedom" does not reduce the number of allocated channels as illustrated in the above example.

In contrast to the occupancy of a functional unit r_k for a specific time $w(v_i, r_k)$, the situation for communication resources is different since production and consumption of a data item is not directly coupled. Consider, for instance, two processors P_a and P_b and a variable that should be written to an output port of P_a, transfered over a channel to an input buffer of P_b, and is processed later in time.

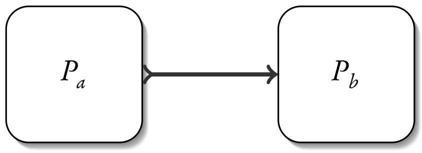


Figure 3.28: Illustration of the communication model of two processors P_a and P_b.

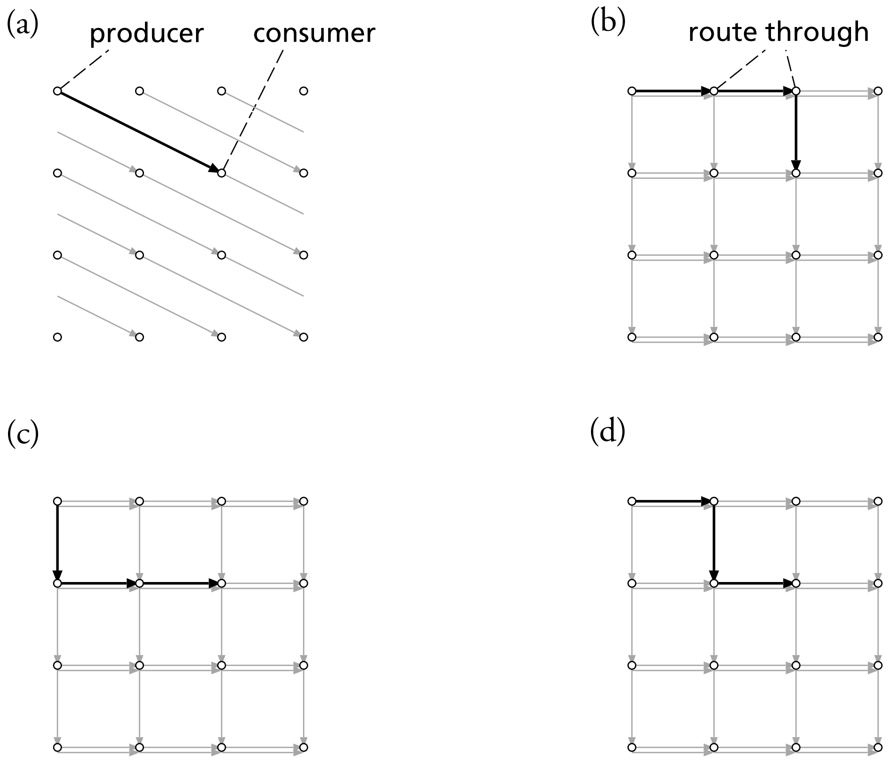


Figure 3.29: In (a), a dependency in direction $(2\ 1)^T$ is shown. Three possible decompositions (routings) are shown in (b)-(d).

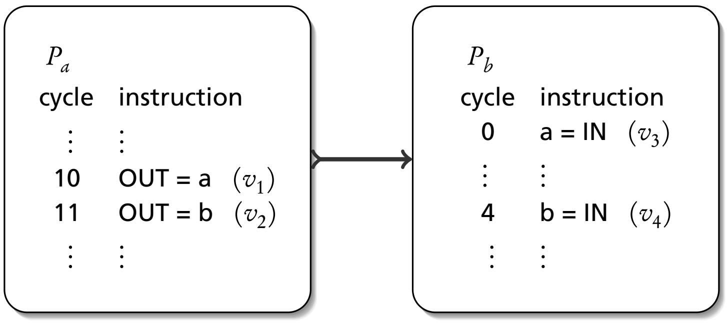
Here, the problem is that if several data dependencies have to share the same input buffer, data might be processed out of order.
In general, two cases can be distinguished:

Case 1: If a dedicated input buffer is available for each inter-processor data dependency, the conflict will not arise. The FIFO semantics holds since both processors work iteratively with the same iteration interval. Thus, the first processor produces every iteration one data item and the second processor consumes every iteration one data item.

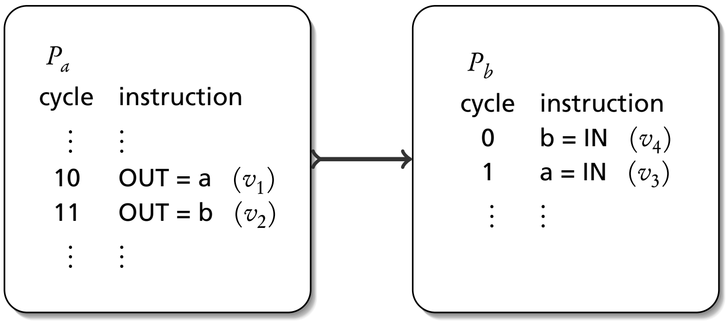
Case 2: If two or more inter-processor data dependencies have to share the same input buffer, it could happen that the FIFO semantics does not hold any longer. That is, the communication pattern might be out of order. For illustration, two examples are considered.

Example 1:
Two processors P_a and P_b are connected via one channel. In each processor, a program is executed reiteratively. The cycle numbers within the processors

are not global, that is, they vary from each other by an offset defined by the schedule Λ. First, in cycle 10, a variable a is written by processor P_a to its output port. After this, variable b is written to the same output port. Some time later, the data is read in the same order (first a then b) by processor P_b. Hence, in this example, the FIFO semantics hold. Note that the variables in parentheses right to the instructions denote an RDG node v_i that is associated with the operation. Consequently, the cycle number corresponds to $\tau(v_i)$. In the example below, there exist two data links: (v_1, v_3) and (v_2, v_4).



Example 2:
This example is similar to the first one, with the same data links, (v_1, v_3) and (v_2, v_4). But, the reading order of processor P_b is reversed, so that variable b is read before variable a. Thus, the FIFO semantics is violated.



To distinguish from related work: In [SM06d, WKTH08c], primarily the routing of data links has been discussed. Furthermore, it is assumed that enough random access memory is available within a PE in order to store received data. But, the order of memory accesses is not accounted for. In contrast, we do not consider the spatial routing but the right communication order when dealing with FIFOs as input buffers such as in case of WPPAs. In order to ensure the correct data flow, the semantics of a FIFO has to be incorporated into the mixed integer program. For this purpose, additional constraints have to be developed in the following.

At first, the afore discussed examples are considered, where only two data links share the same channel (input buffer). The write times of processor P_a are denoted by $\tau'(v_1)$ and $\tau'(v_2)$, whereas the read times of processor P_b are denoted by $\tau(v_3)$ and $\tau(v_4)$. The time steps $\tau'(v_i)$ are defined as $\tau'(v_i) = \tau(v_i) + w_i$. In this case, the following implications must hold in order to guarantee the FIFO semantics.

$$\begin{aligned} \tau'(v_1) < \tau'(v_2) \quad &\Rightarrow \quad \tau(v_3) < \tau(v_4) \\ \tau'(v_2) < \tau'(v_1) \quad &\Rightarrow \quad \tau(v_4) < \tau(v_3) \end{aligned}$$

Generally, if an input buffer is shared by several data links, for each pair the afore stated implications have to be satisfied.

Definition 3.15 (FIFO semantics). *Consider a system consisting of a source processor, a target processor, and a channel in between. The channel can buffer data since the time steps of data production and consumption can be different. Let an RDG $G = (V, E, D)$ be given. Denote $\tau'(p_1), \ldots, \tau'(p_n)$ as n time steps of data production at the source processor, with $p_1, \ldots, p_n \in V_{out} \subseteq V$. At each time step only one piece of data can be written to an output port of the source processor, $\tau'(p_i) \neq \tau'(p_j)$, $\forall i, j \in [1..n] \wedge i \neq j$. Further, let $\tau(c_1), \ldots, \tau(c_n)$, with $c_1, \ldots, c_n \in V_{in} \subseteq V$, be the n time steps, where the data produced at time steps $\tau'(p_i)$, are consumed at the target processor. It is also assumed that at each time step only one piece of data can be read from the input port of the target processor, $\tau(c_i) \neq \tau(c_j)$, $\forall i, j \in [1..n] \wedge i \neq j$. Causality is assured by the data dependency constraint as defined in Equation (3.13) on page 73. Then, the channel has* FIFO semantics *if the following implications are satisfied.*

$$\begin{aligned} \tau'(p_i) < \tau'(p_j) \quad &\Rightarrow \quad \tau(c_i) < \tau(c_j) \qquad \forall i, j \in [1..n] : i \neq j \\ \tau'(p_j) < \tau'(p_i) \quad &\Rightarrow \quad \tau(c_j) < \tau(c_i) \qquad \forall i, j \in [1..n] : i \neq j \end{aligned} \tag{3.62}$$

Theorem 3.10 (FIFO constraint). *The FIFO semantics according to Definition 3.15 and the implications in Equation (3.62) are equivalent to the following constraints*

$$\left.\begin{aligned} \tau'(p_i) - \tau'(p_j) \; &< \; (1 - \beta_{i,j})(h'(p_i) - l'(p_j) + 1) \\ \tau(c_i) - \tau(c_j) \; &< \; (1 - \beta_{i,j})(h(c_i) - l(c_j) + 1) \end{aligned}\right\} \begin{aligned} &\forall i, j \in [1..n] : i \neq j \\ &\wedge\, p_i, p_j \in V_{out} \\ &\wedge\, c_i, c_j \in V_{in} \end{aligned}$$

where $\beta_{i,j} \in \{0, 1\}$ are binary variables, $l(c_j)$ denotes the earliest and $h(c_i)$ the latest possible starting time of the respective nodes. $l'(p_j)$ denotes the earliest and $h'(p_i)$ the latest possible finishing time of node p_j and node p_i.

Proof. Since the FIFO semantics is defined for each data link pair separately, without loss of generality, only two data links that share the same channel are considered in the following. Let two binary variables $\beta_p \in \{0,1\}$ and $\beta_c \in \{0,1\}$ be associated to the conditions within the implications.

$$\left.\begin{array}{lcl} \tau'(p_1) < \tau'(p_2) & \Rightarrow & \beta_p = 1 \\ \tau'(p_1) > \tau'(p_2) & \Rightarrow & \beta_p = 0 \end{array}\right\} \Leftrightarrow \tau'(p_1) < \tau'(p_2) + M_p(1-\beta_p) \tag{3.63}$$

$$\left.\begin{array}{lcl} \tau(c_1) < \tau(c_2) & \Rightarrow & \beta_c = 1 \\ \tau(c_1) > \tau(c_2) & \Rightarrow & \beta_c = 0 \end{array}\right\} \Leftrightarrow \tau(c_1) < \tau(c_2) + M_c(1-\beta_c) \tag{3.64}$$

M_p and M_c are big-M constants that should be chosen as small as possible. Since the implications in Equation (3.63) and in Equation (3.64) are of the same type, only the last one is considered for finding an adequate upper bound of M_c. In order to obey the implications in Equation (3.64), it is sufficient if M_c equals $\tau(c_1) - \tau(c_2) + 1$. Since $\tau(c_1)$ and $\tau(c_2)$ are variables of the MIP, their predetermined ASAP (l) and ALAP (h) times are used for the estimation.

$$M_c = \max\{\tau(c_1) - \tau(c_2)\} + 1 = h(c_1) - l(c_2) + 1$$

Analogous M_p can be bounded by

$$M_p = \max\{\tau'(p_1) - \tau'(p_2)\} + 1 = h'(p_1) - l'(p_2) + 1$$

The FIFO semantics is considered again.

$$\begin{array}{lclclcl} (\tau'(p_1) < \tau'(p_2) & \Rightarrow & \tau(c_1) < \tau(c_2)) & \Leftrightarrow & (\beta_p = 1 & \Rightarrow & \beta_c = 1) \\ (\tau'(p_2) < \tau'(p_1) & \Rightarrow & \tau(c_2) < \tau(c_1)) & \Leftrightarrow & (\beta_p = 0 & \Rightarrow & \beta_c = 0) \end{array}$$

In the first implication, β_p and β_c are both equal to one and in the second implication both are equal to zero. Consequently, the binary variables are in logical conjunction, and thus they can be equated to $\beta = \beta_p = \beta_c$.

$$\begin{array}{r} \tau'(p_1) < \tau'(p_2) + (h'(p_1) - l'(p_2) + 1)(1-\beta) \\ \tau(c_1) < \tau(c_2) + (h(c_1) - l(c_2) + 1)(1-\beta) \end{array} \qquad \square$$

In order to formulate appropriate resource constraints for handling the number of available channels, the RDG edges that are associated with input and output ports have to be determined. The set E_{IO} contains all edges $e \in E$ for which $\Phi d_e \neq 0$.

$$\begin{array}{c} E_{IO} = \{e \in E \mid \Phi d_e \neq 0\} \\ \Rightarrow V_{out} = \{p \mid (p,c) \in E_{IO}\} \quad \text{and} \quad V_{in} = \{c \mid (p,c) \in E_{IO}\} \end{array}$$

It should be noted that input and output variables are not treated separately since data dependencies that leave the iteration space (array) at the borders can be used instead. That is, an input or output port is either occupied by a data link (inter processor communication) or by global I/O.

The resource graph is extended by further resources, namely each input port instance $p_{k^{in}}$ and output port instance $p_{k^{out}}$ of a processor is added to the set V_T. It is important to consider each port separately in order to have a unique binding from nodes to ports, because nodes that are bound to the same input port must have a common FIFO constraint.

A binary variable $\beta_{i,k^{out},t} \in \{0,1\}$ denotes that node $p_i \in V_{out}$ occupies the output port $p_{k^{out}} \in V_T$ at time step t. Analogous, a binary variable $\beta_{i,k^{in},t} \in \{0,1\}$ denotes that node $c_i \in V_{in}$ occupies the input port $p_{k^{in}} \in V_T$ at time step t.

Consider an edge (p_i, c_i) that reflects an inter-processor data dependency from processor P_a to processor P_b. At time step $\tau'(p_i)$, processor P_a has to write data to its output port, and at time step $\tau(c_i)$, processor P_b reads the data from its input port. For all nodes p_i that belong to set V_{out}, the following constraints are formulated.

$$\sum_{\forall k^{out}\,:\,(p_i, p_{k^{out}}) \in E_R} \; \sum_{t=l'(p_i)}^{h'(p_i)} t\beta_{i,k^{out},t} = \tau'(p_i) \qquad \forall p_i \in V_{out} \tag{3.65}$$

$$\sum_{\forall k^{out}\,:\,(p_i, p_{k^{out}}) \in E_R} \; \sum_{t=l'(p_i)}^{h'(p_i)} \beta_{i,k^{out},t} = 1 \qquad \forall p_i \in V_{out} \tag{3.66}$$

The constraint in Equation (3.65) ensures that an output port of the processor is exactly occupied at the finishing time of the operation associated with node p_i. The constraint in Equation (3.66) denotes that a node p_i should be executed every time on the same output port instance and exactly once per iteration.

Similar constraints have to be considered for the input ports. Here, the only difference is that in the constraint in Equation (3.67) the starting time of node c_i is synchronized with the port access.

$$\sum_{\forall k^{in}\,:\,(c_i, p_{k^{in}}) \in E_R} \; \sum_{t=l(c_i)}^{h(c_i)} t\beta_{i,k^{in},t} = \tau(c_i) \qquad \forall c_i \in V_{in} \tag{3.67}$$

$$\sum_{\forall k^{in}\,:\,(c_i, p_{k^{in}}) \in E_R} \; \sum_{t=l(c_i)}^{h(c_i)} \beta_{i,k^{in},t} = 1 \qquad \forall c_i \in V_{in} \tag{3.68}$$

The FIFO constraints must be generated for all nodes that share the same output or input buffer. Since the first time, the binding of nodes to ports is determined, is during the scheduling, the FIFO constraints have to be modified in such a way that they affect only nodes grouped to the same port. Hence, the idea is to generate a set of FIFO constraints for each port that contains binary variables $\beta_{i,k^{out},t}$ and $\beta_{j,k^{out},t}$

or $\beta_{i,k^{in},t}$ and $\beta_{j,k^{in},t}$ as a selector to indicate if the constraint is active or if it has no impact.

$$\begin{aligned}\tau'(p_i)-\tau'(p_j) &< (1-\beta_{i,j})(h'(p_i)-l'(p_j)+1)\\ &\quad + M_p(2-\sum_{t=l'(p_i)}^{h'(p_i)} \beta_{i,k^{out},t} && \forall (p_i,c_i),(p_j,c_j)\in E_{IO} : p_i \neq p_j\\ &\quad && \forall k^{out} : (p_i,p_{k^{out}})\in E_R\\ &\quad - \sum_{t=l'(p_j)}^{h'(p_j)} \beta_{j,k^{out},t}) && \wedge (p_j,p_{k^{out}})\in E_R\end{aligned}$$

$$\begin{aligned}\tau(c_i)-\tau(c_j) &< (1-\beta_{i,j})(h(c_i)-l(c_j)+1)\\ &\quad + M_c(2-\sum_{t=l(c_i)}^{h(c_i)} \beta_{i,k^{in},t}) && \forall (p_i,c_i),(p_j,c_j)\in E_{IO} : c_i \neq c_j\\ &\quad && \forall k^{in} : (c_i,p_{k^{in}})\in E_R\\ &\quad - \sum_{t=l(c_j)}^{h(c_j)} \beta_{j,k^{in},t}) && \wedge (c_j,p_{k^{in}})\in E_R\end{aligned}$$

It can be easily verified that suited lower bounds for the big-M constants are defined by $M_p = h'(p_i) - l'(p_j)$ and $M_c = h(c_i) - l(c_j)$. Then, in summary, the afore introduced MIPs can be augmented by the following constraints in order to handle channel constraints.

Channel constraints

Additional input:

- Set V_T of resource types augmented by all input and out port instances

Additional output:

- Binding of input and output variables to processor ports if a valid schedule can be obtained for the given channel constraints

Additional constraints:

$$\sum_{\forall k^{out}\,:\,(p_i,p_{k^{out}})\in E_R}\ \sum_{t=l'(p_i)}^{h'(p_i)} t\beta_{i,k^{out},t} = \tau'(p_i) \qquad \forall p_i \in V_{out}$$

$$\sum_{\forall k^{out}\,:\,(p_i,p_{k^{out}})\in E_R}\ \sum_{t=l'(p_i)}^{h'(p_i)} \beta_{i,k^{out},t} = 1 \qquad \forall p_i \in V_{out}$$

$$\sum_{\forall k^{in}\,:\,(c_i,p_{k^{in}})\in E_R}\ \sum_{t=l(c_i)}^{h(c_i)} t\beta_{i,k^{in},t} = \tau(c_i) \qquad \forall c_i \in V_{in}$$

$$\sum_{\forall k^{in}\,:\,(c_i,p_{k^{in}})\in E_R}\ \sum_{t=l(c_i)}^{h(c_i)} \beta_{i,k^{in},t} = 1 \qquad \forall c_i \in V_{in}$$

$$\begin{aligned}\tau'(p_i)-\tau'(p_j) \;&<\; (1-\beta_{i,j})(h'(p_i)-l'(p_j)+1)\\ &\quad + M_p(2-\sum_{t=l'(p_i)}^{h'(p_i)}\beta_{i,k^{out},t} \qquad \forall (p_i,c_i),(p_j,c_j)\in E_{IO}\ :\ p_i\neq p_j\\ &\quad -\sum_{t=l'(p_j)}^{h'(p_j)}\beta_{j,k^{out},t}) \qquad \forall k^{out}\ :\ (p_i,p_{k^{out}})\in E_R \wedge (p_j,p_{k^{out}})\in E_R\end{aligned}$$

$$\begin{aligned}\tau(c_i)-\tau(c_j) \;&<\; (1-\beta_{i,j})(h(c_i)-l(c_j)+1)\\ &\quad + M_c(2-\sum_{t=l(c_i)}^{h(c_i)}\beta_{i,k^{in},t}) \qquad \forall (p_i,c_i),(p_j,c_j)\in E_{IO}\ :\ c_i\neq c_j\\ &\quad -\sum_{t=l(c_j)}^{h(c_j)}\beta_{j,k^{in},t}) \qquad \forall k^{in}\ :\ (c_i,p_{k^{in}})\in E_R \wedge (c_j,p_{k^{in}})\in E_R\end{aligned}$$

where

$$E_{IO}=\{e\in E \mid \Phi d_e\neq 0\} \qquad V_{out}=\{p \mid (p,c)\in E_{IO}\}$$
$$V_{in}=\{c \mid (p,c)\in E_{IO}\} \qquad M_p=h'(p_i)-l'(p_j) \qquad M_c=h(c_i)-l(c_j)$$

3.9 Summary

After elaboration and differentiation from related work, a continuous and modular scheduling concept has been formulated in this chapter. Since in data flow dominated areas as, for instance, multimedia and other digital signal processing, throughput is often the major optimization goal, exact scheduling methods based on mixed integer programming have been chosen. Starting from well-known results on linear scheduling, further constraints were developed in a step-by-step procedure. The constraints can be classified as global allocation, which includes projection, partitioning, and multi-level partitioning, as well as local allocation, consisting of the number of functional units, module selection, and functional pipelining, and can serve as modular extensions to MIPs. The proposed methods allow the simultaneous optimization of schedules on different levels within the processors and at processor array level, with respect to the aforementioned resource constraints. Particular worthy of mention, are the novel serialization concepts for scheduling partitioned algorithms for arbitrary parallelotope-shaped tiles.

Furthermore, for the first time, an exact method has been derived that regards software pipelining for programs with multi-dimensional data flow in consideration of iteration dependent as well as run-time dependent conditions and awareness of limited resources at different levels (for instance, number of functional units within a processor and the total number of processors). The concept was enabled by the introduction of the so-called AND-XOR-trees (AXT) that express the relationship between operations. Based on the concept of AXTs, new MIP constraints have been presented.

The modularity of the proposed concept has been utilized in order to incorporate further constraints for the number of available registers and communication channels between processors that are necessary to target a special class of tightly-coupled, programmable processor arrays, called weakly-programmable processor arrays.

Chapter 4

Target Code Generation

In this chapter, it is described how the information obtained during the allocation and scheduling phases can be utilized in order to generate code for a given target architecture. As discussed earlier, these target architectures can be either dedicated hardware accelerators or weakly-programmable processor arrays. In the first case, the question is, how to generate a synthesizable hardware description that can be further refined by standard vendor tools for FPGA or ASIC designs. In case of WP-PAs, assembly code for each processor and reconfiguration data for the interconnect between the processors has to be generated.

For a better understanding of the entire design methodology, the PARO synthesis tool, which has been developed in the course of this thesis, is introduced in this chapter. Afterwards, the code generation for both dedicated hardware accelerators and WPPAs are briefly described. As an introduction, the chapter starts with an overview of related work.

The major contributions of this chapter can be summarized as follows:

- Related work in the area of high-level synthesis tools and embedded processor arrays is given in Section 4.1 with an emphasis on recent approaches for accelerating loop programs.
- The entire design flow of the PARO synthesis tool is presented at a glance in Section 4.2. Its uniqueness is the applicability to target different architectures, which are classically tackled with totally different design methods.
- A systematic approach for the synthesis of dedicated hardware accelerators consisting of several processor elements, interconnect, and control structures, is presented in Section 4.3. For the first time, a method for the code generation for tightly-coupled, programmable processor arrays is described (see Sec-

tion 4.4). The approach starts from almost the same allocation and scheduling information as the synthesis method for dedicated hardware accelerators.

4.1 Related Work

This section about related work considers two aspects. The first part is concerned with the synthesis of processor arrays and current high-level synthesis tools from industry as well as approaches from academia. The second part examines mapping approaches for coarse-grained reconfigurable architectures and embedded multi-processor arrays.

4.1.1 High-Level Synthesis Approaches and Tools

In the following, related work in the field of systolic and VLSI processor arrays is reviewed first. Subsequently, several approaches that specially focus on the synthesis of computationally intensive programs are discussed. Finally, selected general purpose high-level synthesis approaches, based on C, C++, and SystemC, are presented.

4.1.1.1 Design Tools for Systolic and VLSI Processor Arrays

There is a long history in the systematic design of systolic and VLSI processor arrays [Pla99]. Many authors present tools and methods for the design of such arrays. The following list of older approaches is neither exhaustive nor discussed in detail. Examples of tools and approaches for the design of processor arrays include DIASTOL [FGQ86], ADVIS [Mol87], DECOMPOSER [HOI88], VACS [KJ88], SYSTARS [Omt88], SDEF [EC89], HiFi [AD88], PRESAGE [Don88, Don92], MSSM [HH92], VASS [YCJ96], and DG2VHDL [SM00]. All the afore mentioned tools consider algorithms only at iteration level and not the functionality within an iteration. The approaches neither start from a well defined input language nor provide the possibility to synthesize a hardware description. Furthermore, only the design of *full size* processor arrays is possible, that means, the size of the processor array depends on the problem size.

Approaches that employ a programming language as input specification are presented in [OF88,GMQS89,LMQ91,NGCD91,ATT92,TA93,Tei93,Bur94,Veh95, SFS$^+$95]. RAB [OF88] is a programming tool for the design of bit level processor arrays starting from an input description in the language C. The authors in [GMQS89] present the Alpha Du Centaur environment for the design of parallel regular algorithms. Here, a proprietary language named Alpha [LMQ91] is used as input and interpretable LUSTRE code [CPHP87] is generated as output.

In [NGCD91], the authors present Cathedral, a high-level synthesis script for the design of DSP applications. In order to describe the behavior of an input algorithm the language Silage [Hil85] is used. COMPAR [ATT92, TA93] and CASPAR [Tei93] are compilers for the design of application-specific processor arrays. As design entry a subset of the parallel programming language UNITY [CM88] is considered. Starting from a piecewise linear algorithm formulated in this input language, Teich [Tei93] presents an entire design trajectory down to an abstract array description. The main steps in his proposed design flow include *localization*, *control generation*, and *hardware matching*. However, an appropriate schedule has to be defined manually and the abstract array description has to be refined by hand in order to obtain a synthesizable array structure. In [Bur94], Burleson present the ARREST environment for the design of fine-grained processor arrays. The arrays are specified by a structural hardware description language that contains constructs for the repetitive generation of regular structures. DECOMP [Veh95] and the LISA design environment [SFS+95] use Pascal as input language and generate an EDIF netlist as output. However, the generality and applicability of the proposed environments is questionable since the approach is only applied to a single simple algorithm.

4.1.1.2 Tools and Approaches for the Acceleration of Loop Programs

The following approaches all describe work based on loop parallelization techniques in the polytope model [Len93, Fea96], which is similar to our methodology. They aim at the synthesis of one dedicated hardware accelerator for a given loop program.

MMAlpha. MMAlpha [GQR03] is a transformation tool-box developed at IRISA in Rennes, France. Programs written in the Alpha language are considered [LMQ91] as design entry. A program written in Alpha describes a system of recurrence equations. MMAlpha consists of a set of Mathematica packages [Wol96b] enabling a design trajectory down to synthesizable VHDL code. Scheduling techniques for both, one and multi-dimensional time, are available [Fea92a, Fea92b]. However, MMAlpha does not contain any architectural modeling possibilities as for instance, multi-cycle operations, module selection, or resource constrained scheduling.

PICO-NPA. The PICO (program in, chip out) project developed at the Hewlett-Packard Laboratories [SAR+00b, SAR+00a, KAS+02] automates the design of application-specific embedded computer systems. It uses an architecture template that consists of one VLIW processor and an optional non-programmable accelerator (NPA). The second part is referred to as PICO-NPA [SAM+02] and aims at the generation of hardware accelerators for computationally intensive functions, expressed as loop nests in C. PICO-NPA includes transformations for the minimization of global memory accesses (instantiation of local registers and exploitation of uniform data dependencies) and rectangular tiling. In case of tiling, the loop nest is extended by one outer loop that is implemented in software on the host processor, and the inner loops are synthesized as an NPA. Scheduling is performed in a hierarchical fashion. First, a schedule between the iterations is determined, before the offsets for each operation within the loop body are computed subsequently. The research project has been commercialized by Synfora [Syn09] in the product PICO Express.

SA-C Compiler. The SA-C compiler [RCP+01, NBD+03] within the Cameron project provides a mapping method from single assignment C (SA-C) source code to executable FPGA configurations. SA-C is a variant of the programming language C. It exploits instruction-level and loop-level parallelism. The compiler focuses on image processing application. SA-C programs are compiled into data flow graphs. The method tries to maximize locality by reusing data and common subexpressions. The final translation into VHDL is based on a template library.

Compaan/Laura. The Compaan tool translates Matlab applications into a *polyhedral reduced dependence graph* [KRD00]. For that, Compaan performs an exact analysis of the data dependencies by taking the instances of arrays into account. The method is based on *parametric integer programming* [Fea88]. Once obtained, the reduced dependence graph is converted into a Kahn process network (KPN) [Kah74, LP95]. Such a network consists of nodes, executing processes, and channels with FIFO semantics in between. The Laura tool [ZSKD03, SZT+04] generates synthesizable VHDL code from the KPN specification. Here, pre-synthesized coarse-grained IP cores should be used as process nodes in order to keep the number of FIFOs in the design small. Out-of-order communication is handled by the consideration of reordering memories [TKD05]. In conclusion, the Compaan/Laura approach targets communicating processes/loops rather than exploiting software pipelining and instruction-level parallelism of a single loop nest.

ROCCC. In [GN06, BGN06], the authors present the Riverside Optimizing Compiler for Configurable Computing (ROCCC). ROCCC is a compiler for FPGAs based on SUIF2 [SUI] and Machine-SUIF [SH02], which employs many conventional transformations such as loop unrolling or scalar replacement. As input a subset (no pointers, no break or continue statements, only static for loops, linear indexing functions, etc.) of C is used. ROCCC has a strong emphasis on filter and image processing algorithms with window operators. Here the authors developed the concept of so-called *smart buffers* [GBN04] in order to decouple memory access from computations and to increase data reuse. For address generation, a parameterizable

VHDL template of a finite state machine is used. Nevertheless, the concept is not able to store entire rows of an image, as required by windowing algorithms (for instance, two-dimensional convolution). Thus, some main memory addresses have to be accessed several times [DZD^{+}08]. This effect is tolerable since the authors target a SGI RASC RC100 blade [SGI09] with a high-speed NUMAlink direct memory connection. ROCCC is also able to synthesize full-size systolic arrays [BN08] but not partitioned arrays.

The authors in [BRS07] present a framework for the mapping of perfectly nested loops with uniform dependencies onto the FPGA extension of the Cray XD1 high performance computer. The approach considers LSGP partitioning and purely linear scheduling without any affine part. The degree of automation seems to be in a very early state since matrix multiplication is the only example discussed.

The authors in [CHM08] and [KLB08] present approaches that are based on a VLIW architecture template and standard compilers.

Bluespec. Bluespec [Blu09] performs hardware synthesis, starting from an operation centric description, where the behavior of a system is described as a collection of atomic operations in the form of rules [HA00]. Such a rule is typically defined by a predicate condition and an effect on the state of the system. The main advantage of considering atomic operations is that each rule can be formulated as if the rest of the system is static. Originally, Bluespec's design entry was done in form of a Haskell like syntax. Currently, Bluespec uses a subset of System Verilog, called BSV that makes use of *term rewriting systems* [HA99]. During synthesis, the Bluespec compiler tries to schedule as many *conflict-free* rules as possible.

4.1.1.3 High-Level Synthesis Tools and Approaches based on C, C++, or SystemC

There exist a number of high-level synthesis tools and approaches that is based on the languages C, C++, or SystemC [SSV08].

Commercial Systems. Commercial examples of such systems include Catapult-C from Mentor Graphics [Men09], Forte Cynthesizer [For09], Agility Compiler from Celoxica [CEL09], PICO Express from Synfora [Syn09], AutoPilot [ZFJ^{+}08], ImpulseC [Imp09], Nios II C-to-Hardware Acceleration Compiler (C2H) from Altera [Alt08], and CHiMPS from Xilinx [PBD^{+}08].

Apart from commercial systems, there exist several C-based synthesis approaches in academia. Here, we only mention some well-known and recent approaches. For instance, the DEFACTO [BDD+99, DHP+05] synthesis system employs scalar replacement for data reuse and *unroll-and-jam* for code reordering.

In the Nimble framework [LCD+00], a hardware/software partitioning algorithm is contained that partitions applications into control code, running on a CPU, and computationally intensive code for the datapath (the FPGA).

SPARK [GDGN03, GGDN04] is a framework for compiling applications, specified in a subset of C, to FPGAs. Here, compiler transformations have been reinstrumented for synthesis by incorporating ideas of mutual exclusive operations, resource sharing and hardware cost models. The SPARK methodology is particularly targeted to control-intensive microprocessor functional blocks, as well as multimedia and image processing applications. A hardware/software partitioning algorithm that distributes applications to a CPU and an FPGA is also contained in the framework. However, SPARK is restricted to one-dimensional arrays.

The ASC system [Men06] is a stream compiler for computation on FPGAs. It provides optimizations on the algorithm level, the architecture level, the arithmetic level, and the bit level within the same given C++ program.

Comrade [GK07] is a compiler for combined hardware/software solutions of reconfigurable systems. It uses speculation techniques in order to increase the performance.

Trident [TGP07] is an FPGA compiler framework for floating-point algorithms based on LLVM [LA04].

SystemCoDesigner [KSS+09] is a tool at electronic system level (ESL) that includes the automatic optimization of an implementation consisting of hardware and software, with respect to several objectives. Behavioral descriptions written in SystemC are used as design entry. SystemCoDesigner extracts the mathematical model, performs behavioral synthesis, and explores the multi-objective design space, using multi-objective optimization algorithms. During design space exploration, a single design point is evaluated by simulating performance models, which are also automatically derived from the SystemC description and the behavioral synthesis results. Since SystemCoDesigner is an actor-oriented approach, the parallelization of an application at lower levels, such as instruction and data level, might introduce a large number of communication (buffers) and control primitives.

All of the above mentioned design tools start from a subset of sequential C, C++, or SystemC code. However, starting with sequential languages has the disadvantage that their semantics force a lot of restrictions on the execution order of the program (cf. Chapter 2). Another disadvantage of C-based hardware design is that most design tools only support a limited subset of the language. Porting existing, highly optimized C code to such a design environment is a time consuming task and often ends in completely rewriting the code from scratch.

Furthermore, most existing tools do not allow high-level program transformations in order to match the input program to given architectural constraints (like available memory or I/O bandwidth), or only to a limited extend.

4.1.2 Mapping Approaches for Coarse-Grained and Embedded Multi-Processor Arrays

Some research work has been performed, studying compilation techniques for coarse-grained reconfigurable architectures. For instance, the authors of [VNK$^+$03] describe a compiler framework for analyzing SA-C programs, perform optimizations, and automatically map the application onto MorphoSys [SLL$^+$00], a row-parallel or column-parallel SIMD (Single Instruction-stream Multiple Data-Stream) architecture. This approach is limited since the synthesis order is predefined by the loop order and no data dependencies between iterations are allowed. Another approach for mapping loops onto coarse-grained reconfigurable architectures is presented by Dutt and others in [LCD03]. Outstanding in their compilation flow is the target architecture, the DRAA, a generic reconfigurable architecture template, which can represent a wide range of coarse-grained reconfigurable arrays. The mapping technique itself is based on loop pipelining and partitioning of the program tree into clusters, which can be placed on a line of the array.

The aforementioned approaches focus on the placement of operations in space and time, whereas the authors in [PFM$^+$08] propose a method that primarily focuses on the routing problem, where a schedule is developed by routing each edge in the data flow graph.

Other existing approaches that are based on compilation techniques are closely related to approaches from the DSP world. These approaches employ several loop transformations, like pipelining [WL01] or temporal partitioning, but, are not able to exploit the full parallelism of a given algorithm and the computational potential of a typical 2-dimensional array.

4.1.3 Unified Approaches

Only very few approaches are know that try to unify the synthesis of different architectures and the compilation to different targets, respectively.

MATCH. A compiler that starts from Matlab as input description is presented by Banerjee et alii in [BSC$^+$00]. The compiler, called MATCH (MATlab Compiler for Heterogeneous computing systems), targets digital signal processors as well as the generation of code for FPGAs. For the DSP code generation the, *single-program, multiple-data* (SPMD) paradigm of parallel execution is employed. The modus operandi for hardware synthesis targeting FPGAs is rather straight forward. Nothing about the support of high-level transformations is known.

Streamroller. Mahlke and others [KFM06, Kud08] present the Streamroller synthesis system that is a unified compilation and synthesis system for streaming applications. The system can target the Cell multi-core architecture, but custom hardware accelerators can be created by high-level synthesis as well. However, this high-level synthesis is limited since it is compilation-centric, that means, starting with the SUIF compiler infrastructure [WFW$^+$94], the application is converted to assembly code for the Trimaran [Tri09] compiler tool chain. Thus, the resulting architecture is based on a VLIW template.

A similar approach is presented in [FKDM09] by the same authors. Here they extend the VLIW template by control structures in order to handle multiple loop programs. This class of loop accelerators is called *semi-programmable* by the authors.

4.2 The PARO Synthesis Tool

The PARO synthesis tool is a design environment for parallelizing and mapping nested loop programs onto acceleration engines, which are typically part of an SoC. A dedicated accelerator can be either derived from a single algorithm by high-level synthesis or PARO can be used to derive the processor schedules in a WPPA. PARO is entirely object-oriented written in the programming language C++. In the course of this thesis, the development[1] of PARO started in the end of 2004. Currently, it is comprised of approximately 110 000 lines of code.

An overview of PARO's design flow is depicted in Figure 4.1. The front end of the system starts with a program written in the PAULA language, as described in Section 2.2.2. An input program is parsed to obtain an internal representation,

[1]It should be mentioned that the term *PARO* as a *design system* and the term *PARO methodology* are used considerably longer. A distinction from this older works follows: Bednara [BT03, Bed04] presents a design system, which is based on the CASPAR design system [Tei93]. Around this system, a number of loosely-coupled tools for the parallelization of C code [Bey02], scheduling and exploration [HT01], and hardware synthesis [Bed04] were build. Scheduling is restricted to projection as global allocation. Also, the hardware generation is restricted to projection along a vector in the first direction (that means, the first iteration variable denotes the time axis).

The term *PARO methodology* is used in several previous works only for handcrafted mappings of certain algorithms onto FPGAs [HT04a, DHT$^+$06e], coarse-grained architectures [HDT04b, HDT06], and tightly-coupled, programmable processor arrays [DHT06d]. In [RDHT05], the generation of dedicated FPGA-based hardware accelerators for a single parameterizable digital signal processing application was demonstrated.

