Estimation and Optimization of the Performance of Polyhedral Process Networks

Haastregt, S. van

Estimation and Optimization of the Performance of Polyhedral Process Networks

Sven van Haastregt

Estimation and Optimization of the Performance of Polyhedral Process Networks

Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof.mr. C.J.J.M. Stolker,
volgens besluit van het College voor Promoties
te verdedigen op dinsdag 17 december 2013
klokke 12:30 uur

door

Sven van Haastregt
geboren te Rijpwetering
in 1985

Samenstelling promotiecommissie:

promotor	Prof.dr. Ed Deprettere	
co-promotor	Dr. Bart Kienhuis	
overige leden:	Prof.dr. Joost Kok	
	Prof.dr. Harry Wijshoff	
	Prof.dr. Koen Bertels	Technische Universiteit Delft
	Dr. Hristo Nikolov	

This manuscript was edited by the author using Vim, and typeset using LaTeX 2$_\varepsilon$, BibTeX, and *MakeIndex* in a process automated using GNU Make. Graphics were produced mostly using Inkscape, and occasionally using Xfig or gnuplot. Git over ssh was used for revision tracking, synchronization, and backup purposes.

Cover design by Marcel IJssennagger.

Estimation and Optimization of the Performance of Polyhedral Process Networks
Sven van Haastregt. -
Thesis Universiteit Leiden. - With index, ref. - With summary in Dutch
190 pages, 47988 words, 176 index entries, 162 references.
ISBN 978-94-6182-383-0

CONTENTS

NOTATION

$\lvert \cdot \rvert$	Cardinality: $\lvert\mathcal{S}\rvert \equiv$ the number of elements in $\mathcal{S}$, page 20.
$[\cdot, \cdot)$	Interval: $[a, b) = \{x \in \mathbb{Z} \mid a \leq x < b\}$.
$\lceil \cdot \rceil$	Least integer: $\lceil x \rceil = n \Leftrightarrow n \in \mathbb{N} \wedge n - 1 < x \leq n$.
$\cdot \prec \cdot$	Lexicographical order, page 18.
D_p	Iteration domain of process p, page 27.
$d(e)$	Number of initial tokens on edge e, page 22.
δ_c	Process reading from channel c, page 27.
$\mathcal{E}$	The set of channels of a PPN, page 27.
II_F	Initiation interval of function F, page 38.
IPD_i^k	k-th Input Port Domain of process i, page 27.
Λ_F	Latency (input-to-output delay) of function F, page 38.
M_c	Channel relation of channel c, page 27.
$\mathbb{N}$	The set of natural numbers, including 0.
$\mathbb{N}^+$	The set of positive natural numbers, excluding 0.
OPD_i^k	k-th Output Port Domain of process i, page 27.
$\mathcal{P}$	The set of processes of a PPN, page 27.
$\mathbb{Q}$	The set of rational numbers.
σ_c	Process writing to channel c, page 27.
T_p	Period of a process p, page 55.
$t(a)$	Execution time of data flow node a, page 22.
τ_p	Throughput of a process p, page 55.
$\theta(\mathbf{i})$	Application of schedule θ to iteration vector $\mathbf{i}$, pages 111, 112.
$\mathbb{Z}$	The set of integers.

CHAPTER 1

INTRODUCTION

ELEMENT NUMBER 14, or silicon, has been important for many ancient civilizations, albeit mostly as a constituent of sand and rocks. Silicon was essential for the construction of houses, temples, and roads, which together formed the centers of society. In 1954, a new and very different use for silicon was found that would have a dramatic impact on the established centers of society: Gordon Teal and his team produced the first silicon transistor [Che04]. Many electronic devices have become available since then, in which silicon transistors are an essential component. By miniaturization, more and more transistors could be fit onto a small area, thereby enabling the construction of complex processing systems. Contemporary examples of such processing systems include the special purpose processors found in automotive, mobile communications, medical, industrial, and entertainment application domains. Many of these processing systems are tightly coupled to their environment and perform a specific task, and are therefore classified as *embedded systems* [LS11, Mar11, SB00]. Central to this dissertation is the design of the special purpose processors in these embedded systems.

1.1 Problem Context

The special purpose processors in embedded systems are highly optimized to perform their application-specific computations in a fast and area- and energy-efficient way. The design of those processors is becoming increasingly challenging due to increasing application complexity, the ever-increasing demand for computational power, and worldwide time-to-market pressure. To satisfy the demand for computational power, *Multi-Processor System-on-Chip* (*MPSoC*) solutions are deployed in modern

embedded systems. Such MPSoCs consist of many different components such as programmable processing components, specialized processing components, memory components, and input/output interfaces. By letting multiple components work in parallel, the demand for computational power is met. Unfortunately, the design of an MPSoC is even more challenging than the design of a single-processor system. The challenge for the designer is to distribute computations over different processors of the MPSoC. While doing so, the designer should guarantee functional correctness of the system and at the same time make tradeoffs between orthogonal design aspects such as circuit area and performance [Mar06]. Thus, the shift to multi-processor systems may address the demand for computational power, but this comes at the expense of a further increase in design complexity.

Traditionally, processors have been designed at the *Register Transfer Level* (*RTL*). An RTL specification of a processor consists of registers that are interconnected by signals and combinational logic. RTL design of modern MPSoCs is becoming increasingly error-prone and time-consuming because of the abundance of registers, signals, and combinational logic needed for a modern MPSoC's functionality. To cope with the design complexity of modern MPSoCs, the designer needs to work at a level of abstraction above the RTL. This has led to the emergence of *Electronic System-Level* (*ESL*) design methodologies [GAGS09, BM10]. In such a design methodology, the designer first specifies a system at a high level of abstraction. Next, the designer constructs an RTL implementation from the initial specification with the aid of system-level design automation tools.

An example system-level design tool set is the open-source Daedalus tool set which has been developed at the Leiden Embedded Research Center (LERC) [NSD08b, Lei08]. We leverage the Daedalus tool set in this thesis. This means that we want to develop the special purpose processors of an embedded system with Daedalus. An overview of the Daedalus system-level tool flow is depicted in Figure 1.1. Daedalus enables a designer to obtain a deployable gate-level specification from a system-level specification in a fully automated way. The functional behavior of the system-level specification is specified as a sequential C program, as shown at the upper right part of Figure 1.1. The elaboration from one specification level to a lower specification level is done in a fully automated way. We discuss the different aspects of Daedalus in the following paragraphs.

Many applications in the embedded systems domain are specified using an imperative model of computation, in for example the C language. Such models are well-suited and widely adopted to specify the functionality of single-processor systems. Unfortunately, mapping an imperative specification onto a multi-processor system is difficult because of two mismatches. First, the sequential nature of an imperative specification does not match the parallel nature of a multi-processor system. Sec-

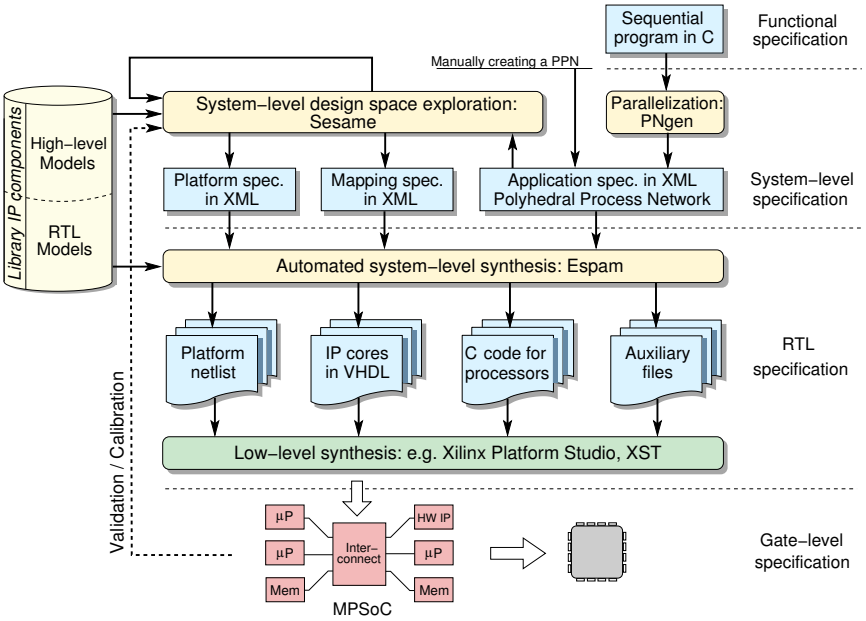


Figure 1.1: Daedalus system-level tool flow overview [NSD08b].

ond, an imperative specification assumes shared memory which is likely to become a performance bottleneck on a multi-processor system. A distributed memory model better matches a multi-processor system, but it is not possible in the general case to extract a distributed memory model from an imperative specification.

The functionality of a multi-processor system is more naturally specified using a parallel model of computation such as a network of processes communicating over channels. A model that has gained widespread popularity is the Kahn Process Network model [Kah74]. Specifying the functionality of a system using a parallel model of computation is considered more difficult compared to using an imperative model of computation. This is because the human brain tends to solve problems as a sequence of steps, which matches the sequential nature of an imperative model of computation. Moreover, in a parallel specification deadlocks and race conditions may occur that are very difficult to detect or predict beforehand [Mar06]. Such difficulties do not occur in a sequential specification. As such, many designers prefer specifying an application using a sequential specification, despite the subsequent difficulties of implementing the specification as an MPSoC. The mismatch between the programmer-preferred sequential specifications and the parallel specifications desired for multi-processor systems is known as the *specification gap* [Ste04].

Various approaches exist to bridge the specification gap. One approach is to extend a sequential program with library calls or compiler pragma directives to indicate tasks that can execute concurrently. Examples of this approach include pthreads, OpenCL [Khr08], and OpenMP [Ope97]. Another approach is to automatically extract concurrent tasks from a sequential program using a parallelizing compiler such as LooPo [GL97], Polaris [BEF+94], Pluto [BBK+08], or PNGEN [VNS07]. The latter is part of Daedalus to bridge the specification gap. PNGEN generates a parallel specification from a sequential program written in a subset of the C language. We discuss PNGEN in more detail in Section 2.3.

A system-level specification lacks many details that are present in the RTL specification because these details are irrelevant at the system level. For example, at the system level the designer reasons about sending data from one processor to another without specifying the registers and logic that implement such communication in the RTL specification. Not exposing the designer to such implementation details allows a designer to better cope with complex systems. However, the omission of implementation details opens up a gap between the system-level specification and the RTL implementation, which is known as the *implementation gap* [NSD08b]. To obtain a functional implementation from a system-level specification, the implementation gap needs to be bridged by adding low-level implementation details to the system-level specification. This is done by a system-level synthesis tool which refines a system-level specification into an RTL specification in a systematic and automated way.

The Daedalus tool set provides the ESPAM tool for automated system-level synthesis. A system-level specification for ESPAM is composed of three individual specifications: an implementation platform specification describing the number and types of processing and interconnect components of the system; a parallel application specification consisting of a network of communicating tasks; and a mapping specification that maps the application tasks onto processing components. The ESPAM tool generates an RTL specification from the three specifications. This RTL specification is then taken through commercial *low-level synthesis* tools that convert the RTL into a gate-level specification. Place-and-route tools take such a gate-level specification and create a layout of the circuit which can be implemented on a *Field-Programmable Gate Array* (*FPGA*) or provided to an *Application-Specific Integrated Circuit* (*ASIC*) manufacturing process. This last step yields a complete MPSoC implementation.

1.2 Problem Statement

Existing system-level design tools such as Daedalus present a forward synthesis flow to bridge the specification and implementation gaps. This allows a designer to obtain

a working prototype of a system in only a few hours of time [NSD08b]. However, many different implementations of an application specification are possible that have identical functionality but differ in performance and implementation cost aspects. This presents the designer with another problem: selecting an implementation from a vast *design space* of possible implementations. Only a subset of the *design points* in this design space represent implementations that satisfy a set of given *design constraints* on performance and circuit area. Thus, solely closing the specification and implementation gaps still leaves open the problem of selecting the design point that best matches a set of design constraints.

A Daedalus system-level specification consists of the application, platform, and mapping subspecifications, as described in the previous section and shown in Figure 1.1. Each of these subspecifications may be transformed to yield a functionally equivalent implementation that has different performance and resource cost properties, as described by the Y-chart approach [KDWV02]. For example, a designer can transfrom the platform specification by adding or removing processors, or transform the mapping specification by moving a task from one processor to another, or transform the application specification by splitting a tasks into smaller subtasks and thereby exposing more parallelism. Many combinations of such *transformations* are possible and this number grows rapidly as application and platform sizes increase. As a result, the design space for a modern MPSoC is typically very large.

Despite the existence of fully automated system-level synthesis tools, implementing and evaluating all design points is infeasible for modern MPSoC design because of the large design space. Therefore, the design space should be explored in such a way that only the "promising" design points need to be implemented and evaluated. Finding the promising design points is a non-trivial multi-objective optimization problem. Many *Design Space Exploration* (*DSE*) techniques have been proposed to efficiently search large design spaces [Gri04]. Daedalus incorporates the SESAME DSE tool to explore the design space using an evolutionary algorithm [PEP06]. SESAME relies on trace-based simulation to estimate the performance of candidate design points. Although SESAME's simulation is intended for fast performance analysis, conducting many simulations may still take a considerable amount of time [PP12]. This leads to unreasonably long design times.

An alternative way of finding a satisfactory design point is the (naive) iterative design flow depicted in Figure 1.2. The design flow starts with an initial system-level specification. A parallel application specification is automatically derived from an imperative program using for example PNGEN, thereby bridging the specification gap. The designer synthesizes this system-level specification into an FPGA prototype to verify if for example performance constraints are satisfied. The designer uses a system-level synthesis tool such as ESPAM in this step, thereby bridging the

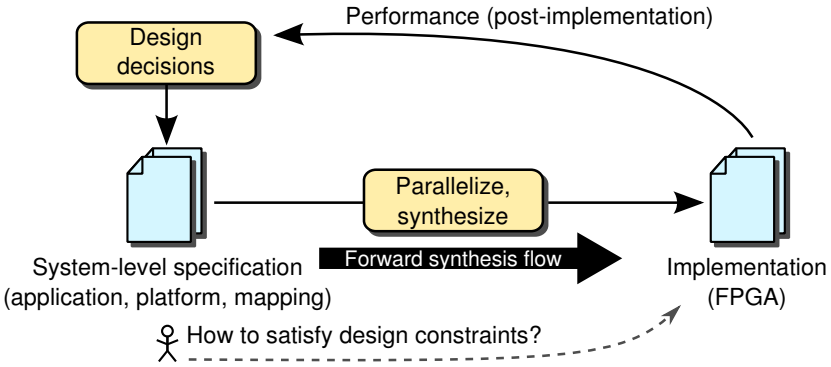


Figure 1.2: Iterative system-level design flow.

implementation gap. If a performance constraint is not satisfied, the system-level specification is transformed based on performance and cost metrics obtained from the prototype implementation. These transformations entail modifying the application by rewriting the C code, modifying the platform by adding processors, or modifying the mapping by assigned tasks to different processors. The designer relies on experience and expertise to come up with transformations that most likely have the desirable effect on performance and cost aspects. Building up this knowledge is referred to as the "acquisition of insight" [Spe97]. However, it is not trivial to predict beforehand if and by how much a certain transformation affects performance and cost aspects. At this moment, the best a designer can do is to perform a new time-consuming synthesis step after transforming the system-level specification. This procedure is repeated until an implementation is obtained that satisfies performance constraints. The designer can then proceed with the actual manufacturing of the system.

A naive iterative design flow may appear to be more deterministic than a random-search driven DSE flow. Because the designer iteratively transforms a system-level specification in a pragmatic manner, a system that satisfies all performance constraints should eventually be the result. However, this only holds if the designer always makes the optimal decisions. This does not always happen in practice, because the designer may for example overlook solutions or ignore solutions that seem counter-intuitive. Another problem with a naive iterative design flow is that evaluation of a single specification may easily take a few hours of time. This reduces the number of iterations a designer can make in a given time frame, increasing time-to-market.

The naive iterative design flow bridges both the specification and implementation gaps by employing advanced parallelizing compilation techniques and system-level

synthesis tools. However, it does not address the following problem: given a performance constraint, which transformations should the designer apply to obtain an implementation that meets this performance constraint? For example, consider the scenario in which a designer constructs a video processing system under the constraint that the system should meet a throughput of 20 frames per second. After synthesizing the system-level specification, the designer finds that the system works at only 11 frames per second. This puts a burden on the designer to transform the system-level specification such that the performance constraint of 20 frames per second is met. We therefore argue that solely bridging the specification and implementation gaps is not sufficient to solve a design problem.

In this dissertation, we consider the iterative system-level design flow of Figure 1.2 and address a designer's problem that is currently not addressed. That is, we ask how to modify this design flow to obtain a constraint-satisfying implementation of a system in a short amount of time. This modification is needed as synthesizing a design in the current flow takes too long, keeping the designer in the dark whether the design will satisfy the designer's constraints. Performance estimation methods are lacking that could provide an early indication of whether a design will satisfy the designer's constraints at all. After obtaining an implementation not meeting the constraints, there is little guidance to help a designer transform his design in such a way that his performance constraints will be satisfied. In this context, we formulate our three central research problems as follows:

1. **Synthesis**: How to automatically obtain efficient RTL implementations from a high-level specification that enable application of established transformations such as splitting, merging, stream multiplexing, and scheduling?
2. **Performance estimation**: How to assess the absolute performance of a design point, possibly in different ways by trading off evaluation speed against accuracy?
3. **Optimization**: How to obtain an implementation that satisfies a performance constraint while reducing the number of design iterations?

Only after addressing these three problems from a designer's perspective, Daedalus can become a powerful system-level synthesis tool capable of solving design problems.

1.3 Related Work

We address the central problems listed above in this dissertation by leveraging and extending the underlying theory of the Daedalus methodology [NSD08b, Lei08]. The

Daedalus methodology addresses the problem of obtaining an efficient FPGA implementation from a high-level application specification in a short amount of time. As such, Daedalus provides an important stepping stone to address the three central problems, ultimately leading to an extended Daedalus design flow that also considers performance constraints.

Daedalus is only one of many methodologies to (semi-)automatically obtain special purpose processor implementations from high-level application and system specifications. In this section, we give a brief overview of related approaches to obtain RTL implementations from high-level application specifications. We discuss related high-level synthesis techniques in Section 1.3.1 and related electronic system-level synthesis techniques in Section 1.3.2. Related work specific to each of the three central problems is discussed separately in Chapters 3, 4, and 5.