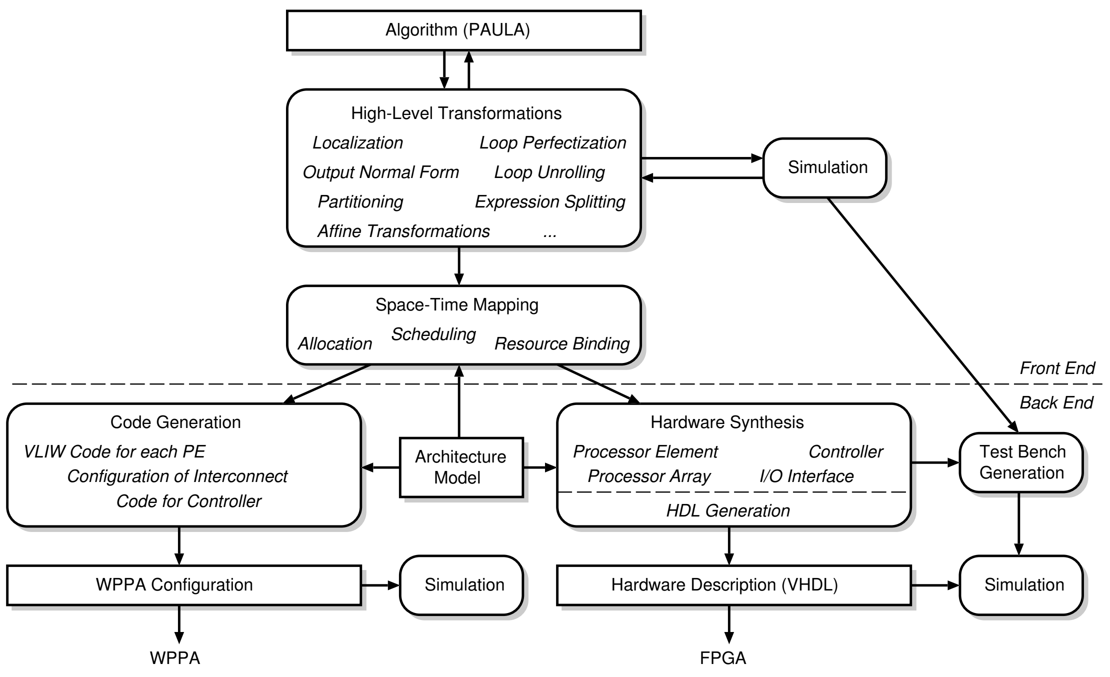


Figure 4.1: PARO design flow.

which strongly relies on matrices and polyhedral objects. For instance, a comprehensive class for matrices offers a large number of operations, starting from operators (for instance, * or +) for these objects to methods that determine, amongst others, the determinant, rank, inverse, adjugate, Hermite normal form of a matrix. The elements of a matrix are rational numbers, which are overflow save, since, if necessary, the data type for the nominator or denominator of a rational can be switched dynamically to a data type with arbitrary precision (GMP [The08]) at run-time. As backbone for operations in the polyhedral model, PolyLib [Wil93, Pol07] is used. It was initially developed by Doran Wilde at IRISA in Rennes and continuously extended at the University of Strasbourg. Among others, we added a completely symbolic handling of iteration variables and parameter names. Furthermore, we implemented data structures for handling linearly bounded lattices and operations on them, such as intersection, union, and others.

Based on a given algorithm, various source-to-source compiler transformations [Muc97] and optimizations can be applied within the PARO design system. Among others, these transformations include:

Constant and variable propagation. The propagation of variables and constants leads to a more compact code and decreased register usage.

Common sub-expression elimination. By data flow analysis, identical expressions within a program can be identified. Subsequently, it can be analyzed if it is worthwhile to replace an expression with an intermediate variable to store the computed value.

Loop perfectization. Loop perfectization transforms non-perfectly nested loop program into perfectly nested loops [Xue97].

Dead-code elimination. By static program analysis, program code can be determined that does not affect the program at all. This code, called *dead code*, can either be code that is unreachable or it affects variables that are neither defined as output variables nor used somewhere else in the program. Dead code might result from other transformations such as *common sub-expression elimination*.

Affine transformations. Affine transformations of the iteration space are a popular instrument for the parallelization of algorithms. Transformations such as *loop reversal*, *loop interchange*, and *loop skewing* can be expressed by affine transformations [Wol96a]. In addition, affine transformations can be used to embed variables of lower dimension into a common iteration space.

Strength reduction of operators. Strength reduction is a compiler transformation that systematically replaces operations by less expensive ones. For instance, in PARO, multiplications and divisions by constant values can be replaced by shift and add operations.

Loop unrolling. Loop unrolling is a major optimization transformation, which exposes parallelism in a loop program. Loop unrolling expands the loop kernel by a factor of n by copying $n-1$ consecutive iterations, which leads to larger data flow graphs at the benefit of possibly more instruction level parallelism.

Localization. Algorithms with non-uniform data dependencies are usually not suitable for the mapping onto regular processor arrays as they result in expensive global communication in terms of memory access. For that reason, a well known transformation called *localization* [TR91] exists, which replaces affine dependencies by regular dependencies. That means, it converts global communication into short propagation links to increase the regularity of the processor array. Thus, localization transforms a DPLA into a DPRA.

Global allocation. Global allocation may be achieved through either projection or diverse (hierarchical) partitioning schemes that can be selected (see Section 3.2). In case of already partitioned algorithms, it is not necessary to keep information about the tile shape/size. It is sufficient to symbolically denote the variable names which should be considered as *local sequential* or/and *global sequential* during scheduling. Internally, the PARO system reconstructs other necessary information (tile shape).

The heart of PARO is built by the allocation and scheduling methods, as described in detail in Chapter 3. Here, latency optimal schedules under resource constraints are derived. For the formulation and management of mixed linear integer programs, a generic intermediate layer has been developed. This intermediate layer has the advantage to decouple the formulation of the problem from the solving process. Several back ends to commercial solvers CPLEX [ILO06] as well as to freely available solvers (GLPK [GLP09], lp_solve [BDEN08], MiniSat [ES06], PIP [Fea88]) were implemented.

Furthermore, several modules for the visualization of graphs (reduced dependence graphs, AND-XOR-trees) and scheduling results (Gantt charts) were implemented.

4.3 Synthesis of Dedicated Hardware Accelerators

This section briefly describes the synthesis of a dedicated hardware accelerator. The synthesis can be subdivided into three parts:

1. Synthesis of the processor elements
2. Synthesis of the control structure
3. Synthesis of the interconnect structure

The synthesis of all parts generates a completely platform and language independent register transfer level (RTL) description of the hardware, which is further optimized and finally converted into HDL code of choice. Before describing the synthesis of the three different parts, the processor elements are classified into different types.

4.3.1 Classification of Processor Elements

A result of the allocation as described in Section 3.3, is the processor space $\mathcal{P} \subset \mathbb{Z}^s$. The regular structure of a given DPRA leads to regular processor arrays, which may be mapped to FPGAs or ASICs. However, not all processors $p \in \mathcal{P}$ need to execute all equations of an algorithm. For example, in a regular processor array, only the border processors are supposed to communicate with memory or I/O FIFOs. As a result, the processor space $\mathcal{P}$ can be expressed as a disjoint union of several *processor types* $\mathcal{P}_i \subseteq \mathcal{P}$:

$$\mathcal{P} = \bigcup_{i=1}^{k} \mathcal{P}_i \qquad \wedge \qquad \mathcal{P}_i \cap \mathcal{P}_j = \emptyset \,:\, i \neq j$$

Such a processor array is called a *piecewise regular array*. In order to reduce hardware costs, it is beneficial to generate specialized hardware implementations for each individual processor type, which contain only those functional units that are required to execute all equations defined for the corresponding processor type.

The first step of processor type classification is to calculate for each statement S_i within a given algorithm the set of processors $\mathcal{P}_{S_i}$, which executes this statement at least once. Let the following equation of a DPRA in output normal form be given.

$$(4.1) \qquad S_i : \quad x_i[I] = \mathcal{F}_i(\ldots, x_j[I - d_{ji}], \ldots) \qquad \text{if } \mathcal{C}_i^{\mathrm{I}}(I)$$

Then, the corresponding set of processors $\mathcal{P}_{S_i}$ can be obtained as follows.

$$\mathcal{P}_{S_i} = \left\{ p \in \mathcal{P} \mid p = \Phi I + \phi \wedge I \in \{\mathcal{I} \cap \mathcal{I}_{\mathcal{C}_i^{\mathrm{I}}(I)}\} \right\}$$

The processor types $\mathcal{P}_i$ can be obtained by finding intersection sets by pairwise comparison of all sets $\mathcal{P}_{S_i}$. Although the complexity is quadratic, in practice the decomposition into several sets is feasible within a fraction of a second. For further details, we refer to [Ruc06, HRDT08].

After processor type classification, the RTL structure for each individual processor type $\mathcal{P}_i$ can be synthesized (see Section 4.3.3) and each processor $p \in \mathcal{P}$ can be instantiated in the array structure. In order to facilitate regular placement, especially for 2-dimensional arrays on the chip, it is important that there exist only few different processor types of similar size. The logical placement of a processor is equivalent to its index p, while for the physical placement, data must be calculated depending on its required area and the geometry of the chip, taking special purpose structures of certain architectures into account, for instance hard-wired multipliers on FPGAs.

4.3.2 Synthesis of Interconnection Structure

In order to synthesize the processor interconnection structure, the data dependencies of the DPRA have to be analyzed. With the help of Equation (4.1) and a space-time mapping, as in Equation (3.1), Section 3.2, the synthesis of a processor interconnection for the data dependency d_{ji} is done by first determining the *processor displacement* as follows.

$$d_{ji}^{p} = \Phi(I + d_{ji}) - \Phi I = \Phi d_{ji}$$

For each pair of processors p_t and $p_s = p_t + d^p_{ji}$ with $p_s \in \mathcal{P} \wedge p_t \in \mathcal{P}$, a corresponding connection has to be created from the source processor p_s to the target processor p_t.

The *time displacement* denotes the number of time steps, the value of variable $x_j[I + d_{ji}]$ must be stored, before it is used to compute variable $x_i[I]$ (cf. with lifetime in Section 3.7). Let v_i and v_j denote the corresponding RDG nodes for x_i and x_j. Then, the time displacement is given as follows.

$$\begin{aligned} d^t_{ji} &= \Lambda(I + d_{ji}) + \tau(v_j) + w_j - \Lambda I - \tau(v_i) \\ &= \Lambda d_{ji} + \tau(v_j) + w_j - \tau(v_i) \end{aligned} \tag{4.2}$$

As recapitulation, $\tau(v_j)$ is the relative start time of the execution of node v_j, and w_j is the computation time of the associated operation (see Section 3.5). In processor arrays, the time displacement is equal to the number of delay registers on the respective processor interconnection. If the number of delay registers is large, controlled delay registers, [BT01], [Bed04], FIFOs, or embedded memories are used. Of course, the whole procedure must be repeated for each data dependency.

4.3.3 Synthesis of Processor Elements

A processor element consists of a local control logic (see Section 4.3.4) and the data path, where the actual computations are done. The synthesis of a processor element from the reduced dependence graph is performed as follows. If the instance binding is not handled during the scheduling phase, it is done afterwards, using a modified left-edge algorithm [Tei97, Ruc06].

During the phase of binding, a functional unit is assigned to each operation in the loop body that will execute the operation. In case of reuse of functional units, input multiplexers are required in order to select the correct operands in every time step. The interconnection between the functional units can be directly derived from the reduced dependence graph, whereas the number of time steps, an intermediate result must be stored, can be computed by Equation (4.2). However, for data dependencies with $d_{ji} = 0$, that is, intra-iteration dependencies, instead of shift registers, only simple registers are used to hold the intermediate results. Just like functional units, these registers may also be reused during one iteration. Register binding is also done using a modified left-edge algorithm, similar to resource instance binding.

4.3.4 Synthesis of Control Structure

In the context of regular processor arrays, synthesis of efficient control structures is of high importance. The purpose of processor array control structures is to produce

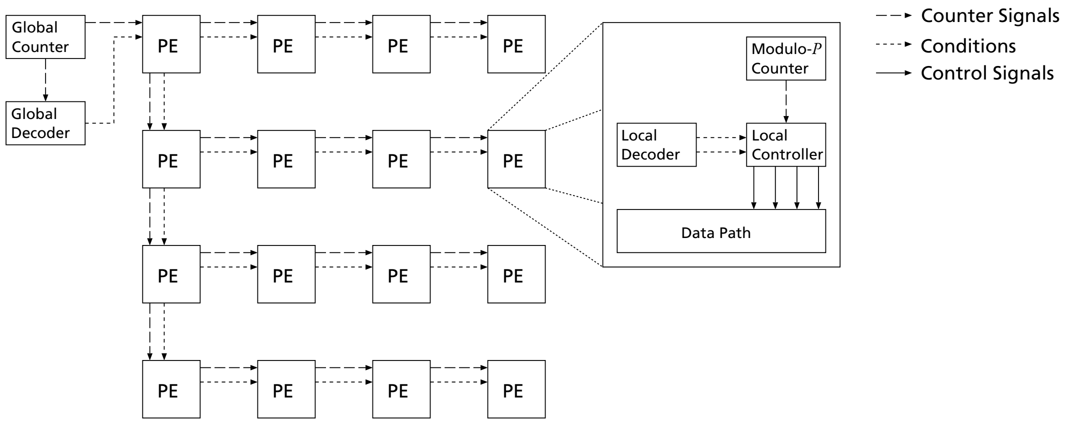


Figure 4.2: Overview of control architecture. Data path interconnect is not shown. Shown is a distributed global/local control scheme.

control signals to orchestrate the correct computation of the algorithm on the array. With this in mind, several different types of control signals must be generated:

- For algorithms with iteration dependent conditions, each processor must perform different operations and use different input data, depending on the current iteration point I. That is, control signals for functional units as well as for input multiplexers must be generated.
- If functional units are reused for multiple operations in the loop body, the input multiplexers must be controlled to select the correct input data. The same concept applies in the case of reusing internal registers.
- Multifunctional units like ALUs need to know which operation must be performed in a certain time step.
- Finally, access to I/O memory and FIFOs requires additional control signals and addresses to be generated.

In order to not limit the excellent scalability of regular processor arrays, it is important that the size of the control path remains nearly constant, regardless of the problem size (that means, the size of iteration space $|\mathcal{I}|$) and the size of the processor array $|\mathcal{P}|$.

In [DHT06b] and [DHRT07], we present a complete control methodology, that provides all the required control functionality while being very efficient in terms of area. This architecture is depicted in Figure 4.2 and briefly summarized in the following section.

The key characteristic of our control methodology is the use of combined global and local control facilities. All control signals that are common to all processor elements, except for a time delay, are generated by a global controller, outside (by a host) or at the border of the array, and propagated through the array, whereas local control is only necessary for signals that differ among the processor types. This strategy reduces the required area and, in general, improves the clock frequency. Because of the cyclic nature of iterative algorithms, most of the control demands can be met using nested counters that produce counter signals, which are subsequently decoded into specific control signals for multiplexers and functional units.

The central component of the control architecture is a global counter, which generates the non-constant parts of the iteration vector. The non-constant parts are the iteration variables that not directly resemble a processor index p. The variable parts of the iteration vector are obtained by scanning the tiles, as defined by loop matrices and the scheduling vector. The sequence of counter signals is used to compute the iteration dependent conditions. Here, one can identify conditions, which are independent of the current processor index and thus can be evaluated by a global decoder unit. Only the processor dependent conditions are subject to evaluation by local decoders within each processor. The globally evaluated conditions are propagated as binary signals along with the counter signals and the appropriate delay through the processor array [DHRT07].

Both the globally and locally decoded conditionals are transformed by an additional local controller into control signals for multiplexers and functional units. The global iteration counter signals and conditions are only related to iteration points. But in general, there are several operations scheduled at each iteration point (with offset $\tau(v_i)$), so additional logic is required to assure the correct execution behavior within one iteration period. In an iterative schedule, the iteration interval P defines the number of time steps between the start of two subsequent iteration points. Since the same sequence of control signals must be generated during every iteration interval, the required control functionality can be implemented by a modulo-P counter. Its output is connected to a decoder logic, which takes globally and locally evaluated iteration dependent conditions into account, and finally generates the control signals for the functional units and multiplexers. Of course, this counter could also be implemented globally. However, cost analysis has shown that the required delay registers would be more expensive than local modulo-P counters [DHRT07].

4.4 Code Generation for Weakly-Programmable Processor Arrays

As written earlier, when regarding weakly-programmable processor arrays as target architecture, a considerable amount of additional mapping constraints have to be taken into account. For instance, the number of registers and I/O ports per processor that have to be considered during scheduling (cf. Section 3.8). Other constraints, such as the size of the instruction memory within each processor, the number of available control signals within the processor array, and the number of control signals that can be simultaneously evaluated by the branch unit, are assumed to be sufficiently available. Indeed, the control flow could be serialized with similar constraints as for the data flow but then control would become the dominant part, which would be contradictory to the philosophy of our methodology where control should have almost no overhead in terms of execution time. Furthermore, the architecture should have a certain number of feedback shift registers within the processors and memory structures at the borders of the array that can serve as intermediate memory when partitioning an algorithm.

With the aforementioned assumptions and the characteristics of an algorithm as stated in Section 3.8, the mapping flow could be summarized as follows.

4.4.1 Mapping Flow

In addition to an algorithm to be mapped, let a space mapping Φ (global allocation) be given. The space mapping is defined by an appropriate partitioning, which should match the algorithm with the architectural constraints in terms of number and size of memories and I/O bandwidth. For further details, how to estimate necessary memory sizes and I/O traffic, we refer to the following works [Eck01, Dut04, DHT06d].

The pseudo code of Algorithm 4 describes the mapping flow for WPPAs. In lines 1-7 of the algorithm, the number m of internal variables ($\Phi d_e = 0$) with a lifetime greater than one period is counted. In line 8, the counted number m is compared with the number of available feedback shift registers (FSR). If enough FSRs are available, the mapping flow can continue with scheduling in line 11 of the algorithm. Otherwise, for the given space mapping no solution exists and the algorithm returns false (line 9). It may happen that the scheduling procedure cannot determine a schedule because of the resource constraints are too strict or because of the algorithm structure. In this case, the Boolean return variable β is false and the mapping procedure also returns false.

Once a valid schedule has been determined, the binding of operations to functional units and registers is done (line 15). Here, the same binding methods are ap-

Algorithm 4: Mapping flow for WPPAs

```
Input  : 1. Specification of the architecture (resource graph and allocation),
         2. Algorithm in form of the RDG $G = (V, E, D)$,
         3. Space mapping $\Phi$
Output: If possible: Local schedule (assembly program) for each processor
         and interconnect reconfiguration

1  $m = 0$;
2  foreach edge $e \in E$ do
3      if $(d_e \neq 0) \wedge (\Phi d_e = 0)$ then
4          $m = m + 1$;
5          $D_{reuse} = D_{reuse} \cup \{d_e\}$;
6      end
7  end
8  if $m > \alpha(FSR)$ then
       // For the given $\Phi$, the algorithm is not mappable. Retry with other mapping!
9      return false;
10 end
11 $\beta$ = doScheduling ();
12 if !$\beta$ then
       // No valid schedule exist.
13     return false;
14 end
15 doBinding ();
16 generateAssemblyCode ();
17 generateConfigCode ();
```

plied as when synthezising dedicated hardware accelerators. Remember that the instance binding for the I/O ports of a processor is directly determined during scheduling. Also, the allocation of feedback shift registers to variables is easily determined within this method since the FSRs are not shared by multiple variables.

Afterwards (program line 16), from the internal schedule and the binding, the assembly code for each processor can be generated. The cycle number (start time) of an instruction word is directly given by the calculated schedule. The operation type of an RDG node together with its resource binding defines the *issue slot*. Finally, taking the operands (registers, FSRs, constants, I/O ports) into account, the assembler instruction inclusive operands can be emitted. Furthermore, in dependence on possible iteration and run-time dependent conditions, code for the parallel branch unit is generated (examples follow in Section 5.4). For time-variant iteration dependent conditions, similar to the hardware generation, it is assumed that appropriate control signals are generated by a global controller and propagated through the control network of the array. These signals are then evaluated by each branch unit of a processor. In case of time-invariant iteration dependent conditions, different programs for groups or even single processors are generated (cf. processor classification in Section 4.3.1).

Finally, in line 17 of the mapping algorithm, the reconfiguration code for the interconnect structure and the length of input delay registers and FSRs be emitted. This information is combined with the different programs according to the processor classification. Considering each processor and its surrounding interconnect as a whole has the advantage that the *multicast reconfiguration* [KHKT06b] feature of WPPAs can be employed. Thus, both, the size of the configuration bit stream and the reconfiguration time are reduced.

4.5 Summary and Conclusions

In this chapter, related work in the area of high-level synthesis tools and embedded processor arrays with the main emphasis on recent approaches has been presented and discussed.

Afterwards, the entire design flow of the PARO synthesis tool, which has been developed in the course of this thesis, has been presented in Section 4.2. PARO's uniqueness is the applicability to target different architectures that are classically tackled with totally different method, either high-level synthesis approaches or standard compilation methods.

A systematic approach for the synthesis of dedicated hardware accelerators consisting of several processor elements, interconnect, and control structures, has been presented. Also, for the first time, a method has been briefly presented how to generate code for tightly-coupled, programmable processor arrays (WPPAs) from almost the same allocation and scheduling information as for the generation of dedicated processor arrays.

The unified design approach of the PARO synthesis tool might be easily adapted to further array architectures, consisting of EPIC or VLIW processors. Moreover, the back end of PARO could be retargeted to generate configuration code for coarse-grained reconfigurable array architectures. For instance, in [HDT04a, HDT04b, HDT06], we have adapted our methodology to target the PACT XPP64-A architecture [BEM$^+$03], which is a high performance run-time reconfigurable processor array.

Furthermore, the rich set of high-level transformations included in PARO might also be used for code generation for multi-core architectures such as modern graphics processors.

The introduction of the theoretic foundations and their practical implementations (target code generation) allow to quantitatively evaluate the proposed approaches in the next chapter.

Chapter 5

Experiments and Evaluation

After the presentation of new scheduling techniques in Chapter 3 and the description of the PARO design system in Chapter 4, several experimental results are obtained by applying the new methods on different examples in this chapter.

In Section 5.1, the proposed sequentialization constraints are quantitatively compared with an existing method. In Section 5.2, examples for predicated and conditional execution are given. Scheduling with register constraints is discussed in Section 5.3. Scheduling and code generation for WPPAs is demonstrated in Section 5.4. Scheduling as well as synthesis results for a number of algorithms chosen from different benchmarks are presented in Section 5.5. A comparison of loop unrolling with our proposed method of loop partitioning follows in Section 5.6. Finally, in Section 5.7, a complex real world application stemming from the area of image processing is presented.

5.1 Sequentialized Scheduling

In this section, we quantitatively compare the novel sequentialization approach, introduced in Chapter 3, with the method presented by Teich, Thiele, and Zhang in [TTZ96, TTZ97]. As comparison, we consider a one-level partitioning where the iterations within the tile should be executed sequentially.

For a given loop matrix R, the sequentialization constraint presented in [TTZ96, TTZ97] (named *Method TTZ* in the following) is defined as follows.

$$\Lambda_{seq} R \begin{pmatrix} 1/h'_{1,1} & -1 & \cdots & -1 \\ 0 & 1/h'_{2,2} & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1/h'_{s,s} \end{pmatrix} \geq (1\ 0\ \ldots\ 0)$$

The entries $h_{i,i}$ of the above matrix are determined by decomposing the loop matrix R in such a way that $R = UH'$, where U is a unimodular matrix, the matrix H' is in upper right triangular form, and $h'_{i,i} > 0$ for all $1 \leq i \leq s$.

$$H' = \begin{pmatrix} h'_{1,1} & h'_{1,2} & \cdots & h'_{1,s} \\ 0 & h'_{2,2} & \cdots & h'_{2,s} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & h'_{s,s} \end{pmatrix}$$

As repetition, our method (named *Method Hannig* in the following) presented in Section 3.5.2 is briefly summarized. The sequentialization constraint is defined by

$$\Lambda_{seq} T^{-1} \underbrace{\left(S' + \begin{pmatrix} 0 & \bar{h}_{1,2} & \ldots & \bar{h}_{1,n} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \bar{h}_{n-1,n} \\ 0 & \ldots & \ldots & 0 \end{pmatrix} \right)}_{\vec{S}'} \geq (1\ 1\ \ldots\ 1)$$

where the coefficients $\bar{h}_{i,j}$ for all $1 \leq i \leq n-1$ and $i < j \leq n$ are calculated as follows.

$$\bar{h}_{i,j} = h_{i,j} - \left\lfloor \frac{h_{i,j} + l_i}{h_{i,i}} \right\rfloor h_{i,i} \qquad l_i = \frac{\sigma \det(R) - w_i}{w_i}$$

$$T = \sigma W^{-1} \mathrm{adj}(R) \qquad \sigma = \frac{\det(R)}{|\det(R)|} \qquad W = \begin{pmatrix} w_1 & 0 & \ldots & 0 \\ 0 & w_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & w_n \end{pmatrix}$$

$$w_i = \frac{1}{g_i} \prod_{j=1}^{n} g_j \quad \forall 1 \leq i \leq n \qquad g_k = \gcd_{\forall l = \{1,2,\ldots,n\}} (r_{l,k}) \quad \forall k \in \{i,j\}$$

In the following, the two sequentialization constraints are constructed and compared for six different parallelotopes as tiles or rather loop matrices $R_1, \ldots, R_6$.

Example 1

Given loop matrix: $R_1 = \begin{pmatrix} -3 & 3 \\ 3 & 6 \end{pmatrix}$

Method TTZ:

$$R_1 = U_{R_1} H'_{R_1} = \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 3 & 6 \\ 0 & 9 \end{pmatrix} \quad \Rightarrow \quad R_1 \begin{pmatrix} 1/3 & -1 \\ 0 & 1/9 \end{pmatrix} = \begin{pmatrix} -1 & 10/3 \\ 1 & -7/3 \end{pmatrix}$$

$\Rightarrow$ sequentialization constraint:

$$-\Lambda_{seq_1} + \Lambda_{seq_2} \geq 1 \quad \wedge \quad 10\Lambda_{seq_1} - 7\Lambda_{seq_2} \geq 0$$

Method Hannig:

$$g_{R_1} = \begin{pmatrix} 3 \\ 3 \end{pmatrix} \qquad w_{R_1} = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$$

$$\det(R_1) = -27 \qquad \operatorname{adj}(R_1) = \begin{pmatrix} 6 & -3 \\ -3 & -3 \end{pmatrix} \qquad \sigma_{R_1} = -1$$

$$T_{R_1} = \begin{pmatrix} -2 & 1 \\ 1 & 1 \end{pmatrix} \qquad T_{R_1}^{-1} = \frac{1}{3} \begin{pmatrix} -1 & 1 \\ 1 & 2 \end{pmatrix}$$

$$H_{R_1} = T_{R_1} U_{R_1} \Leftrightarrow \begin{pmatrix} 3 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} -2 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} -1 & 0 \\ 1 & 1 \end{pmatrix} \Rightarrow S'_{R_1} = \begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\vec{S}'_{R_1} = \begin{pmatrix} 3 & -8 \\ 0 & 1 \end{pmatrix} \Rightarrow \vec{S}_{R_1} = T_{R_1}^{-1} \vec{S}'_{R_1} = \begin{pmatrix} -1 & 3 \\ 1 & -2 \end{pmatrix}$$

$\Rightarrow$ sequentialization constraint:

$$-\Lambda_{seq_1} + \Lambda_{seq_2} \geq 1 \quad \wedge \quad 3\Lambda_{seq_1} - 2\Lambda_{seq_2} \geq 1$$

Example 2

Given loop matrix: $R_2 = \begin{pmatrix} 6 & 4 \\ 2 & -2 \end{pmatrix}$

Method TTZ:

$$R_2 = U_{R_2} H'_{R_2} = \begin{pmatrix} 3 & -2 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 2 & 8 \\ 0 & 10 \end{pmatrix}$$

$$\Rightarrow \quad R_2 \begin{pmatrix} 1/2 & -1 \\ 0 & 1/10 \end{pmatrix} = \begin{pmatrix} 3 & -28/5 \\ 1 & -11/5 \end{pmatrix}$$

$\Rightarrow$ sequentialization constraint:

$$3\Lambda_{seq_1} + \Lambda_{seq_2} \geq 1 \quad \wedge \quad -28\Lambda_{seq_1} - 11\Lambda_{seq_2} \geq 0$$

Method Hannig:

$$g_{R_2} = \begin{pmatrix} 2 \\ 2 \end{pmatrix} \qquad w_{R_2} = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

$$\det(R_2) = -20 \qquad \operatorname{adj}(R_2) = \begin{pmatrix} -2 & -4 \\ -2 & 6 \end{pmatrix} \qquad \sigma_{R_2} = -1$$

$$T_{R_2} = \begin{pmatrix} 1 & 2 \\ 1 & -3 \end{pmatrix} \qquad T_{R_2}^{-1} = \frac{1}{5}\begin{pmatrix} 3 & 2 \\ 1 & -1 \end{pmatrix}$$

$$H_{R_2} = T_{R_2} U_{R_2} \Leftrightarrow \begin{pmatrix} 5 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 1 & -3 \end{pmatrix} \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix} \Rightarrow S'_{R_2} = \begin{pmatrix} 5 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\vec{S}'_{R_2} = \begin{pmatrix} 5 & -9 \\ 0 & 1 \end{pmatrix} \Rightarrow \vec{S}_{R_2} = T_{R_2}^{-1} \vec{S}'_{R_2} = \begin{pmatrix} 3 & -5 \\ 1 & -2 \end{pmatrix}$$

$\Rightarrow$ sequentialization constraint:

$$3\Lambda_{seq_1} + \Lambda_{seq_2} \geq 1 \quad \wedge \quad -5\Lambda_{seq_1} - 2\Lambda_{seq_2} \geq 1$$

Example 3

Given loop matrix: $R_3 = \begin{pmatrix} 10 & 0 \\ 0 & 4 \end{pmatrix}$

Method TTZ:

$$R_3 = U_{R_3} H'_{R_3} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 10 & 0 \\ 0 & 4 \end{pmatrix}$$

$$\Rightarrow \quad R_3 \begin{pmatrix} 1/10 & -1 \\ 0 & 1/4 \end{pmatrix} = \begin{pmatrix} 1 & -10 \\ 0 & 1 \end{pmatrix}$$

$\Rightarrow$ sequentialization constraint:

$$\Lambda_{seq_1} \geq 1 \quad \wedge \quad -10\Lambda_{seq_1} + \Lambda_{seq_2} \geq 0$$

Method Hannig:

$$g_{R_3} = \begin{pmatrix} 10 \\ 4 \end{pmatrix} \qquad w_{R_3} = \begin{pmatrix} 4 \\ 10 \end{pmatrix}$$

$$\det(R_3) = 40 \qquad \operatorname{adj}(R_3) = \begin{pmatrix} 4 & 0 \\ 0 & 10 \end{pmatrix} \qquad \sigma_{R_3} = 1$$

$$T_{R_3} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad T_{R_3}^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$H_{R_3} = T_{R_3} U_{R_3} \Leftrightarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \Rightarrow S'_{R_3} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\vec{S}'_{R_3} = \begin{pmatrix} 1 & -9 \\ 0 & 1 \end{pmatrix} \Rightarrow \vec{S}_{R_3} = T_{R_3}^{-1} \vec{S}'_{R_3} = \begin{pmatrix} 1 & -9 \\ 0 & 1 \end{pmatrix}$$

$\Rightarrow$ sequentialization constraint:

$$\Lambda_{seq_1} \geq 1 \quad \wedge \quad -9\Lambda_{seq_1} + \Lambda_{seq_2} \geq 1$$

Example 4

Given loop matrix: $R_4 = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 4 \end{pmatrix}$

Method TTZ:

$$R_4 = U_{R_4} H'_{R_4} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 4 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 4 \end{pmatrix}$$

$$\Rightarrow \quad R_4 \begin{pmatrix} 1/4 & -1 & -1 \\ 0 & 1/4 & -1 \\ 0 & 0 & 1/4 \end{pmatrix} = \begin{pmatrix} 1 & -4 & -4 \\ 0 & 1 & -4 \\ 0 & 0 & 1 \end{pmatrix}$$

$\Rightarrow$ sequentialization constraint:

$$\Lambda_{seq_1} \geq 1 \quad \wedge \quad -4\Lambda_{seq_1} + \Lambda_{seq_2} \geq 0 \quad \wedge \quad -4\Lambda_{seq_1} - 4\Lambda_{seq_2} + \Lambda_{seq_3} \geq 0$$

Method Hannig:

$$g_{R_4} = \begin{pmatrix} 4 \\ 4 \\ 4 \end{pmatrix} \qquad w_{R_4} = \begin{pmatrix} 16 \\ 16 \\ 16 \end{pmatrix}$$

$$\det(R_4) = 64 \qquad \operatorname{adj}(R_4) = \begin{pmatrix} 16 & 0 & 0 \\ 0 & 16 & 0 \\ 0 & 0 & 16 \end{pmatrix} \qquad \sigma_{R_4} = 1$$

$$T_{R_4} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad T_{R_4}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$H_{R_4} = T_{R_4} U_{R_4} \quad \Leftrightarrow \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\Rightarrow \quad S'_{R_4} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\vec{S}'_{R_4} = \begin{pmatrix} 1 & -3 & -3 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{pmatrix} \quad \Rightarrow \quad \vec{S}_{R_4} = T^{-1}_{R_4} \vec{S}'_{R_4} = \begin{pmatrix} 1 & -3 & -3 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{pmatrix}$$

$\Rightarrow$ sequentialization constraint:

$$\Lambda_{seq_1} \geq 1 \quad \wedge \quad -3\Lambda_{seq_1} + \Lambda_{seq_2} \geq 1 \quad \wedge \quad -3\Lambda_{seq_1} - 3\Lambda_{seq_2} + \Lambda_{seq_3} \geq 1$$

Example 5

Given loop matrix: $R_5 = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 7 & 0 \\ 0 & 0 & 5 \end{pmatrix}$

Method TTZ:

$$R_5 = U_{R_5} H'_{R_5} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 4 & 0 & 0 \\ 0 & 7 & 0 \\ 0 & 0 & 5 \end{pmatrix}$$

$$\Rightarrow \quad R_5 \begin{pmatrix} 1/4 & -1 & -1 \\ 0 & 1/7 & -1 \\ 0 & 0 & 1/5 \end{pmatrix} = \begin{pmatrix} 1 & -4 & -4 \\ 0 & 1 & -7 \\ 0 & 0 & 1 \end{pmatrix}$$

$\Rightarrow$ sequentialization constraint:

$$\Lambda_{seq_1} \geq 1 \quad \wedge \quad -4\Lambda_{seq_1} + \Lambda_{seq_2} \geq 0 \quad \wedge \quad -4\Lambda_{seq_1} - 7\Lambda_{seq_2} + \Lambda_{seq_3} \geq 0$$

Method Hannig:

$$g_{R_5} = \begin{pmatrix} 4 \\ 7 \\ 5 \end{pmatrix} \qquad w_{R_5} = \begin{pmatrix} 35 \\ 20 \\ 28 \end{pmatrix}$$

$$\det(R_5) = 140 \qquad \operatorname{adj}(R_5) = \begin{pmatrix} 35 & 0 & 0 \\ 0 & 20 & 0 \\ 0 & 0 & 28 \end{pmatrix} \qquad \sigma_{R_5} = 1$$

$$T_{R_5} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad T^{-1}_{R_5} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$H_{R_5} = T_{R_5} U_{R_5} \iff \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\Rightarrow S'_{R_5} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\vec{S}'_{R_5} = \begin{pmatrix} 1 & -3 & -3 \\ 0 & 1 & -6 \\ 0 & 0 & 1 \end{pmatrix} \Rightarrow \vec{S}_{R_5} = T_{R_5}^{-1} \vec{S}'_{R_5} = \begin{pmatrix} 1 & -3 & -3 \\ 0 & 1 & -6 \\ 0 & 0 & 1 \end{pmatrix}$$

$\Rightarrow$ sequentialization constraint:

$$\Lambda_{seq_1} \geq 1 \quad \wedge \quad -3\Lambda_{seq_1} + \Lambda_{seq_2} \geq 1 \quad \wedge \quad -3\Lambda_{seq_1} - 6\Lambda_{seq_2} + \Lambda_{seq_3} \geq 1$$

Example 6

Given loop matrix: $R_6 = \begin{pmatrix} 4 & 1 & 5 \\ 4 & 7 & 10 \\ 4 & 10 & 5 \end{pmatrix}$

Method TTZ:

$$R_6 = U_{R_6} H'_{R_6} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 2 & -1 \\ 1 & 3 & -2 \end{pmatrix} \begin{pmatrix} 4 & 1 & 5 \\ 0 & 3 & 10 \\ 0 & 0 & 15 \end{pmatrix}$$

$$\Rightarrow R_6 \begin{pmatrix} 1/4 & -1 & -1 \\ 0 & 1/3 & -1 \\ 0 & 0 & 1/15 \end{pmatrix} = \begin{pmatrix} 1 & -11/3 & -14/3 \\ 1 & -5/3 & -31/3 \\ 1 & -2/3 & -41/3 \end{pmatrix}$$

$\Rightarrow$ sequentialization constraint:

$$\Lambda_{seq_1} + \Lambda_{seq_2} + \Lambda_{seq_3} \geq 1 \wedge -11\Lambda_{seq_1} - 5\Lambda_{seq_2} - 2\Lambda_{seq_3} \geq 0$$
$$\wedge -14\Lambda_{seq_1} - 31\Lambda_{seq_2} - 41\Lambda_{seq_3} \geq 0$$

Method Hannig:

$$g_{R_6} = \begin{pmatrix} 4 \\ 1 \\ 5 \end{pmatrix} \qquad w_{R_6} = \begin{pmatrix} 5 \\ 20 \\ 4 \end{pmatrix}$$

$$\det(R_6) = -180 \qquad \operatorname{adj}(R_6) = \begin{pmatrix} -65 & 45 & -25 \\ 20 & 0 & -20 \\ 12 & -36 & 24 \end{pmatrix} \qquad \sigma_{R_6} = -1$$

$$T_{R_6} = \begin{pmatrix} 13 & -9 & 5 \\ -1 & 0 & 1 \\ -3 & 9 & -6 \end{pmatrix} \qquad T_{R_6}^{-1} = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 7 & 2 \\ 1 & 10 & 1 \end{pmatrix}$$

$$H_{R_6} = T_{R_6} U_{R_6} \quad \Leftrightarrow \quad \begin{pmatrix} 9 & 6 & 5 \\ 0 & 3 & 1 \\ 0 & 0 & 3 \end{pmatrix} = \begin{pmatrix} 13 & -9 & 5 \\ -1 & 0 & 1 \\ -3 & 9 & -6 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 3 & 2 \\ 1 & 4 & 2 \end{pmatrix}$$

$$\Rightarrow \; S'_{R_6} = \begin{pmatrix} 9 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

$$\vec{S}'_{R_6} = \begin{pmatrix} 9 & -30 & -31 \\ 0 & 3 & -8 \\ 0 & 0 & 3 \end{pmatrix} \;\Rightarrow\; \vec{S}_{R_6} = T_{R_6}^{-1}\vec{S}'_{R_6} = \begin{pmatrix} 1 & -3 & -4 \\ 1 & -1 & -9 \\ 1 & 0 & -12 \end{pmatrix}$$

$\Rightarrow$ sequentialization constraint:

$$\Lambda_{seq_1} + \Lambda_{seq_2} + \Lambda_{seq_3} \geq 1 \;\wedge\; -3\Lambda_{seq_1} - \Lambda_{seq_2} \geq 1 \;\wedge\; -4\Lambda_{seq_1} - 9\Lambda_{seq_2} - 12\Lambda_{seq_3} \geq 1$$

Comparison

In order to compare the two methods, a schedule Λ_{seq} is determined only for the sequentialization constraint. That means, no further resource or precedence constraints are considered. Then, each iteration point within the given tile, specified by a loop matrix R_i, can be executed in one time step. Thus, the number of iteration points within a tile defines the minimal latency $L_{min} = |\det(R_i)|$. The *tightness* T is a measure of how close the derived linear schedule is to the optimal free schedule. For this, the minimal latency L_{min} is set in proportion to the latency L, associated to a schedule vector Λ_{seq}, $T = L_{min}/L$. In Table 5.1, the results for the afore determined

Table 5.1: Comparison of sequentialization methods.