1.3.1 High-Level Synthesis

Automated synthesis of RTL implementations from specifications above the register transfer level, known as *High-Level Synthesis* (*HLS*), has been subject of research since the late 1980s [MK88, MPC88, PK89]. Since then, many academic and commercial HLS tools have been developed. In 1994, electronic design automation company Synopsys released its Behavioral Compiler tool that is widely regarded as the first commercial HLS tool [CM08]. This tool took a behavioral description of a design in VHDL or Verilog as input and generated a cycle-accurate VHDL or Verilog description. During synthesis, the tool allowed the designer to trade off throughput against chip area. Since then, many different HLS tools have been released by different companies, with varying degrees of commercial success. As of 2013, three of the major commercially available HLS tools are Synopsis SynphonyC [Syn10], Xilinx Vivado HLS [Xil13], and Calypso Catapult [Cal11]. A difference with Behavioral Compiler is that modern commercial HLS tools have anchored on C, C++, or SystemC input specifications instead of input specifications using Hardware Description Languages like VHDL or Verilog [Fin10]. Meeus et al. conducted a comparison between twelve different commercial and open-source high-level synthesis tools [MVBG+12].

Next to commercial tools, numerous academic high-level synthesis tools have been developed. One of the early academic tools was Hercules [MK88] which has been integrated in the Olympus Synthesis System [MKMT90]. Olympus takes an input specification written in HardwareC and provides synthesis and simulation of designs. HardwareC is a C-like language in which a system is described as a set of concurrent modules. The modules are interconnected using communication primitives. This requires the designer to split the application functionality across different modules and

interconnect them manually using communication primitives. Advanced compilation techniques such as those employed by Daedalus allow tools to automatically derive such interconnects from a sequential specification. The ROCCC tool takes a subset of the C language as input and generates RTL targeted towards FPGAs [GNB08]. ROCCC requires that loop iterators are used in at most one array dimension. This poses a problem when expressing for example a loop skewing transformation. Such a restriction is not necessary in Daedalus as any affine expression of loop iterators is analyzable using the polyhedral model. Other HLS approaches that employ the polyhedral model include for example PARO [HRDT08] and MMALPHA [GQR03]. These approaches use functional languages as input, while commercial tools and Daedalus all use an imperative language. Related early work included modeling affine nested loop programs using uniform recurrence equations to generate systolic array implementations [Qui84]. The FCUDA approach takes C code annotated using NVIDIA's CUDA primitives and generates C code annotated with AutoESL pragmas to obtain an FPGA implementation [PGS+09]. This allows a designer to express parallelism in a single specification and target both GPU and FPGA platforms. Besides the main FPGA backend, Daedalus also includes a GPU backend, allowing a designer to also target both GPU and FPGA platforms. Unlike FCUDA, Daedalus does not require CUDA-like annotations of the C code.

High-level synthesis should not be confused with design entry using a high-level language, because the use of a high-level language does not necessarily imply that the design is specified at a high level of abstraction. For example, Handel-C is a subset of the C language with extensions to describe hardware succinctly [Pag96]. Parallel behavior is expressed using the Communicating Sequential Processes (CSP) model of computation [Hoa85]. Similar to RTL design, the designer should perform scheduling and pipelining manually, whereas this is performed automatically in an HLS flow. Cobble is a language similar to Handel-C [TCL05] with support for custom compilation schemes. This allows a designer to define how a particular pattern in the source program should be mapped to hardware. Cobble is compiled into Pebble, which is a simplified hardware description language supporting design parameterization and run-time reconfiguration [LM98]. MyHDL allows a designer to specify hardware in the Python language [Dec03]. MyHDL still requires the designer to specify the behavior of the hardware at the register-transfer level using constructs provided by the MyHDL Python package.

Daedalus may be regarded as an HLS tool to some extent, since it generates RTL from a specification in the C language when a process is mapped onto an application-specific hardware processor. But in contrast to a conventional HLS tool, Daedalus only generates the control path RTL of a hardware processor. Daedalus does not generate data path RTL, as it relies on the designer to provide IP cores that implement

the data path [NSD08a]. Moreover, Daedalus generates complete heterogeneous MPSoC implementations, which is a task that is beyond the scope of high-level synthesis. Another difference between HLS tools and Daedalus is the model used to represent applications. HLS tools predominantly employ *Control Data Flow Graphs* (*CDFGs*) [MPC88, CGMT09], whereas Daedalus employs a process network based model [VNS07].

1.3.2 Electronic System-Level Synthesis

During the late 1990s, electronic system-level synthesis gained interest of system designers as it provided means to cope with the increasing design complexity of systems. A system-level synthesis flow focuses on an entire system possibly containing programmable processors. In contrast, a high-level synthesis flow focuses on a highly optimized application-specific RTL architecture implementing one or more kernels. Many different system-level synthesis tools exist besides Daedalus. SystemCoDesigner takes a set of SystemC modules as input and implements a system by mapping these modules onto hardware and software components [KSS+09]. Ptolemy is an environment for simulation and prototyping of heterogeneous systems [BHLM94, EJL+03]. A system design in Ptolemy may consist of subsystems that employ different models of computation, such as continuous time or process network based models. PeaCE [HKL+08] provides a system-level design environment based on Ptolemy, but restricts itself to an extension of the synchronous data flow model and an extension of the finite state machine model. SystemCoDesigner, Ptolemy, and PeaCE require the designer to specify a system as a set of actors interconnected using communication channels, while Daedalus automatically derives actors (processes) and channels from sequential code. The StreamIt approach [GTA06] requires the designer to specify an application graph using actors and communication channels in a custom language. StreamIt employs the Synchronous Data Flow (SDF) model, which is more restrictive than the PPN model employed by Daedalus. The System-on-Chip Environment (SCE) [DGP+08] uses the SpecC language [ZDG97] to describe system behavior. The SCE design flow consists of similar design steps as Daedalus, such as parallelization, communication synthesis, and RTL generation. However, the parallelization step is automated in Daedalus, whereas SCE requires the designer to explicitly specify the system as a set of concurrent tasks interconnected using communication channels. The MPSoC Application Programming Studio (MAPS) [LC10] is a framework that aids the MPSoC designer with C application parallelization. The parallelization in MAPS is relies on profiling information, whereas parallelization in Daedalus is static. Like Daedalus, MAPS can also incorporate already parallelized applications specified as a (Kahn) process network. Unlike

Daedalus, MAPS does not provide an automated way to obtain a parallelized variant of a sequential application. The Multi-Application and Multi-Processor Synthesis (MAMPS) flow maps synchronous dataflow graphs onto homogeneous MPSoCs. In contrast, Daedalus uses the more expressive PPN model and targets heterogeneous MPSoCs. MAMPS on the other hand supports multiple applications at once, whereas the Daedalus version used in this thesis supports only one application at once, although the Daedalus^{RT} extension does support multiple applications [BZNS12].

Bluespec SystemVerilog (BSV) is a high-level hardware description language intended to describe complete systems [NC10]. In a comparison conducted by Nikolov et al., a C specification of an H.264 video decoder was implemented using both the automated Daedalus flow and as a semi-custom design in BSV [NRD$^+$09]. The Daedalus design employed programmable components as processing elements, on which the C specification was mapped. The Bluespec design employed dedicated hardware processing elements, requiring manual conversion of the C specification to BSV. The authors found that the design time for the Daedalus approach was roughly 6 times shorter than the design time for the Bluespec design. This difference was mainly caused by the manual conversion and verification in the BSV design. However, the shorter design time in Daedalus came at the expense of higher resource cost caused by the use of programmable processors. Replacing programmable processors with dedicated RTL cores may reduce the resource cost footprint in the Daedalus flow. Such cores can be obtained automatically from C using HLS tools. We discuss the integration of HLS in Daedalus in Chapter 3.

Several ESL tools focus on graphical entry of a system-level design. For example, a system is specified in Koski using Unified Modeling Language (UML) [KKO$^+$06]. Xilinx System Generator provides a block-based design environment [Xil02]. A System Generator design can be compiled into a netlist, which can then be synthesized onto an FPGA. The latest Vivado design suite from Xilinx integrates System Generator, AutoESL, and RTL synthesis into a single ESL design environment.

Many of the discussed high-level synthesis and system-level design tools do not address the specification gap, as they require the designer to provide a parallel specification. On the other hand, Daedalus employs the PNGEN tool flow to bridge the specification gap as it can automatically find a parallel specification. As the designer does not have to provide a parallel specification, the design process is accelerated. A key challenge of automated parallelization is detecting the statements that are independent of each other, such that they can execute in parallel. PNGEN employs exact data dependence analysis to precisely find the dependence relations between statements [Fea91]. Obtaining exact data dependence information is complicated and is not always possible for arbitrary code. Most HLS and ESL tools that start from a sequential specification therefore rely on approximate data dependence anal-

ysis techniques such as Banerjee's test [Ban88]. But as a result of the approximate nature, tools may need to conservatively assume a data dependence exists between statements, possibly preventing any parallel execution. Such false data dependences can be circumvented using tool-specific compiler pragmas that need to be inserted by the designer. Daedalus requires that the application is specified as a SANLP (cf. Section 2.3) for which exact data dependence analysis is always feasible. This eliminates the need for conservative data dependence assumptions, freeing the designer from having to manually analyze data dependences.

1.4 Contributions and Outline

In Section 1.2, we identified three central problems in an iterative system-level design flow. We solve these three problems in this dissertation in the context of the Daedalus methodology [NSD08b], which we discuss in more detail in **Chapter 2**. The Daedalus methodology is the result of many dissertations in the LERC group [Rij02, Ste04, Tur07, ZI08, Nik09, Mei10, Nad12, Bal13]. We want to build further upon the contributions made in these dissertations. The research conducted in this thesis has led to the following four contributions:

Contribution I [HK09, NHS+11, HK12]: As a solution to the synthesis problem, we add extensions to the Daedalus methodology. These extensions enable us to obtain FPGA implementations from C programs which were already accepted by Daedalus, but for which an FPGA implementation was not yet feasible. With these extensions, we are now able to obtain complete FPGA implementations for industrially relevant applications, like the sphere decoder application discussed in Chapter 6. We have shown that we can characterize and integrate functional kernels (IP cores) from a broad set of conventional HLS tools like the industrial tools Vivado HLS and Synphony C, and the academic tool DWARV. Our extensions to the Daedalus methodology provide an enabling step to realize the complete conventional "forward" system-level synthesis flow for FPGAs in the flow shown in Figure 1.3. The position of these extensions is indicated by the ❶ in Figure 1.3 and are discussed in **Chapter 3**.

Contribution II [HHK10]: As a solution to the performance estimation problem, we investigate four different performance estimation techniques, that differ in accuracy and assessment effort. We want to emphasize on two techniques in particular. We show a novel analytical approach to estimate the performance of cyclic PPNs. The analytical approach is based on the well-known *Maximum Cycle Mean* (MCM) theory from the HSDF model of computation, but avoids exponential complexity explosion in the PPN-to-HSDF conversion. Although providing a theoretical basis, the practical use is limited due to unknown accuracy of the result. In that respect, an-

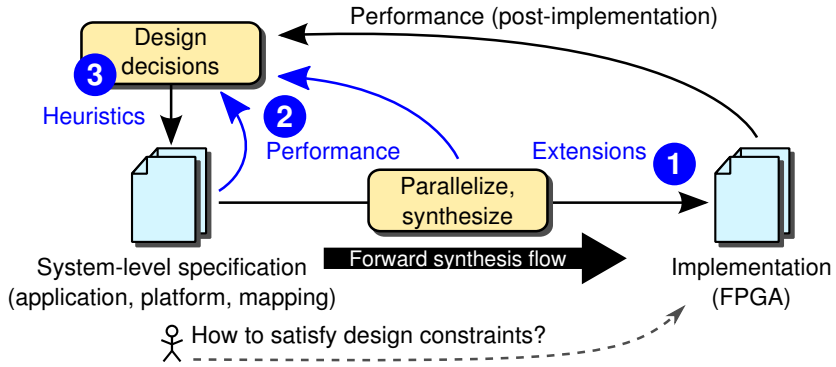


Figure 1.3: Contributions positioned in the iterative design flow.

other contribution of this thesis is *cprof*, which is a novel profiling based approach that completely bypasses the forward synthesis flow. The approach is robust as it relies only on an ordinary C++ compiler to obtain accurate performance estimates of PPNs. Moreover, the approach allows for early estimation of the effects of transformations. The approach provides the designer with an upper bound on the degree of parallelism in an application specification. This allows a designer to assess at a very early stage in the design flow whether he can meet his constraints. The position of the alternative performance estimation techniques in the overall design flow is indicated by the ❷ in Figure 1.3 and is discussed in **Chapter 4**.

Contribution III [HK12]: As a solution to the optimization problem, we provide heuristics to optimize a design by leveraging the insight gained from the performance estimation techniques discussed in Chapter 4. The heuristics provide a concrete set of criteria that guide the designer in selecting standard transformations such as splitting, merging, stream multiplexing, and scheduling. This position of the heuristics is indicated by the ❸ in Figure 1.3 and is discussed in **Chapter 5**.

Contribution IV [HNVK11, NNH+10, NNH+11]: We have shown that we can apply the extended forward system-level synthesis flow depicted in Figure 1.3 on an industrially relevant application. This case study also shows that PPNs are a feasible alternative to conventional CDFG-based C-to-RTL flows. Using the heuristics from Chapter 5, in particular merging, we were able to transform the design to obtain a new pareto design point that was not achievable with a state-of-the-art industrial HLS tool. The use of the profiling-based cprof performance estimation technique presented in Chapter 4 was essential to gain insight in the application performance and the optimization opportunities. The case study is discussed in **Chapter 6**.

We summarize our work and conclude in **Chapter 7**.

CHAPTER 2

BACKGROUND

In this chapter, we introduce concepts and notations that are used throughout this thesis. In Section 2.1, we introduce the polyhedral model which we employ for analysis of programs. In Section 2.2, we review various models of computation that are widely employed to represent applications. We focus on the polyhedral process network model of computation employed by Daedalus and review in Section 2.3 how such networks can be derived from a particular class of sequential programs. In Section 2.4, we review how processes of polyhedral process networks can be implemented in hardware.

2.1 Polyhedral Model

The streaming applications that we consider in this thesis are data-driven: a sequence of computations is repeatedly applied on an incoming data stream, such as a stream of images produced by a video camera. These streaming applications spend most of their execution time in loops that perform computations on data stored in arrays. For example, edge detection algorithms consist of loop nests that iterate over all pixels of the input image that is stored in a 2-dimensional array. These loop nests are the primary candidates for optimization, since most of the time is spent there. To select and apply optimizations, one needs means to reason about iterations of loops and relations between statements contained in loop nests. This is possible with the polyhedral model [Pug91, Fea96] which is employed by modern compilers like GCC [PCB+06] and LLVM/Polly [GZA+11]. The polyhedral model allows a compact representation of loop nests while providing sufficient means to express advanced optimizations such as loop skewing [SKD02]. We use polyhedra to compactly represent loop nests

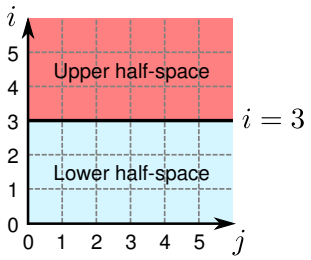


Figure 2.1: A 1-dimensional hyperplane (i.e., a line) $H_1 = \{(j, i) \in \mathbb{Q}^2 \mid i = 3\}$ dividing a 2-dimensional space.

in the polyhedral model. A polyhedron can be defined using hyperplanes.

Definition 2.1 (Hyperplane)**.**
A *hyperplane* H is a subspace of dimension $d - 1$ inside a d-dimensional space, that is,

$$H = \{\mathbf{x} \in \mathbb{Q}^d \mid \mathbf{a}^T\mathbf{x} = c\},$$

where $\mathbf{a}$ is a non-zero vector of size d and c is a constant [Rij02].

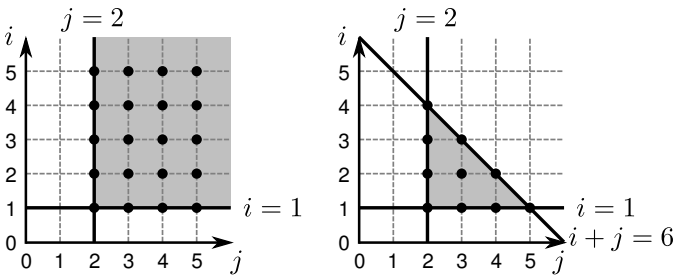
A hyperplane is a generalization of a conventional 2-dimensional plane to $n \in \mathbb{N}$ dimensions. A 1-dimensional hyperplane dividing a 2-dimensional space is shown in Figure 2.1. A hyperplane divides a space into an upper and a lower *half-space*. We distinguish *open half-spaces* which do not include the dividing hyperplane itself, and *closed half-spaces* which include the dividing hyperplane. We use hyperplanes to define subspaces of $\mathbb{Q}^d$, known as rational polyhedra:

Definition 2.2 (Rational Polyhedron)**.**
A *rational polyhedron* $\mathcal{P}$ is a subspace of $\mathbb{Q}^d$ that is bounded by a finite set of m hyperplanes, that is,

$$\mathcal{P} = \{\mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} \geq \mathbf{c}\},$$

where A is an integral $m \times d$ matrix and $\mathbf{c}$ is an integral vector of size m [Ver10].

The shaded rectangular area in Figure 2.2a represents a 2-dimensional rational polyhedron that is bounded by the closed upper half-spaces of two 1-dimensional hyperplanes $i = 1$ and $j = 2$. This rational polyhedron extends into infinity in both dimensions. By adding the closed lower half-space of the hyperplane $i + j = 6$ to the bounds, we obtain a rational polyhedron that is fully enclosed by its bounding hyperplanes, as shown in Figure 2.2b. Such an enclosed rational polyhedron containing a finite number of integral points is called a *rational polytope*.



a) Rational polyhedron.

b) Rational polytope.

```
for (j=2;j<=5;j++) {
  for (i=1;i<=6-j;i++) {
    F(...);
  }
}
```

c) Loop nest represented by b).

```
for (j=2;j<=N;j++) {
  for (i=1;i<=6-j;i++) {
    F(...);
  }
}
```

d) Loop with parametric bound.

```
for (j=2;j<=5;j++) {
  for (i=1;i<=6-j;i++) {
    if (i%2 == 0)
      F(...);
  }
}
```

e) Statement with modulo guard.

Figure 2.2: a) A 2-dimensional rational polyhedron; b) a 2-dimensional rational polytope; c) a loop nest of depth two that can be represented by the 2-dimensional rational polytope given in b); d) a loop nest where the outer loop has a parametric upper bound; and e) a statement with a modulo guard.