Loop	**Method TTZ**			**Method Hannig**			**Difference**	
matrix	Λ_{seq}	L	T	Λ_{seq}	L	T	**abs.**	**rel.**
R_1	(3 4)	33	82%	(3 4)	33	82%	0	0%
R_2	(3 -8)	28	71%	(3 -8)	28	71%	0	0%
R_3	(1 10)	40	100%	(1 10)	49	100%	0	0%
R_4	(1 4 20)	76	84%	(1 4 16)	64	100%	12	19%
R_5	(1 5 40)	194	72%	(1 4 28)	140	100%	54	39%
R_6	(-17 50 -32)	254	71%	(-16 47 -30)	240	75%	14	6%

sequentialization constraints are presented. For both methods (TTZ and Hannig), the optimal linear schedule vector Λ_{seq}, the corresponding latency L, and the tightness T are given. In the last two columns, the absolute difference of the latency, as well as the relative difference between the two methods are given. For the first three considered loop matrices, which are all two-dimensional, both methods result in the same schedule vector. Only for the rectangular tile (R_3), a tight schedule is obtained. In case of three-dimensional tiles the situation is different. Here, our new method is better in all cases. Remarkably different is the situation for the cube as tile (R_4) and the cuboid (R_5), where the new sequentialization constraint leads to an improvement of 19% and 39%, respectively.

Based on the construction of our method, one can conclude that for any n-dimensional orthotope, a tight linear schedule can be obtained.

5.2 Predicated and Conditional Execution

In this section, several programs with run-time dependent conditions are presented and subsequently solved by application of the scheduling method presented in Section 3.6. By these examples, some special cases of nested conditions are discussed. For the sake of compactness and readability, the examples are constructed and fictive.

Example

The code fragment is given as pseudo code as well as PAULA program. The pseudo code contains two nested if-then-else statements. The number of variables that is influenced by these statements is six, which can be easily verified by the PAULA program.

Pseudo code

```
$C_1[i] = (a[i] > 8);$
$C_2[i] = (b[i] > 0);$
$C_3[i] = (a[i] > 5);$
IF ($C_1[i]$)
   $b[i] = a[i] + 2;$
   IF ($C_2[i]$)
      $c[i] = a[i] + 4;$
   ELSE
      $c[i] = a[i] * 5;$
   ENDIF
ELSE
   $b[i] = a[i] * 3;$
   $c[i] = a[i] * 6;$
ENDIF
IF ($C_3[i]$)
   $d[i] = a[i] * 2;$
   IF ($C_1[i]$)
      $e[i] = a[i] + 4;$
   ELSE
      $e[i] = a[i] * 5;$
   ENDIF
ELSE
   $d[i] = a[i] * 3;$
   $e[i] = a[i] * 6;$
ENDIF
```

PAULA program

```
S0:  C1[i]  = (a[i]>8);
S1:  C2[i]  = (b[i]>0);
S2:  C3[i]  = (a[i]>5);
S3:  b[i]   = ifrt(C1[i],b1[i],b0[i]);
S4:  b1[i]  = a[i] + 2;
S5:  b0[i]  = a[i] * 3;
S6:  c[i]   = ifrt(C1[i],c1[i],c0[i]);
S7:  c1[i]  = ifrt(C2[i],c11[i],c10[i]);
S8:  c11[i] = a[i] + 4;
S9:  c10[i] = a[i] * 5;
S10: c0[i]  = a[i] * 6;
S11: d[i]   = ifrt(C3[i],d1[i],d0[i]);
S12: d1[i]  = a[i] * 2;
S13: d0[i]  = a[i] * 3;
S14: e[i]   = ifrt(C3[i],e1[i],e0[i]);
S15: e1[i]  = ifrt(C1[i],e11[i],e10[i]);
S16: e11[i] = a[i] + 4;
S17: e10[i] = a[i] * 5;
S18: e0[i]  = a[i] * 6;
```

From the given program, a reduced dependence graph is generated according to the extended Definition 2.7. Two versions are possible: One is dedicated for *predicated execution* and the other for *conditional execution*. The RDG for predicated execution is depicted in Figure 5.1. Here, the edges denote the precedence of the nodes. Apart from that, if enough resource are available, the nodes might be executed in parallel. For instance, consider the nodes of statement S_4, S_5, and S_0. All three operations (multiplication in S_5, addition in S_4, comparison in S_0) might be executed in parallel since they are independent of each other. The scheduling method is eager to compute both branches in parallel. Afterwards, just the right result has to be selected by the merge node (S_3). A corresponding schedule with an iteration interval of $P = 4$ and an allocation of two adders, two multipliers (two cycles latency with a pipeline rate of one), and two comparators, is shown in the following.

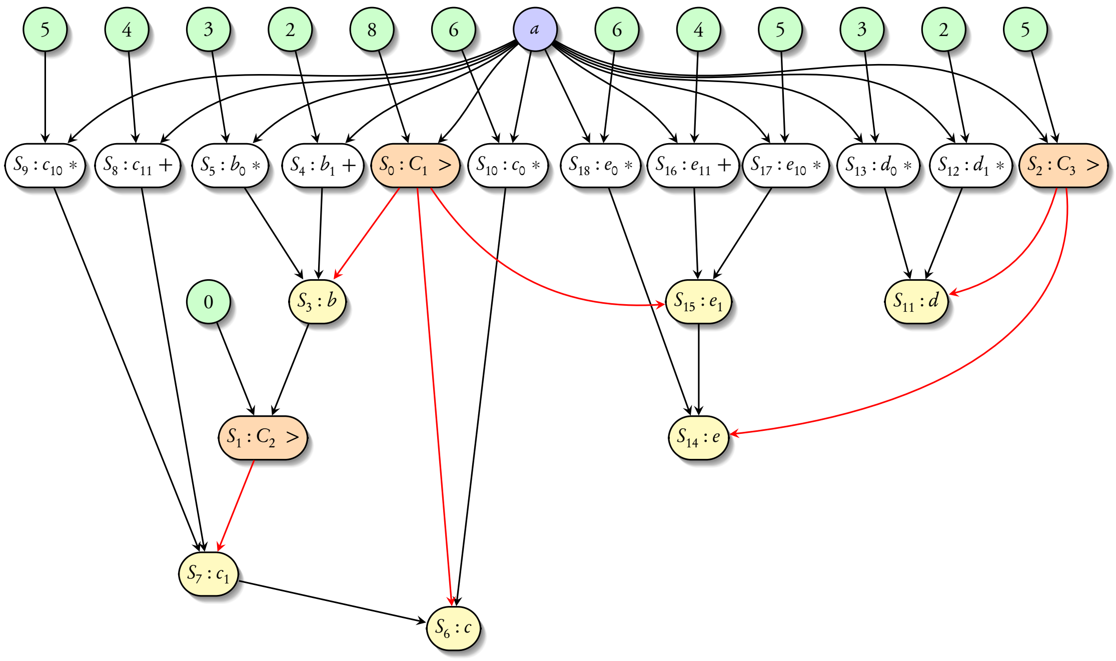


Figure 5.1: Reduced dependence graph for predicated execution.

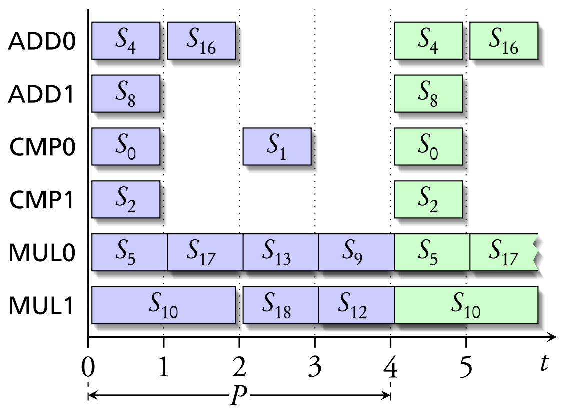


Figure 5.2: Gantt chart of the scheduled algorithm according to the RDG in Figure 5.1. Note, multiplications have two cycles latency but can start in each cycle a new operation. Thus, the bars for the multipliers may overlap each other. For instance, statement S_5 executed on multiplier **MUL0** ends its execution not already at time step one but at two.

It should be mentioned that an iteration interval of four can also be obtained with only one adder and comparator.

The other possibility is the conditional execution shown in Figure 5.3. For this,

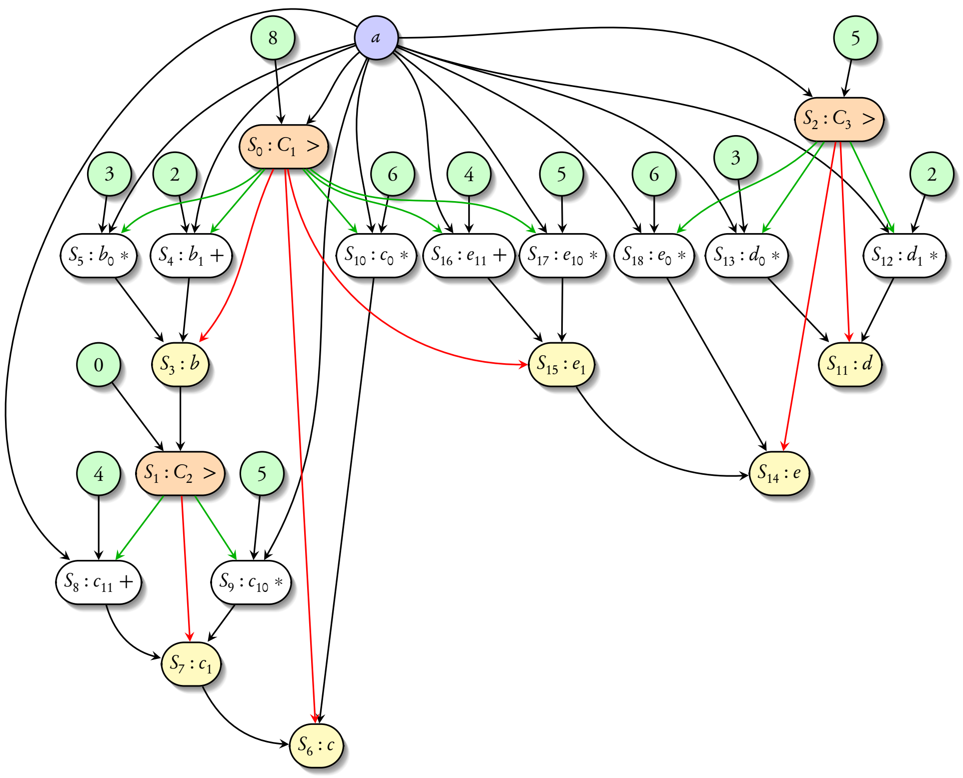


Figure 5.3: Reduced dependence graph for conditional execution.

serialization edges are added to the RDG (green edges) in order that first a comparison is evaluated and then in dependence on it, the right branch is executed. Besides the serialization edges, a corresponding AND-XOR-tree is created, which is shown in Figure 5.4.
In the AXT, it can be seen that the nesting of the conditions depends on the program. Particular interesting is that conditional C_1 is used in different branches and at different levels. The derived schedule is visualized as a Gantt chart in Figure 5.5.

Thanks to conditional execution, the iteration interval has been reduced to $P = 3$. The example demonstrates that even operations from different iterations can mutually exclusive share the same resource at the same time. This is the case for statements S_5 and S_9, which are executed on multiplier MUL0, and is denoted by the color gradient in the Gantt chart. Statement S_5 belongs to the actual iteration and S_9 to the previous one.

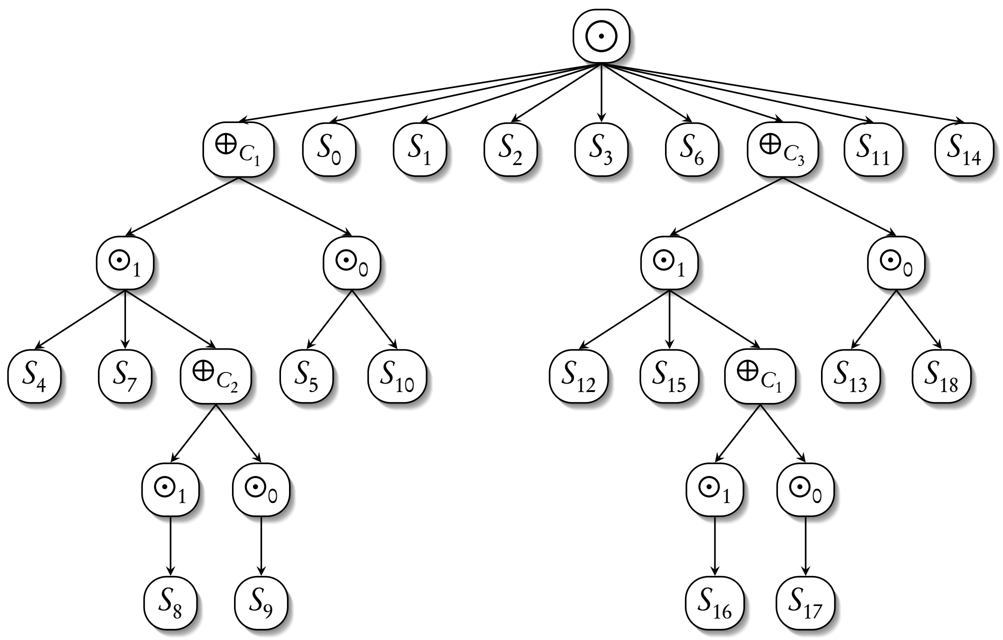


Figure 5.4: AND-XOR-tree corresponding to the reduced dependence graph shown in Figure 5.3.

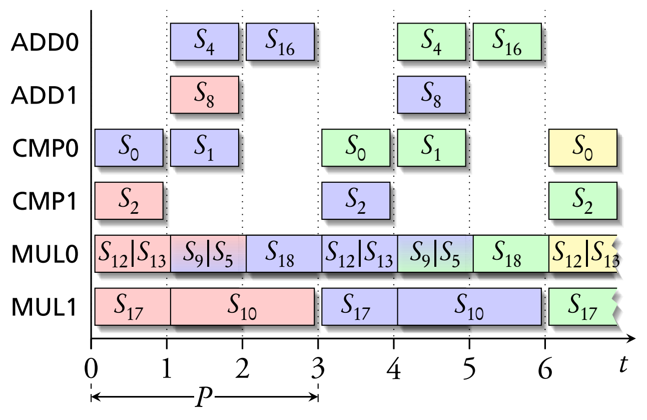


Figure 5.5: Gantt chart of the scheduled algorithm according to the RDG in Figure 5.3 and the AXT in Figure 5.4.

5.3 Scheduling with Register Constraints

Scheduling with register constraints can become challenging if the lifetime of a variable increases. This might be the case if, due to limited resources, the execution has to be serialized and several longer lifetimes exist, which can be traded for the shorter ones of other variables. Such a situation, with large scheduling freedom, is given by a dependence graph in form of a regular tree structure. Thus, an example for demonstration of the proposed scheduling method with register constraints (cf. Section 3.8.2) is given by considering an adder tree that sums up 16 values. That means, 15 add operations are necessary. Of course, this example should also be scheduled to achieve maximal throughput (minimal iteration interval). Simultaneously, also the local latency L_l is minimized.

Table 5.2: Local latency L_l and iteration interval P for different adder and register allocations.

Number of adders	**Number of registers**	P	L_l	**Number of adders**	**Number of registers**	P	L_l
16	16	1	4	3	8	5	7
16	8	2	5	3	7	5	7
8	16	2	5	3	6	5	7
8	8	2	5	3	5	5	8
8	7	3	5	3	4	5	8
8	6	3	5	2	8	8	8
8	5	4	6	2	7	8	8
8	4	5	6	2	6	8	8
7	8	3	5	2	5	8	9
6	8	3	5	2	4	8	10
5	8	3	6	1	8	15	15
4	8	4	6	1	7	15	15
4	7	4	6	1	6	15	15
4	6	4	6	1	5	15	15
4	5	4	7	1	4	15	15
4	4	5	7				

In Table 5.2, the results for different numbers of adders and registers are shown. One can see that adders might be traded for registers. Consider, for instance, an iteration interval of length five. Then, the allocations (no. of adders, no. of registers), (8,4), (4,4), (3,6), and (3,4) lead to the same iteration interval and thus to an interesting multi-objective exploration problem. Because, if only the cost will be minimized for a given iteration interval, the overall latency might be suboptimal.

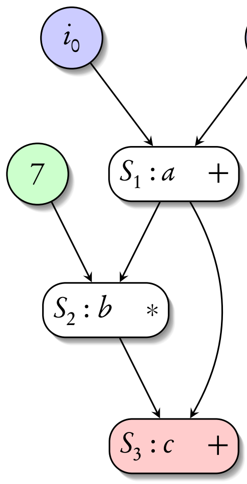
5.4 Scheduling for WPPA

In Section 3.8 a scheduling method and in Section 4.4 a design flow (code generation) for weakly-programmable processor arrays have been proposed. For illustration, these techniques are applied to several examples in this section.

As first example, the following simple program consisting of three statements is considered. The program has to compute two additions and one multiplication.

```
par (i>=1 and i<=N and j>=1 and j<=N)
{ S1:  a[i,j] = i0[i,j] + i1[i,j];
  S2:  b[i,j] = a[i,j] * 7;
  S3:  c[i,j] = a[i,j] + b[i,j];
}
```

Variables i_0 and i_1 are inputs and variable c is an output as denoted by the different colors of the nodes in the corresponding reduced dependence graph, shown in the following.

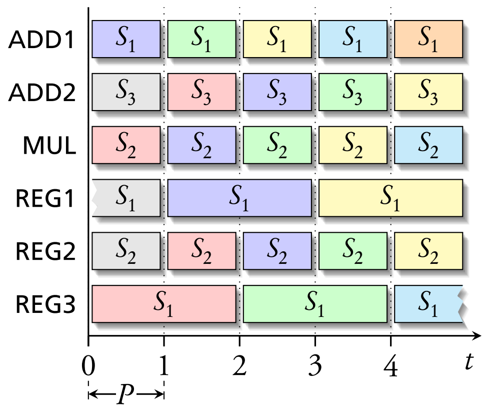


The program does not contain any loop-carried dependencies, thus the global allocation plays a secondary role. It is assumed that both additions and the multiplication are single-cycle operations. Scheduling is studied for the following set of resource allocations.

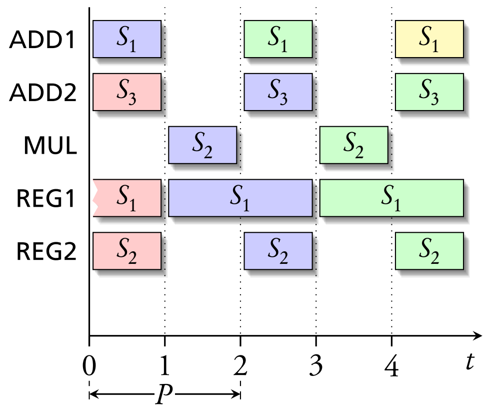
Allocation	α**(ADD)**	α**(REG)**	α**(MUL)**
A_1	2	3	1
A_2	2	2	1
A_3	1	3	1
A_4	1	2	1

As result, the following schedules have been determined using MIP techniques as proposed in Section 3.8.

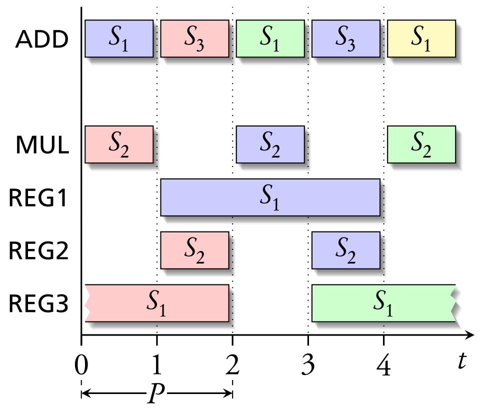
Gantt chart for allocation A_1



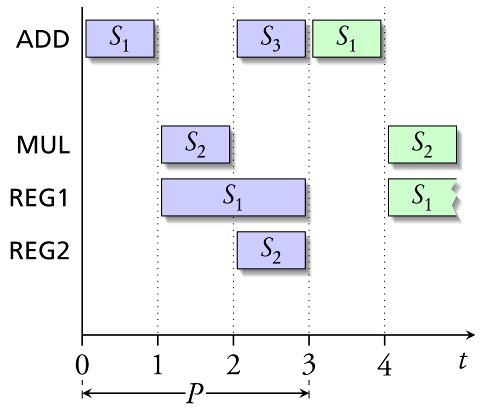
Gantt chart for allocation A_2



Gantt chart for allocation A_3



Gantt chart for allocation A_4



As expected, the first allocation (A_1) leads to the best throughput (iteration interval of one). All resources, functional units as well as the three registers, are 100% utilized. In the time interval drawn, from zero to five, resource occupancies from overall seven iterations are shown (denoted by the different colors). As can be seen from the program and the RDG, an execution in the order of $S_1 \prec S_2 \prec S_3$ has to be satisfied for each iteration. Considering the Gantt chart of allocation A_1 and the first iteration (shown in blue), according to the execution order, the computation starts with S_1 in cycle zero, whereas the result of the addition is written to register one at the end of the cycle (time step one). Since the register occupancy is caused by statement S_1, it is written in the corresponding bar of the chart. The value of variable a is needed in statement S_2 and S_3. Thus, the lifetime of variable a is two cycles. Since the next iteration (shown in green) starts already at cycle one, the result of statement S_1 has to be written to register three because register one is still occupied by the previous overlapping iteration.

The Gantt charts for allocation A_2 and A_3 are interesting in the sense of that they have the same throughput (iteration interval of $P = 2$). That means, one adder can be traded for one register.

On the first glance at the schedule of allocation A_2, one could think that one adder of the two is dispensable since both are only utilized to 50%. But this appearance is deceiving as the schedule for allocation A_4 demonstrates, which only leads to an iteration interval of $P = 3$ due to the limited number of registers.

From the determined schedules, a corresponding VLIW program can be easily generated. Here, it must be pointed out that the program length does not directly correspond to the iteration interval. In order to determine the program length, two congruent allocation patterns with a minimal distance in time have to be considered. This search method can be interpreted as an *autocorrelation*. As an example, consider the Gantt chart of allocation A_1. Although the blue and green allocation patterns are not congruent, the blue and yellow patterns are. Thus, the VLIW program length is two.

VLIW programs in accordance to the different allocations are shown in the following. The first column of a program denotes the line number, which is followed by the number of instructions that are executed in parallel within this cycle. The last column contains information about with which program line to continue in the next cycle.

WPPA program for allocation A_1

Line	**ADD1**	**ADD2**	**MUL**	**BRANCH**
0	ADD R1,IN1,IN2	ADD OUT1,R1,R2	MUL R2,R3,#7	NEXT
1	ADD R3,IN1,IN2	ADD OUT1,R3,R2	MUL R2,R1,#7	JMP 0

WPPA program for allocation A_2

Line	**ADD1**	**ADD2**	**MUL**	**BRANCH**
0	ADD R1,IN1,IN2	ADD OUT1,R1,R2	NOP	NEXT
1	NOP	NOP	MUL R2,R1,#7	JMP 0

WPPA program for allocation A_3

Line	**ADD**	**MUL**	**BRANCH**
0	ADD R1,IN1,IN2	MUL R2,R1,#7	NEXT
1	ADD OUT1,R3,R2	NOP	NEXT

```
  2     ADD R3,IN1,IN2  MUL R2,R1,#7  NEXT
  3     ADD OUT1,R1,R2  NOP           JMP 0
```

WPPA program for allocation A_4

```
Line    ADD             MUL           BRANCH
  0     ADD R1,IN1,IN2  NOP           NEXT
  1     NOP             MUL R2,R1,#7  NEXT
  2     ADD OUT1,R1,R2  NOP           JMP 0
```

On closer inspection of the four VLIW programs, allocations A_2 and A_3 are of special interest once again since both have the same throughput but the program lengths differ by a factor of two. A study of the cost associated to the different allocations would shed light on the most appropriate allocation. However, without exact estimation of the cost, it is hard to decide which of the two allocations, A_2 or A_3, is better. Regarding the area cost of an adder and a register, they are of similar size. Because of the possibility that all functional units can simultaneously access the register file, its size and the size of the multiplexer structures are increased substantially. In this context, allocation A_3 might be better. But from the point of instruction memory size, allocation A_2 would be better.

As second example, a *median filter* is considered. This filter is commonly used in image pre-processing for noise reduction. It is especially useful to reduce *salt and pepper noise* and *speckle noise*. We consider a 1×3 window that is sliding in horizontal direction over the processed image. The values in the window are sorted in numerical order. Then, the value in the center of the window, the so-called median value, is selected as output value. For example, let a window of values (4, 13, 6) be given. Sorting the window results in (4, 6, 13), and thus the median value is 6. Furthermore, the image size is full HDTV resolution with 1920×1080 pixels. The borders of the image should be treated by repeating the edge values, which results in an iteration space that is increased by one in horizontal direction. A first version of the median filter is given by the following program, where `pi` denotes the input image and `po` the output image.

```
par (x >= 0 and x < 1921 and y >= 0 and y < 1080)
{ p[x,y] = pi[x,y]                                if (x<1920);
  m[x,y] = 0                                      if (x==0);
  m[x,y] = median(p[x,y],p[x-1,y],p[x-1,y])       if (x==1);
  m[x,y] = median(p[x,y],p[x-1,y],p[x-2,y])       if (x>1 and x<1920);
  m[x,y] = median(p[x-2,y],p[x-1,y],p[x-1,y])     if (x==1920);
  po[x-1,y] = m[x,y]                              if (x>=1);
}
```

The median function for the borders ($x = 1$ and $x = 1921$) is trivial, thus we obtain.

```
par (x >= 0 and x < 1921 and y >= 0 and y < 1080)
{ p[x,y] = pi[x,y]                             if (x<1920);
  m[x,y] = 0                                   if (x==0);
  m[x,y] = p[x-1,y]                            if (x==1);
  m[x,y] = median(p[x,y],p[x-1,y],p[x-2,y]) if (x>1 and x<1920);
  m[x,y] = p[x-1,y]                            if (x==1920);
  po[x-1,y] = m[x,y]                           if (x>=1);
}
```

The filter should be mapped onto a 1×4 WPPA. Thus, the algorithm is partitioned into four stripes resulting in the following program.

```
par (x>=0 and x<481 and y>=0 and y<1080 and z>=0 and z<=3)
{ p[x,y,z] = pi[x,y,z]                          if (x<480);
  m[x,y,z] = 0                                  if (x==0);
  m[x,y,z] = p[x-1,y,z]                         if (x==1 and z==0);
  m[x,y,z] = p[x+479,y,z-1]                     if (x==1 and z>0);
  m[x,y,z] = median(p[x,y,z],p[x-1,y,z],p[x-2,y,z])
                                                if (x>1 and x<480);
  m[x,y,z] = p[x-1,y,z]                         if (x==480);
  po[x-1,y,z] = m[x,y,z]                        if (x>=1);
}
```

Subsequently, the median functions in the algorithm are replaced by compare operations. The median `m = median(a,b,c)` of three variables `a`, `b`, and `c` can be computed as follows.

```
  C1 = (a>b);
  C2 = (c>d);
  C3 = (f>e);
  d = ifrt(C1, a, b);
  e = ifrt(C1, b, a);
  f = ifrt(C2, d, c);
  m = ifrt(C3, f, e);
```

The above substitution leads to:

```
par (x>=0 and x<481 and y>=0 and y<1080 and z>=0 and z<=3)
{ S0:  p[x,y,z] = pi[x,y,z]                    if (x<480);
  S1:  m[x,y,z] = 0                            if (x==0);
  S2:  m[x,y,z] = p[x-1,y,z]                   if (x==1 and z==0);
  S3:  m[x,y,z] = p[x+479,y,z-1]               if (x==1 and z>0);
  S4:  C1[x,y,z]= (p[x,y,z]>p[x-1,y,z])        if (x>1 and x<480);
  S5:  C2[x,y,z]= (p[x-2,y,z]>d[x,y,z])        if (x>1 and x<480);
  S6:  C3[x,y,z]= (f[x,y,z]>e[x,y,z])          if (x>1 and x<480);
  S7:  d[x,y,z] = ifrt(C1[x,y,z],p[x,y,z],p[x-1,y,z])
                                               if (x>1 and x<480);
  S8:  e[x,y,z] = ifrt(C1[x,y,z],p[x-1,y,z],p[x,y,z])
                                               if (x>1 and x<480);
  S9:  f[x,y,z] = ifrt(C2[x,y,z],d[x,y,z],p[x-2,y,z])
                                               if (x>1 and x<480);
  S10: m[x,y,z] = ifrt(C3[x,y,z],f[x,y,z],e[x,y,z])
                                               if (x>1 and x<480);
  S11: m[x,y,z] = p[x-1,y,z]                   if (x==480);
  S12: po[x-1,y,z] = m[x,y,z]                  if (x>=1);
}
```

The variables with data reuse `p[x-1,y,z]` and `p[x-2,y,z]` are replaced by intermediate variables. Furthermore, all iteration dependent conditions that do not access I/O ports or feedback shift registers (FSR) can be removed. Also, statement S_1 can be removed since it covers only the case $x = 0$ but the output port is only written for $x \geq 1$.

```
par (x>=0 and x<481 and y>=0 and y<1080 and z>=0 and z<=3)
{ S0:  p[x,y,z] = pi[x,y,z]      if (x < 480); // read from input port
  S1:  p1[x,y,z]= p[x,y,z]       if (x < 480); // write to FSR1
  S2:  p2[x,y,z]= p[x,y,z]       if (x < 480); // write to FSR2
  S3:  p3[x,y,z]= p1[x-1,y,z]    if (x >= 1);  // read from FSR1
  S4:  p4[x,y,z]= p2[x-2,y,z]    if (x >= 2);  // read from FSR2
  S5:  m[x,y,z] = p3[x,y,z]      if (x == 1 and z == 0);
  S6:  m[x,y,z] = p[x+479,y,z-1] if (x == 1 and z >= 1);
  S7:  C1[x,y,z]= p[x,y,z] > p3[x,y,z];
  S8:  C2[x,y,z]= p4[x,y,z] > d[x,y,z];
  S9:  C3[x,y,z]= f[x,y,z] > e[x,y,z];
  S10: d[x,y,z] = ifrt(C1[x,y,z],p[x,y,z],p3[x,y,z]);
  S11: e[x,y,z] = ifrt(C1[x,y,z],p3[x,y,z],p[x,y,z]);
  S12: f[x,y,z] = ifrt(C2[x,y,z],d[x,y,z],p4[x,y,z]);
  S13: m[x,y,z] = ifrt(C3[x,y,z],f[x,y,z],e[x,y,z])
                                if (x >= 2 and x < 480);
  S14: m[x,y,z] = p3[x,y,z]       if (x == 480);
  S15: po[x-1, y, z] = m[x,y,z]   if (x >= 1); // write to output port
}
```

A corresponding VLIW code fragment obtained after scheduling is given in the following. The line below each VLIW instruction is a comment denoting to which statement the operation belongs. In case of the branch unit, the comment denotes two parts separated by the symbol '|'. The left part denotes the actual condition. The right part denotes the control signals to be evaluated and in dependence on the evaluation, the corresponding branch targets.

Line	**ADD0**	**ADD1**	**BRANCH**
0	NOP	NOP	IF IC0,IC1 JMP 1, 2, 3, -
	–	–	$--$ | IC0$=(x<480)$, IC1$=(x\geq 1)$
1	MOV RD2,IN1	MOV RD3,RF1	IF IC0 JMP 4, 5
	S_0	S_3	$1\leq x<480$ | IC0$=(x\geq 1)$
2	MOV RD2,IN1	NOP	IF IC0 JMP 4, 5
	S_0	–	$x<1$ | IC0$=(x\geq 1)$
3	NOP	MOV RD3,RF1	IF IC0 JMP 4, 5
	–	S_3	$x\geq 480$ | IC0$=(x\geq 1)$
4	CMP RC1,RD2,RD3	MOV OUT1,RD5	IF IC0,RC2 JMP 6, 7, 8, 9
	S_7	S_{15}	$x\geq 1$ | IC0$=(x<480)$, $(RC2)$
5	CMP RC1,RD2,RD3	NOP	IF IC0,RC2 JMP 6, 7, 8, 9
	S_7	–	$x<1$ | IC0$=(x<480)$, $(RC2)$
6	MOV RF2,RD2	MOV RD1,RD6	IF IC0,RC1 JMP 10,11,12,13
	S_2	S_{12}	$x<480, RC2$ | IC0$=(x\geq 2)$, $(RC1)$
...			

For explanation, in the first line of the VLIW program, only the branch unit is active and decides which part of the image is actually considered. Depending on the two control signals IC0 and IC1, generated by a global controller, the program flow is chosen. For instance, if $x < 1$, the program continues in line 2 with statement S_0 of the algorithm, where input IN1 is read to register RD2. Simultaneously, the branch unit checks IC0, which denotes whether x is greater or equal to one ($x \geq 1$) in the next iteration. According to this evaluation the program continues with line 4 or 5. In both cases statement S_7 is executed, where the values of RD2 (variable p) and RD3 (variable p3) are compared and the result is written to control register RC1, which can be evaluated by the branch unit in the next instruction, as for instance, in line 6.

5.5 High-Level Synthesis for FPGA Targets

Several algorithms from various application domains, some of which are taken from the well-known MediaBench suite [LPM97], have been synthesized. In this context, we profiled the JPEG and MPEG2 algorithms and identified some of the most computationally intensive loop kernels. For these and some further algorithms, characteristics, such as the size of the processor array, the number of nodes and edges of the reduced dependence graph, and if an algorithm has run-time (RT) or iteration (IT) dependent conditions, are summarized in Table 5.3. For most of these examples, the resource-constrained scheduling was performed within a split of a second to a second depending on how many MIP instances (bisection method to find the optimal iteration interval, test of several loop vector permutations in case of partitioning) had to be generated. All algorithms were implemented using 16-bit integer or fixed point arithmetics and synthesized using the *Xilinx ISE 6.3i* toolchain targeting a *Virtex-II 8000* FPGA. The synthesis results are shown in Table 5.4.

For each example, the cost in terms of FPGA primitives, the maximum achievable clock frequency, the total execution time for the algorithm in clock cycles, and the average number of clock cycles between the availability of two successive output instances (for example samples or pixels) is given. The latter is the inverse of the throughput, that is, a smaller number denotes a higher throughput. The initial latency does not affect the throughput. Note that an output interval less than 1 denotes that more than one piece of output data is produced per clock cycle. The Gaussian filter was partitioned, so that implementation costs remain mostly constant for larger image sizes. In this case of course, the latency raises. For the FIR filter, we used partitioning to trade throughput and cost for constant latency. The projected implementation is fast but very expensive, whereas partitioning allows for a fine-grained design space exploration. The results for the matrix multiplication benchmark are similar but partitioning was applied in such a way, that the total execution time could also be selected according to the user's requirements.

Generally, the maximum achievable clock speed is lower for partitioned implementations. This is caused by the more complex control logic. Although global control signals are propagated efficiently through the array, the local controller inside each PE is currently a pure combinational circuit. Here, the performance can be increased, by applying additional pipelining.

In all experiments, the HDL synthesis done by PARO took only a few seconds for each example, whereas the subsequent logic synthesis and place-and-route performed by the tools of the FPGA vendor required several minutes. The time for performing the place-and-route could be reduced by employing hard macro generation for each processor type [Bed04].

Table 5.3: Characteristics of the considered algorithms.

Algorithm	Abbrv.	No. of PEs	No. of RDG nodes / edges	Conditions RT / IT
Edge Detection				
– 100×100 image, sequential	ED1	1	15 / 21	yes / yes
– 100×100 image, partitioned	ED2	1×4	31 / 44	yes / yes
– 1000×1000 image, sequential	ED3	1	15 / 21	yes / yes
– 1000×1000 image, partitioned	ED4	1×4	31 / 44	yes / yes
Gaussian Filtering				
– 100×100 image, 3×3 mask	GAUSS1	3×3	25 / 65	no / yes
– 1000×1000 image, 3×3 mask	GAUSS2	3×3	25 / 65	no / yes
– 2000×2000 image, 3×3 mask	GAUSS3	3×3	25 / 65	no / yes
– 1000×1000 image, 5×5 mask	GAUSS4	5×5	25 / 65	no / yes
– 2000×2000 image, 5×5 mask	GAUSS5	5×5	25 / 65	no / yes
FIR Filter				
– 64 Taps, sequential	FIR1	1	19 / 36	no / yes
– 64 Taps, partitioned	FIR2	1×4	30 / 109	no / yes
– 64 Taps, partitioned	FIR3	1×8	30 / 109	no / yes
– 64 Taps, projected	FIR4	1×64	19 / 36	no / yes
Matrix Multiplication				
– 6×6 matrix size, sequential	MM1	1	16 / 28	no / yes
– 6×6 matrix size, partitioned	MM2	2×2	22 / 59	no / yes
– 6×6 matrix size, projected	MM3	6×6	16 / 28	no / yes
– 100×100 matrix size, partitioned	MM4	2×2	22 / 59	no / yes
Discrete Cosine Transformation	DCT	2	176 / 168	no / no
Elliptical Wave Digital Filter	EWDF	1	80 / 104	no / no
Partial Differential Equation Solver	PDE	1	29 / 44	no / yes
MPEG2 Quantisizer	QUANT	1	21 / 25	yes / no
JPEG Loop 1	JPEG1	1	21 / 31	yes / yes
JPEG Loop 2	JPEG2	1	17 / 21	yes / no

In order to evaluate the scalability of the scheduler, the considered discrete cosine transformation (DCT) was fully unrolled and factorized, which led to an RDG with 9 244 nodes and 9 180 edges. The number of available multipliers was restricted to 24, whereas unlimited adders/subtracters were assumed. Solving the MIP took 10.5 minutes for the minimum possible iteration interval of 18 cycles.

Table 5.4: Synthesis results.

Algorithm	No. of LUTs	No. of FFs	No. of MULTs	No. of BRAMs	Max. Clock (MHz)	Exec. time (cycles)	Avg. output interval (cycles)
ED1	475	324	0	4	146	$3.0 \cdot 10^4$	3
ED2	2962	1455	0	4	143	$7.7 \cdot 10^3$	3
ED3	508	344	0	20	148	$3.0 \cdot 10^6$	3
ED4	2997	1913	0	44	120	$7.5 \cdot 10^5$	3
GAUSS1	655	1439	9	2	171	$1.0 \cdot 10^4$	1
GAUSS2	683	1463	9	2	171	$1.0 \cdot 10^6$	1
GAUSS3	696	1472	9	4	169	$4.0 \cdot 10^6$	1
GAUSS4	1538	3909	25	4	171	$1.0 \cdot 10^6$	1
GAUSS5	1555	3922	25	8	169	$4.0 \cdot 10^6$	1
FIR1	130	106	1	0	162	68	64
FIR2	773	834	4	0	125	71	16
FIR3	1915	1014	8	0	132	71	8
FIR4	5782	9089	64	0	167	68	1
MM1	204	157	1	0	131	250	6
MM2	829	795	4	0	115	72	1.5
MM3	1888	4067	36	0	166	20	0.28
MM4	1736	913	4	505	97	$2.6 \cdot 10^5$	3614
DCT	1754	1152	8	1	130	94	0.65
EWDF	1169	624	1	0	94	$2.5 \cdot 10^5$	1
PDE	619	502	1	0	128	$1.2 \cdot 10^5$	(1 result)
QUANT	637	1190	1	0	141	222	2.95
JPEG1	82	79	0	0	224	63	1
JPEG2	570	1139	0	0	158	126	1

5.6 Loop Unrolling versus Loop Partitioning

In this section, our proposed parallelization approach (loop partitioning) is compared with loop unrolling, which is commonly used in order to increase the throughput. Figure 5.6(a) shows the iteration space with omitted data dependencies of a 4-tap FIR filter, which is used to illustrate the fundamental difference between the two approaches. The pseudo code of an N-tap FIR filter is given in Figure 5.7.