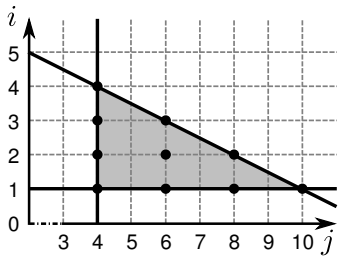
Definition 2.3 (Parametric Rational Polyhedron)**.**
A *parametric rational polyhedron* $\mathcal{P}(\mathbf{s})$ is a family of rational polyhedra in $\mathbb{Q}^d$ that is parametrized by parameters $\mathbf{s} \in \mathbb{Q}^n$:

$$\mathbf{s} \mapsto \mathcal{P}(\mathbf{s}) = \{\mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} + B\mathbf{s} \geq \mathbf{c}\},$$

where A is an integral $m \times d$ matrix, B is an integral $m \times n$ matrix, and $\mathbf{c}$ is an integral vector of size m [Ver10].

A parametric rational polyhedron can represent a loop nest that iterates over a finite, possibly parameterized set of iterations. By assuming that the iterators of such a loop nest are integers, we can represent a loop nest as a set of integral points in a (parametric) rational polyhedron. For example, the loop nest shown in Figure 2.2c can be represented by the rational polytope shown in Figure 2.2b. Each iteration of the loop nest has a corresponding point in the rational polytope. The loop nest shown in Figure 2.2d can be represented by a parametric rational polytope.

When for example a statement is guarded with an expression containing a modulo operator, we are interested in only a subset of the points of a parametric rational polyhedron. In the example shown in Figure 2.2d, function `F` is called only for even values of iterator `i`. We define the polyhedral set to represent a subset of points in a parametric rational polyhedron.



$$\begin{aligned}\mathcal{S} &= \left\{(j,i) \in \mathbb{Z}^2 \mid \exists e \in \mathbb{Z} : \begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ -1 & -2 \end{pmatrix} \cdot \begin{pmatrix} j \\ i \end{pmatrix} \right. \\ &\qquad \left. + \begin{pmatrix} 2 \\ -2 \\ 0 \\ 0 \\ 0 \end{pmatrix} \cdot (e) \geq \begin{pmatrix} 0 \\ 0 \\ 4 \\ 1 \\ -12 \end{pmatrix} \right\} \\ &= \{(j,i) \in \mathbb{Z}^2 \mid \exists e \in \mathbb{Z} : 2e = j \wedge j \geq 4 \wedge i \geq 1 \wedge \\ &\qquad 2i + j \leq 12\}\end{aligned}$$

Figure 2.3: Example polyhedral set.

Definition 2.4 (Polyhedral Set)**.**
A *polyhedral set* $\mathcal{S}$ is a finite union of basic integer sets, $\mathcal{S} = \bigcup_i \mathcal{S}_i$, of type $\mathbb{Q}^n \rightarrow 2^{\mathbb{Q}^d}$, where each basic integer set $\mathcal{S}_i$ is defined as

$$\mathcal{S}_i = \mathbf{s} \mapsto \mathcal{S}_i(\mathbf{s}) = \{\mathbf{x} \in \mathbb{Z}^d \mid \exists \mathbf{z} \in \mathbb{Z}^e : A\mathbf{x} + B\mathbf{s} + D\mathbf{z} \geq \mathbf{c}\},$$

where A is an integral $m \times d$ matrix, B is an integral $m \times n$ matrix, D is an integral $m \times e$ matrix, and $\mathbf{c}$ is an integral vector of size m. The *parameter domain* of $\mathcal{S}$, $\{\mathbf{s} \in \mathbb{Z}^n \mid \mathcal{S}(\mathbf{s}) \neq \emptyset\}$, is a polyhedral set containing all parameter values $\mathbf{s}$ for which $\mathcal{S}$ is non-empty. A polyhedral set with an empty parameter domain (i.e., $n = 0$) is called a *non-parametric* polyhedral set, and denoted with "$\mathbf{s} \mapsto$" omitted. The parameter domain of a polyhedral set is always non-parametric [Ver10].

The polyhedral set depicted in Figure 2.3 contains only a subset of the integral points of its bounding rational polytope. In particular, it only contains the integral points for even values of j, which can be expressed as "$j \bmod 2 = 0$". Such constraints are enforced using the existentially quantified variables $\mathbf{z}$ in Definition 2.4. For example, the constraint "$j \bmod 2 = 0$" is represented by a condition $2e = j$ and the requirement that e is integral.

To allow reasoning about the execution order of different iterations of a program, we define the lexicographic order on the points of a polyhedral set:

Definition 2.5 (Lexicographic Order)**.**
The *lexicographic order* is a total order on the elements of a polyhedral set. An element $\mathbf{a}$ is lexicographically smaller than an element $\mathbf{b}$, denoted as $\mathbf{a} \prec \mathbf{b}$, if

$a_i < b_i$ for the first dimension i in which both elements differ, or, equivalently,

$$\mathbf{a} \prec \mathbf{b} \equiv \bigvee_{i=1}^{n} \left(a_i < b_i \wedge \bigwedge_{j=1}^{i-1} a_j = b_j \right).$$

For example, an element $a = (2, 3, 5)$ is lexicographically smaller than an element $b = (2, 4, 0)$, because the first difference between both elements is in the second dimension, and the value 3 in the second dimension of a is less than the value 4 in the second dimension of b.

Loop optimizations such as skewing transform iteration domains that we represent using polyhedral sets. A transformation of a polyhedral set can be expressed as a relation between the original polyhedral set and the transformed polyhedral set. We define the polyhedral map to express such relations:

Definition 2.6 (Polyhedral Map)**.**
A *polyhedral map* $\mathcal{M}$ is a finite union of basic polyhedral maps, $\mathcal{M} = \bigcup_i \mathcal{M}_i$, of type $\mathbb{Q}^n \to 2^{\mathbb{Q}^{d_1+d_2}}$, where each basic polyhedral map is defined as

$$\begin{aligned}\mathcal{M}_i &= \mathbf{s} \mapsto \mathcal{M}_i(\mathbf{S}) \\ &= \{(\mathbf{x_1}, \mathbf{x_2}) \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid \exists \mathbf{z} \in \mathbb{Z}^e : A_1\mathbf{x_1} + A_2\mathbf{x_2} + B\mathbf{s} + D\mathbf{z} \geq \mathbf{c}\},\end{aligned}$$

where A_1 is an integral $m \times d_1$ matrix, A_2 is an integral $m \times d_2$ matrix, B is an integral $m \times n$ matrix, D is an integral $m \times e$ matrix, and $\mathbf{c}$ is an integral vector of size m [Ver10].

The polyhedral set

$$\mathbf{s} \mapsto \{\mathbf{x_1} \in \mathbb{Z}^{d_1} \mid \exists \mathbf{x_2} \in \mathbb{Z}^{d_2} : (\mathbf{x_1}, \mathbf{x_2}) \in M(\mathbf{s})\}$$

is the *domain* of a polyhedral map M. The polyhedral set

$$\mathbf{s} \mapsto \{\mathbf{x_2} \in \mathbb{Z}^{d_2} \mid \exists \mathbf{x_1} \in \mathbb{Z}^{d_1} : (\mathbf{x_1}, \mathbf{x_2}) \in M(\mathbf{s})\}$$

is the *range* of a polyhedral map M. In this thesis, we denote polyhedral maps as

$$\mathcal{M} = \mathbf{s} \mapsto \{\mathbf{x_1} \to \mathbf{x_2} \mid \dots\}.$$

An example polyhedral map consisting of only one basic polyhedral map is

$$\mathcal{M}_1 = \{(j_1, i_1) \to (j_2, i_2) \mid j_2 = 2j_1 \wedge i_2 = i_1\}. \tag{2.1}$$

We use polyhedral maps to manipulate points or polyhedral sets by *application* of the

polyhedral map. For example, applying $\mathcal{M}_1$ to a point $(2, 1)$ yields $(4, 1)$, denoted as

$$\mathcal{M}_1(2, 1) = (4, 1).$$

If we apply this polyhedral map to the polyhedral set of Figure 2.2b, that is, if we compute $\mathcal{M}_1(\mathcal{S}_1)$, we obtain the polyhedral set depicted in Figure 2.3. The points in this new polyhedral set result from application of $\mathcal{M}_1$ to each point in the original polyhedral set $\mathcal{S}_1$.

We sometimes need to know the size of a polyhedral set or map, for example to judge whether a certain transformation is beneficial to a given program. The number of elements in a polyhedral set or polyhedral map is given by the cardinality:

Definition 2.7 (Cardinality)**.**
The *cardinality* of a polyhedral set $\mathcal{S}$, denoted as $|\mathcal{S}|$, represents the number of elements in $\mathcal{S}$.
The cardinality of a polyhedral map $\mathcal{M}$, denoted as $|\mathcal{M}|$, represents the number of elements in the range of $\mathcal{M}$ associated to any element in the domain of $\mathcal{M}$.

We use the `barvinok` library to analytically determine the cardinality of polyhedral sets and maps [VSB+07, Ver03a]. The cardinality is expressed as a piecewise quasipolynomial. A piecewise quasipolynomial consists of one or more quasipolynomials:

Definition 2.8 (Quasipolynomial)**.**
A *quasipolynomial* $q(\mathbf{x})$ is a polynomial expression in greatest integer parts of affine expressions of variables in $\mathbf{x}$. The coefficient of each term may include a constant integer division [Ver10].

Definition 2.9 (Piecewise Quasipolynomial)**.**
A *piecewise quasipolynomial* $q(\mathbf{x})$ consists of one or more quasipolynomials. Each quasipolynomial $q_i(\mathbf{x})$ is defined only for a disjoint piece $\mathcal{D}_i$ of a domain $\mathcal{D}$. For a given point $\mathbf{x} \in \mathcal{D}$, the piecewise quasipolynomial evaluates to

$$q(\mathbf{x}) = \begin{cases} q_i(\mathbf{x}) & \text{if } \mathbf{x} \in \mathcal{D}_i, \\ 0 & \text{otherwise} \end{cases} \quad \text{[Ver10]}.$$

For example, the cardinality of the polyhedral set $\mathcal{S}_2$ of Figure 2.3 is expressed using the piecewise quasipolynomial

$$|\mathcal{S}_2| = \begin{cases} 10 & \text{if } 1 \geq 0 \end{cases}.$$

The cardinality of $\mathcal{S}_2$ is constant because all bounding hyperplanes are constant. Therefore, the cardinality is not dependent on any parameters or variables and con-

sists of only one piece that is selected using the tautology $1 \geq 0$. The quasipolynomial has a constant value of 10, as $\mathcal{S}_2$ consists of 10 points.

The cardinality of the polyhedral map $\mathcal{M}_1$ of Equation (2.1) is expressed using the piecewise quasipolynomial

$$|\mathcal{M}_1|(j_1, i_1) = \begin{cases} 1 & \text{if } (j_1, i_1) \in \mathbb{Z}^2, \\ 0 & \text{otherwise.} \end{cases}$$

This means that applying $\mathcal{M}_1$ to any point (j_1, i_1) that is in $\mathbb{Z}^2$ always yields exactly one new point (j_2, i_2).

2.2 Models of Computation

Designers specify the behavior of a system in a structured way using a *Model of Computation* (*MoC*). To facilitate programming of multi-processor systems, a parallel MoC is needed such that the tasks for each processor and the communication and synchronization mechanisms can be specified. Different MoCs have been proposed and evaluated for their use in design automation in literature [LSV98, JS05]. For example, HDL simulators often employ a timed discrete-event MoC in which all events are ordered globally in time. A global ordering is often not desired for a multi-processor system because different parts of the system may execute in parallel. Our interest is in untimed dataflow process network based MoCs such as Kahn Process Networks (KPNs) defined by Kahn [Kah74]. The dataflow-based MoCs that we consider in this thesis have several properties that make them attractive for specification of multi-processor systems [SZT+04]. One desirable property is deterministic behavior, such that a given input sequence always results in the same output sequence regardless of variations in computation or communication times. Another desirable property is that each task behaves autonomously, such that each processor of a multi-processor system can be considered in isolation. This allows designers to better cope with complex multi-processor systems.

Many different specializations of dataflow process network based MoCs have been proposed in literature for the design of streaming applications. A major reason for the abundance of different specializations is to allow different tradeoffs of expressiveness against analyzability. With *expressiveness* of a model we refer to the ability to express an application in that model in a succinct way. Although more general models often can be converted to more specialized equivalent models, such a conversion often increases the size of the application model making it no longer succinct. With *analyzability* of a model we refer to the existence and complexity of compile-time analysis algorithms to compute for example static schedules, buffer sizes, or

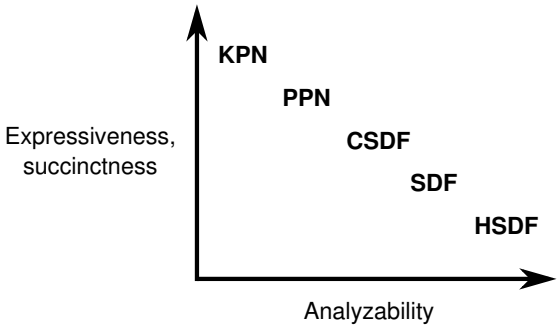


Figure 2.4: Different models and their expressiveness and analyzability.

throughput. In Figure 2.4, we depict five different models and compare their expressiveness and analyzability. For example, many applications can be expressed in the KPN MoC, but due to the genericity of the model, the compile-time analyzability is limited. In contrast, the HSDF model has a lower expressiveness but this allows for full analyzability. We now review four dataflow-based models that we use in the remainder of this thesis for specification and analysis of MPSoCs: the HSDF, SDF, CSDF, and PPN models of computation.

2.2.1 Homogeneous Synchronous Dataflow

The most restricted model of computation that we consider in this thesis is the homogeneous synchronous dataflow model, which is also known as the single-rate dataflow model [GGS$^+$06]. The more generic models that we discuss later extend the homogeneous synchronous dataflow model. We use the following definition, in line with the notation used by e.g. Moreira et al. [MBGS10]

Definition 2.10 (Homogeneous Synchronous Dataflow Graph)**.**
A *Homogeneous Synchronous Data Flow* (*HSDF*) graph is a directed graph defined by a tuple (V, E, t, d), where

- V is a set of vertices representing computation *nodes*,
- E is a set of *edges* representing communication channels that carry *tokens*,
- $t(i), i \in V$ represents the time needed for a single execution of node i, and
- $d(e), e \in E$ represents the number of *initial tokens* on edge e, also referred to as the *delay* of edge e.

An HSDF graph consisting of four nodes and six edges is shown in Figure 2.5. Shown in the upper half of each node is a label that we assign for convenient referencing. Shown in the lower half of each node is the node's execution time $t(i)$. For

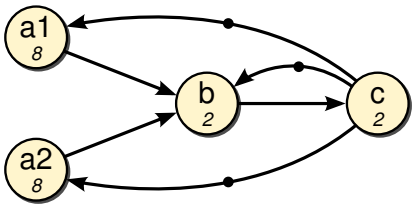


Figure 2.5: An HSDF graph.

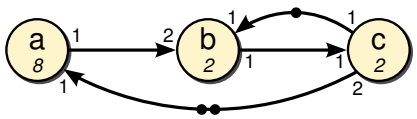
example, node b has an execution time $t(b) = 2$ time units. Initial tokens $d(e)$ for each edge are shown as dots on the edges. For example, the edge connecting node c to *a2* contains one initial token, that is, $d(c \rightarrow a2) = 1$. For clarity reasons, we may visualize multiple initial tokens by a single dot and a number above or below the dot.

Edges transfer units of data referred to as *tokens*. A node is said to be *enabled* if each of its incoming edges contains at least one token. An enabled node is said to *fire* when it consumes a token from each incoming edge, performs a computation on these tokens, and then produces a token on each of its outgoing edges. If none of the nodes is enabled, then the graph is in a *deadlock* state. If all nodes of a graph can fire infinitely often, then the graph is *live*. An HSDF graph is said to be *consistent* if every token written to an edge is eventually consumed, such that the graph can be executed under bounded memory conditions. An *iteration* of an HSDF graph is defined as each node executing exactly once.

Different firings of a node may start at the same time, such that overlapped execution between firings of the same node occurs. For example, if edge $c \rightarrow a1$ in Figure 2.5 would contain two initial tokens, then two firings of *a1* can start simultaneously. Such overlapped execution of firings of the same node is referred to as *auto-concurrency*. By adding an edge from a node to itself, referred to as a *selfloop*, we can regulate auto-concurrency of a node. The number of initial tokens on that selfloop limits the number of parallel firings. By putting one initial token on the selfloop, auto-concurrency is fully prevented. In such a case, the node consumes the initial token from the selfloop at the first firing, and only produces a new token on the edge once it finishes its firing. The node is not enabled for any subsequent firings until the first firing has finished, meaning no overlap between firings occurs.

2.2.2 Synchronous Dataflow

HSDF graphs are a special case of the more general synchronous dataflow graphs defined by Lee and Messerschmitt [LM87].



$$\Gamma = \begin{bmatrix} 1 & -2 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \\ -1 & 0 & 2 \end{bmatrix}$$

Figure 2.6: An SDF graph and its topology matrix Γ.

Definition 2.11 (Synchronous Dataflow Graph)**.**
A *Synchronous Data Flow* (*SDF*) graph is a directed graph defined by a tuple (V, E, t, d, p, c), where

- V, E, t, and d follow those in Definition 2.10,
- $p(e), e \in E$ represents the number of tokens placed on edge e when the corresponding source node fires, referred to as the *production rate*, and
- $c(e), e \in E$ represents the number of tokens consumed from edge e when the corresponding destination node fires, referred to as the *consumption rate*.

An SDF graph consisting of three nodes and four edges is shown in Figure 2.6. The numbers depicted at the location where edges connect to nodes represent the production and consumption rates. For example, when node c fires it consumes $c(b \rightarrow c) = 1$ token from edge $b \rightarrow c$, and it produces $p(c \rightarrow b) = 1$ token on edge $c \rightarrow b$ and $p(c \rightarrow a) = 2$ tokens on edge $c \rightarrow a$.

The structure and production and consumption rates of an SDF graph are compactly represented by a *topology matrix* Γ. The columns of Γ represent the nodes and the rows of Γ represent the edges. A positive entry $\Gamma(i, j)$ means that node j produces $\Gamma(i, j)$ tokens on edge i. A negative entry $\Gamma(i, j)$ means that node j consumes $-\Gamma(i, j)$ tokens from edge i. A zero entry $\Gamma(i, j)$ means that node j does not read or write to edge i. A selfloop can be represented in Γ by the net difference between production and consumption [LM87, p. 27].