Figure 5.6(b) shows the loop unrolling approach, where the innermost loop is completely unrolled and mapped onto a processor element (PE) with 4 MUL and ADD units. In addition to full loop unrolling, one can partially unroll the loop nest for an iteration variable. The unroll factor u denotes how often the loop body is duplicated. In the algorithm shown in Figure 5.8, the iteration variable `j` of the N-tap FIR filter is unrolled by a factor of 2.

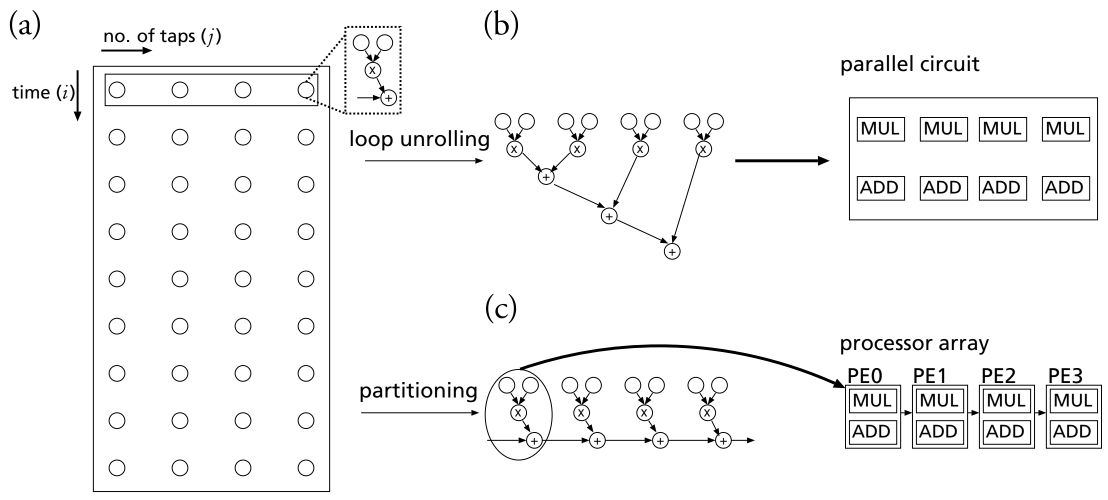


Figure 5.6: In (a), the iteration space of a loop program is sketched. In (b), the synthesis problem in the case of a partially unrolled loop is shown. In (c), the synthesis problem for the partitioned loop program is depicted.

```
FORALL (i >= 0 and i <= T-1)        // T: number of input samples
{ FORALL (j >= 0 and j <= N-1)      // N: number of filter taps
  { IF (i==0) THEN a[i,j] = a_in[j];  // Read filter coefficient
              ELSE a[i,j] = a[i-1,j];
    IF (j==0) THEN
    { u[i,j] = u_in[i];                    // Read input sample
      y[i,j] = a[i,j] * u[i,j];
    }
    ELSE
    { y[i,j] = y[i,j-1] + a[i,j] * u[i,j];
      IF (i==0) THEN u[i,j] = 0;
                ELSE u[i,j] = u[i-1,j-1];  // Enables data reuse
    }
    IF (j == N-1) y_out[i] = y[i,j];       // Write output
  }
}
```

Figure 5.7: Pseudo code of an N-tap FIR filter.

Figure 5.6(c) shows the FIR filter in the case of the partitioning approach. Here, the iterations within the tile are processed in parallel by 4 PEs. Each PE contains one MUL and one ADD unit. The tiles are processed *global sequentially*. The pseudo code of the partitioned FIR filter is given in Figure 5.9.
The major questions that need to be answered on basis of several algorithms are:

```
FORALL (i >= 0 and i <= T-1)
{ FORALL (j >= 0 and j <= N/2-1)
  { IF (i==0) THEN
    { a0[i,j] = a_in0[j];
      a1[i,j] = a_in1[j];
      u1[i,j] = 0;
    }
    ELSE
    { a0[i,j] = a0[i-1,j];
      a1[i,j] = a1[i-1,j];
      u1[i,j] = u0[i-1,j];
    }
    IF (j==0) THEN
    { u0[i,j] = u_in[i];
      y0[i,j] = a0[i,j] * u0[i,j];
    }
    ELSE
    { y0[i,j] = y1[i,j-1] + a0[i,j] * u0[i,j];
      IF (i==0) THEN u0[i,j] = 0;
                ELSE u0[i,j] = u1[i-1,j-1];
    }
    y1[i,j] = y0[i,j] + a1[i,j] * u1[i,j];
    IF (j == N-1) y_out[i] = y1[i,j];
  }
}
```

Figure 5.8: FIR filter unrolled by a factor of u=2.

- What is the quantitative trade-off in terms of hardware cost, performance, and power between loop unrolling and partitioning?
- What should be the optimal granularity of resources in parallel processors for efficient mapping?

In the following, we answer the above questions by a quantitative analysis of the hardware generated for the transformed loop programs by the PARO design system. In more detail, we compare the two methods described before with respect to resource usage, performance (clock frequency and throughput), and power. Several experiments with different setups have been performed. Also the difference between the usage of dedicated DSP elements of the target devices or the synthesis purely in LUTs has been studied. The synthesis results were obtained from *Xilinx ISE 9.1* for a *Xilinx Virtex 4* FPGA (xc4vlx100-12ff1513). For the estimation of the dynamic power, *Xilinx XPower* was used in combination with the *post-place & route simulation*

```
FORALL (i >= 0 and i <= T-1)
{ FORALL (j >= 0 and j <= P-1)        // P: number of PEs
  { FORALL (k >= 0 and k <= N/P-1)    // Iteration over
                                      // different tiles
    { IF (i==0) THEN
      { a[i,j,k] = a_in[j+k];
      }
      ELSE
      { a[i,j,k] = a[i-1,j,k];
        IF (j>0)     THEN
          u[i,j,k] = u[i-1,j-1,k];     // Intra-tile comm.
        ELSEIF (k>0) THEN
          u[i,j,k] = u[i-1,j+P-1,k-1]; // Inter-tile comm.
      }
      IF (j==0 and k==0) THEN
      { u[i,j,k] = u_in[i];
        y[i,j,k] = a[i,j,k] * u[i,j,k];
      }
      ELSEIF (i==0) THEN u[i,j,k] = 0;
      IF (j>0) THEN
        y[i,j,k] = y[i,j-1,k] + a[i,j,k] * u[i,j,k];
      ELSEIF (k>0) THEN
        y[i,j,k] = y[i,j+P-1,k-1] + x[i,j,k];
      IF (j==P-1 and k==N/P-1)
        y_out[i] = y1[i,j,k];
    }
  }
}
```

Figure 5.9: Partitioned FIR filter.

models of the designs. The usage of BRAMs was disabled throughout all experiments except for the matrix multiplication where we allowed BRAMs for the storage of intermediate data.

In the first experiment, an 8-bit 64-tap FIR filter is considered. The coefficients of the filter are reconfigurable, that means, they were implemented as inputs. The results are shown in Table 5.5, where the first column denotes the unroll factor *u* in case of the standard high-level synthesis (HLS) approach, and in case of a processor array approach, the number of processing elements (#PE). This number also corresponds to the total number of available multipliers and adders for the data-path implementation in both variants. The results for the HLS approach and for the processor follows in the table. Finally, in the last column of the table, the number of

Table 5.5: Resource usage and performance of different 64-tap FIR filter implementations.

			HLS			DSP48
u	**LUTs**	**FFs**	**Clock**	**Through-**	**Power**	**slices**
[no.]	[no.]	[no.]	[MHz]	**put** [MB/sec]	[mW]	[no.]
2	207	152	189	5.6	19	0
4	366	252	197	11.7	24	0
8	723	501	201	24.0	37	0
16	1428	999	168	40.1	49	0
32	3037	2143	164	78.2	93	0
64	6681	4540	156	148.8	189	0
2	150	126	163	4.9	18	2
4	215	192	192	11.4	27	4
8	347	324	193	23.0	38	8
16	636	727	185	44.1	60	16
32	1471	1623	182	86.8	114	32
64	3284	3333	143	136.4	240	64

					Processor Array					DSP48
#PE	**LUTs**	**Diff.**	**FFs**	**Diff.**	**Clock**	**Diff.**	**Through-**	**Power**	**Diff.**	**slices**
[no.]	[no.]	[%]	[no.]	[%]	[MHz]	[%]	**put** [MB/sec]	[mW]	[%]	[no.]
2	242	16.9	154	1.3	218	15.3	6.5	23	21.1	0
4	395	7.9	256	1.6	219	11.2	13.1	29	20.8	0
8	734	1.5	471	-6.0	224	11.4	26.7	41	10.8	0
16	1401	-1.9	861	-13.8	218	29.8	52.0	62	26.5	0
32	2748	-9.5	1698	-20.8	210	28.0	100.1	103	10.8	0
64	4907	-26.6	4321	-4.8	222	42.3	211.7	187	-1.1	0
2	185	23.3	140	11.1	231	41.7	6.9	27	50.0	2
4	249	15.8	224	16.7	213	10.9	12.7	32	18.5	4
8	413	19.0	396	22.2	218	13.0	26.0	47	23.7	8
16	738	16.0	736	1.2	185	0.0	44.1	64	6.7	16
32	1382	-6.1	1418	-12.6	212	16.5	101.1	130	14.0	32
64	2631	-19.9	2794	-16.2	192	34.3	183.1	219	-8.8	64

consumed DSP48 slices is given. Two columns depict the cost in terms of the number of look-up tables (LUTs) and slice flip-flops (FFs), two other columns represent the performance metrics clock frequency and throughput followed by the dynamic power consumption. Next to each column (LUTs, FFs, clock, power) of the processor array implementations, the relative difference compared with the HLS approach is given. Since the throughput is proportional to the clock frequency, the relative difference is the same and is omitted in the table.

In the upper half of Table 5.5, the usage of dedicated DSP48 slices was disabled. Here, for the lowest resource usage ($u = 2$), the processor array implementation is more than 15% faster than the unrolled variant but also requires more resources, that is 16.9% more LUTs and 1.3% more flip-flops. Note that the power dissipation is proportional to the clock frequency. It can be noticed that for increasing u, the clock frequency of the HLS approach is decreasing whereas for the processor array implementations it is almost constant. It seems that the place and route routines do not perform so well for larger designs (flattened register-transfer circuits), whereas the clustering of operations into several processor elements performs much better in

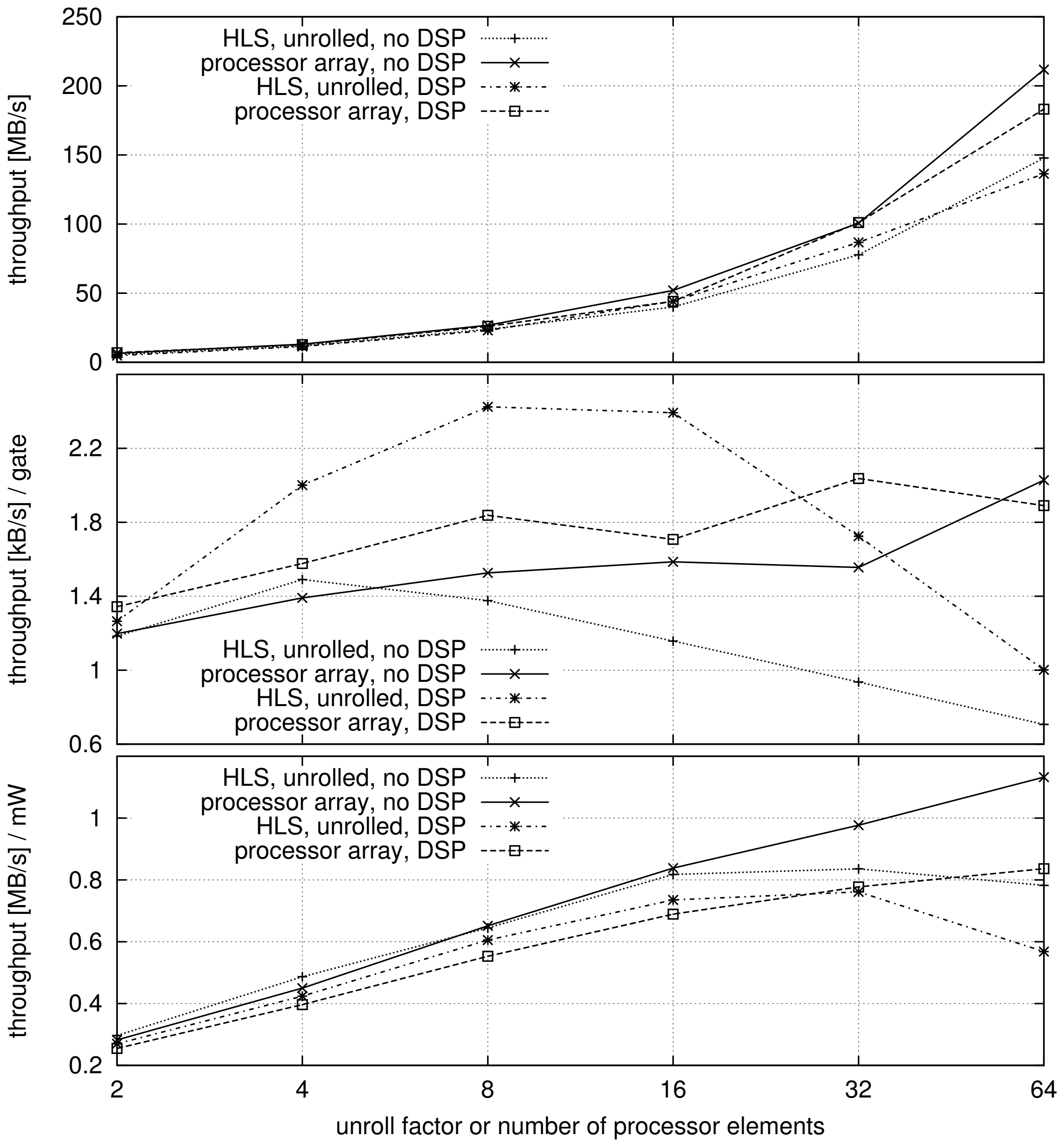


Figure 5.10: Throughput, throughput per gate, and the throughput per mW for different unroll factors and numbers of processor elements, respectively.

terms of clock speed. A closer look at the placed and routed designs reveals that longer wires and more multiplexing are the reasons for the lower clock frequency and for the higher amount of LUTs for the loop unrolled versions compared with the partitioned versions.

In a second run of experiments, the multiplications in the FIR algorithms were implemented by the Xilinx DSP48 slices. The results for different unroll factors and number of processing elements are also shown in Table 5.5. Because of the prede-

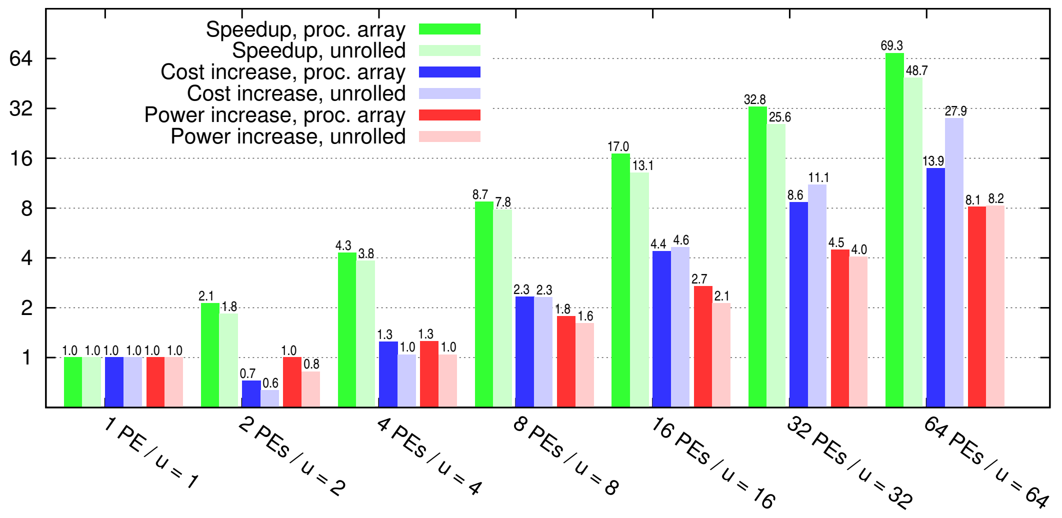


Figure 5.11: The speedup (with respect to throughput), cost and power increase for different 64-tap FIR filter implementations in relation to a sequentialized implementation, where only a single processing element with one multiplier and one adder was available, is shown.

termined location of the DSP48 slices, the values are fluctuating more than in the previous case (no DSP48 slices). However, apart from one outlier, the throughput of the processor array approach is 11 to 42% higher as compared with the unrolled approach. In Figure 5.10, the throughput itself, normalized by the gate count, and the throughput per mW is shown. In Figure 5.11, the speedup characterizes the performance gain with respect to the throughput for the FIR filter algorithm. The cost increase is related to the gate count of the designs. In the single PE solution of the FIR filter, the 64 iterations are executed sequentially within one PE. For this solution both, throughput and cost, are normalized to 1.0. Partitioning the algorithm to 2, 4, . . . , 64 PEs theoretically enables also a higher throughput by the same factor. The superlinear speedup in case of the processor array implementations is because of the increasing clock frequency for larger numbers of PEs. The moderate cost and power increase is caused by the decreasing amount of intermediate data, which have to be stored internally in the processor array.

Since the difference for the $u = 64$ implementations was tremendous, several other fully unrolled/full size versions have been studied. The results are shown in Figure 5.12. When enabling the DSP48 slices, the throughput for the unrolled approach sharply falls for higher numbers of taps. Note that these results should not be used to compare the achievable throughput between DSP48 and LUT-based implementations since an optimal pipelined version of the DSP48 slices has not been used in the experiments.

As a second algorithm, a DCT width 8-bit I/O and internally up to 16 bits datapath was studied. The results for different numbers of available multipliers, implemented in DSP48 slices, are shown in Table 5.6. Unlike the FIR filter, the DCT has no loop-carried data dependencies. The loop body of the DCT was unrolled four times and a processor array consisting of 4 processing elements was considered. The unrolled loop body contains 96 multiplications and therefore, the results for the cost (LUTs, FFs) of the last experiment, where 96 multipliers were available, are quite close. However, in three of four cases, the processor array implementation has a better throughput of up to 21% along with an increased power dissipation by only 11%.

For the last evaluation, an algorithm for the multiplication of two 64×64 matrices was considered. Using the processor array approach, again a higher clock frequency and throughput as compared with the loop unrolling approach using the same number of resources (multipliers and adders) was achieved (see Table 5.7). The higher area cost in terms of LUTs for the processor array approach is caused by the con-

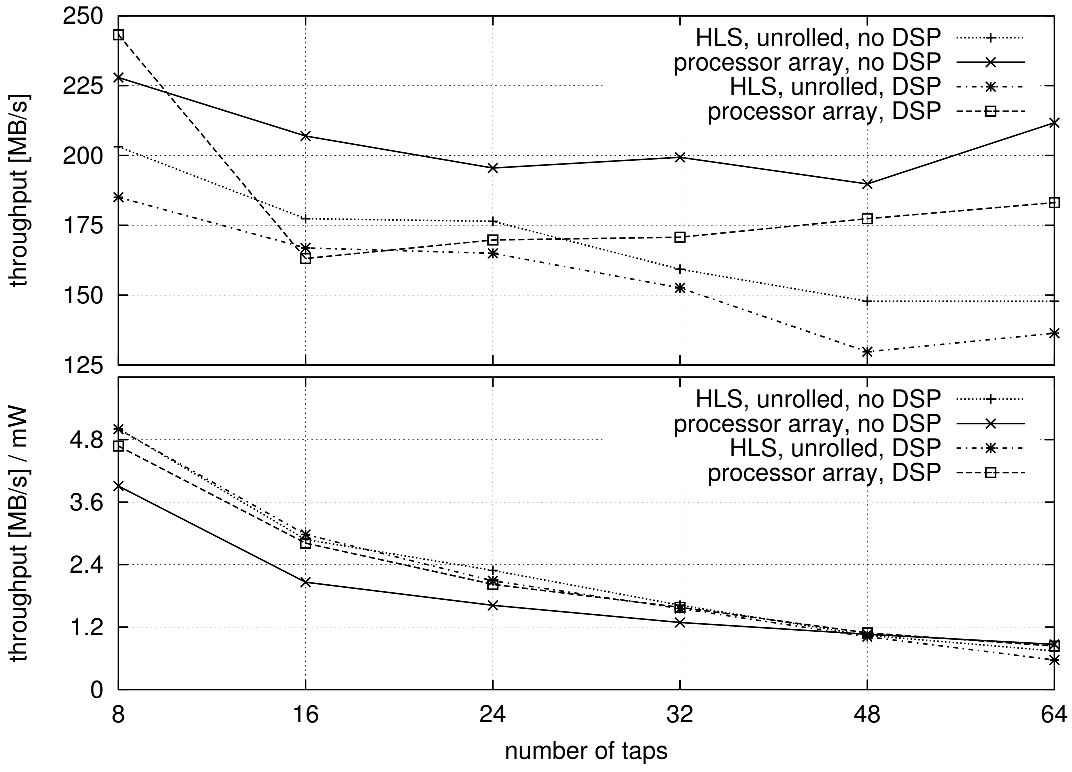


Figure 5.12: Throughput of different fully unrolled and full size FIR filter implementations.

Table 5.6: Resource usage and performance of different DCT implementations.

no of mult.	LUTs [no.]	FFs [no.]	HLS Clock [MHz]	Through-put [MB/sec]	Power [mW]	DSP48 slices [no.]
16	3935	1717	152	48.3	438	16
32	3944	2644	186	118.3	528	32
48	3590	3141	185	176.4	700	48
96	3023	3556	190	362.4	867	96

no of mult.	LUTs [no.]	Diff. [%]	FFs [no.]	Diff. [%]	Processor Array Clock [MHz]	Diff. [%]	Through-put [MB/sec]	Power [mW]	Diff. [%]	DSP48 slices [no.]
16	3811	-3.2	1760	2.5	163	7.2	51.8	483	10.3	16
32	3808	-3.4	2543	-3.8	175	-5.9	111.3	638	20.8	32
48	3457	-3.7	3080	-1.9	224	21.1	213.6	778	11.1	48
96	3067	1.5	3600	1.2	207	8.9	394.8	880	1.5	96

Table 5.7: Resource usage and performance of different matrix multiplication implementations.

u [no.]	LUTs [no.]	FFs [no.]	HLS Clock [MHz]	Through-put [MB/sec]
2	784	474	148	9.2
4	1459	812	145	18.2
8	3049	1724	123	30.8
16	5937	3166	125	62.4

#PE [no.]	LUTs [no.]	Diff. [%]	FFs [no.]	Processor Array Diff. [%]	Clock [MHz]	Diff. [%]	Through-put [MB/sec]
2	895	14.2	603	27.2	153	3.4	9.5
4	1629	11.7	1082	33.3	149	2.8	18.6
8	3030	-0.1	2038	18,2	152	23.6	38.1
16	5895	-0.1	3991	26,0	172	37.6	85.8

trol overhead when partitioning the algorithm onto several processor elements. 8 BRAMs were instantiated for data reuse in both the variants.

In addition, we also performed the experiments with Altera tools and devices. Here, we obtained similar but closer results between unrolling and partitioning since Altera's placement routines seem to uncover better locality in many cases.

5.7 Real World Case Study

A multiresolution algorithm, such as used in digital video or medical image processing, has been considered as a real world case study. Characteristic for these filters is the processing of an image at different resolutions (see Figure 5.13). Here, in a decomposition phase, two image pyramids with subsequently reduced resolutions ($g_0(1024 \times 1024)$, $g_1(512 \times 512)$, … and $l_0(1024 \times 1024)$, $l_1(512 \times 512)$, …) are constructed. The images g_i, $1 \leq i \leq 5$ are so called Gaussian images that are obtained by downsampling the previous image g_{i-1} by a factor of two in both directions and subsequent smoothing by a 3×3 Gaussian filter. The second pyramid (l_j, $0 \le j \le 4$) represents the edges in the image at different resolutions. The edges are obtained by a Laplacian filter, hence the images l_j are called Laplacian images. At each layer, a Gaussian and a Laplacian image is fed to a filter, except to the last filter, where two times g_5 is used. Filtering of an image at different resolutions has the advantage that each feature of the image can be processed at its most appropriate scale, while the filter kernel can be kept small simultaneously. After filtering, the image is reconstructed from the bottom up by expanding and adding two images at each layer.

The operations (downsampling, upsampling, lowpass) within the decomposition and reconstruction phase have little computational complexity. The situation is different for the filter kernels. Here, a non-linear gradient adaptive filter is used. The filter reduces noise in the images significantly, while fine and sharp image structure are preserved.

The filter has been formulated in one non-perfectly nested PAULA program. It mainly consists of the following parts:

- Border treatment (*mirroring*),

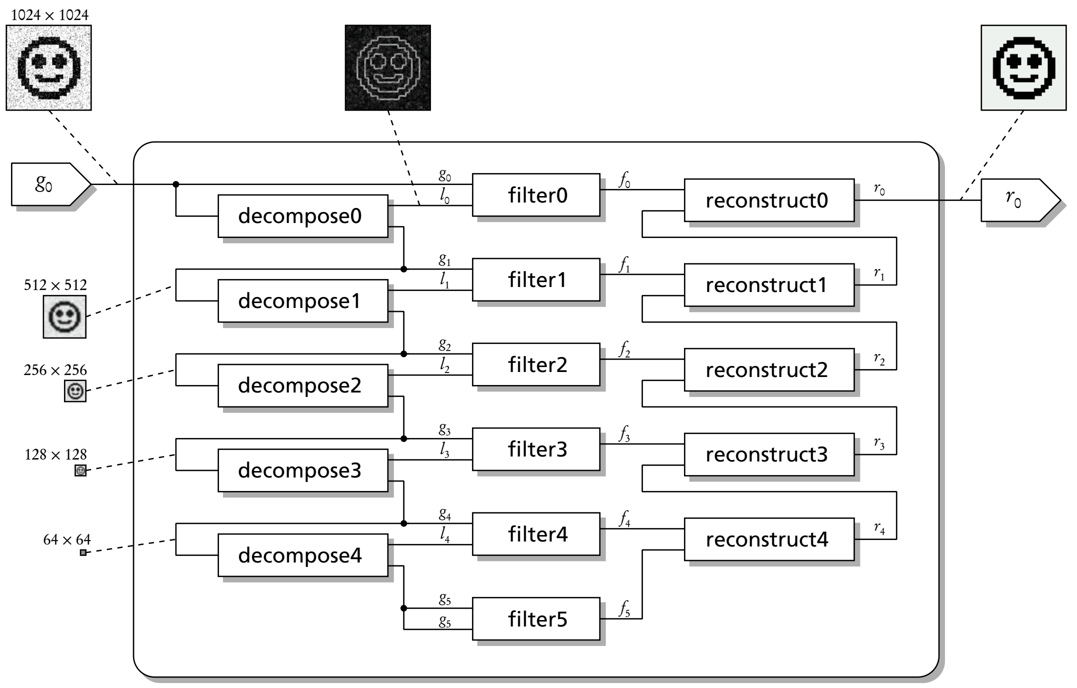


Figure 5.13: Multiresolution filter with five layers and six filter kernels, respectively.

- Gradient computation,
- Weighting kernel computation,
- Weighting kernel normalization,
- Convolution.

Overall, the program consists of 306 lines of code, where half of the code is devoted to the border treatment.

Table 5.8: Multiresolution filter: Number of operations within the filter kernel.

Operation	**Number**
data type conversions	3
comparisons	3
additions	26
divisions	2
exponential functions	9
multiplications	35
shift operations	33
subtractions	16
subtractions (unary)	9

After applying several high-level transformations to the program (cf. Section 4.2), the number of operations for one iteration or rather for the computation of one pixel is known. An overview of types and number of operations is given in Table 5.8.

Table 5.9: Multiresolution filter: Resource allocation for each filter kernel.

Filter	P	**mult**	**exp**	**div1**	**div4**	**div8**
		$w=5$	$w=23$	$w=53$	$w=56$	$w=56$
		$\delta=1$	$\delta=1$	$\delta=1$	$\delta=4$	$\delta=8$
filter0	2	19	5	1	0	0
filter1	8	5	2	0	1	0
filter2	32	2	1	0	0	1
filter3	128	1	1	0	0	1
filter4	512	1	1	0	0	1
filter5	2048	1	1	0	0	1

The numerous propagations and assignment operations, for instance for the border treatment are not counted in Table 5.8. The corresponding reduced dependence graph consists of 384 nodes and 633 edges (see Figure 5.14). The input data streams with 50 million pixels per second through the input port. Thus, the assumption was

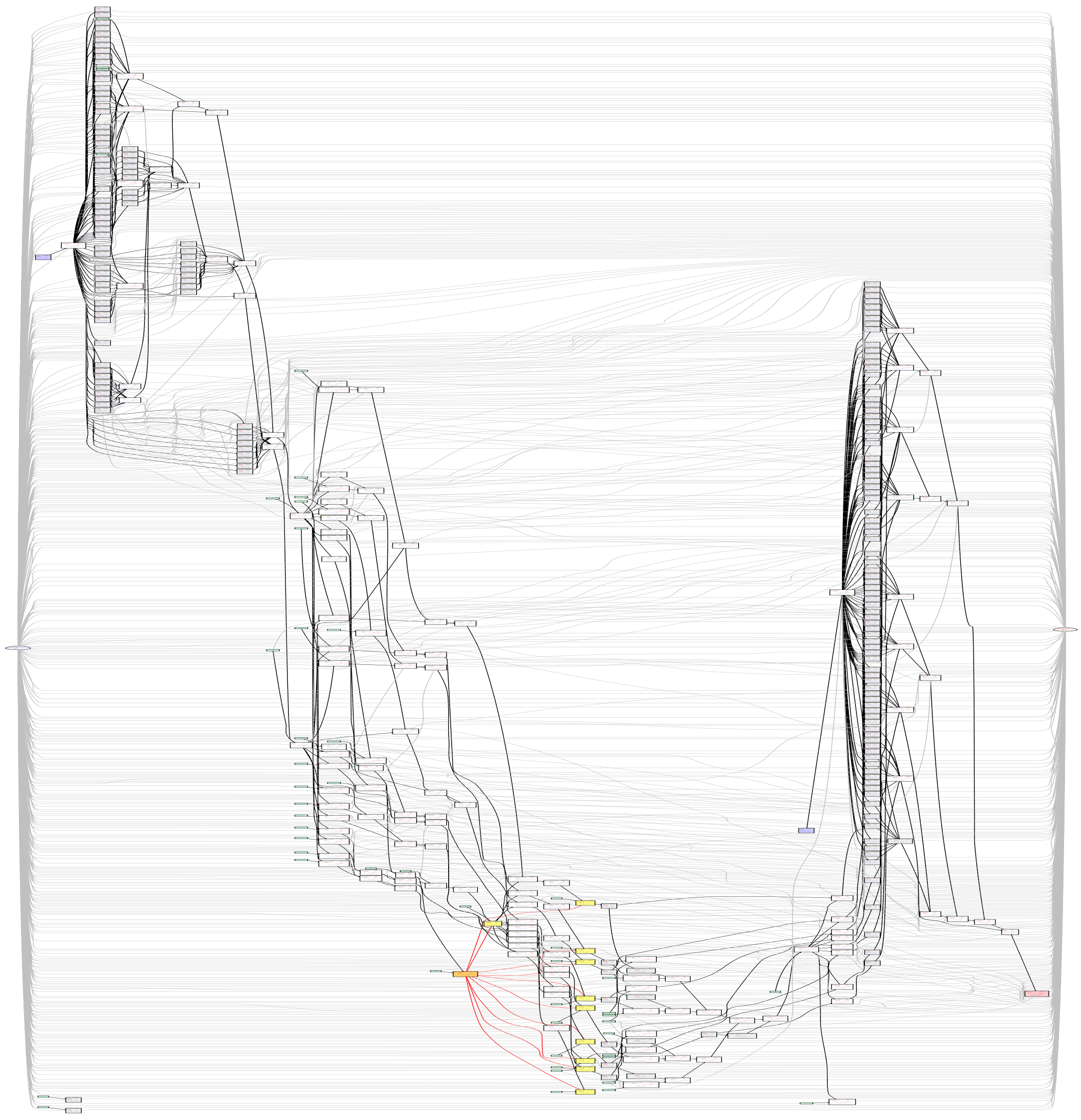


Figure 5.14: Reduced dependence graph of filter kernel, automatically generated by the PARO design system.

made that a filter can be synthesized with a clock frequency of 100 MHz leading to an iteration interval of $P = 2$ for *filter0*. That means, in every second cycle, a new pixel has to be processed by *filter0*. The data streams row by row into the array, thus an appropriate LSGP partitioning scheme has been chosen. Since divisions, exponential functions, and multiplications are very costly in hardware, the corresponding functional units should be shared as much as possible. The 35 multiplications in Table 5.8 have to be performed within two cycles. In the ideal case, this can be achieved with an allocation of $\lceil 35/2 \rceil = 18$ multipliers that can start a new operation in each cycle. Likewise, a divider should be chosen that can start a new operation at each cycle, which is sufficient for computing the two divisions within the filter kernel.

In *filter1*, only a quarter of the pixels have to be processed. Therefore, the iteration interval can be quadrupled to $P = 8$ and the number of resources can be reduced. For instance, one divider, that can start a new operation in every second cycle, and five multipliers are sufficient.

The different allocations for each filter are given in Table 5.9. The execution times w and pipeline rates δ of the functional units are also given. Scheduling and synthesis is as easy as changing the parameters of Table 5.9 within the allocation section of the program.

Table 5.10: Results for each filter.

Filter	P	**local latency**	**solving time**	**parallel iterations**
filter0	2	169	2.1 sec	85
filter1	8	177	3.2 sec	23
filter2	32	175	2.9 sec	6
filter3	128	175	6.9 sec	2
filter4	512	175	23.1 sec	1
filter5	2048	175	88.0 sec	1

The solution times and scheduling results are shown in Table 5.10. For each of the first four filters *filter0* to *filter3* an optimal schedule was found in less than ten seconds. For larger iteration intervals ($P = 512$ and $P = 2048$), a schedule was found in 23 and 88 seconds, respectively. Here, the possible scheduling window is significantly larger than the derived local latency of 175 cycles. The last column of the table denotes the number of iterations that run simultaneously at the same time. For example, *filter0* processes a new pixel in every second cycle but because of the long execution times of the complex operations, local latency amounts to 169 cycles. That is, computations from $\lceil 169/2 \rceil = 85$ iterations are executed in parallel.

5.8 Summary

In this chapter, the new scheduling techniques presented in Chapter 3 have been evaluated in several experiments, ranging from small descriptive examples of several selected algorithms from benchmarks to a complex real world case study.

The proposed sequentialization constraints have been quantitatively compared to an existing method. Here, already few randomly selected examples demonstrate an improvement of up to 39% in execution time (latency of the found solution).

Examples for predicated as well as conditional execution have been presented. For the first time, scheduling with register and channel constraints for WPPAs has been demonstrated for descriptive examples. Here, also the corresponding VLIW programs (target code) have been presented.

Afterwards, the full application of the PARO synthesis tool has been demonstrated, where more than 25 well-known benchmark algorithms have been scheduled and synthesized. The complexity of the corresponding RDGs ranged from dozens to more than 150 nodes. An optimal schedule could be derived within seconds.

It has been shown in Section 5.6 that the proposed array approach (loop partitioning) is extremely powerful, when compared with loop unrolling. The usage of the same design tool (PARO) enabled, for the first time, a fair quantitative evaluation of the two approaches for a set of computationally intensive algorithms. Because of its regularity and the possible clustering of resources into several processing elements, the processor array approach achieves a far better throughput, of up to 42% more compared with loop unrolling in all experiments for FPGA targets. For the cost and power metrics, it can be noted that the loop unrolling results are better for smaller designs whereas for larger designs, the processor array approach achieves smaller cost and power values.

Finally, a complex (RDG with 384 nodes and 633 edges) real world application for image processing has been presented. Moderate times for finding optimal schedules that interleave operations from up to 85 iterations simultaneously, demonstrate the applicability, even also for very complex examples.

Chapter 6

Conclusions and Future Work

In this chapter, the key contributions of this thesis are briefly summarized. Furthermore, directions of future work in this important research area are outlined.

6.1 Summary

This monograph has presented novel contributions in the areas of modeling and resource-constrained scheduling for nested loop programs.

More specific, a new class of algorithms called *dynamic piecewise linear algorithms* and a corresponding graph representation for modeling iterative, multi-dimensional data flow has been presented in the thesis. This class of dynamic piecewise linear algorithms extends well-known models, that are based on systems of recurrence equations, defined over polyhedral iteration domains. Its novel extension enables the modeling of a specific type of dynamic data dependencies arising in many important algorithm classes that existing loop parallelizers were just not able to handle. By this enhancement, the range of applications with multi-dimensional data flow, that can be parallelized and mapped onto massively parallel processor arrays, is significantly increased. For instance, a lot of computationally intensive applications for video and image processing, which consist of nested loop programs with only few and simple run-time dependent conditions, are now parallelizable and mappable to either dedicated hardware accelerators or tightly-coupled, programmable processor arrays.

On the basis of the class of dynamic piecewise linear algorithms, the language PAULA has been introduced. It allows modeling dataflow-intensive applications. It is intended for designing highly parallel algorithms at instruction, data, and loop-level parallelism. The PAULA language permits very compact and efficient behavioral descriptions and serves as design entry when generating dedicated hardware accelerators, or might be used as high-level programming language for tightly-coupled multi-processor architectures. The language covers a broad range of applications from the areas of digital image, video and other signal processing, linear algebra, cryptography, and many other scientific computing domains where efficient parallelization techniques and hardware accelerators are indispensable. The intention was not to propagate a new parallel programming language but to have one specially tailored for the concepts described in the thesis. It should be noted that the proposed language features may also be expressed in a 'C-like' or any other high-level programming language. But, in order to not mislead a programmer, the catalog of restrictions and modifications would be as long as the description of the PAULA language itself.

Exact and holistic scheduling techniques have been developed, that incorporate both, the local allocation and one or more levels of global allocation. By this close integration, performance-optimal schedules (in terms of throughput and overall execution time of an algorithm) are derived. In order to obtain these schedules, an approach based on mixed integer programming has been developed. For the first time, the local schedule as well as the global schedule for possibly several levels of partitioning is optimized at the same time. In this context, a new *serialization constraint* has been presented that leads to better schedules (up to 39%) than existing approaches. The second main contribution in the area of scheduling has been the formulation of resource constraints that take the mutual exclusivity of iteration dependent conditions as well as of run-time dependent conditions into account. In addition, the MIP-based techniques have been extended by further constraints to cope with the constraints induced by a given fixed programmable processor array (WPPA). These constraints consider local register constraints within the processors and channel constraints of the communication structure.

Several experimental results demonstrate the strength of the proposed methods. Here, more than 25 selected algorithms have been scheduled and synthesized automatically as test cases. The complexity of the corresponding RDGs ranged from dozens to more than 150 nodes. In each case, an optimal schedule was derived within seconds.

The powerfulness of the proposed array approach (loop partitioning) compared with loop unrolling has been demonstrated. For the first time, the usage of the same design tool (PARO) enabled a fair, quantitative evaluation of the two approaches for a set of computationally intensive algorithms. Because of its regularity and the clustering of resources into several processing elements, the processor array approach achieves in all experiments a far better throughput, up to 42% more compared with loop unrolling. For the cost and power metrics, it can be noted, that for smaller designs, the loop unrolling results are better, whereas, for larger designs, the processor array approach is better.