An SDF graph can be converted into an equivalent HSDF graph [SB00, Chapter 3]. However, such a conversion may cause an exponential increase in the number of nodes in the worst case. The HSDF graph of Figure 2.5 is the result of converting the SDF graph of Figure 2.6. An *iteration* of an SDF graph is defined as each node of the equivalent HSDF graph executing exactly once. If an SDF graph is consistent, then a *repetition vector* $\mathbf{q}$ exists which contains for every node the number of times the node has to fire to return the SDF graph to its initial state. The repetition vector is the smallest non-trivial positive integer vector that is a valid solution to the *balance equation* $\Gamma \cdot \mathbf{q} = \mathbf{0}$.

For the graph of Figure 2.6, the smallest non-trivial solution to the balance equation is the repetition vector $\mathbf{q} = [2, 1, 1]^T$. This means that if node *a* fires twice, node *b*

fires once, and node c fires once, then the number of initial tokens on each edge is the same as before the execution of these four firings.

An SDF node always consumes tokens from all input edges and produces tokens on all output edges during a firing. Consequently, the SDF model cannot describe a node that for example reads from different input ports during different firings. This means that applications in which such behavior occurs cannot be modeled as an SDF graph.

2.2.3 Cyclo-Static Dataflow

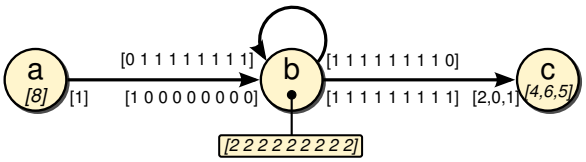
An extension to the SDF model that allows such behavior is the cyclo-static dataflow model [BELP96]. This model allows a compact representation of applications with a cyclically changing, but predefined behavior.

Definition 2.12 (Cyclo-Static Dataflow Graph)**.**
A *Cyclo-Static Data Flow* (*CSDF*) graph is a directed graph defined by a tuple $(V, E, \mathbf{f}, \mathbf{t}, d, \mathbf{p}, \mathbf{c})$, where

- V, E, and d follow those in Definition 2.10,
- $\mathbf{f}_j, j \in V$ represents the function repertoire for node j, which is a sequence of functions $[f_j(0), f_j(1), \cdots, f_j(S_j - 1)]$ of *phase length* S_j,
- $\mathbf{t}_j(i), j \in V$ represents the time needed for an execution of function i in $\mathbf{f}_j$,
- $\mathbf{p}_e(i), e \in E$ is a sequence of integers representing the number of tokens produced on edge e after e's source node fires its i-th function, and
- $\mathbf{c}_e(i), e \in E$ is a sequence of integers representing the number of tokens consumed from edge e before e's destination node fires its i-th function.

Each node in a CSDF graph executes the functions in its function repertoire in a cyclic fashion. At the start of the n-th firing of node j, $\mathbf{c}_e(n \bmod S_j)$ tokens are consumed from incoming edge e. Then, function $f_j(n \bmod S_j)$ is executed which takes $\mathbf{t}_j(n \bmod S_j)$ time units. After the function finishes execution, $\mathbf{p}_e(n \bmod S_j)$ tokens are produced on outgoing edge e.

Similar to the topology matrix of an SDF graph, we can define a *topology matrix* Γ for a CSDF graph. A positive entry $\Gamma(i, j)$ means that node j produces in total $\Gamma(i, j)$ tokens on edge i for a complete execution sequence, that is, $\Gamma(i, j) = \sum_{k=0}^{S_j - 1} \mathbf{p}_i(k)$. A negative entry $\Gamma(i, j)$ means that node j consumes in total $\Gamma(i, j)$ tokens from edge i for a complete execution sequence, that is, $\Gamma(i, j) = -\sum_{k=0}^{S_j - 1} \mathbf{c}_i(k)$. All other entries are zero.



$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 9 & -3 \end{bmatrix}$$

$$S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

Figure 2.7: A CSDF graph, its topology matrix Γ, and its phase matrix S.

To obtain the repetition vector of a CSDF graph, one first solves the *balance equation* $\Gamma \cdot \mathbf{r} = \mathbf{0}$. The repetition vector then equals

$$\mathbf{q} = S \cdot \mathbf{r}, \qquad \text{where } S(i,j) = \begin{cases} S_j & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases} \tag{2.2}$$

Matrix S in Equation (2.2), whose diagonal contains the phase lengths of all nodes, is referred to as the *phase matrix*.

A CSDF graph consisting of three nodes and three edges is shown in Figure 2.7. The function repertoire of node *c* contains three functions with latencies 4, 6, and 5, as shown in the bottom part of the node. Thus, the phase length of node *c* $S_c = 3$. Node *c* has one incoming edge $b \rightarrow c$. In the $0 \pmod 3$-th execution of node *c*, two tokens are consumed from this edge; in the $1 \pmod 3$-th execution of node *c*, no tokens are consumed; and in the $2 \pmod 3$-th execution of node *c*, one token is consumed from this edge.

The topology matrix of the CSDF graph is shown in the upper right part of Figure 2.7. Since the CSDF graph contains a selfloop, the second row of Γ consists entirely of zeros. The phase matrix of the CSDF graph is shown in the lower right part of Figure 2.7. For example, the lower right element of this matrix equals node *c*'s phase length $S_c = 3$. The smallest non-trivial solution to the balance equation is $\mathbf{r} = [1, 1, 3]^T$. Hence, the repetition vector of the CSDF graph $\mathbf{q} = [1, 9, 9]^T$.

The phase lengths and production and consumption patterns $\mathbf{p}$ and $\mathbf{c}$ may be large for applications that have mainly regular, but occasionally irregular behavior. This is for example found in image edge detection algorithms, whose behavior is regular for most pixels, but irregular for pixels at the image borders. Large phase lengths make a CSDF representation impractical for analysis and synthesis tools. We therefore present another model in which complex patterns can be captured in a compact way using the polyhedral model.

2.2.4 Polyhedral Process Networks

The Daedalus system-level design tool set that was introduced in Section 1.1 (cf. Figure 1.1) employs polyhedral process networks as its application model. The polyhedral process network model was first coined by Meijer et al. [MNS10] and was later formally defined by Verdoolaege [Ver10]. The definition of Verdoolaege differs from the classical definitions employed by the Compaan and Daedalus tools, as presented by for example Turjan [Tur07], Nikolov et al. [NSD08b], and Rijpkema [Rij02]. Throughout this thesis, we use the definition of the latter references. A conversion from the definition of Verdoolaege to the definition used by Daedalus is possible and is extensively used in the Daedalus tool flow [Ver03b].

Definition 2.13 (Polyhedral Process Network)**.**
A *Polyhedral Process Network* (*PPN*) is a directed graph $(\mathcal{P}, \mathcal{E})$ where $\mathcal{P}$ is a set of vertices representing processes and $\mathcal{E}$ is a set of edges representing communication channels. Each process $p_i \in \mathcal{P}$ is characterized by:

- a function F_i,
- a process dimensionality d_i,
- a polyhedral set $D_i \subseteq \mathbb{Z}^{d_i}$ defining the process' domain.
- a set of *input ports* IP_i, where the k-th input port IP_i^k is bound to an input argument of F_i and has an associated *Input Port Domain* (*IPD*) $IPD_i^k \subseteq D_i$, and
- a set of *output ports* OP_i, where the k-th output port OP_i^k is bound to an output argument of F_i and has an associated *Output Port Domain* (*OPD*) $OPD_i^k \subseteq D_i$.

Each channel $c_i \in \mathcal{E}$ is characterized by:

- a source process $\sigma_i \in \mathcal{P}$,
- a destination process $\delta_i \in \mathcal{P}$,
- a source process' output port $OP_{\delta_i}^j$,
- a destination process' input port $IP_{\sigma_i}^k$,
- a polyhedral map $M_i \subseteq D_{\sigma_i} \times D_{\delta_i}$ mapping iterations from the destination process domain back to the source process domain.
- a channel type T_i, which is *FIFO*, *sticky FIFO*, or *out-of-order* (cf. Section 2.3.1), and
- a piecewise quasipolynomial S_i representing the buffer size.

The parameters that occur in the process domains, channel maps and buffer sizes are *static*, meaning that their values are fixed at run-time. A more general model which

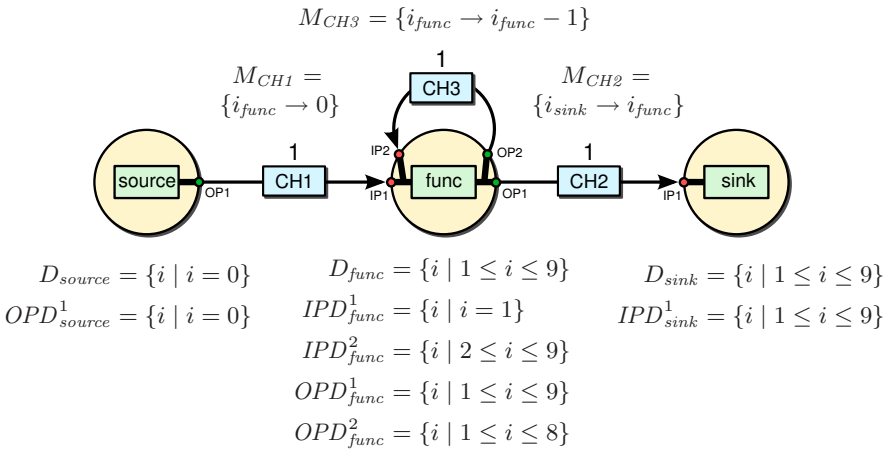


Figure 2.8: A polyhedral process network.

also includes *dynamic* parameters is the Parameterized Polyhedral Process Network (P^3N) model [ZNS11]. Such dynamic parameters enable the P^3N model to cope with applications that adapt their behavior at runtime. Another related model is the Approximated Dependence Graph (ADG) [SD03]. The ADG model supports the class of weakly dynamic programs, which is more generic than the class of static affine nested loop programs that we consider in this thesis.

In this thesis, we are dealing with instances of PPNs for which all static parameters have known fixed values. We replace the static parameters by their fixed values, thereby removing the paramters, for the sake of simplicity.

An example PPN consisting of three processes and three channels is depicted in Figure 2.8. In this thesis, we only consider PPNs that consist of exactly one *connected component*. That is, if one replaces all directed edges in the graph by undirected edges, then a path from u to v exists for every pair of vertices u, v. The PPNs that we consider may contain zero or more *strongly connected component*s. A strongly connected component is a subgraph in which a path from any vertex in the subgraph to any other vertex in the subgraph exists.

If a process does not have any input ports, that is, $IP_i = \emptyset$, then the process is called a *source process*. Likewise, if a process does not have any output ports, that is, $OP_i = \emptyset$, then the process is called a *sink process*. The function of a process should be a *pure function*, that is, it should always yield the same output for a given input and it should not have any side effects. Exceptions to this requirement are source and sink processes, which often serve as an abstraction for the input and output interfaces of a system. As such, input and output operations are desired side-effects for functions of

source and sink processes.

In the PPN of Figure 2.8, *source* is a source process with one output port and *sink* is a sink process with one input port. The ports of a process are depicted by the dots on the border of each process. The output argument of the *source* function is connected to the output port of the process. Similarly, the input port of the *sink* process is connected to the input argument of the function. The *func* process has two input ports and two output ports. The *func* function has one input argument and one output argument. Both input ports connect to the same input argument and the output argument is connected to both output ports. Port multiplexing and demultiplexing is performed at run-time, as computations are distributed as a result of data flow analysis. Input and output tokens of a process need to be communicated from and to different processes at different iterations through process input and output ports.

The process and port domains are depicted below the processes in Figure 2.8. For example, the domain of the *sink* process consists of the integral points from 1 to 9. The IPD of its input port is identical to the process domain, which means that in every iteration a token is read from this input port.

The channels in the PPN of Figure 2.8 are shown as rectangles. All channels in this PPN are FIFO channels of size one, as denoted by the number above each channel. The map for each channel is shown above the channel sizes. Channel *CH1* maps an iteration of the *func* process to iteration $i = 0$ of the *source* process. Channel *CH2* maps iterations of the *sink* process to iterations of the *func* process. Channel *CH3* maps iterations of the *func* process to its previous iteration. In the remainder of this thesis, we depict channels in a more compact way as a single arrow with a number specifying the buffer size.

Operational Semantics

Each process of a PPN executes autonomously according to a three-stage program that is executed for each point in the process domain: a read stage, an execute stage, and a write stage [ZI08]. This is an important property that we exploit throughout this thesis. First, in the *read stage*, the input arguments to the process function are read from the input ports whose IPD contains the current iteration. If the channel connected to an input port does not contain any tokens, then the process blocks until a token becomes available. Second, in the *execute stage*, the process function is executed with the input data obtained during the read stage. Third, in the *write stage*, the output arguments of the process function are written to the output ports whose OPD contains the current iteration. If the channel connected to an output port does not have sufficient room to store another token, then the process blocks until a free slot becomes available in the channel.

A process traverses the points in its iteration domain D_i in the lexicographical order, in a sequential fashion. Thus, two iterations cannot start at the same time.

2.3 Derivation of PPNs from Sequential Programs

Polyhedral process networks can be derived automatically from sequential programs known as static affine nested loop programs [RDK00, VNS07].

Definition 2.14 (Static Affine Nested Loop Program)**.**
A *static affine nested loop program* (*SANLP*) is a program consisting of statements enclosed by zero or more loops, where:

- all loops have a constant integral stride,
- loop bounds, if-conditions, and array index expressions are affine combinations of constants and enclosing loop iterators, and
- communication between statements is explicit, that is, statements do not exchange data through hidden variables.

The SANLP for the example of Figure 2.8 is shown in Figure 2.10. This PPN can be derived from the SANLP using the `c2pdg`, `pn`, and `pn2adg` tools from the `isa` tool set [Ver03b]. The tool flow is depicted in Figure 2.9. First, the `c2pdg` tool converts the SANLP into a Polyhedral Dependence Graph (PDG). This PDG contains the statements of the SANLP, the iteration domain of each statement, and the variable and array accesses performed by each statement. The `pn` tool extends the PDG with dependence information [VNS07] obtained using exact dataflow analysis [Fea91]. The `pn2adg` tool converts the extended PDG into an Approximated Dependence Graph (ADG). The PPN model that we introduced in Section 2.2.4 is a subset of this ADG model, as we do not handle dynamic parameters. We therefore consider the output of `pn2adg` as the actual PPN, assuming the input C code does not result in an ADG that lies beyond our PPN model. In this thesis, we refer to the consecutive execution of the `c2pdg`, `pn`, and `pn2adg` tools as PNGEN.

For each of the three function calls in the SANLP of Figure 2.10, PNGEN constructs a process. The domain of each process is derived from the for-loops and if-statements

Figure 2.9: PNGEN: Tool flow to convert a SANLP written in C into a PPN.

surrounding the function call. For function calls not enclosed in any for-loop, such as *source*, a 1-dimensional domain containing a single point is constructed.

PNGEN determines which processes should be interconnected by channels using exact dataflow analysis [Fea91] which is based on parametric integer programming techniques [Fea88]. For each read operation of a variable or array element, exact dataflow analysis reports the latest write operation that wrote the variable or array element. For example, at line 3 of Figure 2.10 we read array element `a[0]` during iteration $i = 1$. This array element is written in line 1. Therefore, PNGEN adds channel *CH1* to the PPN which connects the write operation (that is, the *source* process) to the read operation (that is, the *func* process). As another example, consider the read operations of array elements `a[1]` to `a[8]` at line 3. The read operations are performed during iterations $2 \leq i \leq 9$. Exact dataflow analysis reports that these array elements are written in line 3 during iterations $1 \leq i \leq 8$. Therefore, PNGEN adds channel *CH3* to the PPN which connects the *func* process to itself. The corresponding OPD contains the iterations $1 \leq i \leq 8$ during which the array elements are written. The corresponding IPD contains the iterations $2 \leq i \leq 9$ during which the array elements are read. This corresponds to the domains shown in Figure 2.8.

2.3.1 Channel Type Determination

Channels in a PPN are not all FIFOs, but need to be further classified [TKD07]. Each channel is either of type FIFO, sticky FIFO, or out-of-order, as defined in Defintion 2.13. To distinguish between out-of-order and (sticky) FIFO channel types, PNGEN first verifies if the values written to a channel are read in the same order as the order in which they were written. That is, communication over a channel is *in-order* if for any pair of write operations $(\mathbf{w_1}, \mathbf{w_2})$, the corresponding read operations $(\mathbf{r_1}, \mathbf{r_2})$ execute in the same order. If a pair of write operations exists for which the corresponding read operations occur in the opposite order, then the channel is marked as *out-of-order*.

```
1 source(&a[0]);
2 for (i=1; i<=9; i++) {
3   func(a[i-1], &a[i]);
4 }
5 for (i=1; i<=9; i++) {
6   sink(a[i]);
7 }
```

Figure 2.10: SANLP for the polyhedral process network of Figure 2.8.

In the example of Figure 2.10, all array elements are written and read exactly once. All communication is in-order, which causes the channels to be classified as FIFOs. If we would surround the for-loop containing the *sink* function call by another for-loop, then the elements of array `b` are written once and read multiple times. PNGEN employs a *reuse detection* technique to identify channels from which a single token is read multiple times. Reuse detection results in the construction of a *data reuse channel pair*, consisting of two FIFO channels. The first FIFO channel propagates data from the write operation to the first read operation. The second FIFO channel is a selfloop which propagates data from a read operation to a later read operation by the same process. If the same token is used by multiple subsequent iterations, then the reuse channel pair can be optimized further into a *sticky FIFO*. This means the selfloop is replaced by a register. We refer to Section 3.3 for an example of reuse detection, and we refer to Section 3.4 for an example of a sticky FIFO.

2.3.2 Buffer Size Computation

Each channel of a PPN has an associated buffer size specifying the number of tokens that can be stored. The buffer size has to be chosen under the following constraints. Choosing a buffer size that is too small results in an *artificial deadlock*, a condition in which none of the process can make progress because one or more processes are blocked on a write operation. Choosing an arbitrary large buffer size prevents artificial deadlocks, but increases memory cost. Therefore, careful selection of buffer sizes is required.

The buffer size computation performed by PNGEN consists of the following steps. First, PNGEN computes a global schedule for all processes. That is, it determines a single execution sequence containing all iterations of all processes. PNGEN ensures that the schedule is valid, meaning that each value is always written before it is read. Next, for each channel a buffer size is determined for the computed schedule. The schedule specifies a relative order on any pair of iterations from the same or a different process. Therefore, for a read iteration $\mathbf{r}$ (i.e., an iteration performing a read operation), the number of read iterations $nR(\mathbf{r})$ and the number of iterations performing a write iteration $nW(\mathbf{r})$ preceding $\mathbf{r}$ is known. The buffer size is then the maximal value of $nW(\mathbf{r}) - nR(\mathbf{r})$ over all read iterations $\mathbf{r}$. For the non-parametric PPNs that we consider, this maximal value can be computed symbolically or obtained using simulation. The symbolic approach works by computing an upper bound on a quasi-polynomial [CFGV09]. The simulation-based approach works by simulating the write and read iterations according to the schedule and tracking the maximal amount of tokens stored in the channel.

Computing minimal deadlock-free buffer sizes or a deadlock-free schedule is a non-

trivial optimization problem. PNGEN employs a greedy algorithm to compute a deadlock-free schedule. As a result, the buffer sizes computed by PNGEN are not guaranteed to be minimal, but at least a deadlock-free execution exists for the computed buffer sizes. The schedule is used for buffer size computation. Execution of a PPN is not bound to the schedule, as processes in a PPN only synchronize using blocking read and write operations.

2.4 Code Generation

We employ the ESPAM tool [NSD08b] to implement PPNs derived by PN. We illustrate the ESPAM tool flow in Figure 2.11. The input to ESPAM is a *System-Level Specification*, consisting of three components. First, we provide the *application specification* in the form of a PPN which is derived from a C program using PNGEN. Second, we provide a target *platform specification* describing the amount and types of processors and peripherals, and the type of interconnect. For example, the designer can populate a platform with programmable processors such as MicroBlazes and function-specific hardware IP cores. Third, we provide a *mapping specification* which maps the processes of the PPN onto the processors. The platform and mapping specifications are at a high level of abstraction, omitting low-level details such as the processor memory organization. ESPAM automatically elaborates the specifications to the required degree of detail. After elaboration and possible refinement, one of the backends at the right part of Figure 2.11 generates code which implements the specified system.

ESPAM offers different backends such that a given system-level specification can be implemented in different forms. We distinguish two classes of backends:

- **Implementation backends** produce a fully functional implementation of a

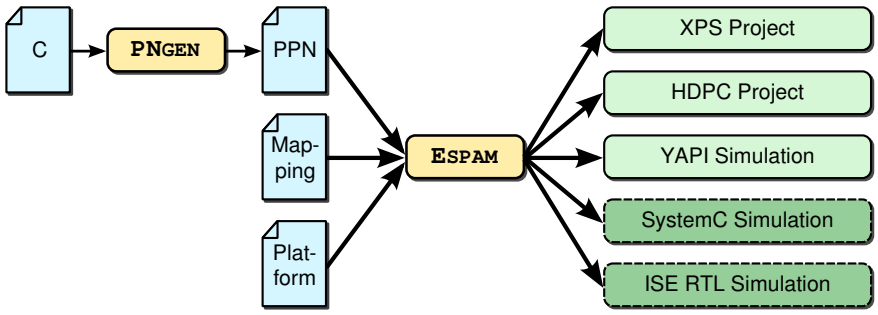


Figure 2.11: The ESPAM tool flow.

given system. ESPAM contains a Xilinx Platform Studio (XPS) backend which generates an FPGA project at the register transfer level, that can be synthesized using vendor-specific low-level synthesis tools to obtain a working prototype. ESPAM also contains a Heterogeneous Desktop Parallel Computing (HDPC) backend which generates a software implementation for a general purpose desktop computer containing for example a GPU device.

- **Simulation backends** produce an environment in which a given system can be simulated. For example, ESPAM contains a YAPI backend [KES+00] which enables fast functional verification of a parallelized application.

In Chapter 4 of this thesis, we present two new backends to ESPAM, which are depicted at the bottom right part of Figure 2.11. The SystemC simulation backend provides fast performance assessment. It works at a raised level of detail compared to RTL simulation, thereby increasing the simulation speed at the expense of lower accuracy. When more accurate performance and resource cost metrics are needed, the ISE backend can be employed. The ISE backend produces a Xilinx ISE project that implements the system entirely in VHDL. This project can be simulated and synthesized in the Xilinx ISE tool to obtain accurate execution time and resource usage metrics. Since the ISE simulation works at a more detailed level, obtaining metrics is more time-consuming compared to a SystemC simulation. For a small application like QR decomposition, a SystemC simulation takes a few seconds, whereas an ISE simulation may take about a minute.

2.4.1 Integrating Dedicated IP Cores

In systems with tight throughput constraints, performing all computations on programmable processors may not be feasible due to the limited performance of such processors. To increase the overall system throughput, designers offload the heaviest computations onto dedicated hardware *IP cores*. These IP cores are custom architectures that are optimized to perform a specific task. Such IP cores are traditionally written in RTL or may be generated from code written in a high-level language using a high-level synthesis tool. The LAURA Virtual Processor model was proposed to include such IP cores in the Daedalus tool flow [ZSKD03, NSD08a].

A representation of a process as C code is shown in the left part of Figure 2.12. The LAURA processor for this process is depicted in the right part of Figure 2.12. A LAURA processor consists of a read, execute, write, and control unit to implement the operational semantics of a PPN process. The read and write units iterate over the process domain D_p and ensure at runtime that the proper channel is being read or written during each iteration. The execute unit contains an IP core which implements the process' functionality, that is, the function F. The read, execute, and write units

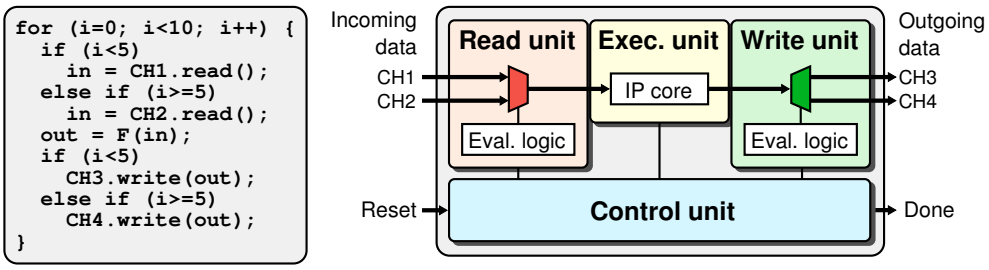


Figure 2.12: Example process code and the corresponding LAURA processor.

operate in a pipeline fashion, as depicted in Figure 2.13. For example, when iteration 0 enters the execute stage, the LAURA processor can initiate the read stage of the next iteration. The control unit orchestrates execution of the read, execute, and write units. For example, it stalls the read and execute units if the write unit reports a blocking write condition. In Figure 2.13, a blocking read condition occurs during iteration 3. In such a case, the controller ensures that previous iterations that are already in the pipeline continue executing, while iteration 3 is stalled until data is available. This leads to a bubble in the pipeline, which is indicated by a "–" in Figure 2.13.

The read unit is connected to the incoming channels of the process. For each input argument of the process' function F, a *read multiplexer* is instantiated. This multiplexer selects the incoming channels from which the argument is read. The selection is driven by the read unit's *evaluation logic* block. The evaluation logic employs a set of counters that iterate over the process domain. For each input port, the evaluation logic contains an expression in terms of the iterators that selects when that port has to be read. As such, data from the appropriate input ports is forwarded to the execute unit according to the current iteration.

The execute unit implements the process' function F. It provides an insertion slot for an IP core that implements the function F. The execute unit passes the argument values selected by the read unit to the IP core. The IP core processes the input data and produces output data after a delay that is specific to the IP core. The output data of the IP core is passed to the write unit. An IP core is often implemented in a pipeline fashion to provide high throughput. By employing pipelined IP cores, execution of subsequent independent process firings can overlap in time, thereby increasing the process' throughput.

The write unit is connected to the outgoing channels of the process. For each output argument of the process' function F, a demultiplexer selects the appropriate output channel to which the argument is written. The selection is driven by the write unit's evaluation logic block, which functions similarly to the evaluation logic of the read

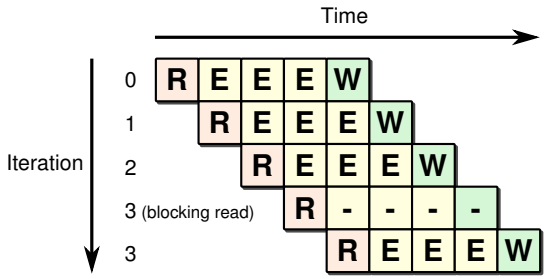


Figure 2.13: Pipelined execution of a LAURA processor containing a 3-stage IP core.

unit.

The control unit is responsible to implement the blocking read and write behavior of a PPN process. It enables or disables the read, execute, and write units based on information provided by those three units. The read unit reports a blocking read condition if one of the selected input channels does not contain any tokens. During a blocking read condition, the read unit is stalled while the execute and write units may still continue to process iterations pending in the pipeline. The write unit reports a blocking write condition if one of the selected output channels does not have sufficient room to store another token. During a blocking write condition, all units are stalled until the external consumer process clears the blocking write condition by reading a token from the full channel. The delay of an IP core may vary per firing of a process. For example, the delay of a variable length encoder IP core may depend on the input data. To integrate such IP cores in LAURA, the IP core should indicate when it is ready to accept or produce data. The execute unit forwards this information to the control unit, which then enables the read and write units accordingly.

CHAPTER 3

SYNTHESIZING PPNS

In Chapter 2, we introduced the Polyhedral Process Network model of computation and the PNGEN tool flow which automatically derives PPNs from sequential static affine nested loop programs written in C. We then introduced the ESPAM tool which employs the LAURA model to obtain synthesizable RTL implementations of PPNs. In this chapter, we focus on optimizing the RTL in the aforementioned tool flow. We first investigate shortcomings of the current state-of-the-art techniques and then propose extensions to facilitate more efficient RTL implementations.

3.1 Motivation & Contributions

When implementing industrially relevant applications, such as the sphere decoder application discussed in Chapter 6, and when applying transformations discussed in Chapter 5, we encountered four limitations of the LAURA model and the ESPAM tool. These limitations comprise characterization of functions, incorporation of novel front-end optimizations, handling of more complex domains, and handling out-of-order communication. In this chapter, we present solutions to these four limitations.

First, in the original work describing the LAURA model, only the delay metric of an IP core was considered [ZSKD03, NSD08a]. Such a simplified characterization does not suffice when integrating IP cores generated by HLS tools or when reasoning about system composition. In Section 3.2, we therefore present a more elaborate characterization of IP cores.

Second, the PN tool performs several optimizations that were not taken into account in the original LAURA model. In Section 3.3 and 3.4, we show how data reuse and sticky FIFO optimizations can be leveraged in the LAURA model to obtain more

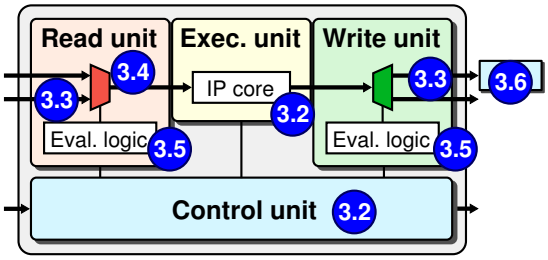


Figure 3.1: Position of the contributions of this chapter in the LAURA model.

efficient implementations.

Third, for complex iteration domains, the evaluation logic of a LAURA processor may become part of the critical path limiting the maximum achievable clock frequency of a system. As a result, the overall throughput of the system is limited. In Section 3.5, we investigate two different approaches to reduce the degradation of the maximum achievable clock frequency.

Fourth, applications with reordering communication could not be implemented using the ESPAM tool. Moreover, the known reordering buffer implementations suffered from read and write penalties with regards to non-reordering buffers [ZTKD02]. In Section 3.6, we present a new reordering buffer design with single-cycle read and write latencies that has been integrated in ESPAM. The particular design enables effortless integration in ESPAM-generated MPSoCs with point-to-point communication. In Section 3.7, we summarize this chapter. The positions of the contributions to the LAURA model have been indicated in Figure 3.1.

3.2 IP Core Characterization

The original LAURA model assumes that the IP core that is integrated into the execute unit comes from an external library. Such a library contains IP cores for different functions and possibly multiple IP cores for the same function that differ in performance and resource cost metrics. Being able to characterize an IP core in a concise way is important when considering performance estimations of PPNs in Chapter 4. To systematically distinguish between different IP cores which possibly implement the same function, we introduce the notion of a function implementation.

Definition 3.1 (Function Implementation)**.**
A *function implementation* is a particular implementation of a process function F. A function implementation is characterized by

- a latency Λ_F and
- an initiation interval II_F,

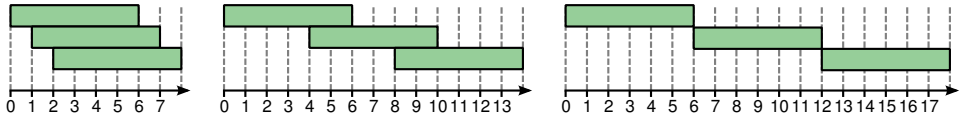
where $\Lambda_F \in \mathbb{N}^+$ is the input-to-output delay in clock cycles, and $II_F \in \mathbb{N}^+$ is the initiation interval in clock cycles.

The delay Λ_F represents the time between the start of a function execution and the moment at which all output has been produced. In Figure 3.2c, we show a time line of three sequential executions of a function implementation with $\Lambda_F = 6$.

The *initiation interval* II represents the amount of time between successive starts of a function implementation. Figure 3.2a depicts a function implementation with $II_F = 1$, allowing an execution of a function to be started every clock cycle. As a result, different executions of the function overlap in a pipeline fashion. Figure 3.2b depicts a function implementation with $II_F = 4$, allowing an execution to be started only every four clock cycles. The amount of overlap between different executions is less than the previous scenario. Figure 3.2c depicts a function implementation with $II_F = \Lambda_F = 6$, resulting in fully sequential executions of the function. This scenario resembles a non-pipelined function implementation. In this thesis, we set $II_F = \Lambda_F$ to model an implementation on a programmable processor on which no overlapped execution of function invocations occurs. A low II implies that the function implementation can deliver a high throughput. However, a low II reduces the opportunities for resource sharing inside a function implementation, resulting in higher resource cost compared to function implementations with a higher II. As such, the II is a key tool in trading off throughput and resource cost of the function implementation.

3.2.1 IP Core Integration

The function implementations in the IP core library may originate from various sources. The corresponding IP cores may be implemented in RTL manually, or the RTL can be automatically derived from a high-level specification using HLS tools. We have successfully implemented IP cores generated by the PICO [Syn10], AutoESL [Xil11], and DWARV HLS tools [YBK+07]. The RTL generated by PICO and



a) $II_F = 1$. b) $II_F = 4$. c) $II_F = \Lambda_F$.

Figure 3.2: Different initiation intervals for an IP core with delay $\Lambda_F = 6$.

AutoESL can be integrated in a straightforward way by connecting the clock, reset, enable, and data ports to the execute unit [HK09]. The RTL generated by DWARV assumes a shared memory model which is different from the distributed memory model employed in the PPN context. Therefore, integrating DWARV cores requires an additional wrapper which transfers data to and from a memory that connects to the DWARV core [NHS+11].

HLS tools such as PICO or AutoESL characterize a generated fixed-latency core by its latency Λ and initiation interval II [Fin10]. In the original LAURA model, only the latency was taken into account and the II value was assumed to be one. To integrate a fixed-delay IP core characterized by Λ and II values, we have extended the LAURA model to take IP cores with $II > 1$ into account. Both Λ and II are incorporated in the control unit of the generated LAURA HDL. Using the delay value, the control unit enables the write unit at the appropriate times, that is, when valid data is produced by the execute unit. Using the II value, the control unit enables the read unit only at valid II boundaries.

Function implementations with a variable delay cannot be characterized accurately by a single number. Instead, a designer may choose to set Λ to the average or worst-case delay value for performance analysis purposes. When integrating a variable-delay IP core, the values Λ and II are not taken into account in the LAURA HDL. Instead, the control unit requires the IP core to indicate when it is ready to accept or produce data.

3.3 Data Reuse

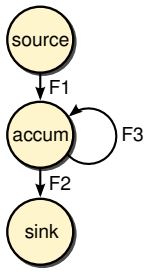
In applications such as filters, often a variable or array element is written once and subsequently read multiple times. For example, the array element `a[1]` in Figure 3.3a is written once when $i = 1$ and read when $j = 1$ (for argument `a[j]`) and $j = 2$ (for argument `a[j-1]`). In a PPN derived from the C code, both reads of `a[1]` are performed by the *accum* process. For the relation from *source* to *accum*, the compiler detects *data reuse*, which means the same token is read more than once from this relation.

A PPN derived from the C code using PNGEN is shown in Figure 3.3b. Channels *F1* and *F3* implement the data reuse channel pair for the relation from *source* to *accum*. Channel *F1* is a regular FIFO which transfers a token when *accum* needs it for the first time. Channel *F3* is a regular FIFO which propagates the token to subsequent iterations of *accum*.

In Figure 3.4, we depict part of a LAURA processor for the *accum* process of Figure 3.3c. Its read unit contains two multiplexers. The lower multiplexer passes tokens

```
for (i=0; i<5; i++) {
  source(&a[i]);
}
for (j=1; j<5; j++) {
  accum(a[j], a[j-1], &b[j]);
}
for (k=1; k<5; k++) {
  sink(b[k]);
}
```

a) C code.



b) PPN with data reuse.

Figure 3.3: A program with data reuse.

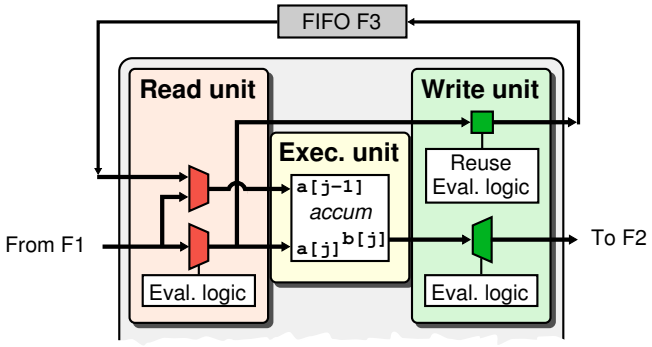


Figure 3.4: Handling data reuse in a LAURA processor.

from FIFO *F1* to the first input of the *accum* IP core. The upper multiplexer selects between FIFO *F1* that is read during the first iteration and FIFO *F3* that is read during subsequent iterations, and passes the token to the second argument of the IP core. The write unit contains a single demultiplexer which propagates the IP core output to FIFO *F2*. To handle the reuse, we extend the write unit with another output port connected to FIFO *F3*. The output port is driven by the first input to the IP core. A separate reuse evaluation logic block ensures that only tokens that need to be propagated to subsequent iterations are written to *F3*. The reuse evaluation logic block duplicates the expressions from the write unit's evaluation logic for the reuse ports to select the correct output port. Tokens that are reused in subsequent iterations can be written to *F3* immediately after reading them, irrespective of the IP core latency. We therefore connect the counters of the read unit to the reuse evaluation logic block.