Finally, a complex (RDG with 384 nodes and 633 edges) real world application from the area of image processing has been presented. Moderate times for finding optimal schedules, which interleave operations from up to 85 iterations simultaneously, demonstrate the applicability, also for very complex examples.

Thanks to the modularity of the proposed scheduling techniques, the methodology can be easily extended to cope with further architectural properties. For instance, in [SMHT08], we extended the approach to processor arrays with subword parallelism.

6.2 Future Work Directions

The contributions in the area of modeling and languages, scheduling, and target code generation pave the way for a multitude of further works.

One important field of activity is the integration of accelerators into a system-on-a-chip and the communication with other accelerators. Concerning the system integration, different solutions are possible. For instance, a tight coupling via registers or a loosely-coupled integration using FIFOs or addressable buffer memories can be considered. The latter concept is needed when partitioning data into several chunks and transporting them to local memories of the accelerator. The main challenges in this scenario are the generation of adequate address generators and rate-matched scheduling of operations within the processor array and communication resources. From the modeling perspective, such communication primitives could be fairly easily integrated into the PAULA language.

In the case of communicating loops (accelerators), modeling can be done, for example, by the concept of a task graph, where each task denotes one loop program. Finding appropriate transformations for each loop program, in order to maximize the overall performance or to minimize intermediate memories, can lead to a difficult exploration problem.

Another exciting research topic might be the development of a symbolic design methodology. Such a methodology could be especially beneficial when considering dynamically reconfigurable or programmable architectures such as WPPAs. For instance, a symbolic mapping in dependence on the number of available processors would lead to a reduction of configuration memory and may also decrease the time in order react on resource changes or failures. The implementation of such a symbolic design methodology requires appropriate symbolic linear algebra methods (for example as provided in Maple [Hec93]) and methods for determining symbolic schedules. Here, for instance, *parametric integer programming* (PIP) [Fea88] might be applied. However, the performance of PIP, compared with commercial solvers, such as CPLEX, is restricted. Furthermore, because of different parameter combinations, a *state explosion* can happen and might lead to an intractable problem. Nevertheless, for one-dimensional arrays (one parameter), projections as allocation, or rectangular tile shapes, the proposed approach might lead to promising solutions.

Appendix A

Linear Programming

At this place, only a short introduction to linear programming is given. For an elaborate study, we refer, for instance, to the following textbooks, [HL95, Mur81, PS82, Sch86].

A.1 Linear Programming in a Nutshell

Linear programming (LP) is a mathematical technique for optimizing a linear function. This objective function is subject to a set of linear inequalities.

$$\begin{array}{lrl} \min & c^{\mathrm{T}}x & c, x \in \mathbb{R}^n \\ \text{subject to} & Ax \geq b & A \in \mathbb{R}^{m\times n},\ b \in \mathbb{R}^m \\ & x \geq 0 & \end{array}$$

Commonly, the above representation is called the *primary problem* or the *canonical form* of linear programming. Herein, x is a vector of variables that have to be determined. The matrix A and the vectors b and c are known coefficients. The term $c^{\mathrm{T}}x$ is called objective function and can be either minimized or maximized. The inequalities $Ax \geq b$ are the constraints, which represent the intersection of m half-spaces and thus specify a convex polyhedron over which the objective function is to be optimized. The matrix A is typically not square since in that case a linear program could be easily solved by inverting matrix A, if A has full rank. Usually, the matrix A has more columns than rows, that is, $Ax \geq b$ is most times highly under-determined, which offers a lot of optimization freedom.

As illustration of the optimization procedure, consider once again the problem in form of a convex polyhedron. It can be easily imagined, when traversing the hull

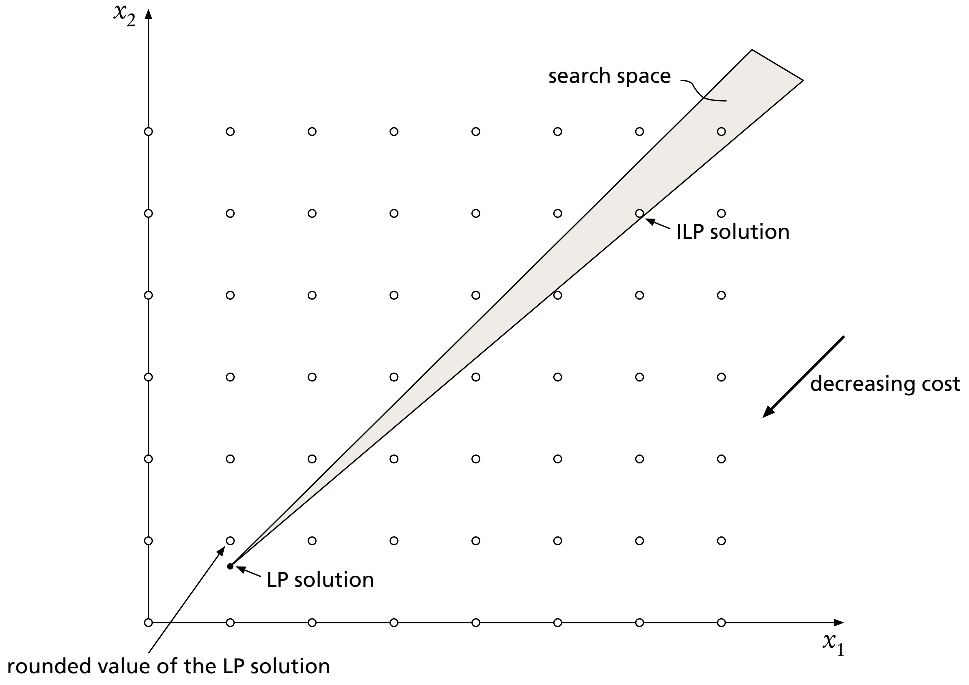


Figure A.1: A hypothetical example of an integer linear optimization problem (from [PS82]).

of the polyhedron, that the optimal solution must correspond to one of the vertices of the polyhedron.

One efficient technique for solving linear programs goes back to the 1940s and is called *simplex method*. Briefly described, this method selects a number of sub-matrices of A and solves them for x. In this way, the solution successively is improved until no more advancement can be made. Another group of algorithms for solving linear programs are the so-called *Karmarkar* and *barrier methods*. They are especially suited for solving high dimensional optimization problems. The methods share the characteristic that an optimal solution is found by traversing the interior of the feasible region whereas the simplex method investigates only the vertices.

Every primary problem (as defined above) can be converted into a so-called *dual problem*, which is defined as follows.

$$\begin{array}{lrl} \text{max} & b^{\mathrm{T}}y & y \in \mathbb{R}^m \\ \text{subject to} & A^{\mathrm{T}}y \leq c & \\ & y \geq 0 & \end{array}$$

In the dual problem y is used instead of x as the variable vector. A main result of this duality is that every feasible solution for a linear program gives a bound on the optimal value of the objective function of its dual problem. Likewise, if the primary problem is unbounded then the dual problem is infeasible and vice versa.

If all variables of the program are required to be integral, the problem is called an *integer linear program* (ILP) problem. On the first view, ILP problems do not seem harder to solve than conventional linear programs. One could think of it as to solve a corresponding linear program and to round the solution to the next integral point. This approach works to some extent, especially for large objective values. However, the difficulties that might arise are illustrated in Figure A.1.

Often, integer constraints are used to model combinatorial conditions or nonlinear constraints. In contrast to linear programming, which can be solved efficiently in the worst case, integer programming problems are in general NP-hard. The special case when the variables are required to be either 0 or 1 is called *0-1 integer programming* or *binary integer programming* (BIP). If a problem contains both real and integer variables it is called a *mixed integer linear programming* (MILP) or *mixed integer programming* (MIP) problem. These problems are also NP-hard in general. However, there exist some ILP and MIP subclasses that are efficiently solvable. In order to solve linear programs that contain integer variables, heuristics and other advanced algorithms such as *cutting-plane methods*, *branch and bound* or hybrid versions of them (for instance *branch and cut* algorithms) are often employed.

A.2 Formulation of Maximum-Constraints

At several places in the thesis, constraints in form of a maximum function are used. In this section different modeling possibilities of these constraints are described.

Let $a_1, a_2, \ldots, a_n$ be variables of a mixed integer program. Then, the maximum $b = \max(a_1, a_2, \ldots, a_n)$ can be determined in two ways.

1. By adding appropriate constraints and modification of the objective function to the MIP.

2. By adding constraints that contain binary variables and big-M constants.

A.2.1 Modification of the Objective Function

A term $b = \max(a_1, a_2, \ldots, a_n)$ can be calculated in a minimization problem by n inequalities $b \geq a_i$, $i = 1 \ldots n$ and the addition of b to the objective function, $f(x) + b$.

A.2.2 Usage of Binary Variables

First, let us only consider the maximum $b = \max(a_1, a_2)$ of two variables a_1 and a_2.

Theorem A.1 (Max2-Constraint)**.** *Let a_1, a_2, and b be variables of a MIP. Then, the term $b = \max(a_1, a_2)$ is equal to the following system of four inequalities.*

$$b \geq a_1 \tag{A.1}$$
$$b \geq a_2 \tag{A.2}$$
$$b \leq a_1 + \beta M_1 \tag{A.3}$$
$$b \leq a_2 + (1 - \beta) M_2 \tag{A.4}$$

Where β is a binary variable and M_1 and M_2 are big-M constants.

Proof. We use a proof by exhaustion. Thus, the following three cases have to be considered.

1. Case: $a_1 > a_2 \Rightarrow b = a_1$
   From the assumption, it follows that constraint (A.2) is redundant since it is dominated by constraint (A.1). The binary variable β cannot be 1 because otherwise constraint (A.4) will result in $b \leq a_2$, which is a contradiction to the assumption. Thus β is forced to be zero. Hereby, constraint (A.3) becomes $b \leq a_1$, which finally leads to $b = a_1$.

2. Case: $a_2 > a_1 \Rightarrow b = a_2$
   The rationale is analogous to the one of the first case.

3. Case: $a_1 = a_2 \Rightarrow b = a_1 = a_2$
   If a_1 equals a_2, also constraint (A.1) and constraint (A.2) are equal. The value of β is not important since in case of $\beta = 0$ as well as in case of $\beta = 1$, both constraints, (A.3) and (A.4), are satisfied and either the one or the other becomes $b \leq a_{\{1,2\}}$. From this, it follows directly that $b = a_1 = a_2$.

Within the proof, it is assumed that the constants M_1 and M_2 are large enough to satisfy the corresponding constraints. $\square$

Corollary A.1 (MaxN-Constraint)**.** *The constraints for deriving the maximum b out of n variables a_i, $b = \max(a_1, a_2, \ldots, a_n)$, can be directly obtained by the recursive application of Theorem A.1.*

$$b = \max(a_1, a_2, \ldots, a_n) = \max(a_1, \max(a_2, \ldots, a_n)) = \ldots$$

Appendix B

Further Details of the PAULA Language

B.1 Lexical Structure

The lexical structure of the PAULA language is very similar to other common programming languages like C or Perl. The following section briefly discusses the relevant issues.

B.1.1 White Space

PAULA, like many other languages, is a free-format language, which means that in most cases white space (that is, spaces, tabs and line breaks) does not play any role with respect to syntax or semantics of the language. The only place where a white space is important is when it appears between characters that would be interpreted differently when the white space was not present. For example, '`andb`' is completely different from '`and b`'.

B.1.2 Comments

There are three ways of including *comments* in the source code:

- C-style comments. Anything between '`/*`' and '`*/`' is ignored. As in C, such comments can span multiple lines.
- C++-style comments. This sort of comment starts with '`//`' and continues up to the end of the current line.

- Unix-style comments (as found for example in *sh*, *make*, and many other programs). This kind of comment starts with '`#`' and terminates at the end of the current line.

Comments are treated like white space, that means, they are ignored by the parser if they do not serve as token separators. For example, '`andb`' is a single token, while '`and/*foo*/b`' will be interpreted as the token '`and`' followed by the token '`b`'.

B.1.3 Tokens

A *token* is a sequence of characters and represents a terminal symbol in the syntactic grammar. The types of tokens in the PAULA language are the same as in most other programming languages and are described in the following.

B.1.3.1 Words

A *word* starts with a letter ('`a`'–'`z`', '`A`'–'`Z`', or the underscore character '`_`') and then continues with an arbitrary long sequence of letters and digits. Any other character marks the end of the word. Some words have a special meaning in the PAULA language and are called *keywords*. Those keywords are usually printed with a bold font throughout this document. A list of all keywords is given in Section B.3. Any word that is not a keyword is called *identifier*.

B.1.3.2 String Literals

A *string literal* is a sequence of characters enclosed in double quotes (for example, `"hello"`). To include a literal quote sign in the string, it must be escaped like this: `"This program is called \"PARO\"."`. To include a literal '`\`' in the string, use '`\\`'. The sequences '`\n`', '`\r`' and '`\t`' are also available and work just like in C, that is, they represent the line feed (LF), carriage return (CR), and horizontal tab control characters.

B.1.3.3 Integer and Float Literals

Integer literals can be written in normal decimal notation (for example, '`42`'), hexadecimal notation with either uppercase or lowercase letters (for example, '`0x7f`' or '`0X7F`'), or octal notation (for example, '`065`').

Float literals can have one decimal point (for example, '`3.1415`'), an optional signed exponential part with either an uppercase or lowercase delimiter (for example, '`5e6`', '`5E-30`') or both (for example, '`6.022e-23`').

B.1.3.4 Boolean Literals

A *boolean literal* is simply one of the words `true` and `false`.

B.1.3.5 Special Tokens

Special tokens are the punctuators and operators of the language. In most cases, these tokens are exactly one special character (that is, not a letter, digit, or underscore). Examples for such tokens are '+', '(', ';' and many more. However there are some tokens that consist of two special characters without any white space between them, for instance '>>', which is a bitwise shift operator just like in C. A list of those tokens is available in Section B.4. Note that the lexical scanner first tries to find such a two character token before returning separate tokens for each character. So if one needs to write two times the token '>', it is necessary to put a space between the two '>' characters ('> >'), otherwise it would be interpreted as a single token ('>>').

B.1.4 Inclusion of other Files

Like C, the PAULA language offers the possibility to include other source files, which could contain, for instance, common definitions. The syntax is the following:

`include``(`*`filename`*`)`

where *filename* is a string literal as defined in Section B.1.3.2).
Example:

```
include("alu.paro")
```

Note that there is no semicolon after the **`include`** statement. This statement is more like a macro, which can appear anywhere inside the program text. Especially, it does not need to be on a line of its own, like the `#include` statement offered by the C preprocessor.

B.1.5 Special Statements

Like the C preprocessor, the PAULA language also offers special statements to have the system print warning or error messages during parsing.
Example:

```
warning("program has not been tested yet")
```

If the keyword **`warning`** is replaced by **`error`**, the parser stops parsing at this point. These two statement may be used anywhere in the program, just like the **`include`** statement.

B.2 Data Types

B.2.1 Built-in Data Types

The goal of the PARO project is to generate hardware descriptions or to target WPPAs. For that reason, the data types of the PAULA language are tailored to the needs of hardware implementations and VLIW architectures, respectively. Currently, there are four basic data types available: *boolean*, *integer*, *fixed* and *float*, which are described in the following subsections.

B.2.1.1 notype

If a user does not want to specify a data type at all, the pseudo type **`notype`** should be used. This allows to explore all the possible program transformations without taking care of data types.

B.2.1.2 boolean

The *boolean* data type is simply a 1-bit signal that can have the values 'true' or 'false'. In the PAULA language, this type is referenced by the keyword **`boolean`**.

B.2.1.3 integer

The *integer* data type is a bit vector that can be an operand for all basic mathematical and bitwise operations. An integer can be either signed or unsigned, and can have any positive width. An integer type is specified as follows:

```
[unsigned|signed] integer<width>
```

If neither **`signed`** nor **`unsigned`** are given, the integer is signed by default. Integers are represented in two's complement, with values from 0 to $2^{width} - 1$ for unsigned integers, and from $-2^{width-1}$ to $2^{width-1} - 1$ for signed integers.

B.2.1.4 fixed

The *fixed* data type is the preferred type for fixed point arithmetical operations. Like the integer data type, it can be signed or unsigned. A fixed data type has a *width*, which specifies the total number of bits, and a *precision*, which is the number of bits after the binary point. This definition implies that the width must always be greater than or equal to the precision if the fixed type is unsigned. For signed fixed types, the width must be at least one bit larger than the precision to hold the additional sign bit. A fixed type is specified as follows:

```
[unsigned|signed] fixed<width,precision>
```

If neither **signed** nor **unsigned** are given, the fixed is signed by default.

B.2.1.5 float

The PAULA language supports *float* data types where the number of mantissa and exponent bits can be selected by the user. The float type is always signed. The syntax to define a float type is the following:

```
float<mantissa width,exponent width>
```

B.2.2 Type Conversions

Type conversions ("casts") are used to convert one data type into another, provided that the two types are compatible in a certain way. Those conversions are done automatically by the PARO system in several situations, or can be explicitly included in the source code by the programmer.

B.2.2.1 Automatic Conversions

Consider the following expression:

```
a[i,j] = b[i,j] + c[i,j]
```

where `a` is of type `signed integer<32>`, `b` is of type `signed integer<16>` and `c` is of data type `signed integer<8>`. It would of course be very inconvenient if a programmer has to explicitly convert `c` into an integer of width 16 in order to perform the addition, and also convert the 16 bit result to 32 bit so that it can be assigned to `a`. For that reason, the PARO system automatically includes type conversions in such situations. However, since those implicit conversions can lead to subtle problems, only the most basic conversions are done automatically. All remaining conversions need to be written down explicitly in the source code.

Widening Conversions The conversions that are done in the example above are so-called *widening conversions*, which convert one data type into a "wider" data type, where "wider" usually means more bits. Widening conversions are always done in a way that no information gets lost.

Table B.1 shows between which types widening conversions are possible, and what conditions must be fulfilled.

From Data Type	To Data Type	Condition	Remarks (see text)
boolean	*any type*	*false*	1
unsigned integer<w_1>	unsigned integer<w_2>	$w_2 \geq w_1$	
signed integer<w_1>	signed integer<w_2>	$w_2 \geq w_1$	
unsigned integer<w_1>	signed integer<w_2>	$w_2 \geq w_1 + 1$	2
signed integer<w_1>	unsigned integer<w_2>	*false*	3
unsigned integer<w_1>	unsigned fixed<w_2, p_2>	$w_2 - p_2 \geq w_1$	4
signed integer<w_1>	signed fixed<w_2, p_2>	$w_2 - p_2 \geq w_1$	4
unsigned integer<w_1>	signed fixed<w_2, p_2>	$w_2 - p_1 \geq w_1 + 1$	2, 4
signed integer<w_1>	unsigned fixed<w_2, p_2>	*false*	3
unsigned fixed<w_1, p_1>	unsigned integer<w_2>	$w_2 \geq w_1 \wedge p_1 = 0$	5
signed fixed<w_1, p_1>	signed integer<w_2>	$w_2 \geq w_1 \wedge p_1 = 0$	5
unsigned fixed<w_1, p_1>	signed integer<w_2>	*false*	
signed fixed<w_1, p_1>	unsigned integer<w_2>	*false*	
unsigned fixed<w_1, p_1>	unsigned fixed<w_2, p_2>	$w_2 \geq w_1 \wedge p_2 \geq p_1$	
signed fixed<w_1, p_1>	signed fixed<w_2, p_2>	$w_2 \geq w_1 \wedge p_2 \geq p_1$	
unsigned fixed<w_1, p_1>	signed fixed<w_2, p_2>	*false*	
signed fixed<w_1, p_1>	unsigned fixed<w_2, p_2>	*false*	
float	*any type*	*false*	6
any type	float	*false*	6

Table B.1: Widening Conversions.

Remarks:

1. Since the boolean type is regarded as a special type only for logical expressions, no automatic widening conversions are done, although they would be possible (e.g., into integer with the values 0 for false and 1 for true).
2. When converting unsigned into signed, one additional bit is required as sign bit.
3. Converting signed into unsigned is never done automatically (for obvious reasons).
4. The number of bits before the binary point of the destination fixed type must be greater than or equal to the width of the integer.
5. Converting fixed into integer is possible if the fixed type has no bits after the binary point (in which case it is obviously equivalent to an integer).
6. Since the float type has not yet been specified in detail, there are also no widening conversions.

B.2.3 Explicit Conversions

If a conversion is not done automatically by the PARO system, or done in the wrong way, the user is able to explicitly include type casts in the source code. The syntax is similar to the syntax used in C++:

```
cast<newtype>(expression)
```

`newtype` can be either a built-in data type or a type alias.
Example:

```
a[i,j] = cast<signed integer<32> >(b[i,j] + c[i,j]);
```

Note the space between the two '>' characters. Without this space, the sequence would be interpreted as the bitwise right shift operator '> >', which would be a syntax error. See also Section B.1.3.5.

B.3 PAULA Keywords

Here is a list of all keywords in the current version of the PAULA language.

```
allocation
and
bindingpossibility
cast
component
const
constant
do
for
function
if
ifrt
in
input
infinite
not
od
on
or
op
ops
out
output
par
parameter
program
resourcetype
step
simulatorplugin
to
typealias
variable
```

The following keywords are used as big operators:

```
SUM
PRODUCT
MIN
MAX
```

The following keywords are used as data types:

```
boolean
integer
fixed
float
notype
signed
unsigned
```

The following keywords are used as literals for data type *boolean*:

```
false
true
```

The following keywords are used for special purposes:

```
error
include
warning
```

The following keywords are reserved for future extensions to the language:

```
begin
case
define
end
null
switch
type
typedef
while
```

B.4 PAULA Multi-Character Operators

Here is a list of all PAULA operators that consist of more than one character. See also Section B.1.3.5.

```
==

!=

>=

<=
&&
||
<<
>>
```

B.5 EBNF of PAULA

```
adding_operator = '+' | '-';

big_operator = ( 'SUM' | 'PRODUCT' | 'MIN' | 'MAX' ) '['
indexspace ']' '(' expression ')';

bitwise_and_expression = equality { '&' equality };

bitwise_or_expression = bitwise_xor_expression { '|'
bitwise_xor_expression };

bitwise_xor_expression = bitwise_and_expression { '^'
bitwise_and_expression };

builtin_type = "boolean" | "float" | ( ["signed"|"unsigned"]
"integer" "<" integer ">" ) | ( ["signed"|"unsigned"] "fixed"
"<" integer "," integer ">" ) | "notype";

cast = "cast" "<" type ">" "(" expression ")";

constant = [ '+' | '-' ] ( integer | float );

convex_indexspace = indexspace_term { ( 'and' | '&&' )
indexspace_term };

digit = '0' | '1' | ...  | '9';

equality = shift_expression [ relational_operator
shift_expression ];

equation = [ equation_label ] indexed_variable '=' equation_rhs
[ 'if' '(' indexspace ')' ] ';';

equation_label = identifier ':';

equation_rhs = expression | 'ifrt' '(' expression ',' expression
',' expression ')';
```

```
expression = logical_or_expression;

factor = primary | ( unary_operator factor );

float = digit { digit } [ '.'  digit { digit } ] [ ( 'e' | 'E' )
[ '+' | '-' ] digit { digit } ];

for_statement = 'for' '(' identifier '=' integer 'to' integer [
'step' integer ] ')' programblock_body;

function_call = identifier '(' expression { ',' expression }
')';

function_declaration = 'function' identifier '(' type { ',' type
} ')' type ';';

identifier = word;

index_expression = index_term { ( '+' | '-' ) index_term };

index_term = [ '+' | '-' ] ( ( integer [ '/' integer ] ) | (
integer [ '/' integer ] '*' identifier ) | ( identifier ) );

indexed_variable = identifier '[' index_expression { ','
index_expression } ']';

indexspace = [ lattice_definition ':'  ]  convex_indexspace { (
'or' | '||' ) convex_indexspace };

indexspace_term = index_expression ( '>=' | '<=' | '>' | '<' |
'==' ) index_expression;

integer = digit { digit };

lattice_definition = identifier '=' index_expression { ','
identifier '=' index_expression };

letter = 'a' | 'b' | ...  | ' z' | 'A' | 'B' | ...  | 'Z' | '_';

logical_and_expression = bitwise_or_expression { ( '&&' | 'and'
) bitwise_or_expression };

logical_or_expression = logical_and_expression { ( '||' | 'or' )
logical_and_expression };
```

```
multiplication_operator = '*' | '/' | '%';

par_statement = 'par' '(' indexspace ')' programblock_body;

parameter_declaration = 'parameter' identifier [ '=' [ '+' | '-'
] integer ] ';';

primary = indexed_variable | constant | big_operator |
function_call | cast | ( '(' expression ')' );

program = 'program' identifier ';' { typealias_declaration |
variable_declaration | function_declaration |
parameter_declaration } { programblock } 'end' 'program' ';';

programblock = [ programblock_label ] programblock_statement;

programblock_body = 'do' { equation | programblock } 'od';

programblock_label = identifier ':';

programblock_statement = par_statement | for_statement;

relational_operator = ( '==' | '!=' | '>' | '<' | '<=' | '>=' );

shift_expression = simple_expression { shift_operator
simple_expression };

shift_operator = '«' | '»';

simple_expression = term { adding_operator term };

string = '"' ?  any character ?  '"';

term = factor { multiplication_operator factor };

type = builtin_type | typealias;

typealias = identifier;

typealias_declaration = 'typealias' identifier type ';';

unary_operator = '+' | '-' | '!'  | 'not' | ' ';

variable_declaration = 'variable' identifier integer [ 'in' |
'out' ] type ';';

word = letter { letter | digit };
```

German Part

Ablaufplanungsverfahren für Schleifenprogramme
zur Generierung durchsatzoptimierter
Hardware-Beschleuniger

Zusammenfassung

Gegenstand der vorliegenden Dissertationsschrift sind neue Techniken zur *Modellierung* und *Parallelisierung* von berechnungsintensiven Algorithmen in Form von verschachtelten Schleifenprogrammen, sowie Methoden zur *Ablaufplanung und Allokation* für Hardware-Beschleuniger mit hohem Durchsatz. Diese Ansätze bilden zusammen das zentrale Ergebnis dieser Arbeit: Eine einheitliche Methodik zur Abbildung von verschachtelten Schleifenprogrammen. Die Abbildungsmethodik kann sowohl für dedizierte Hardware-Beschleuniger, als auch für bestimmte Klassen von grob-granularen, rekonfigurierbaren Architekturen und Feldern aus eng gekoppelten, programmierbaren Prozessoren eingesetzt werden. Die wesentlichen Beiträge der Arbeit werden im Folgenden kurz zusammengefasst. Im Anschluss werden zukünftige Richtungen in diesem herausfordernden Forschungsgebiet aufgezeigt.

Beiträge

Modellierung. Um berechnungsintensive Programme auf Prozessorfelder systematisch abbilden zu können, ist ein profundes mathematisches Modell von großer Wichtigkeit. In diesem Zusammenhang wurde die neue Algorithmenklasse der *dynamisch stückweise linearen Algorithmen* (engl. *dynamic piecewise linear algorithms*) und eine entsprechende Graph-Repräsentation zur Modellierung von iterativem, mehrdimensionalem Datenfluss entwickelt. Die Klasse von dynamisch stückweise linearen Algorithmen erweitert bekannte Modelle, die auf Systemen von Rekurrenzgleichungen basieren, welche über polyedrische Iterationsräume definiert sind. Die Erweiterung besteht in der Möglichkeit erstmalig auch spezielle dynamische Datenabhängigkeiten zu modellieren, welche in vielen wichtigen Algorithmen auftauchen, jedoch in existierenden Ansätzen zur Schleifenparallelisierung nicht behandelt werden können. Durch diese Neuerung wird das Spektrum an Anwendungen mit mehrdimensionalem Datenfluss, welches parallelisiert und auf massiv-parallele Prozessorfelder abgebildet werden kann, erheblich erweitert. Zum Beispiel verfügen eine Menge von berechnungsintensiven Anwendungen zur Video- und Bildverarbeitung über verschachtelte Schleifenprogramme mit laufzeitabhängigen Bedingungen. Diese Anwendungen sind nun parallelisierbar und können infolgedessen sowohl auf dedizierte Hardware-Beschleuniger als auch auf eng gekoppelte, programmierbare Prozessorfelder abgebildet werden. Auf Basis der Klasse der dynamisch stückweise linearen Algorithmen wurde die Programmiersprache PAULA eingeführt. Dementsprechend fokussiert sich die Sprache ebenfalls auf die Modellierung von datenflussintensiven Anwendungen. Sie ermöglicht die Spezifikation von hoch-parallelen Algorithmen auf Instruktions-, Daten- und Schleifenebene. PAULA ermöglicht sehr kompakte und effiziente Verhaltensbeschreibungen und dient als Entwurfsgrundlage für die

Generierung von dedizierten Hardware-Beschleunigern oder sie kann als höhere Programmiersprache zur Algorithmenabbildung für eng gekoppelte, programmierbare Prozessorfelder verwendet werden. Die Sprache deckt ein weites Feld an Anwendungen ab. Beispiele umfassen die Bereiche der digitalen Bild-, Video- und anderer Formen der Signalverarbeitung, der linearen Algebra, der Kryptographie und viele weitere Domänen des wissenschaftlichen Rechnens, in denen effiziente Parallelisierungstechniken und Hardware-Beschleuniger unabdingbar sind. Die Intention zur Entwicklung einer neuen Sprache war nicht die Verbreitung einer weiteren parallelen Programmiersprache, sondern das Bedürfnis eine maßgeschneiderte Beschreibungsmöglichkeit zu haben, die exakt die in dieser Arbeit vorliegenden Konzepte widerspiegelt. Es wir angemerkt, dass die vorgestellten Spracheigenschaften auch in einer 'C-ähnlichen' oder anderen gängigen höheren Programmiersprache hätten ausgedrückt werden können. Um allerdings eine Programmiererin oder einen Programmierer nicht fehlzuleiten, wäre die Liste an Einschränkungen und Modifikationen ähnlich lang wie die Beschreibung der Sprache PAULA selbst.

Ablaufplanung und Allokation. Exakte und ganzheitliche Ablaufplanungsverfahren werden in der vorliegenden Schrift diskutiert. Diese Verfahren berücksichtigen sowohl eine lokale Allokation (Ressourcen innerhalb eines Prozessors), als auch eine oder mehrere Ebenen globaler Allokation (Algorithmenpartitionierung auf ein Feld von Prozessoren). Durch diese enge Verflechtung werden optimale Ablaufpläne, bezogen auf den Durchsatz beziehungsweise die Gesamtausführungszeit des betrachteten Algorithmus, ermittelt. Zur Ermittlung der Ablaufpläne wurde ein Verfahren basierend auf gemischt-ganzzahliger Programmierung entwickelt. Hierbei wurde erstmals eine Methode präsentiert, die sowohl den lokalen Ablaufplan innerhalb eines Prozessors, als auch den globalen Ablaufplan (zwischen den Prozessoren in einem Feld) für möglicherweise mehrere Partitionierungsebenen gleichzeitig optimiert. In diesem Zusammenhang wurde eine neue Nebenbedingung zur Sequentialisierung der Abarbeitung im Fall beschränkter Ressourcen entwickelt. Der vorgestellte Ansatz resultiert in bis zu 39 % schnelleren Ablaufplänen im Vergleich zu herkömmlichen Ansätzen. Der zweite bedeutende Beitrag im Bereich der Ablaufplanung ist die Formulierung von ressourcengewahren Nebenbedingungen, welche den gegenseitigen Ausschluss im Fall von iterations- oder laufzeitabhängigen Bedingungen berücksichtigen und hierdurch bei beschränkten Ressourcen ebenfalls den Ablauf (Durchsatz) erheblich verbessern können. Zusätzlich wurden weitere neue Nebenbedingungen zur Berücksichtigung von Beschränkungen, die bei der Abbildung auf ein fest vorgegebenes programmierbares Prozessorfeld gegeben sind, entwickelt. Diese Bedingungen betrachten Registerbeschränkungen innerhalb der Prozessoren und Kanalbeschränkungen innerhalb des Prozessorfelds. Gegenüber existierenden Arbeiten, die entweder nur Teilprobleme lösen oder hierarchisch arbeiten, ist der in dieser Arbeit vorliegende Ansatz einzigartig, da sich alle entwickelten Techniken je nach Bedarf zu einem einzigen gemischt-ganzzahligen Programm kombinieren lassen und ganzheitlich ein Ablaufplan optimiert werden kann.

Ergebnisse. Mehrere Experimente veranschaulichen die Stärke der vorgestellten Methodik. Als Fallstudien wurden mehr als 25 Algorithmen ausgewählt. Für diese wurde eine ressourcenbeschränkte Ablaufplanung durchgeführt und im Anschluss automatisch ein entsprechender Hardware-Beschleuniger synthetisiert. Die Komplexität der zugehörigen reduzierten Abhängigkeitsgraphen umfasst hierbei einige Dutzend bis zu mehr als 150 Knoten. In jedem Fall konnte ein optimaler Ablaufplan innerhalb weniger Sekunden ermittelt werden.

Die Mächtigkeit des vorgestellten Ansatzes basierend auf Prozessorfeldern (bzw. Schleifenpartitionierung) im Vergleich zum Abrollen von Schleifen (engl. *loop unrolling*) wurde ebenfalls demonstriert. Durch den Einsatz desselben Entwurfswerkzeugs PARO, welches beide Transformationen beherrscht, wurde eine faire, quantitative Evaluierung der beiden Ansätze für eine Menge von berechnungsintensiven Algorithmen möglich. Aufgrund der Regelmäßigkeit und der Ressourcengruppierung in mehrere Prozessorelemente, erreicht der Prozessorfeld-basierte Ansatz in allen Experimenten einen weitaus höheren Datendurchsatz, bis zu 42 % verglichen mit dem Abrollen von Schleifen.

Letztendlich wurde eine komplexe Anwendung (reduzierter Abhängigkeitsgraph mit 384 Knoten und 633 Kanten) aus dem Bereich der Bildverarbeitung betrachtet. Moderate Zeiten zur Findung von optimalen Ablaufplänen, welche die simultane Ausführung von Operationen von bis zu 85 unterschiedlichen Iterationen verkämmen, demonstrieren die Anwendbarkeit des Ansatzes ebenfalls für sehr komplexe Beispiele.

Dank der Modularität des gewählten Ansatzes konnte die Methodik ebenfalls leicht für Prozessorfelder mit Teilwortparallelismus erweitert werden [SMHT08].

Zukünftige Richtungen

Die Beiträge in den Bereichen Modellierung, Ablaufplanung und Allokation für unterschiedliche Zielarchitekturen ebnen den Weg für zahlreiche fortführende Arbeiten. Ein wichtiges Forschungsgebiet ist die Integration von Beschleunigern in ein Einchipsystem (engl. *System-on-a-Chip*) und die Interaktion mit weiteren Beschleunigern. Bezüglich der Systemintegration sind mehrere Ansätze möglich. Denkbar ist zum Beispiel eine enge Koppelung über Register oder eine lose gekoppelte Integration mittels FIFOs oder adressierbaren Pufferspeichern. Das letztere Konzept wird benötigt, wenn die verwendeten Daten eines Algorithmus in mehrere kleinere Datenblöcke partitioniert und zu lokalen Speichern des Hardware-Beschleunigers transportiert werden müssen. Die größten Herausforderungen bei diesem Ansatz sind die Generierung von geeigneten Adress-Generatoren und die Konstruktion von an die Kommunikationsressourcen angepassten Ablaufplänen. Bezogen auf die Modellierung, könnten solche Kommunikationsprimitive relativ einfach in die Architekturbeschreibung von PAULA integriert werden. Im Fall von kommunizierenden Schleifenprogrammen (Hardware-Beschleunigern), kann die Modellierung beispielsweise durch Task-Graphen erfolgen, wobei jeder Task ein Schleifenprogramm darstellt. Das Finden von geeigneten Transformationen für jedes Schleifenprogramm, um die Gesamtausführungszeit zu maximieren oder den benötigten Puffer zwischen den einzelnen Beschleunigern zu minimieren, kann zu einem schwierigen Explorationsproblem führen.

Ein weiteres hochinteressantes Forschungsgebiet ist die Entwicklung einer symbolischen Entwurfsmethodik. Ein derartiger Ansatz erscheint besonderes gewinnbringend für dynamisch rekonfigurierbare oder programmierbare Architekturen wie zum Beispiel WPPAs (Abkürzung des engl. Begriffs *weakly-programmable processor array*). Eine symbolische Abbildung in Abhängigkeit der verfügbaren Prozessoren würde zu einer Einsparung von Konfigurationsspeicher führen und könnte ebenfalls die Rekonfigurationszeit zur Reaktion auf Ressourcenänderungen oder Fehler erheblich reduzieren. Die Umsetzung einer symbolischen Entwurfsmethodik erfordert entsprechende symbolische Methoden der linearen Algebra (zum Beispiel wie in Maple [Hec93]) und Verfahren zur Bestimmung symbolischer Ablaufpläne. Hierfür könnte beispielsweise parametrische ganzzahlige Programmierung (PIP [Fea88]) verwendet werden. Jedoch ist die Leistungsfähigkeit von PIP im Vergleich zu kommerziellen Lösungsansätzen wie CPLEX beschränkt. Außerdem können unterschiedliche Parameterkombinationen zu einer Explosion des Lösungsraums und somit zur Unlösbarkeit des Problems führen. Nichtsdestotrotz könnte der Ansatz für eindimensionale Prozessorfelder (abhängig von einem Parameter), Projektionen als Allokation oder rechteckigen Partitionen zu Erfolg versprechenden Ergebnissen führen.

Bibliography

[AC94] Mohammed Aloqeely and C. Y. Roger Chen. Sequencer-Based Data Path Synthesis of Regular Iterative Algorithms. In *Proceedings of the 31st Annual Conference on Design Automation (DAC)*, pages 155–160, San Diego, CA, USA, 1994.

[ACD74] Thomas L. Adam, K. Mani Chandy, and J. R. Dickson. A Comparison of List Schedules for Parallel Processing Systems. *Communications of the ACM*, 17(12):685–690, 1974.

[AD88] Jurgen Annevelink and Patrick Dewilde. HIFI: A Functional Design System for VLSI Processing Arrays. In *Proceedings of the International Conference on Systolic Arrays*, pages 413–452, San Diego, CA, USA, May 1988.

[AHM97] David I. August, Wen-mei W. Hwu, and Scott A. Mahlke. A Framework for Balancing Control Flow and Predication. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 92–103, Research Triangle Park, NC, USA, December 1997.

[AJLA95] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[AKN95] Anant Agarwal, David A. Kranz, and Venkat Natarajan. Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(9):943–962, 1995.

[AKPW83] John R. Allen, Ken Kennedy, Carrie Porterfield, and Joe D. Warren. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 177–189, Austin, TX, USA, January 1983.

[Alt08] Altera Corporation. *Nios II C2H Compiler User Guide*. San Jose, CA, USA, November 2008.

[AR94] Rumen Andonov and Sanjay Vishnu Rajopadhye. An Optimal Algo-Tech-Cuit for the Knapsack Problem. Technical Report PI-791, IRISA, Campus de Beaulieu, Rennes, France, January 1994.

[AS06] Manoj Ampalam and Montek Singh. Counterflow Pipelining: Architectural Support for Preemption in Asynchronous Systems using Anti-Tokens. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 611–618, San Jose, CA, USA, November 2006.

[ATT92] Ulrich Arzt, Jürgen Teich, and Lothar Thiele. The concepts of COMPAR: A Compiler for Massive Parallel Architectures. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, pages 681–684, San Diego, CA, USA, May 1992.

[AZ00] Wolfgang Achtziger and Karl-Heinz Zimmermann. Finding Quadratic Schedules for Affine Recurrence Equations Via Nonsmooth Optimization. *The Journal of VLSI Signal Processing*, 25(3):235–260, July 2000.