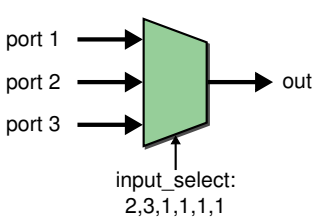
3.4 Sticky FIFOs

As an optimization of data reuse, PNGEN can classify a data reuse channel pair as a sticky FIFO. If the same token is transferred over a FIFO to multiple subsequent iterations of a process, then PN classifies the FIFO as a sticky FIFO and removes the selfloop. During a regular read operation on a sticky FIFO, the receiving process stores the token in a register. Subsequent iterations that need the same token then read from the register instead of the FIFO. This reduces inter-process communication and the number of write operations the producing process has to perform.

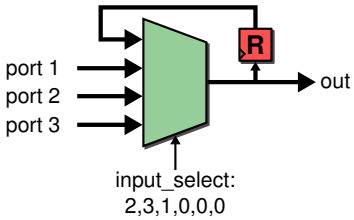
We implement a sticky FIFO by replacing the read multiplexer of a function argument with a "sticky read multiplexer". In Figure 3.5, we illustrate both types of read multiplexers. Figure 3.5a depicts the situation where all of the three input ports of the read multiplexer are connected to regular FIFOs. The read unit's evaluation logic block drives the input_select port of the multiplexer. The output of the multiplexer is propagated to the execute unit. In the example of Figure 3.5a, we first read a token from port 2, then a token from port 3, and then four tokens from port 1, as indicated by the sequence below the input_select port.

Figure 3.5b depicts the situation where port 1 is connected to a sticky FIFO. The output of the multiplexer is both propagated to the execute unit and written into register **R**. The output of register **R** is an additional input to the multiplexer. This additional input is selected when input_select is set to zero. This is illustrated by the sequence below the input_select port. We first read a token from port 2, then a token from port 3, and then a token from port 1. Then, input_select is set to zero which means we reuse the token read from port 1 that is still in **R**. As a result, the process writing to port 1 has to write the token only once.

Since the register is connected to the output of the multiplexer, it also stores tokens read from other ports that can be connected to any type of channel. However, tokens from non-sticky FIFOs are never read from the register, since the semantics of a



a) Port 1 connects to a regular FIFO.



b) Port 1 connects to a sticky FIFO.

Figure 3.5: Read multiplexer architecture.

sticky FIFO ensure that a regular read access is always performed before the token in the register is reused. For the example of Figure 3.5b this means that a zero in the input_select sequence is always preceded by a one, potentially with more zeros in between. Therefore, we do not need a separate register for each sticky FIFO port, but use a single register connected to the multiplexer output.

3.5 Evaluation Logic Optimizations

The main purpose of a LAURA processor is to route tokens from different process ports to the IP core during the appropriate process iterations. The evaluation logic blocks of a LAURA processor select the process ports that are accessed during a given iteration. The evaluation logic is driven by a set of cascaded counters that iterate through the points of the process iteration domain. At each iteration point, an expression is evaluated for each process port. When the expression evaluates to true, the port is accessed in the current iteration. The result of the evaluation is forwarded to the read multiplexer or write demultiplexer of the LAURA processor. In Figure 3.6, we illustrate the internal structure of the evaluation logic by considering the read unit's evaluation logic of Figure 2.12 in more detail. Only one counter is present, because the domain of the process is one-dimensional. The evaluation logic contains an expression for each of the two input ports. Port 1 is accessed during the first five iteration points, as denoted by the bit string in the right part of Figure 3.6. Port 2 is accessed during the remaining five iteration points.

We have identified two problems with the evaluation logic of a LAURA processor. First, the evaluation logic may affect the maximum achievable clock frequency of a LAURA processor, as the expressions become part of the critical path. Second, expressions containing for example `max` or `div` operators are nontrivial to implement. These problems becomes apparent when considering the scheduling transformation discussed in Section 5.1.4, as illustrated in for example Figure 5.11.

We address the first problem by pipelining the evaluation logic, as discussed in Section 3.5.1. We address the second problem by implementing the evaluation logic using ROM tables, as discussed in Section 3.5.2.

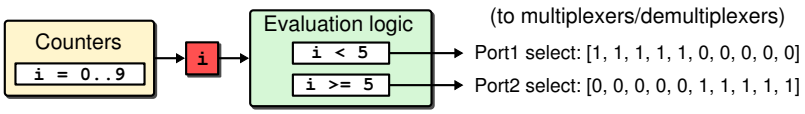


Figure 3.6: Evaluation logic block of a LAURA processor.

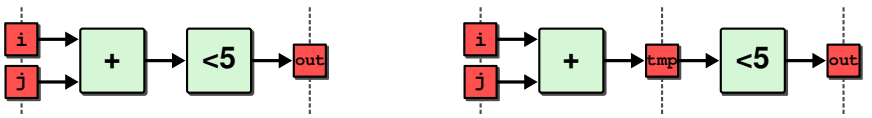
3.5.1 Pipelined Evaluation Logic

To achieve a higher clock frequency, we break long combinational paths into shorter combinational paths that are connected by registers. In Figure 3.7, we illustrate this for the expression $i + j < 5$. Without pipelining, the maximum combinational path length is two because the comparison is connected directly to the addition. In Figure 3.7b, we insert a register between the addition and the comparison. As a result, the maximum combinational path length is reduced to one and therefore the clock cycle period for this circuit can be decreased. However, the evaluation of the expression now takes two clock cycles. Only if subsequent evaluations can execute in an overlapped fashion then a throughput rate of one operation per clock cycle can be sustained at a clock frequency that is higher than the original clock frequency.

The advantage of this solution is that the maximum clock frequency of a LAURA node can be increased at the expense of only a small amount of registers. A disadvantage of this solution is that deciding the amount and insertion points of registers is a non-trivial task. Moreover, control dependencies inside the LAURA model and control dependencies between LAURA processors and other processing or communication components of a system do not allow for unlimited insertion of registers. We have found that pipelining the evaluation logic by one level is still possible.

3.5.2 ROM-Based Evaluation Logic

To implement any non-parametric evaluation logic, we can always resort to a table based implementation. We obtain this table by evaluating all expressions at compile-time and storing the results in a Read-Only Memory (ROM). This technique has already been presented by Derrien et al. [DTZ+05], but was not available in the Daedalus design flow. Derrien et al. already found that ROM based evaluation logic is more expensive in terms of resources than expression based evaluation logic. When realizing designs, we favor expression based evaluation logic, and only use ROM based evaluation logic when expression based evaluation logic requires operators like `max` and `div`, as these operators are not trivial to implement in RTL. Within Daedalus, we can select per processor whether to use expression based evaluation logic or ROM



a) Original: path length of 2 operations. b) Pipelined: max. path length of 1 operation.

Figure 3.7: Expression pipelining.

based evaluation logic.

For each iteration in the process domain, the ROM contains a word that specifies which ports need to be accessed. In a straightforward implementation of ROM-based evaluation logic, all port selection signals for each iteration of the process domain are stored in a table E. For a read or write unit of a process p connected to n ports, such a table E requires

$$n \cdot |D_p| \tag{3.1}$$

bits, where $|D_p|$ is the cardinality of p's process domain. However, many streaming applications exhibit repeating patterns in the ports accessed during subsequent iterations. Like [DTZ+05], we compress such repetition by applying a run-length encoding on the ROM data. This requires an additional table R containing the repetition count of each word in table E.

In Figure 3.8, we show the read unit's evaluation logic of Figure 2.12 implemented using ROM containing run-length encoded port selection patterns. Contrary to Figure 3.6, the evaluation logic block now contains two ROMs instead of a set of expressions. The first ROM shown at the bottom of the evaluation logic block contains table E. A column in this ROM represents the ports that are selected during a set of subsequent iterations. For example, the first column contains the sequence $[1, 0]^T$, meaning the first port is selected while the second port is deselected. The second ROM shown at the top of the evaluation logic block contains table R. It specifies the amount of times each column in E has to be repeated. In Figure 3.8, table R contains $[4, 4]$, meaning that both columns in E should be repeated four times. Thus, the first column is considered in total five times, and then the second column is considered five times. At run time, this results in port 1 being accessed five times, followed by port 2 being accessed five times, as illustrated by the bit strings at the right part of Figure 3.8.

The resource cost of a compressed ROM-based evaluation logic block mainly depends on the sizes of tables E and R. The size of E depends on the number of entries and the number of ports. The size of R depends on the number of entries and the number of bits required to store the largest repetition count occurring in R. This

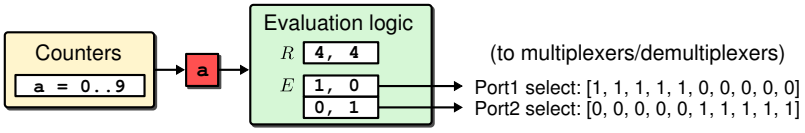


Figure 3.8: Evaluation logic block of a LAURA processor implemented using ROM.

Process - Unit	n	$\lvert D_p \rvert$	$\lvert R \rvert$	$\max(R)$	ROM Size (bits)		
					Uncompr.	Compressed	%
zero-Wr	2	28	13	5	56	65	+16
read-Wr	2	147	42	5	294	210	-29
vectorize-Rd	4	147	42	5	588	294	-50
vectorize-Wr	3	147	42	5	441	252	-43
rotate-Rd	5	441	231	4	2205	1848	-16
rotate-Wr	4	441	212	4	1764	1484	-16
sink-Rd	2	28	13	5	56	65	+16

Table 3.1: Individual ROM sizes for QR decomposition with $K = 21, N = 7$.

yields a total ROM size of

$$|R| \cdot n + |R| \cdot w \tag{3.2}$$

bits, where n is again the number of ports and $w = \lceil \log_2 \max(R) \rceil$. The size of a compressed evaluation logic block may be larger than the size of an uncompressed evaluation logic block in case

$$n \cdot |D_p| \; < \; |R| \cdot n + |R| \cdot w. \tag{3.3}$$

To assess whether this occurs in practice, we consider the QR decomposition application which exhibits complex port selection patterns that reduce compression effectiveness.

In Table 3.1, we show statistics for the individual ROMs of the five processes constituting a QR decomposition application. For example, the third row corresponds to the read unit for the vectorize process. An uncompressed ROM for the vectorize read unit requires $3 \cdot 147$ bits according to Equation (3.1). The compressed ROM requires $42 \cdot 4 + 42 \cdot \lceil \log_2(5) \rceil$ bits according to Equation (3.2). Applying the compression technique to the "zero" and "sink" processes results in ROM sizes that are larger than the sizes of their uncompressed counterparts. This can be attributed to the small domain sizes of these processes. Because each pattern is repeated at most twice, the overhead of table R outweighs the benefits of a smaller number of entries in E.

In Table 3.2, we show the total ROM size with and without using compression for instances of the QR decomposition application. In all cases except the first, the compression technique leads to reduction of the memory cost. For larger values of parameters K and N, the iteration domain sizes of the processes increase. This results in a larger reduction, because the number of additional bits required to store

Parameters		Uncompressed	Compressed	Reduction
K	N	(bits)	(bits)	(%)
3	3	222	226	+1.8
4	4	400	386	-3.5
21	7	5404	4218	-21.9
16	8	5328	3748	-29.7
16	16	20128	8820	-56.2
64	16	78880	34116	-56.7
256	64	4800640	685316	-85.7

Table 3.2: Total ROM sizes for different QR decomposition instances.

higher repetition counts increases more slowly than the number of additional points in the iteration domain.

The worst case for which run-length encoding does not yield any gains is when alternating between two ports. In such a case, the ROM size approaches $n \cdot |D_p|$ bits. The cost of repetition count table R should be added to this, yielding a "compressed" ROM whose size may exceed the size of the uncompressed ROM. However, alternating port selection patterns can often be handled easily using LAURA's conventional expression-based evaluation logic. Therefore, we do not need a ROM-based solution for such cases.

3.5.3 Related Work

All case studies conducted in this dissertation (cf. Chapter 5), the evaluation logic could be successfully implemented in either a pipelined or a ROM-based fashion. However, for applications demanding a clock frequency close to the platform limits, neither a pipelined nor a ROM-based evaluation logic implementation may suffice. In particular the application studied in Chapter 6 demands a high clock frequency of 225 MHz which neither pipelined nor ROM-based evaluation logic can provide. In such a case, one may leverage existing work on control generation. However, this may require non-trivial integration efforts, because the architectures in which the related works are used differ from the LAURA architecture. We present three alternative works that may be considered when further improving the LAURA evaluation logic components.

The CLooGVHDL tool generates a VHDL controller which traverses the points of a set of polytopes according to a predefined order [DBC+07]. The controller consists of a set of communicating automata that iterate over the dimensions of the polytope.

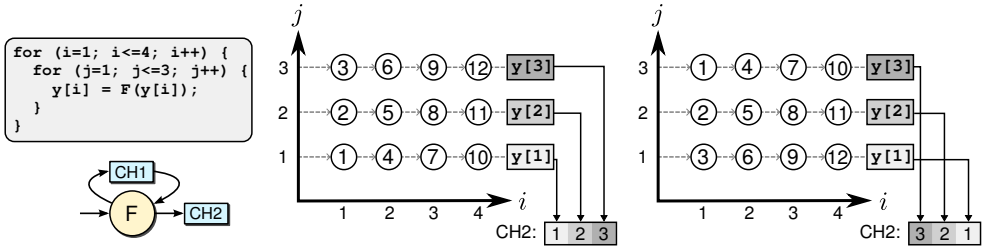
By placing registers between the automata, the maximum achievable clock frequency can be increased. Parallel execution of multiple instances of statements was left as future work. This would be of interest to us, since such parallel execution occurs in the LAURA architecture.

PARO attempts to reduce the resource cost of control logic by identifying counters and control signals that can be shared across different processors [DHRT07]. This approach was shown to lower resource cost particularly for partitioned applications, since the different partitions still have parts in common. However, the efficacy of this is limited for PPNs implemented using LAURA processors because of the globally asynchronous nature of the PPN model. That is, although two processes may share the same process domain and thus have similar control logic, they do not necessarily traverse their domains at the same pace.

Another alternative for the evaluation logic components of a LAURA processor is to implement them using existing HLS tools such as AutoESL [Xil11] or SynphonyC [Syn10]. This has the advantage that a target clock frequency can be specified. The HLS tool then produces a pipelined controller that is optimized for the specified clock frequency. However, we found that in practice the output of such tools have difficulties with the read and write units of a LAURA processor being decoupled [HK09]. For example, stalling the generated controllers on a blocking read condition was not fully supported at the time of our investigation. When such implementation problems have been resolved by HLS tool vendors, using an HLS tool to generate the evaluation logic might be the most favorable alternative solution.

3.6 Out-of-Order Communication

Ideally, a producer process produces tokens in the same order as the consumer process consumes them. Such *in-order* communication allows the channel from producer to consumer to be realized using a relatively inexpensive FIFO buffer. However, the PPNs of some applications do not exhibit solely in-order communication, as explained in Section 2.3.1. On some channels the order in which tokens are produced by the producer process may be different from the order in which tokens are consumed by the consumer process, and vice versa. Such communication is known as *out-of-order communication*. Out-of-order channels cannot be realized using FIFO buffers, because the token order needs to be taken into account to guarantee functional correctness. Instead, more sophisticated interconnects are required, such as *reordering buffers*. Reordering buffers store incoming tokens in order in a private memory and contain reordering logic which outputs the stored tokens in the order required by the consumer. Alternatively, circular buffers with overlapping windows



a) C program and PPN. b) Execution according to original program. c) Execution with inner loop reversed.

Figure 3.9: Two executions of a program with different communication behavior.

can realize out-of-order communication [BBS09]. This solution requires modifications to the producer and consumer process synchronization primitives. The impact on performance and resource cost of these modifications, and the performance and resource cost of the buffer itself is unclear, as no RTL implementation case study has been conducted yet.

In Figure 3.9, we show an example C program and two valid executions of this program. In the first execution shown in Figure 3.9b, we follow the execution order of the original program. That is, we first execute $(i, j) = (1, 1)$, followed by $(1, 2)$, etc. The relative order of iteration executions is illustrated by the number inside the points of Figure 3.9b. Only when $i = 4$, tokens are written to channel *CH2*. Channel *CH2* receives tokens in the order `y[1]`, `y[2]`, `y[3]`. Another valid execution in which the inner loop is traversed in the reverse direction is shown in Figure 3.9c. As a result, channel *CH2* receives tokens in the order `y[3]`, `y[2]`, `y[1]`, which is different from the order shown in Figure 3.9b. If we assume that *CH2*'s consumer process is not modified, the tokens would arrive in reverse order if *CH2* would be implemented using a FIFO buffer. To respect the correct token order, channel *CH2* has to be implemented using a reordering buffer.

Turjan et al. have proposed different realizations of reordering buffers, such as linear, pseudo-polynomial, and Content Addressable Memory (CAM) based implementations [TKD03]. The authors showed that these reordering buffer designs have a considerable negative impact on performance and resource usage. For example, read and write operations of a CAM implementation take four and two clock cycles [ZTKD02], respectively, while read and write operations on a regular FIFO take only one clock cycle.

To avoid counteracting the benefits of an application transformation because of possible reordering communication, we have developed a new reordering buffer [HK12]. The primary difference with previous work is that read and write operations now take

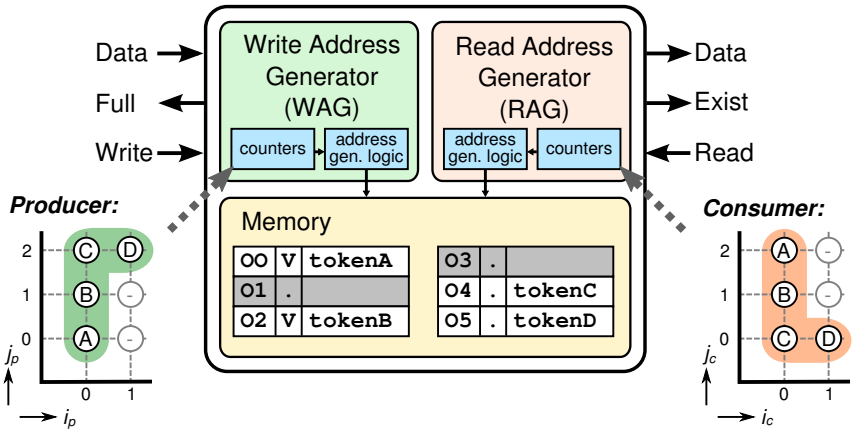


Figure 3.10: Reordering buffer.

only one clock cycle. This means that replacing a FIFO buffer with a reordering buffer increases resource usage, but does not introduce additional delay cycles.