[BAB+06] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H. Loh, Don McCaule, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. Die Stacking (3D) Microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–479, Orlando, FL, USA, December 2006.

[Bal88] M. Balakrishnan. RT-Level Synthesis based on Integrated Scheduling and Binding. Technical Report 8813, Institut für Informatik der Universität Kiel, Olshausenstr. 40-60, Kiel, Germany, December 1988.

[Bas03] Cédric Bastoul. Efficient Code Generation for Automatic Parallelization and Optimization. In *Proceedings of the Second International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 23–30, Ljubljana, Slovenia, October 2003.

[Bas04] Cédric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference* *on Parallel Architecture and Compilation Techniques (PACT)*, pages 7–16, Juan-les-Pins, France, September 2004.

[BCD94] Florin Balasa, Francky Catthoor, and Hugo De Man. Dataflow-Driven Memory Allocation for Multi-Dimensional Signal Processing Systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 31–34, San Jose, CA, USA, 1994.

[BDD90] Jichun Bu, Ed F. Deprettere, and Patrick Dewilde. A Design Methodology for Fixed-Size Systolic Arrays. In *Proceedings of the International Conference on Application Specific Array Processors (ASAP)*, pages 591–602, Princeton, NJ, USA, September 1990.

[BDD$^+$99] Kiran Bondalapati, Pedro C. Diniz, Phillip Duncan, John Granacki, Mary W. Hall, Rajeev Jain, and Heidi Ziegler. DEFACTO: A Design Environment for Adaptive Computing Technology. In *Proceedings of the 11th Workshops Held in Conjunction with the 13th International Parallel Processing Symposium (IPPS) and 10th Symposium on Parallel and Distributed Processing (SPDP)*, volume 1586 of *Lecture Notes in Computer Science (LNCS)*, pages 570–578, San Juan, Puerto Rico, April 1999.

[BDEN08] Michel Berkelaar, Jeroen Dirks, Kjell Eikland, and Peter Notebaert. `http://lpsolve.sourceforge.net/5.5`, 2008.

[Bed04] Marcus Bednara. *Design Automation for Massively Parallel Processor Arrays: Transforming Regular Algorithms to Reconfigurable Hardware.* PhD thesis, University of Erlangen-Nuremberg, Department of Computer Science 12, Erlangen, Germany, 2004.

[BEM$^+$03] Volker Baumgarte, Gerd Ehlers, Frank May, Armin Nückel, Martin Vorbach, and Markus Weinhardt. PACT XPP – A Self-Reconfigurable Data Processing Architecture. *The Journal of Supercomputing*, 26(2):167–184, 2003.

[Bey02] Oliver Beyer. Implementierung eines Verfahrens zur Parallelisierung geschachtelter C-Schleifenprogramme. Diploma thesis, University of Paderborn, Department of Electrical Engineering and Information Technology, Computer Engineering Laboratory, March 2002.

[BF98] Pierre Boulet and Paul Feautrier. Scanning Polyhedra without Do-loops. In *Proceedings of the 13th International Conference on Parallel*

Architecture and Compilation Techniques (PACT), pages 4–11, Paris, France, October 1998.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[BGN06] Betul Buyukkurt, Zhi Guo, and Walid A. Najjar. Impact of Loop Unrolling on Throughput, Area and Clock Frequency in ROCCC: C to VHDL Compiler for FPGAs. In *Proceedings of the 2nd International Workshop on Applied Reconfigurable Computing (ARC)*, volume 3985 of *Lecture Notes in Computer Science (LNCS)*, pages 401–412, Delft, The Netherlands, March 2006.

[BHT01] Marcus Bednara, Frank Hannig, and Jürgen Teich. Boundary Control: A new Distributed Control Architecture for Space-Time Transformed (VLSI) Processor Arrays. In *Proceedings 35th IEEE Asilomar Conference on Signals, Systems, and Computers*, volume 2, pages 468–474, Pacific Grove, CA, USA, November 2001.

[BHT02] Marcus Bednara, Frank Hannig, and Jürgen Teich. *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation – SAMOS*, volume 2268 of *Lecture Notes in Computer Science (LNCS)*, chapter Generation of Distributed Loop Control, pages 154–170. Springer, January 2002.

[BKV+08] Florin Balasa, Per Gunnar Kjeldsberg, Arnout Vandecappelle, Martin Palkovic, Qubo Hu, Hongwei Zhu, and Francky Catthoor. Storage Estimation and Design Space Exploration Methodologies for the Memory Management of Signal Processing Applications. *Journal of Signal Processing Systems*, 53(1-2):51–71, 2008.

[BL93] Joseph T. Buck and Edward A. Lee. Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 1, pages 429–432, April 1993.

[BL01] Sergey V. Bakhanovich and Nickolai A. Likhoded. A Method for Parallelizing Algorithms by Vector Scheduling Functions. *Programming and Computing Software*, 27(4):194–199, 2001.

[Blu09] Bluespec, Inc. `http://www.bluespec.com`, 2009.

[BMP07] Brian Bailey, Grant Martin, and Andrew Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Systems on Silicon. Morgan Kaufmann, March 2007.

[BN08] Betul Buyukkurt and Walid A. Najjar. Compiler Generated Systolic Arrays for Wavefront Algorithm Acceleration on FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 655–658, Heidelberg, Germany, September 2008.

[BRS07] Uday Bondhugula, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. Automatic Mapping of Nested Loops to FPGAs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 101–111, San Jose, CA, USA, 2007.

[BSC$^+$00] Prithviraj Banerjee, Nagaraj Shenoy, Alok Nidhi Choudhary, Scott Alan Hauck, Michael Bachmann, Malay Haldar, Pramod G. Joisha, Alex Keith Jones, Abhay Kanhare, Anshuman Nayak, Suresh Periyacheri, Michael Walkden, and David C. Zaretsky. A MATLAB Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 39–48, Washington, DC, USA, 2000.

[BT01] Marcus Bednara and Jürgen Teich. Synthesis of FPGA Implementations from Loop Algorithms. In *First International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 1–7, Las Vegas, NV, USA, June 2001.

[BT03] Marcus Bednara and Jürgen Teich. Automatic Synthesis of FPGA Processor Arrays from Loop Algorithms. *The Journal of Supercomputing*, 26(2):149–165, 2003.

[Bur94] Wayne Burleson. Using Regular Array Methods for DSP Module Synthesis. In *Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS)*, volume I: Architecture, pages 58–67, Wailea, HI, USA, January 1994.

[But02] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 2002.

[BWSF06] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray Tracing on the Cell Processor. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 15–23, Salt Lake City, UT, USA, September 2006.

[Cam91] Raul Camposano. Path-Based Scheduling for Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(1):85–93, January 1991.

[CBF95] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy Array Dataflow Analysis. In *Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 92–101, 1995.

[CD04] João M. P. Cardoso and Pedro C. Diniz. Modeling Loop Unrolling: Approaches and Open Issues. In Andy D. Pimentel and Stamatis Vassiliadis, editors, *Computer Systems: Architectures, Modeling, and Simulation, 4th International Samos Workshop (SAMOS), Proceedings*, volume 3133 of *Lecture Notes in Computer Science (LNCS)*, pages 224–233, Island of Samos, Greece, July 2004.

[CDF06] Hadda Cherroun, Alain Darte, and Paul Feautrier. Scheduling under Resource Constraints using Dis-Equations. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1067–1072, Munich, Germany, March 2006.

[CDK$^+$02] Francky Catthoor, Koen Danckaert, Chidamber Kulkarni, Erik Brockmeyer, Per Gunnar Kjeldsberg, Tanja van Achteren, and Thierry Omnes. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers, Hingham, MA, USA, 2002.

[CEL09] CELOXICA, Handel-C. `http://www.celoxica.com`, 2009.

[CF07] Hadda Cherroun and Paul Feautrier. An Exact Resource Constrained-Scheduler using Graph Coloring Technique. In *Proceedings of the IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, pages 554–561, Amman, Jordan, May 2007.

[CGR93] Viraphol Chaiyakul, Daniel D. Gajski, and Loganath Ramachandran. High-Level Transformations for Minimizing Syntactic Variances. In *Proceedings of the 30th ACM/IEEE Design Automation Conference (DAC)*, pages 413–418, Dallas, TX, USA, 1993.

[CHM08] Nathan Clark, Amir Hormati, and Scott A. Mahlke. VEAL: Virtualized Execution Accelerator for Loops. *ACM SIGARCH Computer Architecture News*, 36(3):389–400, 2008.

[CHR$^+$03] Charles Consel, Hedi Hamdi, Laurent Réveillère, Lenin Singaravelu, Haiyan Yu, and Calton Pu. Spidle: A DSL Approach to Specifying Streaming Applications. In *In Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE)*, volume 2830 of *Lecture Notes in Computer Science (LNCS)*, pages 1–17, Erfurt, Germany, 2003.

[CJP07] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[CK93] W. H. Chou and Sun-Yuan Kung. Scheduling Partitioned Algorithms with Limited Communication Supports. In Luigi Dadda and Benjamin Wah, editors, *Proceedings of the International Conference on Application Specific Array Processors (ASAP)*, pages 53–64, Venice, Italy, October 1993.

[CLG02] Josep M. Codina, Josep Llosa, and Antonio González. A Comparative Study of Modulo Scheduling Techniques. In *Proceedings of the 16th International Conference on Supercomputing (ICS)*, pages 97–106, New York, NY, USA, June 2002.

[CLL$^+$05] Michael K. Chen, Xiao Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. Shangri-La: Achieving High Performance from Compiled Network Applications while Enabling Ease of Programming. *ACM SIGPLAN Notices*, 40(6):224–236, 2005.

[CLR97] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT electrical engineering and computer science series. MIT Press, Cambridge, Massachusetts, USA, 18. edition, 1997.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, Massachusetts, 1988.

[CM95] Stephanie Coleman and Kathryn S. McKinley. Tile Size Selection using Cache Organization and Data Layout. *ACM SIGPLAN Notices*, 30(6):279–290, 1995.

[CNO$^+$88] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. *IEEE Transactions on Computers*, 37(8):967–979, 1988.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. LUSTRE: A Declarative Language for Real-Time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 178–188, Munich, Germany, January 1987.

[CT90] Richard J. Cloutier and Donald E. Thomas. The Combination of Scheduling, Allocation, and Mapping in a Single Algorithm. In *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC)*, pages 71–76, June 1990.

[Dar91] Alain Darte. Regular Partitioning for Synthesizing Fixed-Size Systolic Arrays. *Integration, the VLSI Journal*, 12(3):293–304, 1991.

[Dar99] Alain Darte. *Algorithms for Parallel Processing*, volume 105 of *The IMA Volumes in Mathematics and its Applications*, chapter Mathematical Tools for Loop Transformations: From Systems of Uniform Recurrence Equations to the Polytope Model, pages 147–183. Springer, 1999.

[DD90] Alain Darte and Jean-Marc Delosme. Partitioning for Array Processors. Technical Report 90-23, LIP, ENS-Lyon, 46 allee d'Italie, 69364 Lyon Cedex 07, France, August 1990.

[DDM06] Abhishek Das, William J. Dally, and Peter Mattson. Compiling for Stream Processing. In *In Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 33–42, Seattle, WA, USA, 2006.

[Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[DH01] James B. Dabney and Thomas L. Harman. *Mastering SIMULINK 4*. Prentice Hall, 1st edition, 2001.

[DHK$^+$07] Hritam Dutta, Frank Hannig, Alexey Kupriyanov, Dmitrij Kissler, Jürgen Teich, Rainer Schaffer, Sebastian Siegel, Renate Merker, and Bernard Pottier. Massively Parallel Processor Architectures: A Co-design Approach. In *Proceedings of the 3rd International Workshop* *on Reconfigurable Communication Centric System-on-Chips (ReCoSoC)*, pages 61–68, Montpellier, France, June 2007.

[DHP^{+}05] Pedro C. Diniz, Mary Hall, Joonseok Park, Byoungro So, and Heidi Ziegler. Automatic Mapping of C to FPGAs with the DEFACTO Compilation and Synthesis System. *Microprocessors and Microsystems*, 29(2-3):51–62, 2005.

[DHRT07] Hritam Dutta, Frank Hannig, Holger Ruckdeschel, and Jürgen Teich. Efficient Control Generation for Mapping Nested Loop Programs onto Processor Arrays. *Journal of Systems Architecture*, 53(5–6):300–309, May 2007.

[DHT05] Hritam Dutta, Frank Hannig, and Jürgen Teich. Control Path Generation for Mapping Partitioned Dataflow-dominant Algorithms onto Array Architectures. Technical Report 03–2005, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, 91058 Erlangen, Germany, November 2005.

[DHT06a] Hritam Dutta, Frank Hannig, and Jürgen Teich. A Formal Methodology for Hierarchical Partitioning of Piecewise Linear Algorithms. Technical Report 04–2006, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, 91058 Erlangen, Germany, April 2006.

[DHT06b] Hritam Dutta, Frank Hannig, and Jürgen Teich. Controller Synthesis for Mapping Partitioned Programs on Array Architectures. In Werner Grass, Bernhard Sick, and Klaus Waldschmidt, editors, *Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS)*, volume 3894 of *Lecture Notes in Computer Science (LNCS)*, pages 176–191, Frankfurt am Main, Germany, March 2006.

[DHT06c] Hritam Dutta, Frank Hannig, and Jürgen Teich. Hierarchical Partitioning for Piecewise Linear Algorithms. In *Proceedings of the 5th International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, pages 153–160, Bialystok, Poland, September 2006.

[DHT06d] Hritam Dutta, Frank Hannig, and Jürgen Teich. Mapping of Nested Loop Programs onto Massively Parallel Processor Arrays with Memory and I/O Constraints. In Friedhelm Meyer auf der Heide and Burkhard Monien, editors, *Proceedings of the 6th International Heinz*

Nixdorf Symposium, New Trends in Parallel & Distributed Computing, volume 181 of *HNI-Verlagsschriftenreihe*, pages 97–119, Paderborn, Germany, January 2006.

[DHT$^+$06e] Hritam Dutta, Frank Hannig, Jürgen Teich, Benno Heigl, and Heinz Hornegger. A Design Methodology for Hardware Acceleration of Adaptive Filter Algorithms in Image Processing. In *Proceedings of the 17th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 331–337, Steamboat Springs, CO, USA, September 2006.

[DHT08] Hritam Dutta, Frank Hannig, and Jürgen Teich. PARO: A Design Tool for Automatic Generation of Hardware Accelerators. In *Proceedings of ACACES 2008 Poster Abstracts: Advanced Computer Architecture and Compilation for Embedded Systems*, pages 317–320, L'Aquila, Italy, July 2008.

[DHT09] Hritam Dutta, Frank Hannig, and Jürgen Teich. Performance Matching of Hardware Acceleration Engines for Heterogeneous MPSoC using Modular Performance Analysis. In *Proceedings of the 22nd International Conference on Architecture of Computing Systems (ARCS)*, volume 5455 of *Lecture Notes in Computer Science (LNCS)*, pages 233–245, Delft, The Netherlands, January 2009.

[DI85] Jean-Marc Delosme and Ilse C. F. Ipsen. An Illustration of a Methodology for the Construction of Efficient Systolic Architectures in VLSI. In *Proceedings of the Second International Symposium on VLSI Technology, Systems and Applications*, pages 268–273, Taipei, Taiwan, May 1985.

[DI86] Jean-Marc Delosme and Ilse C. F. Ipsen. Systolic Array Synthesis: Computability and Time Cones. In *Proceedings of the International Workshop on Parallel Algorithms & Architectures*, pages 295–312, Luminy, France, 1986.

[Dja04] Clémentin Tayou Djamegni. Mapping Rectangular Mesh Algorithms Onto Asymptotically Space-Optimal Arrays. *Journal of Parallel and Distributed Computing*, 64(3):345–359, 2004.

[Dja06] Clémentin Tayou Djamegni. Complexity of Matrix Product on Modular Linear Systolic Arrays for Algorithms with Affine Schedules. *Journal of Parallel and Distributed Computing*, 66(3):323–333, 2006.

[DKH$^+$09] Hritam Dutta, Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, Jürgen Teich, and Bernard Pottier. A Holistic Approach for Tightly Coupled Reconfigurable Parallel Processors. *Microprocessors and Microsystems*, 33(1):53–62, February 2009.

[DKR92] Alain Darte, Leonid Khachiyan, and Yves Robert. Linear Scheduling is Close to Optimality. In *Proceedings of the International Conference on Application Specific Array Processors (ASAP)*, pages 37–46, Berkeley, CA, USA, August 1992.

[DKT87] Paul David Domich, Ravindran Kannan, and Leslie Earl Trotter, Jr. Hermite Normal Form Computation using Modulo Determinant Arithmetic. *Mathematics of Operations Research*, 12(1):50–59, February 1987.

[DLSM81] Scott Davidson, David Landskov, Bruce D. Shriver, and Patrick W. Mallett. Some Experiments in Local Microcode Compaction for Horizontal Machines. *IEEE Transactions on Computers*, 30(7):460–477, 1981.

[DN89] Srinivas Devadas and A. Richard Newton. Algorithms for Hardware Allocation in Data Path Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(7):768–781, July 1989.

[Don88] Vincent van Dongen. PRESAGE, A Tool for the Design of Low-Cost Systolic Circuits. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 3, pages 2765–2768, June 1988.

[Don92] Vincent van Dongen. From Systolic to Periodic Array Design. In Patrice Quinton and Yves Robert, editors, *Proceedings of the International Workshop on Algorithms and Parallel VLSI Architectures II*, pages 151–162, Gers, France, June 1992.

[DQR$^+$09] Clémentin Tayou Djamegni, Patrice Quinton, Sanjay Vishnu Rajopadhye, Tanguy Risset, and Maurice Tchuenté. A Reindexing Based Approach towards Mapping of DAG with Affine Schedules onto Parallel Embedded Systems. *Journal of Parallel and Distributed Computing*, 69(1):1–11, 2009.

[DR92] Alain Darte and Yves Robert. Scheduling Uniform Loop Nests. Technical Report RR92–10, ENSL Lyon, France, 1992.

[DR94a] Alain Darte and Yves Robert. Constructive Methods for Scheduling Uniform Loop Nests. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):814–822, August 1994.

[DR94b] Alain Darte and Yves Robert. Mapping Uniform Loop Nests onto Distributed Memory Architectures. *Parallel Computing*, 20(5):679–710, May 1994.

[DR95] Alain Darte and Yves Robert. Affine-by-statement Scheduling of Uniform and Affine Loop Nests over Parametric Domains. *Journal of Parallel and Distributed Computing*, 29(1):43–59, 1995.

[DRK01] Steven Derrien, Sanjay Vishnu Rajopadhye, and Susmita Sur Kolay. Combined Instruction and Loop Parallelism in Array Synthesis for FPGAs. In *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS)*, pages 165–170, Montréal, Quebec, Canada, September 2001.

[DRR95] Michèle Dion, Tanguy Risset, and Yves Robert. Resource-Constrained Scheduling of Partitioned Algorithms on Processor Arrays. In *Proceedings of the 3rd Euromicro Workshop on Parallel and Distributed Processing (PDP)*, pages 571–580, San Remo, Italy, January 1995.

[DRS00] Steven Derrien, Sanjay Vishnu Rajopadhye, and Susmita Sur Sur-Kolay. Optimal Partitioning for FPGA Based Regular Array Implementations. In *Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, pages 155–159, Trois-Rivières, Quebec, Canada, August 2000.

[DRV00] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization*. Birkhäuser, 2000.

[DSRV02] Alain Darte, Robert Schreiber, B. Ramakrishna Rau, and Frédéric Vivien. Constructing and Exploiting Linear Schedules with Prescribed Parallelism. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(1):159–172, 2002.

[DSV05] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-Based Memory Allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, October 2005.

[Dut04] Hritam Dutta. Mapping of Hierarchically Partitioned Regular Algorithms onto Processor Arrays. Master's thesis, Department of Computer Science 12, University of Erlangen-Nuremberg, October 2004.

[DV95] Alain Darte and Frédéric Vivien. Revisiting the Decomposition of Karp, Miller and Winograd. In *Proceedings of the IEEE International Conference on Application Specific Array Processors (ASAP)*, pages 13–25, Strasbourg, France, July 1995.

[DZD^{+}08] Yazhuo Dong, Jie Zhou, Yong Dou, Lin Deng, and Jinjing Zhao. Impact of Loop Unrolling on Area, Throughput and Clock Frequency for Window Operations Based on a Data Schedule Method. In *Proceedings of the Congress on Image and Signal Processing (CISP), Vol. 1*, pages 641–645, Sanya, Hainan, China, May 2008.

[DZHT09] Hritam Dutta, Jiali Zhai, Frank Hannig, and Jürgen Teich. Impact of Loop Tiling on the Controller Logic of Hardware Acceleration Engines. In *Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 161–168, Boston, MA, USA, July 2009.

[EC89] Bradley R. Engstrom and Peter R. Cappello. The SDEF Programming System. *Journal of Parallel and Distributed Computing*, 7(2):201–231, 1989.

[Eck01] Uwe Eckhardt. *Algorithmus-Architektur-Codesign für den Entwurf digitaler Systeme mit eingebettetem Prozessorarray und Speicherhierarchie*. PhD thesis, Technische Universität Dresden, Dresden, Germany, June 2001.

[EDA95] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Optimum Modulo Schedules for Minimum Register Requirements. In *Proceedings of the 9th International Conference on Supercomputing (ICS)*, pages 31–40, Barcelona, Spain, July 1995.

[EJP92] Marie-Christine Eglin-Leclerc, Jacques Julliand, and Guy-René Perrin. How to Compile Systems of Recurrence Equations into Networks of Communicating Processes. In *Proceedings of the Second Joint International Conference on Vector and Parallel Processing (CONPAR/-VAPP)*, volume 634 of *Lecture Notes in Computer Science (LNCS)*, pages 795–796, Lyon, France, September 1992.

[EKP$^+$98] Petru Eles, Krzysztof Kuchcinski, Zebo Peng, Alexa Doboli, and Paul Pop. Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 132–139, Paris, France, February 1998.

[EM97a] Uwe Eckhardt and Renate Merker. Co-Partitioning – A Method for Hardware/Software Codesign for Scalable Systolic Arrays. In Reiner W. Hartenstein and Viktor K. Prasanna, editors, *Proceedings of the 4th Reconfigurable Architectures Workshop*, pages 131–138, Geneva, Switzerland, April 1997.

[EM97b] Uwe Eckhardt and Renate Merker. Scheduling in Co-Partitioned Array Architectures. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 219–228, Zurich, Switzerland, July 1997.

[EM99] Uwe Eckhardt and Renate Merker. Hierarchical Algorithm Partitioning at System Level for an Improved Utilization of Memory Structures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(1):14–24, 1999.

[ES06] Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[ESO$^+$08] Sven Eisenhardt, Thomas Schweizer, Julio A. de Oliveira Filho, Tobias Oppold, Wolfgang Rosenstiel, Alexander Thomas, Jürgen Becker, Frank Hannig, Dmitrij Kissler, Hritam Dutta, Jürgen Teich, Heiko Hinkelmann, Peter Zipf, and Manfred Glesner. SPP1148 Booth: Coarse-Grained Reconfiguration. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, page 349, Heidelberg, Germany, September 2008.

[ESV$^+$99] Jos T. J. van Eijndhoven, Frans W. Sijstermans, Kees A. Vissers, Evert-Jan D. Pol, Marcel J. A. Tromp, Pieter Struik, Rudi H. J. Bloks, Pieter van der Wolf, Andy D. Pimentel, and Harald P. E. Vranken. Trimedia CPU64 Architecture. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 586–592, Austin, TX, USA, October 1999.

[FBF$^+$00] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: A Technology Platform for Customizable

VLIW Embedded Processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 203–213, Vancouver, British Columbia, Canada, June 2000.

[Fea88] Paul Feautrier. Parametric Integer Programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[Fea92a] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem. I. One-Dimensional Time. *International Journal of Parallel Programming*, 21(5):313–347, 1992.

[Fea92b] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[Fea94] Paul Feautrier. Fine-Grain Scheduling under Resource Constraints. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 892 of *Lecture Notes in Computer Science (LNCS)*, pages 1–15, Ithaca, NY, USA, August 1994.

[Fea96] Paul Feautrier. Automatic Parallelization in the Polytope Model. Technical Report 8, Laboratoire PRiSM, Université des Versailles St-Quentin en Yvelines, 45, avenue des États-Unis, 78035 Versailles Cedex, France, June 1996.

[Fea06] Paul Feautrier. Scalable and Structured Scheduling. *International Journal of Parallel Programming*, 34(5):459–487, 2006.

[Fer07] François de Ferrière. Improvements to the Psi-SSA Representation. In *Proceedingsof the 10th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 111–121, Nice, France, 2007.

[FG00] Jose Fridman and Zvi Greenfield. The TigerSHARC DSP Architecture. *IEEE Micro*, 20(1):66–76, 2000.

[FGQ86] Patrice Frison, Pierrick Gachet, and Patrice Quinton. Designing Systolic Arrays with DIASTOL. In Sun-Yuan Kung, Robert E. Owen, and J. Greg Nash, editors, *Proceedings of the Workshop on VLSI Signal Processing II*, pages 93–105, Los Angeles, CA, USA, November 1986.

[Fim00] Dirk Fimmel. Generation of Scheduling Functions Supporting LSGP-Partitioning. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 349–358, Boston, MA, USA, July 2000.

[Fim02] Dirk Fimmel. *Optimaler Entwurf paralleler Rechenfelder unter Verwendung ganzzahliger linearer Optimierung*. PhD thesis, Technische Universität Dresden, Dresden, Germany, April 2002.

[Fis81] Joseph A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[Fis83] Joseph A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture (ISCA)*, pages 140–150, Stockholm, Sweden, June 1983.

[FKDM09] Kevin Fan, Manjunath Kudlur, Ganesh Dasika, and Scott A. Mahlke. Bridging the Computation Gap Between Programmable Processors and Hardwired Accelerators. In *Proceedings of the 15th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 313–322, Raleigh, NC, USA, February 2009.

[FKPM05] Kevin Fan, Manjunath Kudlur, Hyunchul Park, and Scott A. Mahlke. Cost Sensitive Modulo Scheduling in a Loop Accelerator Synthesis System. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 219–232, Barcelona, Spain, November 2005.

[FLV95] Agustín Fernández, José M. Llabería, and Miguel Valero-García. Loop Transformation using Nonunimodular Matrices. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):832–840, 1995.

[FM97] Dirk Fimmel and Renate Merker. Determination of the Processor Functionality in the Design of Processor Arrays. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 199–208, Zurich, Switzerland, July 1997.

[FM98] Dirk Fimmel and Renate Merker. Determination of Processor Allocation in the Design of Processor Arrays. *Microprocessors and Microsystems*, 22(3–4):149–155, 1998.

[For09] Forte Design Systems. `http://www.forteds.com`, 2009.

[FP84] Jose A. B. Fortes and Francesco Parisi-Presicce. Optimal Linear Schedules for the Parallel Execution of Algorithms. In *Proceedings of the International Conference on Parallel Processing*, pages 322–328, Bellaire, MI, USA, August 1984.

[GAG96] Ramaswamy Govindarajan, Erik Richter Altman, and Guang R. Gao. A Framework for Resource-Constrained Rate-Optimal Software Pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1133–1149, 1996.

[GAK03] Georgios I. Goumas, Maria G. Athanasaki, and Nectarios Koziris. An Efficient Code Generation Technique for Tiled Iteration Spaces. *IEEE Transactions on Parallel and Distributed Systems*, 14(10):1021–1034, October 2003.

[GBN04] Zhi Guo, Betul Buyukkurt, and Walid A. Najjar. Input Data Reuse in Compiling Window Operations onto Reconfigurable Hardware. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 249–256, Washington, DC, USA, June 2004.

[GCC$^+$05] Charles Gray, Matthew Chapman, Matthew Chubb, David Mosberger-Tang, and Gernot Heiser. Itanium — A System Implementor's Tale. In *Proceedings of the USENIX Annual Technical Conference (ATEC)*, pages 265–278, Anaheim, CA, USA, April 2005.

[GCC09] GCC, the GNU Compiler Collection. `http://gcc.gnu.org`, 2009.

[GDF$^+$01] Jean-Luc Gaudiot, Thomas DeBoni, John Feo, Wim Böhm, Walid A. Najjar, and Patrick Miller. The Sisal Project: Real World Functional Programming. In Santosh Pande and Dharma P. Agrawal, editors, *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems*, volume 1808 of *Lecture Notes in Computer Science (LNCS)*, pages 45–72, 2001.

[GDGN03] Sumit Gupta, Nikil D. Dutt, Rajesh Gupta, and Alexandru Nicolau. SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations. In *Proceedings of the 16th International Conference on VLSI Design (VLSID)*, New Delhi, India, January 2003.

[GDWL92] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[GE93] Catherine H. Gebotys and Mohamed I. Elmasry. Global Optimization Approach for Architectural Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(9):1266–1278, September 1993.

[GGDN04] Sumit Gupta, Rajesh K. Gupta, Nikil D. Dutt, and Alexandru Nicolau. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic Publishers, 2004.

[GGL99] Max Geigl, Martin Griebl, and Christian Lengauer. A Scheme for Detecting the Termination of a Parallel Loop Nest. *Parallel Computing*, 25(12):1489–1510, November 1999.

[GJM00] Bruno Gaujal, Alain Jean-Marie, and Jean Mairesse. Computations of Uniform Recurrence Equations using Minimal Memory Size. *SIAM Journal on Computing*, 30(5):1701–1738, 2000.

[GK07] Hagen Gädke and Andreas Koch. Comrade – A Compiler for Adaptive Computing Systems using a Novel Fast Speculation Technique. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 503–504, Amsterdam, The Netherlands, August 2007.

[GK08] Hagen Gädke and Andreas Koch. Accelerating Speculative Execution in High-Level Synthesis with Cancel Tokens. In *Proceedings of the Fourth International Workshop on Applied Reconfigurable Computing (ARC)*, volume 4943 of *Lecture Notes in Computer Science (LNCS)*, pages 185–195, London, United Kingdom, March 2008.

[GL96] Martin Griebl and Christian Lengauer. The Loop Parallelizer LooPo. In Michael Gerndt, editor, *Proceedings of the Sixth Workshop on Compilers for Parallel Computers*, volume 21 of *Konferenzen des Forschungszentrums Jülich*, pages 311–320, December 1996.

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[GLP09] GLPK (GNU Linear Programming Kit). `http://www.gnu.org/software/glpk`, 2009.

[GM86] Joseph A. Goguen and José Meseguer. *Logic Programming: Functions, Relations, and Equations*, chapter EQLOG: Equality, Types, and Generic Modules for Logic Programming, pages 295–363. Prentice Hall, 1986.

[GMQS89] Pierrick Gachet, Christophe Mauras, Patrice Quinton, and Yannick Saouter. Alpha du Centaur: A Prototype Environment for the Design of Parallel Regular Alorithms. In *Proceedings of the 3rd International Conference on Supercomputing (ICS)*, pages 235–243, Crete, Greece, June 1989.

[GN06] Zhi Guo and Walid A. Najjar. A Compiler Intermediate Representation for Reconfigurable Fabrics. In *16th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Madrid, Spain, August 2006.

[GQR03] Anne-Claire Guillou, Patrice Quinton, and Tanguy Risset. Hardware Synthesis for Multi-Dimensional Time. In *Proceedings IEEE 14th International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 40–50, The Hague, The Netherlands, June 2003.

[Gra66] Ron L. Graham. Bounds for Certain Multiprocessing Anomalies. *The Bell System Technical Journal*, XLV(9):1563–1581, November 1966.

[Gra00] Martin Grajcar. Conditional Scheduling for Embedded Systems using Genetic List Scheduling. In *Proceedings of the 13th International Symposium on System Synthesis (ISSS)*, pages 123–128, Madrid, Spain, 2000.

[GTA06] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, USA, October 2006.

[HA99] James C. Hoe and Arvind. Hardware Synthesis from Term Rewriting Systems. In *Proceedings of the IFIP TC10/WG10.5 Tenth International Conference on Very Large Scale Integration (VLSI)*, volume 162, pages 595–619, December 1999.

[HA00] James C. Hoe and Arvind. Synthesis of Operation-Centric Hardware Descriptions. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 511–519, San Jose, CA, USA, November 2000.

[Haj42] Georg Hajós. Über einfache und mehrfache Bedeckung des n-dimensionalen Raumes mit einem Würfelgitter. *Mathematische Zeitschrift*, 47(1):427–467, December 1942.

[HDK+05] Frank Hannig, Hritam Dutta, Alexey Kupriyanov, Jürgen Teich, Rainer Schaffer, Sebastian Siegel, Renate Merker, Ronan Keryell, Bernard Pottier, Daniel Chillet, Daniel Ménard, and Olivier Sentieys. Co-Design of Massively Parallel Embedded Processor Architectures. In *Proceedings of the first International Workshop on Reconfigurable Communication Centric System-on-Chips (ReCoSoC)*, pages 27–34, Montpellier, France, June 2005.

[HDT04a] Frank Hannig, Hritam Dutta, and Jürgen Teich. Mapping of Regular Nested Loop Programs to Coarse-grained Reconfigurable Arrays – Constraints and Methodology. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, NM, USA, April 2004.

[HDT04b] Frank Hannig, Hritam Dutta, and Jürgen Teich. Regular Mapping for Coarse-grained Reconfigurable Architectures. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume V, pages 57–60, Montréal, Quebec, Canada, May 2004.

[HDT06] Frank Hannig, Hritam Dutta, and Jürgen Teich. Mapping a Class of Dependence Algorithms to Coarse-grained Reconfigurable Arrays: Architectural Parameters and Methodology. *International Journal of Embedded Systems*, 2(1/2):114–127, January 2006.

[HDT09] Frank Hannig, Hritam Dutta, and Jürgen Teich. Parallelization Approaches for Hardware Accelerators – Loop Unrolling versus Loop Partitioning. In *Proceedings of the 22nd International Conference on Architecture of Computing Systems (ARCS)*, volume 5455 of *Lecture Notes in Computer Science (LNCS)*, pages 16–27, Delft, The Netherlands, March 2009.

[Hec93] André Heck. *Introduction to Maple*. Springer, 1993.

[Her51] Charles Hermite. Sur l'Introduction des Variables Continues dans la Théorie des Nombres. *Journal für die reine und angewandte Mathematik*, 41:191–216, 1851.

[HH92] Yin-Tsung Hwang and Yu Hen Hu. MSSM—A Design Aid for Multi-Stage Systolic Mapping. *Journal of VLSI Signal Processing Systems*, 4(2-3):125–145, May 1992.

[HH95] Yin-Tsung Hwang and Yu Hen Hu. A Unified Partitioning and Scheduling Scheme for Mapping Multi-Stage Regular Iterative Algorithms onto Processor Arrays. *Journal of VLSI Signal Processing Systems*, 11(1-2):133–150, 1995.

[HHL90] Cheng-Tsung Hwang, Yu-Chin Hsu, and Youn-Long Lin. Optimum and Heuristic Data Path Scheduling under Resource Constraints. In *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC)*, pages 65–70, Orlando, FL, USA, June 1990.

[Hig93] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993.

[Hil85] Paul N. Hilfinger. A High-Level Language and Silicon Compiler for Digital Signal Processing. In *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC)*, pages 213–216, Portland, OR, USA, May 1985.

[Hil92] Daniel D. Hils. Visual Languages and Computing Survey: Data Flow Visual Programming languages. *Journal of Visual Languages & Computing*, 3(1):69–101, 1992.

[HKM$^+$08] Amir Hormati, Manjunath Kudlur, Scott A. Mahlke, David Bacon, and Rodric Rabbah. Optimus: Efficient Realization of Streaming Applications on FPGAs. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 41–50, Atlanta, GA, USA, 2008.

[HKV$^+$07] Qubo Hu, Per Gunnar Kjeldsberg, Arnout Vandecappelle, Martin Palkovic, and Francky Catthoor. Incremental Hierarchical Memory Size Estimation for Steering of Loop Transformations. *ACM Transactions on Design Automation of Electronic Systems*, 12(4):50, 2007.

[HL87] Chua-Huang Huang and Christian Lengauer. The Derivation of Systolic Implementations of Programs. *Acta Informatica*, 24(6):595–632, 1987.

[HL95] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research*. McGraw-Hill, New York, 6 edition, 1995.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, USA, 1985.

[HOI88] Pao-Po Hou, Robert Michael Owens, and Mary Jane Irwin. DECOMPOSER: A Synthesizer for Systolic Systems. In *Proceedings of the 25th ACM/IEEE Conference on Design Automation (DAC)*, pages 650–653, Atlantic City, NJ, USA, 1988.

[HOPS05] Ulisses Kendi Hayashida, Kunio Okuda, Jairo Panetta, and Siang Wun Song. Generating Parallel Algorithms for Cluster and Grid Computing. In *Proceedings of the 5th International Conference on Computational Science (ICCS)*, volume 3514 of *Lecture Notes in Computer Science (LNCS)*, pages 509–516, Atlanta, GA, USA, May 2005.

[HP81] Louis Hafer and Alice C. Parker. A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic. In *Proceedings of the 18th ACM/IEEE Design Automation Conference (DAC)*, pages 846–853, Nashville, TN, USA, June 1981.

[HRDT08] Frank Hannig, Holger Ruckdeschel, Hritam Dutta, and Jürgen Teich. PARO: Synthesis of Hardware Accelerators for Multi-Dimensional Dataflow-Intensive Applications. In *Proceedings of the Fourth International Workshop on Applied Reconfigurable Computing (ARC)*, volume 4943 of *Lecture Notes in Computer Science (LNCS)*, pages 287–293, London, United Kingdom, March 2008.

[HRT08] Frank Hannig, Holger Ruckdeschel, and Jürgen Teich. The PAULA Language for Designing Multi-Dimensional Dataflow-Intensive Applications. In *Proceedings of the GI/ITG/GMM-Workshop – Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 129–138, Freiburg, Germany, March 2008.

[HS93] Chien Ching Chieh Huang and Ponnuswamy Sadayappan. Communication-Free Hyperplane Partitioning of Nested Loops. *Journal of Parallel and Distributed Computing*, 19(2):90–102, 1993.

[HT01] Frank Hannig and Jürgen Teich. Design Space Exploration for Massively Parallel Processor Arrays. In Victor Malyshkin, editor, *Proceedings of the 6th International Conference on Parallel Computing Technologies (PaCT)*, volume 2127 of *Lecture Notes in Computer Science (LNCS)*, pages 51–65, Novosibirsk, Russia, September 2001.

[HT02a] Frank Hannig and Jürgen Teich. Energy Estimation for Piecewise Regular Processor Arrays. In *Proceedings of the Second International Samos Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, Island of Samos, Greece, July 2002.

[HT02b] Frank Hannig and Jürgen Teich. Energy Estimation of Nested Loop Programs. In *Proceedings 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 149–150, Winnipeg, Manitoba, Canada, August 2002.

[HT04a] Frank Hannig and Jürgen Teich. *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, chapter 6, Energy Estimation and Optimization for Piecewise Regular Processor Arrays, pages 107–126. Number 20 in Signal Processing and Communications. Marcel Dekker, New York, USA, January 2004.

[HT04b] Frank Hannig and Jürgen Teich. Dynamic Piecewise Linear/Regular Algorithms. In *Proceedings of the Fourth International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, pages 79–84, Dresden, Germany, September 2004.

[HT04c] Frank Hannig and Jürgen Teich. Resource Constrained and Speculative Scheduling of an Algorithm Class with Run-Time Dependent Conditionals. In *Proceedings of the 15th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 17–27, Galveston, TX, USA, September 2004.

[HT04d] Frank Hannig and Jürgen Teich. Resource Constrained and Speculative Scheduling of Dynamic Piecewise Regular Algorithms. Technical Report 01–2004, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, 91058 Erlangen, Germany, June 2004.

[HT05] Frank Hannig and Jürgen Teich. Output Serialization for FPGA-based and Coarse-grained Processor Arrays. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 78–84, Las Vegas, NV, USA, June 2005.

[IBM07] IBM Systems and Technology Group. *Cell Broadband Engine Programming Handbook, Version 1.1*. Hopewell Junction, NY, USA, April 2007.

[ILO06] ILOG, CPLEX Division. *ILOG CPLEX 10.0, User's Manual*, 2006.

[Imp09] Impulse Accelerated Technologies, Inc. `http://www.impulsec.com`, 2009.

[Inm84] Inmos Corp. *Occam Programming Manual.* Prentice Hall, 1984.

[Int04] Intel Corporation. *Intel Itanium 2 Processor Reference Manual*, May 2004.

[IT88] François Irigoin and Rémi Triolet. Supernode Partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 319–329, San Diego, CA, USA, January 1988.

[Jai86] Kishan Jainandunsing. Optimal Partitioning Scheme for Wavefront/Systolic Array Processors. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 940–943, San Jose, CA, USA, May 1986.

[JPP88] Rajiv Jain, Alice Parker, and Nohbyung Park. Module Selection for Pipelined Synthesis. In *Proceedings of the 25th ACM/IEEE Conference on Design Automation (DAC)*, pages 542–547, Anaheim, CA, USA, June 1988.

[Kah74] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proceedings of International Federation for Information Processing (IFIP) Congress 74*, pages 471–475, Stockholm, Sweden, August 1974.

[KAS$^+$02] Vinod Kathail, Shail Aditya, Robert Schreiber, B. Ramakrishna Rau, Darren C. Cronquist, and Mukund Sivaraman. PICO: Automatically Designing Custom Computers. *Computer*, 35(9):39–47, 2002.

[KDH$^+$05] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David J. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July 2005.

[KDH+09] Joachim Keinert, Hritam Dutta, Frank Hannig, Christian Haubelt, and Jürgen Teich. Model-Based Synthesis and Optimization of Static Multi-Rate Image Processing Algorithms. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 135–140, Nice, France, April 2009.

[KFM06] Manjunath Kudlur, Kevin Fan, and Scott A. Mahlke. Streamroller: Automatic Synthesis of Prescribed Throughput Accelerator Pipelines. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 270–275, Seoul, Korea, 2006.

[KGV83] Scott Kirkpatrick, Charles Daniel Gelatt, Jr., and Mario P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[KH03] Surin Kittitornkun and Yu Hen Hu. Processor Array Synthesis from Shift-Variant Deep Nested Do Loops. *The Journal of Supercomputing*, 24(3):229–249, 2003.

[KHK+06a] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, Rainer Schaffer, and Jürgen Teich. MAML – An Architecture Description Language for Modeling and Simulation of Processor Array Architectures, Part I. Technical Report 03–2006, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, 91058 Erlangen, Germany, March 2006.

[KHK+06b] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, Jürgen Teich, Julien Lallet, Olivier Sentieys, and Sébastien Pillement. Modeling of Interconnection Networks in Massively Parallel Processor Architectures. Technical Report 05–2006, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, 91058 Erlangen, Germany, August 2006.

[KHK+06c] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, Jürgen Teich, Rainer Schaffer, and Renate Merker. An Architecture Description Language for Massively Parallel Processor Architectures. In *Proceedings of the GI/ITG/GMM-Workshop – Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 11–20, Dresden, Germany, February 2006.

[KHK+07] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, Jürgen Teich, Julien Lallet, Olivier Sentieys, and Sébastien Pillement. Modeling of

Interconnection Networks in Massively Parallel Processor Architectures. In Paul Lukowicz, Lothar Thiele, and Gerhard Tröster, editors, *Proceedings of the 20th International Conference on Architecture of Computing Systems (ARCS)*, Lecture Notes in Computer Science (LNCS), pages 268–282, Zurich, Switzerland, March 2007.

[KHKT06a] Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, and Jürgen Teich. A Dynamically Reconfigurable Weakly Programmable Processor Array Architecture Template. In *Proceedings of the 2nd International Workshop on Reconfigurable Communication Centric System-on-Chips (ReCoSoC)*, pages 31–37, Montpellier, France, July 2006.

[KHKT06b] Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, and Jürgen Teich. A Highly Parameterizable Parallel Processor Array Architecture. In *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT)*, pages 105–112, Bangkok, Thailand, December 2006.

[KHKT06c] Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, and Jürgen Teich. Hardware Cost Analysis for Weakly Programmable Processor Arrays. In *Proceedings of the International Symposium on System-on-Chip (SoC)*, pages 179–182, Tampere, Finland, November 2006.

[KHKT08] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, and Jürgen Teich. *Processor Description Languages*, chapter 12, MAML: An ADL for Designing Single and Multiprocessor Architectures, pages 295–327. Systems on Silicon. Morgan Kaufmann, June 2008.

[Khr08] Khronos OpenCL Working Group, Aaftab Munshi (editor). *The OpenCL Specification, version 1.0, rev. 29*, December 2008.

[KHT04a] Alexey Kupriyanov, Frank Hannig, and Jürgen Teich. Automatic and Optimized Generation of Compiled High-Speed RTL Simulators. In *Proceedings of Workshop on Compilers and Tools for Constrained Embedded Systems (CTCES)*, Washington, DC, USA, September 2004.

[KHT04b] Alexey Kupriyanov, Frank Hannig, and Jürgen Teich. High-Speed Event-Driven RTL Compiled Simulation. In Andy D. Pimentel and Stamatis Vassiliadis, editors, *Computer Systems: Architectures, Modeling, and Simulation, 4th International Samos Workshop (SAMOS), Proceedings*, volume 3133 of *Lecture Notes in Computer Science (LNCS)*, pages 519–529, Island of Samos, Greece, July 2004.

[KHT07] Dmitrij Kissler, Frank Hannig, and Jürgen Teich. Schwach programmiert macht stark – Massiv parallele Prozessorfelder. *Design&Elektronik*, (4):34–39, April 2007.

[Kie00] Bart Kienhuis. MatParser: An Array Dataflow Analysis Compiler. Technical Report UCB/ERL M00/9, University of California, Berkeley, CA, USA, February 2000.

[KJ88] Sun-Yuan Kung and Jack S. N. Jean. A VLSI Array Compiler System (VACS) for Array Design. In *Proceedings of the Workshop on VLSI Signal Processing III*, pages 495–508, Monterey, CA, USA, November 1988.

[KK05] Ismail Kadayif and Mahmut Taylan Kandemir. Data Space-Oriented Tiling for Enhancing Locality. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(2):388–414, May 2005.

[KKH$^+$06] Dmitrij Kissler, Alexey Kupriyanov, Frank Hannig, Dirk Koch, and Jürgen Teich. A Generic Framework for Rapid Prototyping of System-on-Chip Designs. In *Proceedings of International Conference on Computer Design (CDES)*, pages 189–195, Las Vegas, NV, USA, June 2006.

[KKHT07] Alexey Kupriyanov, Dmitrij Kissler, Frank Hannig, and Jürgen Teich. Efficient Event-driven Simulation of Parallel Processor Architectures. In *Proceedings of the 10th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 71–80, Nice, France, April 2007.

[KKT90] Konstantinos Konstantinides, Ronald T. Kaneshiro, and Jon R. Tani. Task Allocation and Scheduling Models for Multiprocessor Digital Signal Processing. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 38(12):2151–2161, December 1990.

[KL78] H. T. Kung and Charles E. Leiserson. Systolic Arrays (for VLSI). In Iain S. Duff and G. W. Stewart, editors, *Sparse Matrix Proceedings*, pages 256–282, Philadelphia, PA, USA, 1978. SIAM.

[KLB08] Markus Köster, Wayne Luk, and Geoffrey Brown. A Hardware Compilation Flow for Instance-Specific VLIW Cores. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 619–622, Heidelberg, Germany, September 2008.

[KLY00] Hyung-Sung Kim, Sung-Woo Lee, and Kee-Young Yoo. Partitioned Systolic Architecture for Modular Multiplication in $GF(2^m)$. *Information Processing Letters*, 76(3):135–139, 2000.

[KMAC06] Rahim Khoja, Mehul Marolia, Tinku Acharya, and Chaitali Chakrabarti. A Coprocessor Architecture for Fast Protein Structure Prediction. *Pattern Recognition*, 39(12):2494–2505, 2006.

[KMC+00] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[KMN+00] Kurt Keutzer, Sharad Malik, A. Richard Newton, Jan M. Rabaey, and Alberto Sangiovanni-Vincentelli. System-Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

[KMW67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The Organization of Computations for Uniform Recurrence Equations. *Journal of the Association for Computing Machinery*, 14(3):563–590, 1967.

[KNB+08] Arun Kejariwal, Alexandru Nicolau, Utpal Banerjee, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. Cache-Aware Iteration Space Partitioning. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 269–270, Salt Lake City, UT, USA, 2008.

[KNBP05] Arun Kejariwal, Alexandru Nicolau, Utpal Banerjee, and Constantine D. Polychronopoulos. A Novel Approach for Partitioning Iteration Spaces with Variable Densities. In *Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 120–131, Chicago, IL, USA, 2005.

[KNS+06] Arun Kejariwal, Alexandru Nicolau, Hideki Saito, Xinmin Tian, Milind Girkar, Utpal Banerjee, and Constantine D. Polychronopoulos. A General Approach for Partitioning N-Dimensional Parallel Nested Loops with Conditionals. In *Proceedings of the eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 49–58, Cambridge, MA, USA, 2006.

[Knu71] Donald Ervin Knuth. An Empirical Study of FORTRAN Programs. *Software - Practice and Experience (SPE)*, 1(2):105–133, 1971.

[KR06] DaeGon Kim and Sanjay Vishnu Rajopadhye. An Improved Systolic Architecture for LU Decomposition. In *Proceedings of IEEE 17th International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 231–238, Steamboat Springs, CO, USA, September 2006.

[KRD00] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In Frank Vahid and Jan Madsen, editors, *Proceedings of the 8th International Workshop on Hardware/Software Co-Design (CODES)*, pages 13–17, San Diego, CA, USA, May 2000.

[KSHT08] Dmitrij Kissler, Andreas Strawetz, Frank Hannig, and Jürgen Teich. Power-efficient Reconfiguration Control in Coarse-Grained Dynamically Reconfigurable Architectures. In *Proceedings of the 18th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, volume 5349 of *Lecture Notes in Computer Science (LNCS)*, pages 307–317, Lisbon, Portugal, September 2008.

[KSHT09] Dmitrij Kissler, Andreas Strawetz, Frank Hannig, and Jürgen Teich. Power-efficient Reconfiguration Control in Coarse-grained Dynamically Reconfigurable Architectures. *Journal of Low Power Electronics*, 5(1):96–105, April 2009.

[KSP07] Srikanth Kurra, Neeraj Kumar Singh, and Preeti Ranjan Panda. The Impact of Loop Unrolling on Controller Delay in High Level Synthesis. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 391–396, Nice, France, April 2007.

[KSS$^+$09] Joachim Keinert, Martin Streubühr, Thomas Schlichter, Joachim Falk, Jens Gladigau, Christian Haubelt, Jürgen Teich, and Michael Meredith. SystemCoDesigner – An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications. *ACM Transactions on Design Automation of Electronic Systems*, 14(1):1–23, 2009.

[Kuc98] Krzysztof Kuchcinski. An Approach to High-Level Synthesis using Constraint Logic Programming. In *Proceedings of the 24th EUROMICRO Conference*, volume 1, pages 74–82, Vesteras, Sweden, August 1998.

[Kud08] Manjunath V. Kudlur. *Streamroller: A Unified Compilation and Synthesis System for Streaming Applications*. PhD thesis, University of Michigan, USA, 2008.

[Kuh80] Robert H. Kuhn. Transforming Algorithms for Single-Stage and VLSI Architectures. In *Workshop on Interconnection Networks for Parallel and Distributed Processing*, pages 11–19, West Layfaette, IN, USA, April 1980.

[KW98] Apostolos A. Kountouris and Christophe Wolinski. Hierarchical Conditional Dependency Graphs for Conditional Resource Sharing. In *Proceedings of the 24th EUROMICRO Conference*, volume 1, pages 313–316, Vesteras, Sweden, August 1998.

[KW01] Krzysztof Kuchcinski and Christophe Wolinski. Synthesis of Conditional Behaviors using Hierarchical Conditional Dependency Graphs and Constraint Logic Programming. In *Proceedings of the Euromicro Symposium on Digital Systems Design (DSD)*, pages 220–227, September 2001.

[KWL01] Apostolos A. Kountouris, Christophe Wolinski, and Jean-Christophe Le Lann. High-level Synthesis using Hierarchical Conditional Dependency Graphs in the CODESIS System. *Journal of Systems Architecture*, 47(3-4):293–313, 2001.

[KYLL94] Taewhan Kim, Noritake Yonezawa, Jane W. S. Liu, and C. L. Liu. A Scheduling Algorithm for Conditional Resource Sharing—A Hierarchical Reduction Approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):425–438, April 1994.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, Palo Alto, CA, USA, March 2004.

[Lam74] Leslie Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17(2):83–93, 1974.

[Lam88] Monica Sin Ling Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *ACM SIGPLAN Notices*, 23(7):318–328, 1988.

[LCD$^+$00] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Proceedings of the 37th ACM/IEEE Design Automation Conference (DAC)*, pages 507–512, Los Angeles, CA, USA, June 2000.

[LCD03] Jong-eun Lee, Kiyoung Choi, and Nikil D. Dutt. An Algorithm for Mapping Loops onto Coarse-Grained Reconfigurable Architectures. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 183–188, San Diego, CA, USA, June 2003.

[LCM$^+$08] Yuan Lin, Yoonseo Choi, Scott A. Mahlke, Trevor Mudge, and Chaitaili Chakrabarti. A Parameterized Dataflow Language Extension for Embedded Streaming Systems. In *In Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 10–17, Island of Samos, Greece, July 2008.

[LDG$^+$02] Paul Lindner, Viorel Dragoi, Thomas Glinsner, Christian Schaefer, and Rafiqul Islam. 3D Interconnect through Aligned Wafer Level Bonding. pages 1439–1443, May 2002.

[Len89] Christian Lengauer. Towards Systolizing Compilation: An Overview. In *Parallel Architectures and Languages Europe PARLE*, pages 253–272, 1989.

[Len93] Christian Lengauer. Loop Parallelization in the Polytope Model. In Eike Best, editor, *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR)*, volume 715 of *Lecture Notes in Computer Science (LNCS)*, pages 398–416, Hildesheim, Germany, August 1993.

[LGAV96] Josep Llosa, Antonio González, Eduard Ayguadé, and Mateo Valero. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 80–86, Barcelona, Spain, October 1996.

[LHT09] Vahid Lari, Frank Hannig, and Jürgen Teich. System Integration of Tightly-Coupled Reconfigurable Processor Arrays and Evaluation of Buffer Size Effects on Their Performance. In *Proceedings of the 4th International Symposium on Embedded Multicore Systems-on-Chip (MCSoC)*, pages 528–534, Vienna, Austria, September 2009.

[LK90] Pei-Zong Lee and Zvi Meir Kedem. Mapping Nested Loop Algorithms into Multidimensional Systolic Arrays. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):64–76, January 1990.

[LMQ91] Hervé Leverge, Christophe Mauras, and Patrice Quinton. A Language-Oriented Approach to the Design of Systolic Chips. In Ed F. Deprettere and A. J. van der Veen, editors, *Proceedings of the International Workshop on Algorithms and Parallel VLSI Architectures*, volume A: Tutorials, pages 309–327, Gers, France, June 1991.

[LN07] Edward A. Lee and Stephen Neuendorffer. Tutorial: Building Ptolemy II Models Graphically. Technical Report UCB/EECS-2007-129, EECS Department, University of California, Berkeley, USA, October 2007.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008.

[LP92] Wei Li and Keshav Pingali. A Singular Loop Transformation Framework Based on Non-singular Matrices. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 757 of *Lecture Notes in Computer Science (LNCS)*, pages 391–405, New Haven, CT, USA, August 1992.

[LP94] Wei Li and Keshav Pingali. A Singular Loop Transformation Framework Based on Non-Singular Matrices. *International Journal of Parallel Programming*, 22(2):183–205, 1994.

[LP95] Edward A. Lee and Thomas M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[LPM97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 330–335, Research Triangle Park, NC, USA, December 1997.

[LW85] Guo-Jie Li and Benjamin W. Wah. The Design of Optimal Systolic Arrays. *IEEE Transactions on Computers*, 34(1):66–77, 1985.

[May83] David May. OCCAM. *ACM SIGPLAN Notices*, 18(4):69–79, 1983.

[MC95] Graham M. Megson and Xian Chen. A Synthesis Method of LSGP Partitioning for Given-Shape Regular Arrays. In *Proceedings of the 9th International Symposium on Parallel Processing (IPPS)*, pages 234–238, Santa Barbara, CA, USA, April 1995.

[McG82] James R. McGraw. The VAL Language: Description and Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(1):44–82, 1982.

[Meg93] Graham M. Megson. Mapping a Class of Run-Time Dependencies onto Regular Arrays. In *Proceedings of the 7th International Parallel Processing Symposium (IPPS)*, pages 97–104, Newport Beach, CA, USA, April 1993.

[Men06] Oskar Mencer. ASC: A Stream Compiler for Computing with FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(9):1603–1617, September 2006.

[Men09] Mentor Graphics Corp. `http://www.mentor.com`, 2009.

[MF86] Dan I. Moldovan and Jose A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, 35(1):1–12, 1986.

[MFM04] Jan Müller, Dirk Fimmel, and Renate Merker. Optimal Loop Scheduling with Register Constraints using Flow Graphs. In *Proceedings of the 7th International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN)*, pages 180–186, Hong Kong, China, May 2004.

[MFMS03] Jan Müller, Dirk Fimmel, Renate Merker, and Rainer Schaffer. A Hardware-Software System for Tomographic Reconstruction. *Journal of Circuits, Systems, and Computers*, 12(2):203–229, 2003.

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *In Proceedings of the ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 896–907, San Diego, CA, USA, 2003.

[MHB^{+}94] Scott A. Mahlke, Richard E. Hank, Roger A. Bringmann, John C. Gyllenhaal, Gallagher David M., and Wen-mei W. Hwu. Characterizing the Impact of Predicated Execution on Branch Prediction. In

Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO), pages 217–227, San Jose, CA, USA, November 1994.

[MHDT09] Richard Membarth, Frank Hannig, Hritam Dutta, and Jürgen Teich. Efficient Mapping of Multiresolution Image Filtering Algorithms on Graphics Processors. In Koen Bertels, Nikitas Dimopoulos, Christina Silvano, and Stephan Wong, editors, *Proceedings of the 9th International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, volume 5657 of *Lecture Notes in Computer Science (LNCS)*, pages 277–288, Island of Samos, Greece, July 2009.

[MKD$^+$09] Richard Membarth, Philipp Kutzer, Hritam Dutta, Frank Hannig, and Jürgen Teich. Acceleration of Multiresolution Imaging Algorithms: A Comparative Study. In *Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 211–214, Boston, MA, USA, July 2009.

[ML90] Jaime H. Moreno and Tomás Lang. Matrix Computations on Systolic-Type Meshes: An Introduction to the Multimesh Graph Method. *Computer*, 23(4):32–51, April 1990.

[MLC$^+$92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution using the Hyperblock. *ACM SIGMICRO Newsletter*, 23(1–2):45–54, 1992.

[MM04] Manju Manjunathaiah and Graham M. Megson. Tools for Regularizing Array Designs. *International Journal of Parallel, Emergent and Distributed Systems*, 19(1):51–75, March 2004.

[MMRR01] Manju Manjunathaiah, Graham M. Megson, Sanjay Vishnu Rajopadhye, and Tanguy Risset. Uniformization of Affine Dependance Programs for Parallel Embedded System Design. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 205–213, Valencia, Spain, September 2001.

[Moh06] Stefan Mohl. Using FPGAs in Supercomputers: Breaking with Reconfigurable Computing. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC)*, Tampa, FL, USA, November 2006.

[Mol83] Dan I. Moldovan. On the Design of Algorithms for VLSI Systolic Arrays. *Proceedings of the IEEE*, 71(1):113–120, January 1983.

[Mol87] Dan I. Moldovan. ADVIS: A Software Package for the Design of Systolic Arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(1):33–40, January 1987.

[MP01] Alessandro Marongiu and Paolo Palazzari. Automatic Implementation of Affine Iterative Algorithms: Design Flow and Communication Synthesis. *Computer Physics Communications*, 139(1):109–131, 2001.

[MS90] David May and Roger Shepherd. Occam and the Transputer. In *Advances in Petri Nets 1989, Covers the 9th European Workshop on Applications and Theory in Petri Nets—Selected Papers*, volume 424 of *Lecture Notes in Computer Science (LNCS)*, pages 329–353, 1990.

[Muc97] Steven Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[Mül04] Jan Müller. *Schleifenparallelisierung für Prozessoren mit parallelen Funktionseinheiten.* PhD thesis, Technische Universität Dresden, Dresden, Germany, October 2004.

[Mül06] Jan Müller. Generalised Resource Model for Parallel Instruction Scheduling. In *Proceedings of the 5th International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, pages 89–94, Bialystok, Poland, September 2006.

[Mur81] Bruce A. Murtagh. *Advanced Linear Programming: Computation and Practice*. McGraw-Hill, New York, 1981.

[MW84] Willard L. Miranker and Andrew Winkler. Spacetime Representations of Computational Structures. *Computing*, 32(2):93–114, 1984.

[NBD+03] Walid A. Najjar, A. P. Wim Böhm, Bruce A. Draper, Jeff Hammes, Robert Rinker, J. Ross Beveridge, Monica Chawathe, and Charles Ross. High-Level Language Abstraction for Reconfigurable Computing. *Computer*, 36(8):63–69, 2003.

[NGCD91] Stefaan Note, Werner Geurts, Francky Catthoor, and Hugo De Man. Cathedral-III: Architecture-Driven High-Level Synthesis for High Throughput DSP Applications. In *Proceedings of the 28th ACM/IEEE Design Automation Conference (DAC)*, pages 597–602, San Francisco, CA, USA, June 1991.

[Nik93] Rishiyur S. Nikhil. An Overview of the Parallel Language Id (a foundation for pH, a parallel dialect of Haskell). Technical report, Digital Equipment Corp., Cambridge, MA, USA, September 1993.

[NW99] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience, New York, NY, USA, 1999.

[OF88] Matthew Thomas O'Keefe and Jose A. B. Fortes. Bit Level Processor Arrays: Current Architectures and a Design and Programming Tool. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 3, pages 2751–2755, Espoo, Finland, June 1988.

[Omt88] E. Theodore L. Omtzigt. SYSTARS: A CAD Tool for the Synthesis and Analysis of VLSI Systolic/Wavefront Arrays. In *Proceedings of the International Conference on Systolic Arrays*, pages 383–391, San Diego, CA, USA, May 1988.

[OSY06] Kunio Okuda, Siang Wun Song, and Marcos Tatsuo Yamamoto. Reliable Systolic Computing Through Redundancy. In *Proceedings of the 11th Asia-Pacific Computer Systems Architecture Conference (ACSAC)*, volume 4186 of *Lecture Notes in Computer Science (LNCS)*, pages 423–429, Shanghai, China, September 2006.

[PBD+08] Andrew R. Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, and Prasanna Sundararajan. CHiMPS: A High-Level Compilation Flow for Hybrid CPU-FPGA Architectures. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA)*, pages 261–261, Monterey, CA, USA, 2008.

[PBSF06] Carlos Alba Pinto, Aleksandar Beric, Satendra Pal Singh, and Sachin Farfade. HiveFlex-Video VSP1: Video Signal Processing Architecture for Video Coding and Post-Processing. In *Proceedings of the Eighth IEEE International Symposium on Multimedia (ISM)*, pages 493–500, 2006.

[PFM+08] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. Edge-centric Modulo Scheduling for Coarse-Grained Reconfigurable Architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 166–176, Toronto, Ontario, Canada, October 2008.

[PJ03] Satish Pillai and Margarida F. Jacome. Compiler-Directed ILP Extraction for Clustered VLIW/EPIC Machines: Predication, Speculation and Modulo Scheduling. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 422–427, 2003.

[PK89] Pierre G. Paulin and John P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, June 1989.

[PL06] Saeed Parsa and Shahriar Lotfi. A New Genetic Algorithm for Loop Tiling. *The Journal of Supercomputing*, 37(3):249–269, 2006.

[Pla99] Toomas P. Plaks. *Piecewise Regular Arrays: Application-Specific Computations*. Parallel Processing Series. Gordon and Breach Science Publishers, 1999.

[PM06] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing*. Prentice Hall, 4th edition, April 2006.

[Pol88] Constantine D. Polychronopoulos. Compiler Optimizations for Enhancing Parallelism and their Impact on Architecture Design. *IEEE Transactions on Computers*, 37(8):991–1004, August 1988.

[Pol07] PolyLib – A Library of Polyhedral Functions. `http://icps.u-strasbg.fr/PolyLib`, 2007.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, NJ, USA, 1982.

[PS91] Joseph C. H. Park and Michael S. Schlansker. On Predicated Execution. Technical Report HPL-91-58, Hewlett-Packard Laboratories, Palo Alto, CA, USA, May 1991.

[PS96] Nelson Luiz Passos and Edwin Hsing-Mean Sha. Achieving Full Parallelism using Multidimensional Retiming. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1150–1163, 1996.

[QR02] Patrice Quinton and Tanguy Risset. *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation – SAMOS*, volume 2268 of *Lecture Notes in Computer Science (LNCS)*, chapter Structured Scheduling of Recurrence Equations: Theory and Practice, pages 112–134. Springer, 2002.

[QRW00] Fabien Quilleré, Sanjay Vishnu Rajopadhye, and Doran Wilde. Generation of Efficient Nested Loops from Polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000.

[QSL$^+$08] Meikang Qiu, Edwin Hsing-Mean Sha, Meilin Liu, Man Lin, Shaoxiong Hua, and Laurence T. Yang. Energy Minimization with Loop Fusion and Multi-Functional-Unit Scheduling for Multidimensional DSP. *Journal of Parallel and Distributed Computing*, 68(4):443–455, 2008.

[Qui84] Patrice Quinton. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA)*, pages 208–214, Ann Arbor, MI, USA, June 1984.

[Ram92] Jagannathan Ramanujam. Non-Unimodular Transformations of Nested Loops. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, pages 214–223, Minneapolis, MN, USA, November 1992.

[Ram95] Jagannathan Ramanujam. Beyond Unimodular Transformations. *The Journal of Supercomputing*, 9(4):365–389, 1995.

[Rao85] Sailesh K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays*. PhD thesis, Stanford University, 1985.

[Rau94] B. Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO)*, pages 63–74, San Jose, CA, USA, November 1994.

[RCP$^+$01] Robert Rinker, Margaret Carter, Amitkumar Patel, Monica Chawathe, Charlie Ross, Jeffrey Hammes, Walid A. Najjar, and A. P. Wim Böhm. An Automated Process for Compiling Dataflow Graphs into Reconfigurable Hardware. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):130–139, February 2001.

[RDHT05] Holger Ruckdeschel, Hritam Dutta, Frank Hannig, and Jürgen Teich. Automatic FIR Filter Generation for FPGAs. In Timo D. Hämäläinen, Andy D. Pimentel, Jarmo Takala, and Stamatis Vassiliadis, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation, 5th International Workshop, SAMOS, Proceedings*, volume 3553 of *Lecture Notes in Computer Science (LNCS)*, pages 51–61, Island of Samos, Greece, July 2005.

[RF93] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7(1–2):9–50, 1993.

[RG06] A. N. M. Ehtesham Rafiq and Fayez Gebali. Processor Array Architectures for Deep Packet Classification. *IEEE Transactions on Parallel and Distributed Systems*, 17(3):241–252, 2006.

[RHDR07] Lakshminarayanan Renganarayanan, Manjukumar Harthikote-Matha, Rinku Dewri, and Sanjay Vishnu Rajopadhye. Towards Optimal Multi-Level Tiling for Stencil Computations. In *Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, Long Beach, CA, USA, 2007.

[RK90] Vwani Prasad Roychowdhury and Thomas Kailath. Study of Parallelism in Regular Iterative Algorithms. In *Proceedings of the Second Annual ACM symposium on Parallel Algorithms and Architectures (SPAA)*, pages 367–376, Island of Crete, Greece, July 1990.

[RR02] Fabrice Rastello and Yves Robert. Automatic Partitioning of Parallel Loops with Parallelepiped-Shaped Tiles. *IEEE Transactions on Parallel and Distributed Systems*, 13(5):460–470, 2002.

[RS87] John R. Rose and Guy Lewis Steele, Jr. C*: An Extended C Language for Data Parallel Programming. In *In Proceedings Second International Conference on Supercomputing (ICS)*, volume 2, pages 2–16, San Francisco, CA, USA, May 1987.

[RS92] Jagannathan Ramanujam and Ponnuswamy Sadayappan. Tiling Multidimensional Iteration Spaces for Multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, 1992.

[RST92] B. Ramakrishna Rau, Michael S. Schlansker, and Partha P. Tirumalai. Code Generation Schema for Modulo Scheduled DO-Loops and WHILE-Loops. Technical Report HPL-92-47, Hewlett-Packard Laboratories, Palo Alto, CA, USA, April 1992.

[RTG$^+$07] Hongbo Rong, Zhizhong Tang, Ramaswamy Govindarajan, Alban Douillet, and Guang R. Gao. Single-Dimension Software Pipelining for Multidimensional Loops. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(1):1–44, March 2007.

[RTRK88] Vwani Prasad Roychowdhury, Lothar Thiele, Sailesh K. Rao, and Thomas Kailath. On the Localization of Algorithms of VLSI Processor Arrays. In *Proceedings of the Workshop on VLSI Signal Processing III*, pages 459–470, November 1988.

[Ruc06] Holger Ruckdeschel. Prozessorfeld-Synthese für partitionierte verschachtelte Schleifenprogramme. Diplomarbeit, Department of Computer Science 12, University of Erlangen-Nuremberg, September 2006.

[RYYT89] B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towie. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-Offs. *Computer*, 22(1):12–26, 28–30, 32–35, 1989.

[SAM$^+$02] Robert Schreiber, Shail Aditya, Scott A. Mahlke, Vinod Kathail, B. Ramakrishna Rau, Darren Cronquist, and Mukund Sivaraman. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators. *Journal of VLSI Signal Processing Systems*, 31(2):127–142, 2002.

[SAR$^+$00a] Robert Schreiber, Shail Aditya, B. Ramakrishna Rau, Vinod Kathail, Scott A. Mahlke, Santosh Abraham, and Greg Snider. High-Level Synthesis of Nonprogrammable Hardware Accelerators. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 113–124, Boston, MA, USA, 2000.

[SAR$^+$00b] Robert Schreiber, Shail Aditya, B. Ramakrishna Rau, Vinod Kathail, Scott A. Mahlke, Santosh Abraham, and Greg Snider. High-Level Synthesis of Nonprogrammable Hardware Accelerators. Technical Report HPL-2000-31, Hewlett-Packard Laboratories, Palo Alto, CA, USA, May 2000.

[SB00] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Signal Processing and Communications. Marcel Dekker, 2000.

[SCD$^+$97] Michael S. Schlansker, Thomas M. Conte, James Dehnert, Kemal Ebcioglu, Jesse Z. Fang, and Carol L. Thompson. Compilers for Instruction-Level Parallelism. *Computer*, 30(12):63–69, 1997.

[SCDS04] Alberto Sangiovanni-Vincentelli, Luca Carloni, Fernando De Bernardinis, and Marco Sgroi. Benefits and Challenges for Platform-Based Design. In *Proceedings of the 41st ACM/IEEE Design Automation Conference (DAC)*, pages 409–414, San Diego, CA, USA, June 2004.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. Whily – Interscience series in discrete mathematics. John Wiley & Sons, Chichester, New York, USA, 1986.

[SD03] Todor Stefanov and Ed F. Deprettere. Deriving Process Networks from Weakly Dynamic Applications in System-Level Design. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign & System synthesis (CODES+ISSS)*, pages 90–96, Newport Beach, CA, USA, 2003.

[SDJ84] Bogong Su, Shiyuan Ding, and Lan Jin. An Improvement of Trace Scheduling for Global Microcode Compaction. *ACM SIGMICRO Newsletter*, 15(4):78–85, 1984.

[SF91] Weijia Shang and Jose A. B. Fortes. Time Optimal Linear Schedules for Algorithms with Uniform Dependencies. *IEEE Transactions on Computers*, 40(6):723–742, 1991.

[SFS$^+$95] Mirjam Schönfeld, Jens Franzen, Markus Schwiegershausen, Peter Pirsch, Uwe Vehlies, and Andreas Münzner. The LISA Design Environment for the Synthesis of Array Processors including Memories for the Data Transfer and Fault Tolerance by Reconfiguration and Coding Techniques. *Journal of VLSI Signal Processing Systems*, 11(1-2):51–74, 1995.

[SG04] Arthur Stoutchinin and Guang Gao. If-Conversion in SSA Form. In *Proceedings of the 10th International European Conference on Parallel Computing (Euro-Par)*, volume 3149 of *Lecture Notes in Computer Science (LNCS)*, pages 336–345, Pisa, Italy, August 2004.

[SGG04] Avi Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, 7th edition, 2004.

[SGI09] SGI RASC Technology. `http://www.sgi.com/products/rasc/`, 2009.

[SH02] Michael D. Smith and Glenn Holloway. *An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization*. Harvard University, Cambridge, MA, USA, 2002.

[SHHT05] Thomas Schlichter, Christian Haubelt, Frank Hannig, and Jürgen Teich. Using Symbolic Feasibility Tests during Design Space Exploration of Heterogeneous Multi-Processor Systems. In *Proceedings of the 16th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 9–14, Island of Samos, Greece, July 2005.

[SL96] Mark G. Stoodley and Corinna G. Lee. Software Pipelining Loops with Conditional Branches. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 262–273, Paris, France, December 1996.

[SL99] Eric Stotzer and Ernst Leiss. Modulo Scheduling for the TMS320C6x VLIW DSP Architecture. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 28–34, Atlanta, GA, USA, May 1999.

[SLL$^+$00] Hartej Singh, Ming-Hau Lee, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, and Eliseu M. Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000.

[SM98] SangMin Shim and Soo-Mook Moon. Split-Path Enhanced Pipeline Scheduling for Loops with Control Flows. In *Proceedings of the 31st Annual ACM/IEEE international symposium on Microarchitecture (MICRO)*, pages 93–102, Dallas, TX, USA, November 1998.

[SM00] Andrew Stone and Elias S. Manolakos. DG2VHDL: A Tool to Facilitate the High Level Synthesis of Parallel Processing Array Architectures. *Journal of VLSI Signal Processing Systems*, 24(1):99–120, 2000.

[SM04] Sebastian Siegel and Renate Merker. Optimized Data-Reuse in Processor Arrays. In *Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 315–325, Galveston, TX, USA, September 2004.

[SM06a] Rainer Schaffer and Renate Merker. Parameterized Mapping of Algorithms onto Processor Arrays with Sub-Word Parallelism. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS)*, pages 99–106, July 2006.

[SM06b] Sebastian Siegel and Renate Merker. Efficient Realization of Data Dependencies in Algorithm Partitioning under Resource Constraints. In W. E. Nagel, W. V. Walter, and W. Lehner, editors, *Proceedings of the 12th International European Conference on Parallel Computing (Euro-Par)*, volume 4128 of *Lecture Notes in Computer Science (LNCS)*, pages 1181–1191, Dresden, Germany, August 2006.

[SM06c] Sebastian Siegel and Renate Merker. Minimum Cost for Channels and Registers in Processor Arrays by Avoiding Redundancy. In *Proceedings of IEEE 17th International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 28–32, Steamboat Springs, CO, USA, September 2006.

[SM06d] Sebastian Siegel and Renate Merker. Optimization of Communication Cost within Processor Arrays Caused by I/O. In *Proceedings of the 18th International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 680–685, Dallas, TX, USA, November 2006.

[SMC00] Rainer Schaffer, Renate Merker, and Francky Catthoor. Combining Background Memory Management and Regular Array Co-Partitioning, Illustrated on a Full Motion Estimation Kernel. In *Proceedings of the Thirteenth International Conference on VLSI Design*, pages 104–109, Calcutta, India, January 2000.

[SMC03] Rainer Schaffer, Renate Merker, and Francky Catthoor. Causality Constraints for Processor Architectures with Sub-Word Parallelism. In *Proceedings of the Euromicro Symposium on Digital System Design (DSD)*, pages 82–89, September 2003.

[SMC06] Rainer Schaffer, Renate Merker, and Francky Catthoor. Derivation of Packing Instructions for Exploiting Sub-Word Parallelism. In *Proceedings of the IEEE International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, pages 167–172, Bialystok, Poland, September 2006.

[SMHT06] Sebastian Siegel, Renate Merker, Frank Hannig, and Jürgen Teich. Communication-conscious Mapping of Regular Nested Loop Programs onto Massively Parallel Processor Arrays. In *Proceedings of the 18th International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 71–76, Dallas, TX, USA, November 2006.

[SMHT08] Rainer Schaffer, Renate Merker, Frank Hannig, and Jürgen Teich. Utilization of all Levels of Parallelism in a Processor Array with Subword Parallelism. In *Proceedings of the 11th Euromicro Conference on Digital System Design (DSD)*, pages 391–398, Parma, Italy, September 2008.

[SMN$^+$03] Dinesh C. Suresh, Satya Ranjan Mohanty, Walid A. Najjar, Laxmi N. Bhuyan, and Frank Vahid. Loop Level Analysis of Security and Network Applications. In *Proceedings of the sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pages 44–50, Anaheim, CA, USA, February 2003.

[SO98] Marc Snir and Steve Otto. *MPI—The Complete Reference: The MPI Core*. MIT Press, 2nd edition, 1998.

[SOF94] Weijia Shang, Matthew Thomas O'Keefe, and Jose A. B. Fortes. On Loop Transformations for Generalized Cycle Shrinking. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):193–204, 1994.

[SR00] Michael S. Schlansker and B. Ramakrishna Rau. EPIC: Explicitly Parallel Instruction Computing. *Computer*, 33(2):37–45, February 2000.

[SSM06] Sebastian Siegel, Rainer Schaffer, and Renate Merker. Efficient Realization of the Edge Detection Algorithm on a Processor Array with Sub-Word Parallelism. In *Proceedings of the IEEE International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, pages 173–178, Bialystok, Poland, September 2006.

[SSV08] Scott Sirowy, Greg Stitt, and Frank Vahid. C is for Circuits: Capturing FPGA Circuits as Sequential Code for Portability. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field programmable Gate Arrays (FPGA)*, pages 117–126, Monterey, CA, USA, 2008.

[ST86] Harold S. Stone and Dominique Thibaut. Footprints in the Cache. *ACM SIGMETRICS Performance Evaluation Review*, 14(1):4–8, 1986.

[ST87a] Uwe Schwiegelshohn and Lothar Thiele. A Systolic Algorithm for Cyclic-by-Rows SVD. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 768–770, Dallas, TX, USA, April 1987.

[ST87b] Uwe Schwiegelshohn and Lothar Thiele. A Systolic Algorithm for Cyclic-by-Rows SVD. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 791–794, Dallas, TX, USA, April 1987.

[ST88] Uwe Schwiegelshohn and Lothar Thiele. A Systolic Array for the Assignment Problem. *IEEE Transactions on Computers*, 37(11):1422–1425, 1988.

[Ste04] Todor Stefanov. *Converting Weakly Dynamic Programs to Equivalent Process Network Specifications*. PhD thesis, Leiden University, Leiden, The Netherlands, September 2004.

[Str03] Gilbert Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, third edition, 2003.