Our reordering buffer is composed of a Write Address Generator (WAG), a Read Address Generator (RAG), and a private memory. The memory is dual-ported, with one port being addressed by the WAG and the other port being addressed by the RAG. The WAG and RAG both contain a set of counters which iterate through domains associated to the channel. These counters are used by the address generation logic to compute the next write and read addresses. To avoid delay cycles, the counters and address generation logic are implemented in a pipeline fashion. To minimize the latency of the address generation logic, we employ a linear addressing scheme. This addressing scheme is based on conventional linearization of an n-dimensional array into a 1-dimensional array. As such, the resulting address expressions are linear polynomials that can be realized efficiently in hardware.

The interface of the reordering buffer resembles a point-to-point FIFO buffer interface. This allows straightforward integration of reordering buffers in ESPAM-generated PPN implementations. That is, when a transformation introduces out-of-order communication, we do not have to modify the interfaces of the processes involved in the out-of-order communication. The interface is depicted in Figure 3.10. The outgoing slave interface exposes an output data bus, an exist signal to indicate if a token is available, and a read signal to acknowledge a read operation. The incoming master interface exposes an input data bus, a full signal to block write operations when the buffer is not ready to accept them, and a write signal to acknowledge a write operation.

We illustrate the memory organization of our reordering buffer at the bottom part of Figure 3.10. In the bottom left, we show a producer domain consisting of four points $(0,0)$, $(0,1)$, $(0,2)$, and $(1,2)$. The producer produces four tokens in the order A, B, C, D. We store these tokens according to a linear addressing scheme at address

$$\mathit{wAddr}(i_p, j_p) = i_p + 2 \cdot j_p. \tag{3.4}$$

The slot for each token is shown in the memory of Figure 3.10. For example, token C is produced in iteration $(0,2)$ and is therefore stored at address `04`. Because of the linear addressing scheme, some addresses may remain unused for non-rectangular domains. In our example, this occurs for addresses `01` and `03`. The consumer domain shown on the bottom right consumes the four tokens in the order C, D, B, A. To retrieve these tokens in the correct order from the memory, we compute

$$\mathit{rAddr}(i_c, j_c) = \mathit{wAddr}(M_{p \to c}(i_c, j_c)) \tag{3.5}$$

for each point in the consumer domain. That is, we first apply the channel relation $M_{p \to c}$ as found by the PN compiler. This gives the point (i_p, j_p) in the producer domain that corresponds to the point (i_c, j_c) in the consumer domain. We then compute $\mathit{wAddr}(i_p, j_p)$ to obtain the address from which the token should be read. For the example of Figure 3.10, PN finds the channel relation

$$M_{p \to c}(i_c, j_c) = \begin{bmatrix} i_p \\ 2 - j_p \end{bmatrix}. \tag{3.6}$$

Therefore, the read address function becomes

$$\mathit{rAddr}(i_c, j_c) = i_c + 2 \cdot (2 - j_c). \tag{3.7}$$

For token C, which is consumed in iteration $(0,0)$, the *rAddr* function yields address `04` which is the same address that was computed by the WAG. However, a token may not have been written by the producer yet. For example, token C may not be available yet at address `04`. Therefore, we introduce an additional valid bit for each memory location. The valid bit is set once a token has been written to its address. To comply with the blocking read semantics of the PPN model, the RAG blocks until the token corresponding to the current consumer iteration is written. In the memory of Figure 3.10, tokens A and B have been written, as indicated by the "v"s, whereas tokens C and D have not been written yet, as indicated by the "."s.

3.7 Conclusion and Summary

To realize the complete forward synthesis flow from a C specification to an FPGA implementation (cf. Figure 1.3), we have presented four extensions to the LAURA methodology in this chapter. These extensions include a more flexible characterization of IP core performance and resource cost aspects; support for novel optimizations of the PNGEN tool flow; architectural optimizations to improve the maximum clock frequency and handle complex iteration domains; and a novel reordering buffer implementation that has a lower performance penalty compared to previous reordering buffer implementations. The extensions enable the Daedalus tool flow to support transformations and cope with industrially relevant applications, as we show in the next chapters.

CHAPTER 4

PERFORMANCE ESTIMATION

In the previous chapter, we have presented methods to realize an FPGA implementation of a polyhedral process network. Considering a single optimized design point does not necessarily result in the best tradeoff between area and performance aspects. Instead, a designer wants to consider different design points that provide different tradeoffs between for example area and performance aspects. In this chapter, we present four different methods to estimate the performance of design points specified as polyhedral process networks, that differ in accuracy and assessment effort.

4.1 Motivation

Looking at Figure 1.2, which shows the iterative design flow, it becomes clear that the designer gets feedback very late. Only after time-consuming synthesis and place-and-route steps does the designer get feedback about performance. This limits the number of design points that a designer can evaluate in a given amount of time. Since he can evaluate only a limited number of design points, assessing if his design constraints can be met is difficult and frustrating. Prototyping for example a sobel design consisting of five LAURA processors already takes about twenty minutes [HK09]. Also, the forward synthesis flow requires that synthesizable RTL for all components is available, which is often not the case at the early stage of a design process. Instead, the designer should obtain feedback faster, possibly at the expense of reduced accuracy, allowing him to avoid a time-consuming forward synthesis step if he knows a design will not satisfy his constraints. Ideally, a designer wants to know whether a design meets his constraints before entering the time-consuming forward synthesis step.

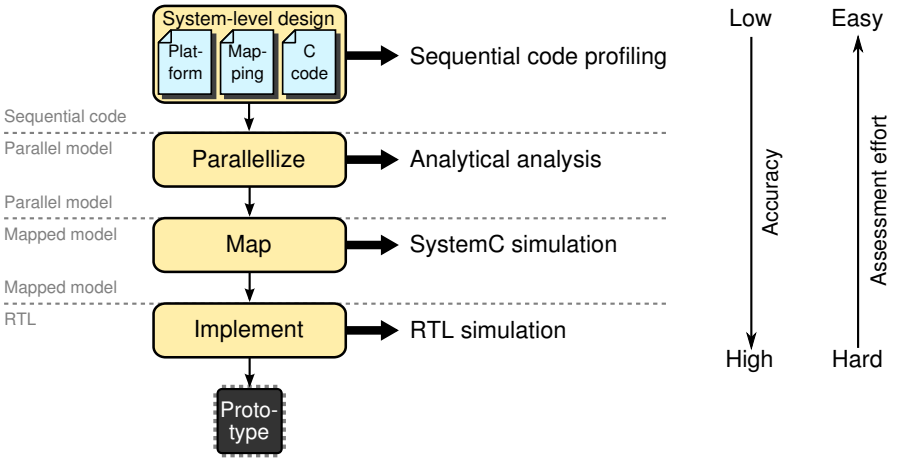


Figure 4.1: Performance assessment at different levels of the Daedalus design flow.

Getting an early performance estimate of a design is not new and has been investigated by for example Meijer et al. [MNS10] and Nikolov [Nik09]. However, these approaches only focus on microprocessor based systems. These approaches are not able to capture the notion of overlap between different iterations of a process and cannot handle cyclic PPNs, rendering them unsuitable for our design flow. Therefore, we investigate in this chapter four different techniques to provide the desired performance estimate.

From the Daedalus design flow, we distill four different levels, as shown in Figure 4.1. For each level, we investigate how to obtain a performance estimate. At the first level, the designer creates a system-level specification consisting of sequential C code and a platform and mapping specification in XML. At the second level, PNGEN parallelizes this C code into a parallel model. At the third level, ESPAM maps the parallel model onto a platform. At the fourth level, commercial synthesis tools implement the low-level RTL model.

We expect that performance assessments at these different levels provide different trade-offs between accuracy and assessment effort [HH96, KDWV02]. A high-level performance assessment can be conducted in a short amount of time, but such high-level performance numbers often deviate from the actual performance of the prototype. On the other hand, low-level performance assessments take a considerable amount of time, but the resulting performance numbers are often very close to the actual performance.

In this chapter, we study the relation between accuracy and assessment effort of performance assessments at the four different levels. We start our investigation from the

RTL at the fourth level and work our way up to the sequential code at the first level. We first present some definitions and concepts that we use to discuss performance estimations in Section 4.2. Performance assessment at the fourth level, RTL simulation, is discussed in Section 4.3. Performance assessment at the third level, SystemC simulation, is discussed in Section 4.4. Performance assessment at the second level, analytical analysis, is discussed in Section 4.5. Performance assessment at the first level, sequential code profiling, is discussed in Section 4.6. After the discussion of performance assessments at the four levels, we compare the four approaches on different aspects such as the ability to incorporate finite buffer sizes in Section 4.7. In Section 4.8, we compare the four approaches by applying each approach on a set of benchmarks. In Section 4.9, we summarize this chapter.

4.2 Definitions

Design constraints on system performance are often expressed as a constraint on throughput [SB00]. To quantify the performance of a process in a PPN, we employ the notion of throughput:

Definition 4.1 (Process Period and Throughput)**.**
The *period* T_p of a PPN process p represents the average time between two subsequent firings of p. The *throughput* $\tau_p = \frac{1}{T_p}$ of a process p represents the average number of firings completed per time unit.

Using the notion of process throughput, we define the throughput of a PPN:

Definition 4.2 (PPN Throughput)**.**
The throughput of a PPN with one sink process equals the throughput of that sink process.

This definition excludes PPNs with more than one sink process. This is not a fundamental limitation because each such PPN can be transformed into a PPN with only one sink process by merging the sink processes. Alternatively, instead of reasoning about the throughput of the entire PPN, one may keep the distinction between different sink processes since they represent different output streams of a system. For example, a video processing system may have a high data rate video output stream and a low data rate control output stream. Combining both data rates is meaningless in practice, and thus it is desirable to keep both throughput rates separated.

The external input and output streams connected to the PPN may affect the throughput achieved by a PPN. For example, if data on the input stream is not delivered fast enough, the throughput of a PPN may drop as the PPN has to wait for data. Similarly,

if data on the output stream is not consumed fast enough, the PPN may be stalled until older data is consumed from the output stream such that storage space for new data becomes available. In this chapter, we are interested in the throughput of a PPN irrespective of environmental factors. Therefore, we employ the notion of isolated throughput:

Definition 4.3 (Isolated Throughput)**.**
The isolated throughput of a PPN is the throughput of the PPN when isolated from external input and output streams.

As such, the isolated throughput represents the theoretical maximum achievable throughput considering only the PPN itself. In the remainder of this chapter we present and review four different techniques to estimate the isolated throughput of PPNs. We want to estimate the throughput of a PPN on a real system, which we refer to as the *absolute throughput*.

Definition 4.4 (Absolute Throughput)**.**
An absolute throughput assessment is used to describe the throughput of an actual FPGA implementation of a PPN.

The goal of each of the four techniques that we present in this chapter is to analyze the performance of a multi-processor system. According to van Gemund, the performance of a parallel system is determined by four key aspects [Gem96]:

- **Conditional synchronization**, which relates to the performance impacts of synchronization due to data dependences. For example, if a PPN process depends on two inputs a and b that become available at times t_a and t_b, then the process should fire no earlier than $\max(t_a, t_b)$.
- **Mutual exclusion**, which relates to contention of processing or communication resources. For example, a processor can only initiate the next PPN process iteration at a valid initiation interval (II) boundary.
- **Basic calibration**, which relates to the performance characteristics of the system constituents. For example, Definition 3.1 provides a systematic way to describe the throughput (II) and latency (Λ) of an IP core that is integrated in a PPN.
- **Conditional control flow**, which relates to non-static control flow inferred by data-dependent control statements. For example, a process function may have a varying latency if the function performs a different computation for different input argument values.

To obtain an accurate assessment of PPN performance, we take the four key aspects into account in the four performance estimation techniques that we present.

4.3 RTL Simulation

At the fourth level in Figure 4.1, we have obtained an FPGA project of the system. This project can be synthesized using vendor-specific low-level synthesis and place-and-route tools to obtain a bitstream. By downloading the bitstream onto an FPGA device, the designer obtains a prototype implementation of the design such that for example functionality and throughput requirements can be verified. However, even for small designs, synthesis of an FPGA project to a bitstream already takes tens of minutes. Obtaining a throughput metric by prototyping is thus a time-consuming approach.

The RTL representation of an FPGA project can be simulated such that low-level synthesis and place-and-route steps are avoided during throughput assessment. The feasibility of such a simulation depends on the types of processors in the platform specification. If one or more programmable processors are involved, an RTL simulation is time-consuming because of the large amount of effort required to simulate a single instruction at the register transfer level, making RTL simulation impractical when using programmable processors. Nevertheless, for platforms consisting entirely of LAURA processors, we found that RTL simulation is a viable approach to obtain a throughput estimate of a design. Therefore, we have extended ESPAM with a backend that produces an RTL simulation project for platforms that consist entirely of LAURA processors. This backend generates a simulation project for the Xilinx ISE simulator.

4.4 SystemC Simulation

At the third level in Figure 4.1, we have obtained a mapped model of the system. As discussed in the previous section, RTL simulation of platforms containing one or more programmable processors is often infeasible in practice. To make simulation of such platforms feasible, we may reduce the amount of simulation details at the expensive of lower accuracy. We achieve this by simulating the mapped model instead of the RTL model, thereby addressing the basic calibration and conditional control flow aspects in less detail. A common solution is to use different simulation techniques for different types of components, known as *co-simulation* [GCD92, Row94]. The different components are then simulated at different levels of detail. Another solution is to use execution traces of an application to simulate different system-level specifications, as done for example by Sesame [PEP06] which is integrated in Daedalus. A widely used standard for simulation of designs with reduced accuracy is the SystemC standard [Sys05]. We have extended ESPAM with two SystemC backends: an untimed SystemC backend and a timed SystemC backend.

The *untimed SystemC* backend generates a functional simulation in the SystemC environment. One of the first backends in the history of ESPAM was the YAPI backend which generates a functional simulation in the YAPI framework [KES+00]. The YAPI backend provides fast functional simulation of a PPN, such that a designer can quickly verify if the functional behavior of a parallelized application is correct. The motivation behind the untimed SystemC backend is to provide similar fast functional simulation, but according to an industry standard. Unlike the YAPI framework, SystemC is an official IEEE standard which implies a more widespread acceptance and better long-term support.

The *timed SystemC* backend generates a functional simulation which includes a notion of time. A designer can use a timed SystemC simulation to obtain throughput metrics in less time than with an RTL simulation. We have explored two different approaches to incorporate programmable processors into timed SystemC simulations. In Section 4.4.1, we present an approach which employs a cycle-accurate instruction set simulator which yields cycle-accurate throughput metrics. In Section 4.4.2, we present an approach which uses fixed execution time estimates, thereby potentially degrading accuracy but further increasing the simulation speed.

4.4.1 Cycle-Accurate Timed SystemC Simulation

To obtain a cycle-accurate simulation environment from a system level specification, we have developed a new backend to ESPAM [HHK10]. This backend generates the C++ code for a SystemC top-level module and the C++ code that has to be run on each processor. The backend currently supports LAURA and MicroBlaze processors. A LAURA processor is simulated using a custom written SystemC module that models the LAURA execution in a cycle-accurate manner. A MicroBlaze processor is simulated using the cycle-accurate GDB-based MicroBlaze *Instruction Set Simulator* (*ISS*) provided by Xilinx. Such an ISS allows for a faster performance assessment, because the ISS simulates only instructions instead of the full RTL implementation of the MicroBlaze processor. However, the MicroBlaze ISS was not designed to operate as a multi-processor simulator. Therefore, ESPAM generates a SystemC top-level module which allows different instances of the ISS to interact.

In Figure 4.2, we show an implementation of the PPN shown in Figure 2.8 using our cycle-accurate timed SystemC simulation model. We map the *source* and *sink* processes onto separate MicroBlaze processors, and the *func1* process onto a LAURA processor. The top-level module contains submodules that implement a simulation model for each processor. The channels of a PPN are implemented using *sc_fifo* primitives from the SystemC standard which interconnect the processor simulation modules.

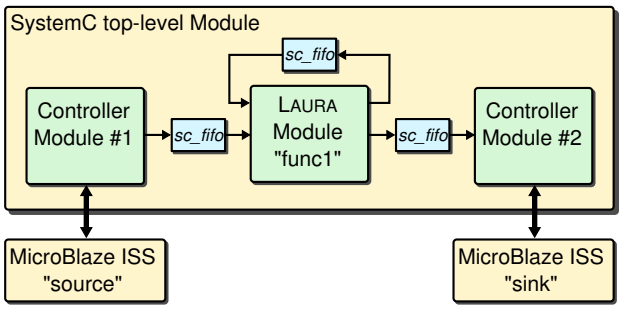


Figure 4.2: A cycle-accurate timed SystemC simulation environment for the PPN of Figure 2.8.

For each LAURA processor, the SystemC top-level module implements a SystemC module that simulates the execution of a process on a LAURA processor described in Section 2.4.1. For each MicroBlaze processor, the SystemC top-level module implements a controller module which drives each MicroBlaze ISS instance, shown in the bottom part of Figure 4.2. This controller module communicates with the ISS instance running as a separate heavy-weight process in the operating system. This allows different ISS instances to run in parallel. A process running on a MicroBlaze processor normally communicates data to other processors via its FSL ports, using the `get` and `put` instructions. The original MicroBlaze ISS does not implement these instructions. We have implemented these instructions in the ISS to send and receive data to and from the controller module associated with an ISS instance. The controller module subsequently transfers data between the other simulated processors using *sc_fifo* instances. The `get` instruction stalls the ISS when no data is available on the *sc_fifo* being read, and the `put` instruction stalls the ISS when the maximum capacity of the *sc_fifo* being written is reached. As such, the ISS implements the blocking read and write primitives according to the semantics of the PPN model.

To ensure cycle-accurate simulation, a global execution time should be maintained across all ISS instances. We have modified the ISS such that each ISS instance keeps track of the global execution time. A straightforward way to synchronize the execution times of all ISS instances is to use a *lockstep* approach. With such an approach, each ISS instance waits for a clock signal from the corresponding SystemC controller module before a MicroBlaze instruction is executed. The lockstep approach guarantees that all simulated MicroBlaze processors advance at the same pace, resulting in a cycle-accurate simulation of the system. Unfortunately, the lockstep approach results in extensive synchronization overhead, because synchronization occurs at every simulated clock cycle. Using the lockstep approach, at most 42000 clock cycles can be

simulated per second for a design containing one MicroBlaze processor. When more MicroBlaze processors are simulated simultaneously, this number drops more or less linearly. As an alternative to the lockstep approach, we can synchronize the execution times of two ISS instances only when these ISS instances interact by communicating a data token. Using such global execution time synchronization, we have observed up to 80 times increases in simulation speed.