[SUI] SUIF Compiler System. `http://suif.stanford.edu`.

[Swa87] Earl E. Swartzlander. *Systolic Signal Processing Systems*. Marcel Dekker, 1987.

[Syn09] Synfora, Inc. `http://www.synfora.com`, 2009.

[SZT$^+$04] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed F. Deprettere. System Design using Kahn Process Networks: The Compaan/Laura Approach. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 340–345, Paris, France, February 2004.

[TA93] Lothar Thiele and Ulrich Arzt. On the Synthesis of Massively Parallel Architectures. *International Journal of High Speed Electronics*, 4(2):99–131, January 1993.

[TE02] Sid-Ahmed-Ali Touati and Christine Eisenbeis. Cyclic Register Pressure and Allocation for Modulo Scheduled Loops. Technical Report RR-4442, INRIA -Université Paris Sud - Paris XI, France, April 2002.

[Tei93] Jürgen Teich. *A Compiler for Application-Specific Processor Arrays*. PhD thesis, Institut für Mikroelektronik, Universität des Saarlandes, Saarbrücken, Germany, 1993.

[Tei97] Jürgen Teich. *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Springer, Heidelberg, 1997.

[Tex08] Texas Instruments. *TMS320C6000 DSP Peripherals Overview*, December 2008.

[TGP07] Justin L. Tripp, Maya B. Gokhale, and Kristopher D. Peterson. Trident: From High-Level Language to Hardware Circuitry. *Computer*, 40(3):28–37, 2007.

[The08] The GMP Team. *GNU MP: The GNU Multiple Precision Arithmetic Library, Edition 4.2.4*, September 2008.

[Thi88] Lothar Thiele. On the Hierarchical Design of VLSI Processor Arrays. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 3, pages 2517–2520, June 1988.

[Thi92] Lothar Thiele. *Computer Systems and Software Engineering: State-of-the-Art*, chapter 4, Compiler Techniques for Massive Parallel Architectures, pages 101–151. Kluwer Academic Publishers, Boston, MA, USA, 1992.

[Thi95] Lothar Thiele. Resource Constrained Scheduling of Uniform Algorithms. *Journal of VLSI Signal Processing*, 10:295–310, 1995.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 1999.

[THR$^+$07] Jürgen Teich, Frank Hannig, Holger Ruckdeschel, Hritam Dutta, Dmitrij Kissler, and Andrej Stravet. A Unified Retargetable Design Methodology for Dedicated and Re-Programmable Multiprocessor Arrays: Case Study and Quantitative Evaluation. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), Invited paper*, pages 14–24, Las Vegas, NV, USA, June 2007.

[TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *In Proceedings of the 11th International Conference on Compiler Construction (CC)*, volume 2304 of *Lecture Notes in Computer Science (LNCS)*, pages 179–196, Grenoble, France, April 2002.

[TKD02] Alexandru Turjan, Bart Kienhuis, and Ed F. Deprettere. A Compile Time based Approach for Solving Out-of-order Communication in Kahn Process Networks. In *Proceedings of the 13th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 17–28, San Jose, CA, USA, July 2002.

[TKD05] Alexandru Turjan, Bart Kienhuis, and Ed F. Deprettere. Solving Out-of-Order Communication in Kahn Process Networks. *Journal of VLSI Signal Processing Systems*, 40(1):7–18, 2005.

[TR91] Lothar Thiele and Vwani Prasad Roychowdhury. Systematic Design of Local Processor Arrays for Numerical Algorithms. In Ed F. Deprettere and A. J. van der Veen, editors, *Proceedings of the International Workshop on Algorithms and Parallel VLSI Architectures*, volume A: Tutorials, pages 329–339, Amsterdam, 1991.

[Tri09] Trimaran, An Infrastructure for Research in Backend Compilation and Architecture Exploration. `http://www.trimaran.org`, 2009.

[TS86] Chia-Jeng Tseng and Daniel P. Siewiorek. Automated Synthesis of Data Paths in Digital Systems. volume 5, pages 379–395, July 1986.

[TT91] Jürgen Teich and Lothar Thiele. Control Generation in the Design of Processor Arrays. *Journal of VLSI Signal Processing Systems*, 3(1-2):77–92, 1991.

[TT92] Jürgen Teich and Lothar Thiele. A Transformative Approach to the Partitioning of Processor Arrays. In *Proceedings of the International Conference on Application Specific Array Processors (ASAP)*, pages 4–20, Berkeley, CA, USA, August 1992.

[TT93] Jürgen Teich and Lothar Thiele. Partitioning of Processor Arrays: A Piecewise Regular Approach. *Integration, the VLSI Journal*, 14(3):297–332, 1993.

[TT02] Jürgen Teich and Lothar Thiele. Exact Partitioning of Affine Dependence Algorithms. In Ed F. Deprettere, Jürgen Teich, and Stamatis Vassiliadis, editors, *Embedded Processor Design Challenges*, volume 2268 of *Lecture Notes in Computer Science (LNCS)*, pages 135–153, March 2002.

[TTZ96] Jürgen Teich, Lothar Thiele, and Li Zhang. Scheduling of Partitioned Regular Algorithms on Processor Arrays with Constrained Resources. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 131–144, Chicago, IL , USA, August 1996.

[TTZ97] Jürgen Teich, Lothar Thiele, and Li Zhang. Scheduling of Partitioned Regular Algorithms on Processor Arrays with Constrained Resources. *Journal of VLSI Signal Processing*, 17(1):5–20, September 1997.

[TWR+88] Chia-Jeng Tseng, Ruey-Sing Wei, Steven G. Rothweiler, Michael M. Tong, and Ajoy K. Bose. Bridge: A Versatile Behavioral Synthesis System. In *Proceedings of the 25th ACM/IEEE Design Automation Conference (DAC)*, pages 415–420, Atlantic City, NJ, USA, 1988.

[VBC06] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral Code Generation in the Real World. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC)*, volume 3923 of *Lecture Notes in Computer Science (LNCS)*, pages 185–201, Vienna, Austria, March 2006.

[Veh95] Uwe Vehlies. Stepwise Transformation of Algorithms into Array Processor Architectures by the DECOMP. *VLSI Design*, 3(1):67–80, 1995.

[vFM+05] Jan van der Veen, Sándor Fekete, Mateusz Majer, Ali Ahmadinia, Christophe Bobda, Frank Hannig, and Jürgen Teich. Defragmenting the Module Layout of a Partially Reconfigurable Device. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 92–101, Las Vegas, NV, USA, June 2005.

[VNK+03] Girish Venkataramani, Walid A. Najjar, Fadi J. Kurdahi, Nader Bagherzadeh, Wim Böhm, and Jeff Hammes. Automatic Compilation to a Coarse-grained Reconfigurable System-on-Chip. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(4):560–589, 2003.

[WC00] Chris H. Walshaw and Mark Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. *Parallel Computing*, 26(12):1635–1660, 2000.

[WFW+94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12):31–37, 1994.

[WHSB92] Nancy J. Warter, Grant E. Haab, Krishna Subramanian, and John W. Bockhaus. Enhanced Modulo Scheduling for Loops with Conditional Branches. *ACM SIGMICRO Newsletter*, 23(1-2):170–179, 1992.

[Wil93] Doran K. Wilde. A Library for Doing Polyhedral Operations. Technical Report 785, IRISA (Institut de Recherche en Informatique et Systèmes Aléatoires), Campus de Beaulieu, 35042 Rennes Cedex, France, December 1993.

[WKTH08a] Christophe Wolinski, Krzysztof Kuchcinski, Jürgen Teich, and Frank Hannig. Area and Reconfiguration Time Minimization of the Communication Network in Regular 2D Reconfigurable Architectures. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 391–396, Heidelberg, Germany, September 2008.

[WKTH08b] Christophe Wolinski, Krzysztof Kuchcinski, Jürgen Teich, and Frank Hannig. Communication Network Reconfiguration Overhead Optimization in Programmable Processor Array Architectures. In *Proceedings of the 11th Euromicro Conference on Digital System Design (DSD)*, pages 345–352, Parma, Italy, September 2008.

[WKTH08c] Christophe Wolinski, Krzysztof Kuchcinski, Jürgen Teich, and Frank Hannig. Optimization of Routing and Reconfiguration Overhead in Programmable Processor Array Architectures. In *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 306–309, Palo Alto, CA, USA, April 2008.

[WL01] Markus Weinhardt and Wayne Luk. Pipeline Vectorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20:234–248, February 2001.

[Wol96a] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[Wol96b] Stephen Wolfram. *The Mathematica Book*. Cambridge University Press, New York, NY, USA, 3rd edition, 1996.

[WPS96] Qingyan Wang, Nelson Luiz Passos, and Edwin Hsing-Mean Sha. Optimal Data Scheduling for Uniform Multidimensional Applications. *IEEE Transactions on Computers*, 45(12):1439–1444, 1996.

[WS91] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 30–44, Toronto, Ontario, Canada, June 1991.

[WS94] Doran Wilde and Oumarou Sié. Regular Array Synthesis using Alpha. In *Proceedings of the International Conference on Application Specific Array Processors (ASAP)*, pages 200–211, San Francsico, CA, USA, August 1994.

[XH98] Jingling Xue and Chua-Huang Huang. Reuse-Driven Tiling for Improving Data Locality. *International Journal of Parallel Programming*, 26(6):671–696, 1998.

[XL91] Jingling Xue and Christian Lengauer. Specifying Control Signals for One-Dimensional Systolic Arrays by Uniform Recurrence Equations. In *Proceedings of the International Workshop on Algorithms and Parallel VLSI Architectures*, pages 181–186, Gers, France, June 1991.

[Xue94] Jingling Xue. Automating Non-Unimodular Loop Transformations for Massive Parallelism. *Parallel Computing*, 20(5):711–728, 1994.

[Xue97] Jingling Xue. Unimodular Transformations of Non-Perfectly Nested Loops. *Parallel Computing*, 22(12):1621–1645, 1997.

[Xue00] Jingling Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[YC88] Yoav Yaacoby and Peter R. Cappello. Scheduling a System of Affine Recurrence Equations onto a Systolic Array. In *Proceedings of the International Conference on Systolic Arrays*, pages 373–382, San Diego, CA, USA, May 1988.

[YC92] Yoav Yaacoby and Peter R. Cappello. Decoupling the Dimensions of a System of Affine Recurrence Equations. *Linear Algebra and its Applications*, 167:157–170, April 1992.

[YCJ96] Jinn-Wang Yeh, Wen-Jiunn Cheng, and Chein-Wei Jen. VASS—A VLSI Array System Synthesizer. *Journal of VLSI Signal Processing Systems*, 12(2):135–158, 1996.

[YKM03] Han-Saem Yun, Jihong Kim, and Soo-Mook Moon. Time Optimal Software Pipelining of Loops with Control Flows. *International Journal of Parallel Programming*, 31(5):339–391, 2003.

[ZA97] Karl-Heinz Zimmermann and Wolfgang Achtziger. Finding Space-Time Transformations for Uniform Recurrences via Branching Parametric Linear Programming. *The Journal of VLSI Signal Processing Systems*, 15(3):259–274, 1997.

[ZA01] Karl-Heinz Zimmermann and Wolfgang Achtziger. Optimal Piecewise Linear Schedules for LSGP- and LPGS-Decomposed Array Processors via Quadratic Programming. *Computer Physics Communications*, 139:64–89, September 2001.

[Zeh96] Eberhard Zehendner. *Entwurf systolischer Systeme: Abbildung regulärer Algorithmen auf synchrone Prozessorarrays*. Teubner, Stuttgart, Leipzig, 1996.

[ZFJ+08] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. *High-Level Synthesis*, chapter Chapter 6: AutoPilot: A Platform-Based ESL Synthesis System, pages 99–112. Springer, 2008.

[Zha07] Xin David Zhang. A Streaming Computation Framework for the Cell Processor. M.Eng. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, August 2007.

[Zim97] Karl-Heinz Zimmermann. A Unifying Lattice-Based Approach for the Partitioning of Systolic Arrays via LPGS and LSGP. *Journal of VLSI Signal Processing Systems*, 17(1):21–41, 1997.

[ZLAV03] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. Hierarchical Clustered Register File Organization for VLIW Processors. page 77, Nice, France, April 2003.

[ZLSL05] Dan Zhang, Zeng-Zhi Li, Hong Song, and Long Liu. A Programming Model for an Embedded Media Processing Architecture. In Timo D. Hämäläinen, Andy D. Pimentel, Jarmo Takala, and Stamatis Vassiliadis, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation, 5th International Workshop, SAMOS, Proceedings*, volume 3553 of *Lecture Notes in Computer Science (LNCS)*, pages 251–261, Island of Samos, Greece, July 2005.

[ZSKD03] Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, and Ed F. Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *Proceedings of the 13th International Conference on Field Programmable*

Logic and Applications (FPL), volume 2778 of *Lecture Notes in Computer Science (LNCS)*, pages 911–920, Lisbon, Portugal, September 2003.

Acronyms

ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AXT	AND-XOR-Tree
BIP	Binary Integer Programming
BRAM	Block Random Access Memory
CPU	Central Processing Unit
CSP	Communicating Sequential Processes
DCT	Discrete Cosine Transform
DPLA	Dynamic Piecewise Linear Algorithm
DPRA	Dynamic Piecewise Regular Algorithm
DSP	Digital Signal Processor / Digital Signal Processing
EDIF	Electronic Design Interchange Format
EPIC	Explicitly Parallel Instruction Computing
ESL	Electronic System-Level
EWDF	Elliptical Wave Digital Filter
FF	Flip-Flop
FIFO	First In, First Out
FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
FSR	Feedback Shift Register
GMP	GNU Multi-Precision library
GPU	Graphics Processing Unit
GS	Globally Sequential
HD DVD	High-Definition Digital Versatile Disc
HDL	Hardware Description Language

HDTV	High-Definition TeleVision
HLS	High-Level Synthesis
HNF	Hermite Normal Form
HPF	High Performance Fortran
IIR	Infinite Impulse Response
ILP	Integer Linear Program / Integer Linear Programming
I/O	Input/Output
JPEG	Joint Photographic Experts Group
LBL	Linearly Bounded Lattice
LP	Linear Program / Linear Programming
LPC	Linear Predictive Coding
LPGS	Locally Parallel, Globally Sequential
LS	Locally Sequential
LSGP	Locally Sequential, Globally Parallel
LUT	Look-Up Table
MILP	Mixed Integer Linear Program / Mixed Integer Linear Programming
MIP	Mixed Integer Program / Mixed Integer Programming
MPI	Message Passing Interface
MPEG	Moving Picture Experts Group
MPSoC	MultiProcessor System-on-a-Chip
PDE	Partial Differential Equation
PE	Processor Element
PIP	Parametric Integer Programming
PLA	Piecewise Linear Algorithm
PRA	Piecewise Regular Algorithm
PRAM	Parallel Random Access Machine
RAM	Random Access Memory
RDG	Reduced Dependence Graph
RIA	Regular Iterative Algorithm
RTL	Register Transfer Level
SAC	Single Assignment Code
SARE	System of Affine Recurrence Equations
SDF	Synchronous Data Flow

SIMD	Single Instruction, Multiple Data
SoC	System-on-a-Chip
SPMD	Single-Program, Multiple-Data
SSA	Static Single Assignment form
SURE	System of Uniform Recurrence Equations
URE	Uniform Recurrence Equation
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLIW	Very Long Instruction Word
WPPA	Weakly-Programmable Processor Arrays
WPPE	Weakly-Programmable Processor Element

Curriculum Vitae

Frank Hannig received his general qualification for university entrance in 1993, followed by 15 months of civilian service. Afterwards, he started his studies in an interdisciplinary course of study in Electrical Engineering and Computer Science at the University of Paderborn, Germany. Besides his studies, he gained industrial research experience in a five month internship in primary development at the Electrolux GPDH Tech-Centre in Fredericia, Denmark (1999), and as working student at the C-LAB (Cooperative Computing & Communication Laboratory) in Paderborn, Germany (1997–2000). After receiving the diploma degree in 2000, he joined as research assistant and Ph.D. student the Chair of Computer Engineering (Prof. Dr.-Ing. Jürgen Teich), Department of Electrical Engineering at the University of Paderborn. Here, he was employed in the DFG Collaborative Research Centre (SFB) 376 "Massive Parallelität". In 2003, Frank Hannig moved to the newly founded Department of Hardware/Software Co-Design at the University of Erlangen-Nuremberg, Germany. Since 2004, he is "Akademischer Rat" (tenured research staff) and leads the Architecture and Compiler Design Group. His main research interests are the design of massively parallel architectures, ranging from dedicated hardware to multi-core architectures, mapping methodologies for domain-specific computing, and architecture/compiler co-design. Frank Hannig is member of IEEE and reviewer for multiple journals and conferences including, amongst others, IEEE SPM, IEEE TVLSI, IEEE TSP, IEEE TCAD, IEEE Design & Test, ASAP, DAC, and DATE. Since 2007, Frank Hannig is a member of the program committees of ERSA (International Conference on Engineering of Reconfigurable Systems and Algorithms) and DASIP (Conference on Design and Architectures for Signal and Image Processing). He is an affiliate member of the European Network of Excellence (NoE) on High Performance and Embedded Architecture and Compilation (HiPEAC).

Author's Own Publications

Contributions in Books and Journals

[8] Dmitrij Kissler, Andreas Strawetz, Frank Hannig, and Jürgen Teich. Power-efficient Reconfiguration Control in Coarse-grained Dynamically Reconfigurable Architectures. *Journal of Low Power Electronics*, 5(1):96–105, 2009.

[7] Hritam Dutta, Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, Jürgen Teich, and Bernard Pottier. A Holistic Approach for Tightly Coupled Reconfigurable Parallel Processors. *Microprocessors and Microsystems*, 33(1):53–62, 2009.

[6] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, and Jürgen Teich. *Processor Description Languages*, chapter 12, MAML: An ADL for Designing Single and Multiprocessor Architectures, pages 295–327. In Systems on Silicon, Morgan Kaufmann, 2008.

[5] Hritam Dutta, Frank Hannig, Holger Ruckdeschel, and Jürgen Teich. Efficient Control Generation for Mapping Nested Loop Programs onto Processor Arrays. *Journal of Systems Architecture*, 53(5–6):300–309, 2007.

[4] Dmitrij Kissler, Frank Hannig, and Jürgen Teich. Schwach programmiert macht stark – Massiv parallele Prozessorfelder. *Design&Elektronik*, (4):34–39, 2007.

[3] Frank Hannig, Hritam Dutta, and Jürgen Teich. Mapping a Class of Dependence Algorithms to Coarse-grained Reconfigurable Arrays: Architectural Parameters and Methodology. *International Journal of Embedded Systems*, 2(1/2): 114–127, 2006.

[2] Frank Hannig and Jürgen Teich. *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, chapter 6, Energy Estimation and Optimization for Piecewise Regular Processor Arrays, pages 107–126. Number 20

in Signal Processing and Communications, Marcel Dekker, New York, USA, 2004.

[1] Marcus Bednara, Frank Hannig, and Jürgen Teich. *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation – SAMOS*, chapter Generation of Distributed Loop Control, pages 154–170. Volume 2268 of Lecture Notes in Computer Science (LNCS), Springer, 2002.

Conference, Symposia, and Workshop Proceedings

[49] Amouri Abdulazim, Farhadur Arifin, Frank Hannig, and Jürgen Teich. FPGA Implementation of an Invasive Computing Architecture. To appear in *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT)*, Sydney, Australia, December 9–11, 2009. IEEE.

[48] Farhadur Arifin, Richard Membarth, Amouri Abdulazim, Frank Hannig, and Jürgen Teich. FSM-Controlled Architectures for Linear Invasion. To appear in *Proceedings of the 17th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Florianópolis, Brazil, October 12–14, 2009. IEEE.

[47] Vahid Lari, Frank Hannig, and Jürgen Teich. System Integration of Tightly-Coupled Reconfigurable Processor Arrays and Evaluation of Buffer Size Effects on Their Performance. In *Proceedings of the 4th International Symposium on Embedded Multicore Systems-on-Chip (MCSoC)*, pages 528–534, Vienna, Austria, September 22–25, 2009. IEEE Computer Society.

[46] Richard Membarth, Frank Hannig, Hritam Dutta, and Jürgen Teich. Efficient Mapping of Multiresolution Image Filtering Algorithms on Graphics Processors. In *Proceedings of the 9th International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, Island of Samos, Greece, July 20–23, 2009, volume 5657 of Lecture Notes in Computer Science (LNCS), pages 277–288, 2009. Springer.

[45] Richard Membarth, Frank Hannig, Hritam Dutta, and Jürgen Teich. Optimization Flow for Algorithm Mapping on Graphics Cards. In *Proceedings of ACACES 2009 Poster Abstracts: Advanced Computer Architecture and Compilation for Embedded Systems*, pages 229–232, Terrassa, Spain, July 12–18, 2009. Academia Press, Ghent.

[44] Hritam Dutta, Jiali Zhai, Frank Hannig, and Jürgen Teich. Impact of Loop Tiling on the Controller Logic of Hardware Acceleration Engines. In *Proceedings of the 20th IEEE International Conference on Application-specific Systems,*

Architectures, and Processors (ASAP), pages 161–168, Boston, MA, USA, July 7–9, 2009. IEEE Computer Society.

[43] Richard Membarth, Philipp Kutzer, Hritam Dutta, Frank Hannig, and Jürgen Teich. Acceleration of Multiresolution Imaging Algorithms: A Comparative Study. In *Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 211–214, Boston, MA, USA, July 7–9, 2009. IEEE Computer Society.

[42] Joachim Keinert, Hritam Dutta, Frank Hannig, Christian Haubelt, and Jürgen Teich. Model-Based Synthesis and Optimization of Static Multi-Rate Image Processing Algorithms. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 135–140, Nice, France, April 20–24, 2009. IEEE Computer Society.

[41] Frank Hannig, Hritam Dutta, and Jürgen Teich. Parallelization Approaches for Hardware Accelerators – Loop Unrolling versus Loop Partitioning. In *Proceedings of the 22nd International Conference on Architecture of Computing Systems (ARCS)*, Delft, The Netherlands, March 10–13, 2009, volume 5455 of Lecture Notes in Computer Science (LNCS), pages 16–27, 2009. Springer.

[40] Hritam Dutta, Frank Hannig, and Jürgen Teich. Performance Matching of Hardware Acceleration Engines for Heterogeneous MPSoC using Modular Performance Analysis. In *Proceedings of the 22nd International Conference on Architecture of Computing Systems (ARCS)*, Delft, The Netherlands, March 10–13, 2009, volume 5455 of Lecture Notes in Computer Science (LNCS), pages 233–245, 2009. Springer.

[39] Dmitrij Kissler, Andreas Strawetz, Frank Hannig, and Jürgen Teich. Power-efficient Reconfiguration Control in Coarse-Grained Dynamically Reconfigurable Architectures. In *Proceedings of the 18th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Lisbon, Portugal, September 10–12, 2008, volume 5349 of Lecture Notes in Computer Science (LNCS), pages 307–317, 2008. Springer.

[38] Christophe Wolinski, Krzysztof Kuchcinski, Jürgen Teich, and Frank Hannig. Area and Reconfiguration Time Minimization of the Communication Network in Regular 2D Reconfigurable Architectures. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 391–396, Heidelberg, Germany, September 8–10, 2008. IEEE.

[37] Sven Eisenhardt, Thomas Schweizer, Julio A. de Oliveira Filho, Tobias Oppold, Wolfgang Rosenstiel, Alexander Thomas, Jürgen Becker, Frank Hannig,

Dmitrij Kissler, Hritam Dutta, Jürgen Teich, Heiko Hinkelmann, Peter Zipf, and Manfred Glesner. SPP1148 Booth: Coarse-Grained Reconfiguration. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, page 349, Heidelberg, Germany, September 8–10, 2008. IEEE.

[36] Christophe Wolinski, Krzysztof Kuchcinski, Jürgen Teich, and Frank Hannig. Communication Network Reconfiguration Overhead Optimization in Programmable Processor Array Architectures. In *Proceedings of the 11th Euromicro Conference on Digital System Design (DSD)*, pages 345–352, Parma, Italy, September 3–5, 2008. IEEE.

[35] Rainer Schaffer, Renate Merker, Frank Hannig, and Jürgen Teich. Utilization of all Levels of Parallelism in a Processor Array with Subword Parallelism. In *Proceedings of the 11th Euromicro Conference on Digital System Design (DSD)*, pages 391–398, Parma, Italy, September 3–5, 2008. IEEE.

[34] Hritam Dutta, Frank Hannig, and Jürgen Teich. PARO: A Design Tool for Automatic Generation of Hardware Accelerators. In *Proceedings of ACACES 2008 Poster Abstracts: Advanced Computer Architecture and Compilation for Embedded Systems*, pages 317–320, L'Aquila, Italy, July 13–19, 2008. Academia Press, Ghent.

[33] Christophe Wolinski, Krzysztof Kuchcinski, Jürgen Teich, and Frank Hannig. Optimization of Routing and Reconfiguration Overhead in Programmable Processor Array Architectures. In *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 306–309, Palo Alto, CA, USA, April 14–15, 2008. IEEE Computer Society.

[32] Frank Hannig, Holger Ruckdeschel, Hritam Dutta, and Jürgen Teich. PARO: Synthesis of Hardware Accelerators for Multi-Dimensional Dataflow-Intensive Applications. In *Proceedings of the Fourth International Workshop on Applied Reconfigurable Computing (ARC)*, London, United Kingdom, March 26–28, 2008, volume 4943 of Lecture Notes in Computer Science (LNCS), pages 287–293, 2008. Springer.

[31] Frank Hannig, Holger Ruckdeschel, and Jürgen Teich. The PAULA Language for Designing Multi-Dimensional Dataflow-Intensive Applications. In *Proceedings of the GI/ITG/GMM-Workshop – Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 129–138, Freiburg, Germany, March 3–5, 2008. Shaker.

[30] Frank Hannig, Hritam Dutta, Holger Ruckdeschel, and Jürgen Teich. Quantitative Evaluation of Behavioral Synthesis Approaches for Reconfigurable Devices. In *Proceedings of the 2nd HiPEAC Workshop on Reconfigurable Computing (WRC)*, pages 73–82, Gothenburg, Sweden, January 27, 2008.

[29] Jürgen Teich, Frank Hannig, Holger Ruckdeschel, Hritam Dutta, Dmitrij Kissler, and Andrej Stravet. A Unified Retargetable Design Methodology for Dedicated and Re-Programmable Multiprocessor Arrays: Case Study and Quantitative Evaluation. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Invited paper, pages 14–24, Las Vegas, NV, USA, June 25–28, 2007. CSREA Press.

[28] Hritam Dutta, Frank Hannig, Alexey Kupriyanov, Dmitrij Kissler, Jürgen Teich, Rainer Schaffer, Sebastian Siegel, Renate Merker, and Bernard Pottier. Massively Parallel Processor Architectures: A Co-design Approach. In *Proceedings of the 3rd International Workshop on Reconfigurable Communication Centric System-on-Chips (ReCoSoC)*, pages 61–68, Montpellier, France, June 18–20, 2007. Univ. Montpellier II.

[27] Alexey Kupriyanov, Dmitrij Kissler, Frank Hannig, and Jürgen Teich. Efficient Event-driven Simulation of Parallel Processor Architectures. In *Proceedings of the 10th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 71–80, Nice, France, April 20, 2007. ACM Press.

[26] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, Jürgen Teich, Julien Lallet, Olivier Sentieys, and Sébastien Pillement. Modeling of Interconnection Networks in Massively Parallel Processor Architectures. In Paul Lukowicz, Lothar Thiele, and Gerhard Tröster, editors, *Proceedings of the 20th International Conference on Architecture of Computing Systems (ARCS)*, Zurich, Switzerland, March 12–15, 2007, In Lecture Notes in Computer Science (LNCS), pages 268–282, 2007. Springer.

[25] Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, and Jürgen Teich. A Highly Parameterizable Parallel Processor Array Architecture. In *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT)*, pages 105–112, Bangkok, Thailand, December 13–15, 2006. IEEE.

[24] Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, and Jürgen Teich. Hardware Cost Analysis for Weakly Programmable Processor Arrays. In *Proceedings of the International Symposium on System-on-Chip (SoC)*, pages 179–182, Tampere, Finland, November 14–16, 2006. IEEE.

[23] Sebastian Siegel, Renate Merker, Frank Hannig, and Jürgen Teich. Communication-conscious Mapping of Regular Nested Loop Programs onto Massively Parallel Processor Arrays. In *Proceedings of the 18th International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 71–76, Dallas, TX, USA, November 13–15, 2006. ACTA Press.

[22] Hritam Dutta, Frank Hannig, and Jürgen Teich. Hierarchical Partitioning for Piecewise Linear Algorithms. In *Proceedings of the 5th International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, pages 153–160, Bialystok, Poland, September 13–17, 2006. IEEE Computer Society.

[21] Hritam Dutta, Frank Hannig, Jürgen Teich, Benno Heigl, and Heinz Hornegger. A Design Methodology for Hardware Acceleration of Adaptive Filter Algorithms in Image Processing. In *Proceedings of the 17th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 331–337, Steamboat Springs, CO, USA, September 11–13, 2006. IEEE Computer Society.

[20] Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, and Jürgen Teich. A Dynamically Reconfigurable Weakly Programmable Processor Array Architecture Template. In *Proceedings of the 2nd International Workshop on Reconfigurable Communication Centric System-on-Chips (ReCoSoC)*, pages 31–37, Montpellier, France, July 3–5, 2006. Univ. Montpellier II.

[19] Dmitrij Kissler, Alexey Kupriyanov, Frank Hannig, Dirk Koch, and Jürgen Teich. A Generic Framework for Rapid Prototyping of System-on-Chip Designs. In *Proceedings of International Conference on Computer Design (CDES)*, pages 189–195, Las Vegas, NV, USA, June 26–29, 2006. CSREA Press.

[18] Hritam Dutta, Frank Hannig, and Jürgen Teich. Controller Synthesis for Mapping Partitioned Programs on Array Architectures. In Werner Grass, Bernhard Sick, and Klaus Waldschmidt, editors, *Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS)*, Frankfurt am Main, Germany, March 13–16, 2006, volume 3894 of Lecture Notes in Computer Science (LNCS), pages 176–191, 2006. Springer.

[17] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, Jürgen Teich, Rainer Schaffer, and Renate Merker. An Architecture Description Language for Massively Parallel Processor Architectures. In *Proceedings of the GIITGGMM-Workshop – Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 11–20, Dresden, Germany, February, 2006. Shaker.

[16] Hritam Dutta, Frank Hannig, and Jürgen Teich. Mapping of Nested Loop Programs onto Massively Parallel Processor Arrays with Memory and I/O Constraints. In Friedhelm Meyer auf der Heide and Burkhard Monien, editors, *Proceedings of the 6th International Heinz Nixdorf Symposium, New Trends in Parallel & Distributed Computing*, Paderborn, Germany, January 17–18, 2006, volume 181 of HNI-Verlagsschriftenreihe, pages 97–119, 2006. Heinz Nixdorf Institut, Universität Paderborn.

[15] Thomas Schlichter, Christian Haubelt, Frank Hannig, and Jürgen Teich. Using Symbolic Feasibility Tests during Design Space Exploration of Heterogeneous Multi-Processor Systems. In *Proceedings of the 16th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 9–14, Island of Samos, Greece, July 23–25, 2005. IEEE Computer Society.

[14] Holger Ruckdeschel, Hritam Dutta, Frank Hannig, and Jürgen Teich. Automatic FIR Filter Generation for FPGAs. In Timo D. Hämäläinen, Andy D. Pimentel, Jarmo Takala, and Stamatis Vassiliadis, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation, 5th International Workshop, SAMOS, Proceedings*, Island of Samos, Greece, July 18–20, 2005, volume 3553 of Lecture Notes in Computer Science (LNCS), pages 51–61, 2005. Springer.

[13] Frank Hannig, Hritam Dutta, Alexey Kupriyanov, Jürgen Teich, Rainer Schaffer, Sebastian Siegel, Renate Merker, Ronan Keryell, Bernard Pottier, Daniel Chillet, Daniel Ménard, and Olivier Sentieys. Co-Design of Massively Parallel Embedded Processor Architectures. In *Proceedings of the first International Workshop on Reconfigurable Communication Centric System-on-Chips (ReCoSoC)*, pages 27–34, Montpellier, France, June 27–29, 2005. Univ. Montpellier II.

[12] Frank Hannig and Jürgen Teich. Output Serialization for FPGA-based and Coarse-grained Processor Arrays. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 78–84, Las Vegas, NV, USA, June 27–30, 2005. CSREA Press.

[11] Jan van der Veen, Sándor Fekete, Mateusz Majer, Ali Ahmadinia, Christophe Bobda, Frank Hannig, and Jürgen Teich. Defragmenting the Module Layout of a Partially Reconfigurable Device. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 92–101, Las Vegas, NV, USA, June 27–30, 2005. CSREA Press.

[10] Frank Hannig and Jürgen Teich. Resource Constrained and Speculative Scheduling of an Algorithm Class with Run-Time Dependent Conditionals. In *Proceedings of the 15th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 17–27, Galveston, TX, USA, September 27–29, 2004. IEEE Computer Society.

[9] Alexey Kupriyanov, Frank Hannig, and Jürgen Teich. Automatic and Optimized Generation of Compiled High-Speed RTL Simulators. In *Proceedings of Workshop on Compilers and Tools for Constrained Embedded Systems (CTCES)*, Washington, DC, USA, September 22, 2004.

[8] Frank Hannig and Jürgen Teich. Dynamic Piecewise Linear/Regular Algorithms. In *Proceedings of the Fourth International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, pages 79–84, Dresden, Germany, September 7–10, 2004. IEEE Computer Society.

[7] Alexey Kupriyanov, Frank Hannig, and Jürgen Teich. High-Speed Event-Driven RTL Compiled Simulation. In Andy D. Pimentel and Stamatis Vassiliadis, editors, *Computer Systems: Architectures, Modeling, and Simulation, 4th International Samos Workshop (SAMOS), Proceedings*, Island of Samos, Greece, July 19–21, 2004, volume 3133 of Lecture Notes in Computer Science (LNCS), pages 519–529, 2004. Springer.

[6] Frank Hannig, Hritam Dutta, and Jürgen Teich. Regular Mapping for Coarse-grained Reconfigurable Architectures. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume V, pages 57–60, Montréal, Quebec, Canada, May 17–21, 2004. IEEE Signal Processing Society.

[5] Frank Hannig, Hritam Dutta, and Jürgen Teich. Mapping of Regular Nested Loop Programs to Coarse-grained Reconfigurable Arrays – Constraints and Methodology. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, NM, USA, April 26–30, 2004. IEEE Computer Society.

[4] Frank Hannig and Jürgen Teich. Energy Estimation of Nested Loop Programs. In *Proceedings 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 149–150, Winnipeg, Manitoba, Canada, August 10–13, 2002. ACM Press.

[3] Frank Hannig and Jürgen Teich. Energy Estimation for Piecewise Regular Processor Arrays. In *Proceedings of the Second International Samos Workshop on*

Systems, Architectures, Modeling, and Simulation (SAMOS), Island of Samos, Greece, July 22–25, 2002.

[2] Marcus Bednara, Frank Hannig, and Jürgen Teich. Boundary Control: A new Distributed Control Architecture for Space-Time Transformed (VLSI) Processor Arrays. In *Proceedings 35th IEEE Asilomar Conference on Signals, Systems, and Computers*, volume 2, pages 468–474, Pacific Grove, CA, USA, November 4–7, 2001. IEEE Computer Society.

[1] Frank Hannig and Jürgen Teich. Design Space Exploration for Massively Parallel Processor Arrays. In Victor Malyshkin, editor, *Parallel Computing Technologies, 6th International Conference, PaCT 2001, Proceedings*, Novosibirsk, Russia, September 3–7, 2001, volume 2127 of Lecture Notes in Computer Science (LNCS), pages 51–65, 2001. Springer.

Technical Reports

[5] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, Jürgen Teich, Julien Lallet, Olivier Sentieys, and Sébastien Pillement. Modeling of Interconnection Networks in Massively Parallel Processor Architectures. Technical Report 05–2006, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, 91058 Erlangen, Germany, August, 2006.

[4] Hritam Dutta, Frank Hannig, and Jürgen Teich. A Formal Methodology for Hierarchical Partitioning of Piecewise Linear Algorithms. Technical Report 04–2006, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, 91058 Erlangen, Germany, April, 2006.

[3] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, Rainer Schaffer, and Jürgen Teich. MAML – An Architecture Description Language for Modeling and Simulation of Processor Array Architectures, Part I. Technical Report 03–2006, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, 91058 Erlangen, Germany, March, 2006.

[2] Hritam Dutta, Frank Hannig, and Jürgen Teich. Controller Synthesis for Mapping Partitioned Programs on Array Architectures. Technical Report 03–2005, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, 91058 Erlangen, Germany, November, 2005.

[1] Frank Hannig and Jürgen Teich. Resource Constrained and Speculative Scheduling of Dynamic Piecewise Regular Algorithms. Technical Report 01–2004, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, 91058 Erlangen, Germany, June, 2004.

Talks and Others

[10] Frank Hannig, Hritam Dutta, and Jürgen Teich. PARO – A Design Tool for the Automatic Generation of Hardware Accelerators. Tool Presentation at the Demo Night of the 20th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP), July 7–9, 2009.

[9] Frank Hannig. Power-Efficient Design of Tightly Coupled Parallel Processors with Dynamic Reconfiguration Capabilities. Talk, HiPEAC Cluster Meeting, Paris, France, November 27–28, 2008.

[8] Hritam Dutta, Frank Hannig, and Jürgen Teich. The PARO Design Tool for Automatic Generation of Hardware Accelerators. Interactive Presentation at Friday Workshop, The New Wave of the High-Level Synthesis, Design, Automation and Test in Europe (DATE), Munich, Germany, March 10–14, 2008.

[7] Jürgen Teich, Frank Hannig, Hritam Dutta, Dmitrij Kissler, and Matthias Hartl. Domain-specific Reconfigurable MPSoC-Systems – Challenges and Trends. Talk at Friday Workshop, Reconfigurable Hardware: Emerging Trade-Offs through Granularity, Heterogeneity and Mixed-Signal Capability in Actual and Future Architectures, Design, Automation and Test in Europe (DATE), Munich, Germany, March 10–14, 2008.

[6] Dmitrij Kissler, Hritam Dutta, Alexey Kupriyanov, Frank Hannig, and Jürgen Teich. A High-Speed Dynamic Reconfigurable Multilevel Parallel Architecture. Hardware and Software Demo at the University Booth at Design, Automation and Test in Europe (DATE), Munich, Germany, March 10–14, 2008.

[5] Frank Hannig. Reconfigurable Computing Activities at University of Erlangen-Nuremberg, Hardware/Software Co-Design. Talk, HiPEAC Cluster Meeting, Cambridge, United Kingdom, November 26–28 , 2007.

[4] Frank Hannig. A Unified Retargetable Design Methodology for Dedicated and Re-Programmable Multiprocessor Arrays – Case Study and Quantitative

Evaluation. Talk, Dagstuhl Seminar No. 07361, Programming Models for Ubiquitous Parallelism, Wadern, Germany, September 2–7, 2007.

[3] Frank Hannig. Architektur und Compiler Co-Design. Talk, DFG SPP 1148 Workshop, Blaubeuren, Germany, October 4, 2004.

[2] Alexey Kupriyanov, Frank Hannig, Jürgen Teich, Dirk Fischer, Michael Thies, and Ralph Weper. ArchitectureComposer. CAD Software Demo at the University Booth at Design, Automation and Test in Europe (DATE), Paris, France, February 16–20, 2004.

[1] Frank Hannig. Mapping of Regular Algorithms to Massively Parallel Architectures. Invited talk, Department of System Simulation, University Erlangen-Nuremberg, Germany, February 12, 2004.