4.4.2 Light-weight Timed SystemC Simulation

Performing a cycle-accurate timed SystemC simulation using ISSs is a delicate task, because code for all processors has to be compiled for the appropriate instruction set, and because communication channels between the ISS instances need to be established. As a light-weight alternative to cycle-accurate timed SystemC simulation, we propose another timed SystemC simulation technique. Instead of relying on an ISS to obtain cycle-accurate execution times, we require the designer to provide function execution times according to Definition 3.1. Thus, for each function f in the PPN, a value Λ_f represents the number of clock cycles taken by a single invocation of f. By considering only a single value Λ_f, we decrease simulation complexity at the expense of lower accuracy. As a result, we simplify the basic calibration aspect as only one Λ_f value per function is required, instead of a list of latency values per instruction. We ignore the conditional control flow aspect, as functions that may contain data-dependent statements are not executed. As a consequence of the simplification, the accuracy of Λ_f determines the accuracy of the final throughput metric.

For each top-level component in a system, that is, a processor in the platform specification and a channel between two processors, we instantiate a SystemC module. The SystemC module is based on a template for the component type. Each SystemC module runs a thread in which the simulation model of the simulated component is updated at each simulated clock cycle. A top-level module interconnects the SystemC modules and invokes the SystemC simulation kernel. The SystemC kernel schedules all threads according to a discrete-event simulation model that is also employed for RTL simulation. Light-weight timed SystemC simulation thus resembles RTL simulation in which only the conditional synchronization, mutual exclusion, and basic calibration aspects are included in the simulation model. Other aspects that do not affect performance significantly, such as the process functionality and input port multiplexing, are specified as C code without invoking the discrete event scheduler of SystemC. Thus, no simulation primitives are constructed for non-essential aspects, which allows faster simulation compared to RTL simulation.

For an example Sobel design mapped on a platform consisting of five processors, we have measured a simulation speed of about 200000 clock cycles per second. In

terms of simulation speed, this approach roughly compares to the cycle-accurate timed SystemC simulation without lockstep synchronization. Assuming the Λ_f values are accurate, light-weight timed SystemC simulation is a feasible alternative to cycle-accurate timed SystemC simulation.

4.5 Maximum Cycle Mean Analysis

At the second level in Figure 4.1, we have obtained a PPN of the application, which is a particular model of computation. Estimating performance for different models of computation is a well-established field of research [LSV98, SB00]. In this section, we want to leverage existing work to find an analytical performance estimation technique for PPNs. We present a novel analytical technique to estimate the throughput of a PPN based on *Maximum Cycle Mean* (*MCM*) analysis. MCM analysis is an established technique to assess the throughput of an HSDF graph [SB00]. MCM analysis is invariant to the application workload because of the analytical nature. This makes this approach appealing compared to the RTL and SystemC approaches, as the assessment effort of the latter aproaches directly depends on the workload. We present an overview of analytical throughput estimation approaches in Section 4.5.1. We discuss the MCM analysis method for HSDF graphs in Section 4.5.2. We explain how we derive an HSDF graph for throughput estimation of a PPN in Section 4.5.3. We conclude this section by applying MCM analysis to two PPNs in Section 4.5.4.

4.5.1 Related Work

Analytical performance assessment of applications modeled as dataflow graphs is a well-studied research field. An analytical method to compare different instances of an application modeled as a PPN was first presented by Meijer et al.[MNS10]. Their technique had two limitiations. First, the scope was limited to acyclic PPNs. Second, the throughput model was developed to obtain relative throughput assessments between two or more PPNs. In contrast, in this chapter we focus on absolute throughput assessments for both acyclic and cyclic PPNs. Thiele et al. have investigated performance analysis for cyclic SDF graphs [TS09]. Because the approach works only on SDF graphs, it cannot cope with varying production and consumption rates that occur in many embedded applications. Such varying rates can be expressed in the PPN model, but no absolute performance analysis currently exists for PPNs.

The period of an HSDF graph can be analytically obtained by computing the maximum cycle mean [DG98, SB00]. Because a PPN is a special case of a CSDF graph, an equivalent HSDF graph can be derived from a PPN using the *conventional method* with which an HSDF graph can be derived from a CSDF graph [BELP96, Fig. 9].

Moonen et al. use this method to compute a conservative bound on the throughput of a CSDF graph [MBBM07]. Unfortunately, the equivalent HSDF graph often exhibits an exponential increase in the number of nodes compared with the CSDF graph. This increases the running time of the algorithm computing the maximum cycle mean, making analysis of large graphs more time-consuming or even impractical. In Section 4.5.3, we present an alternative approach to enable maximum cycle mean analysis on PPNs which avoids the exponential complexity increase.

Ito and Parhi acknowledge the increases in the number of nodes and edges when deriving the equivalent single-rate data flow ("HSDF") graph for a given multi-rate data flow ("SDF") graph [IP95]. Their solution is to remove edges and nodes through procedures called edge degeneration and node degeneration, in such a way that the iteration bound is not affected. The effectiveness of the approach is not guaranteed, as node degeneration is not applicable for certain graphs, as indicated by the authors.

Instead of working on equivalent HSDF graphs, throughput analysis methods exist that operate directly on SDF [GGS+06] and CSDF graphs [SGB08]. These methods construct the state space of the graph by simulating its execution assuming an unlimited number of processor resources. Once a cycle is detected in the state space, the periodic phase is reached. After identification of the periodic phase, the throughput of the graph can be computed. Instead of performing an explicit state-space exploration, one can also perform the state-space exploration symbolically using max-plus algebra [Gei09]. This allows one to obtain an HSDF graph with identical throughput characteristics that often has fewer nodes than an equivalent HSDF graph obtained using the conventional method. However, for some graphs the method of [Gei09] may produce an HSDF graph that has more nodes than the conventionally obtained HSDF graph.

In summary, existing analytical throughput assessment techniques for PPNs cannot cope with cyclic graphs and are only intended for relative throughput assessment. Various techniques exist for HSDF, SDF, and CSDF graphs that can cope with cyclic graphs and provide absolute throughput assessment. However, techniques for HSDF, SDF, and CSDF graphs cannot be applied directly to PPNs, because the succinct PPN representation has to be converted into a more elaborate HSDF, SDF, or CSDF representation. Such a conversion leads to an HSDF or SDF graph with an exponentially large number of nodes, or a CSDF graph with long phase lengths. The conversion takes an large amount of time, and the size of resulting CSDF graph leads to long running times of the analysis methods. To avoid any potential exponential increase in the number of nodes, we look for an approach in which the number of nodes remains equal to the number of processes in the PPN.

4.5.2 Maximum Cycle Mean Analysis

The *iteration period* of an HSDF graph is defined as the time needed to execute an iteration of the graph [SB00, Chapter 5]. A lower bound on the iteration period, called the *iteration bound*, can be obtained by first computing the computation-to-delay ratios of the cycles in the graph [SB00, Chapter 8]. For each cycle C in an HSDF graph G, we compute the computation-to-delay ratio

$$CM(C) = \frac{\sum_{v \in C} t(v)}{\sum_{e \in C} d(e)}, \tag{4.1}$$

which we refer to as the *cycle mean* of C. Thus, the cycle mean of a cycle C equals the sum of the execution times $t(v)$ of all nodes v involved in C divided by the sum of all initial tokens $d(e)$ on the edges e involved in C. The cycle that yields the maximum $CM()$ value is called the *critical cycle* of an HSDF graph. The iteration bound of an HSDF graph G is determined by the critical cycle. Thus, to obtain the iteration bound we compute the *maximum cycle mean*

$$MCM(G) = \max\{CM(C)\}, \quad C \in G. \tag{4.2}$$

The throughput of an HSDF graph G is the reciprocal of the iteration bound, thus

$$\tau(G) = \frac{1}{MCM(G)}. \tag{4.3}$$

Example

We now illustrate the MCM analysis on the HSDF graph of Figure 2.5 on page 23. This graph contains three cycles:

- $c_1 = (a1 \rightarrow b \rightarrow c \rightarrow a1)$,
- $c_2 = (a2 \rightarrow b \rightarrow c \rightarrow a2)$, and
- $c_3 = (b \rightarrow c \rightarrow b)$.

When auto-concurrency is considered, a node may fire multiple times simultaneously. A node mapped onto a programmable processor executes its firings in sequence, such that no auto-concurrency occurs. To explicitly exclude auto-concurrency of the individual nodes, we assume each node i has a selfloop c_i with one initial token. Then, Equation (4.2) yields

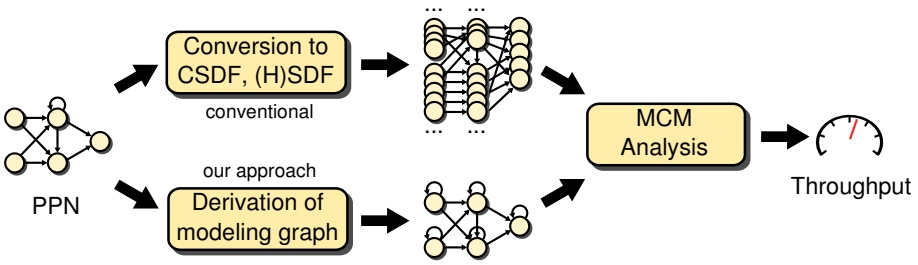


Figure 4.3: Two approaches to apply MCM analysis to PPNs.

$$
\begin{aligned}
MCM(G_{2.5}) &= \max\left\{CM(c_{a1}), CM(c_{a2}), CM(c_b), CM(c_c), CM(c_1), CM(c_2), CM(c_3)\right\} \\
&= \max\left\{\frac{t(a1)}{1}, \frac{t(a2)}{1}, \frac{t(b)}{1}, \frac{t(c)}{1}, \frac{t(a1)+t(b)+t(c)}{d(a1\to b)+d(b\to c)+d(c\to a1)}, \right. \\
&\qquad \left. \frac{t(a2)+t(b)+t(c)}{d(a2\to b)+d(b\to c)+d(c\to a2)}, \frac{t(b)+t(c)}{d(b\to c)+d(c\to b)}\right\} \\
&= \max\left\{\frac{8}{1}, \frac{8}{1}, \frac{2}{1}, \frac{2}{1}, \frac{8+2+2}{0+0+1}, \frac{8+2+2}{0+0+1}, \frac{2+2}{0+1}\right\} \\
&= 12\,.
\end{aligned}
$$

The first four terms in the max-expressions above correspond to the selfloops of the four nodes. The remaining three terms correspond to the three cycles of the graph. The maximum cycle mean is determined by both $CM(c_1)$ and $CM(c_2)$ which both evaluate to 12. Thus, the iteration bound of the HSDF graph of Figure 2.5 is 12, and consequently the throughput is $\frac{1}{12}$.

4.5.3 Derivation of PPN Modeling Graphs

In Section 4.5.1, we mentioned the possibility of applying MCM analysis on PPNs by considering the equivalent CSDF graph and converting the CSDF graph into HSDF. This approach is depicted in the upper part of Figure 4.3. Unfortunately, this results in an exponential increase in the number of nodes. To keep the time needed for the MCM computation within reasonable bounds, we must avoid the exponential increase in the number of nodes, which leads us to a new approach.

We have found a way to derive a more compact HSDF graph from a cyclic PPN, which we depict in the lower part of Figure 4.3. Our approach works by deriving an HSDF graph that models the throughput behavior of a PPN, and then applying conventional MCM analysis to this graph. The number of nodes in our HSDF graph

equals the number of processes in the PPN. The number of edges in our HSDF graph is linearly bounded, as we show in Proposition 4.3. As such, no exponential increase of the graph size occurs, making the approach a suitable alternative for fast performance estimation of PPNs. We divide the derivation in two main steps. First, PPN processes are converted to HSDF nodes. Second, PPN channels are converted to HSDF edges.

Step 1: Constructing Nodes from Processes

The first step in deriving the PPN modeling HSDF graph is to convert PPN processes to HSDF nodes. One possible approach is to interpret the PPN as a CSDF graph [HZ+10, adg2csdf] and then derive an HSDF graph from this CSDF graph using the conventional approach [BELP96, Fig. 9]. This approach causes $\mathbf{q}(p)$ nodes to be instantiated for each process p. For consistent PPNs, $\mathbf{q}(p)$ always equals the number of points in the process domain D_p:

Proposition 4.1 (PPN Repetition Vector)**.** *For each process p of a consistent PPN, the corresponding element of the repetition vector of an equivalent CSDF graph equals the number of points in its process domain, that is, $\forall p \in \mathcal{P} : \mathbf{q}(p) = |D_p|$.*

Proof. In a consistent PPN, for every channel c, the number of points in the corresponding $OPD^j_{\sigma_c}$ is equal to the number of points in the corresponding $IPD^k_{\delta_c}$, thus $|\mathrm{OPD}^j_{\sigma_c}| = |IPD^k_{\delta_c}|$. Therefore, the solution of the balance equation $\Gamma \cdot \mathbf{r} = \mathbf{0}$ is a vector $\mathbf{r}$ which contains a '1' for every process. As a result, the elements of the repetition vector $\mathbf{q} = S \cdot \mathbf{r}$ are equal to the phase lengths of each node, which equals the number of points in the process domain. □

As a result, a separate HSDF node would be instantiated for each iteration of the domain, resulting in large graphs even for small applications. This makes the conventional CSDF-to-HSDF approach infeasible for practical purposes. We can avoid an increase in the number of nodes based on the following observation. In an HSDF graph, all $\mathbf{q}(p)$ nodes originating from a process p may execute in parallel. However, by definition the iterations of a PPN always execute sequentially. This allows us to represent each process p by a single HSDF node h, where node h represents sequential execution of all $\mathbf{q}(p)$ nodes of the conventional equivalent HSDF graph. We multiply the execution time Λ_p of a single firing of process p by $\mathbf{q}(p)$ to model sequential execution of all $\mathbf{q}(p)$ nodes in the equivalent HSDF graph. As a result, the number of nodes in the resulting HSDF graph equals the number of processes in the original PPN.

The execution time $t(h)$ of an HSDF node is set to the total time needed to fire all iterations of the process consecutively without overlapped execution. Included in this execution time are the read and write latencies and the time needed to fire the function. Time spent on a blocking read or write operation is not included, which means our approach does not address the conditional synchronization aspect introduced in Section 4.2. Our approach cannot accurately assess throughput of applications in which read or write operations block on empty or full channels. To exclude auto-concurrency, we add to each HSDF node a selfloop with one initial token. This avoids multiple simultaneous executions of the entire PPN, which is undesirable when determining throughput.

Step 2: Constructing Edges from Channels

The second step in deriving the PPN modeling HSDF graph is to interconnect the HSDF nodes using edges in such a way that the PPN's throughput characteristics are preserved. This is not trivial, because of the different semantics of HSDF edges and PPN channels: HSDF edges have an unbounded capacity and may contain initial tokens, whereas PPN channels have a bounded capacity and do not have a notion of initial tokens. We now discuss how to represent edges in a PPN modeling HSDF graph such that the PPN's throughput characteristics are preserved.

The PPN modeling graph may contain more than one edge between two nodes a and b, if for example the PPN contains multiple channels between two processes. It is sufficient to represent such a collection of channels by a single edge:

Proposition 4.2 (Pruning Multi-Edges in PPN Modeling Graphs)**.** *A collection of PPN channels from process a to process b can be represented by a single edge $(a \rightarrow b)$ in the PPN modeling graph.*

Proof. If an edge $(a \rightarrow b)$ is part of a cycle, then another cycle also exists for each additional edge connecting a to b. The only difference among the cycle means of those cycles is the number of initial tokens that occurs in the denominator of Equation (4.1). A cycle with a larger denominator results in a smaller cycle mean, which implies the cycle mean will not be selected by Equation (4.2). Thus, we only need to consider the cycle with the smallest number of initial tokens, which is the cycle containing the edge with the smallest number of initial tokens. □

We distinguish between three classes of channels: selfloop channels, feedback channels, and feedforward channels.

a) PPN containing a cycle.

b) Corresponding PPN modeling graph.

Figure 4.4: Handling feedback edges in a PPN.

Selfloop channels

For a selfloop channel, which connects a process to itself, no edge is added to the HSDF graph. We omit such selfloops because in step 1 we have already added a selfloop with one initial token to each node. To see why selfloops can be omitted, suppose that the critical cycle of a PPN modeling graph would be a selfloop s of process p with buffer size $S_s \geq 1$. This selfloop could be modeled by adding an edge $(p \rightarrow p)$ with S_s initial tokens. For this newly added cycle C_s, Equation (4.1) yields

$$CM(C_s) = \frac{\Lambda_p \cdot |D_p|}{S_s}.$$

However, for selfloop e added in line 3 of Algorithm 4.1, we already have

$$CM(e) = \frac{\Lambda_p \cdot |D_p|}{1}.$$

Because $CM(e) \geq CM(C_s)$ for all $S_s \geq 1$, we can ignore $CM(C_s)$ in Equation (4.2). Thus, a selfloop of a PPN never forms the critical cycle, and therefore such selfloops can be omitted from the PPN modeling graph without affecting the MCM value.

Feedback channels

Feedback channels are part of a strongly connected component, and are thus the constituents of a cycle. The cycle mean of a cycle in the PPN modeling graph is computed using the sum of all initial tokens on the edges constituting the cycle, as we have shown in Equation (4.1). Hence, the amount of initial tokens on an HSDF edge representing a feedback channel may affect the MCM value of an HSDF graph. Therefore, we should determine the amount of initial tokens for each feedback edge such that an accurate MCM value is obtained.

Each cycle of a PPN contains one process that is the *first process* from that cycle to be fired. The channel of that cycle from which the first process reads is the *last*

channel of that cycle. Initially, we construct for each PPN channel part of a cycle an HSDF edge and assign zero initial tokens to each edge. Only to the edge corresponding to the last channel of the cycle we assign a nonzero number of initial tokens d. For example, suppose process a in Figure 4.4 first reads from a channel outside of the cycle and in the next firing reads from channel $(c \rightarrow a)$ that is part of the cycle $(a \rightarrow b \rightarrow c \rightarrow a)$. As such, process a is the first process of the cycle that can fire. Therefore, edge $(c \rightarrow a)$ is the last edge of the cycle. Selection between edges is not possible in the HSDF model which requires all incoming edges of a node to be read during every firing. Without assigning initial tokens to the last edge $(c \rightarrow a)$ of the cycle, the HSDF graph would be in a deadlock state, preventing meaningful analysis. To avoid this deadlock state, we assign initial tokens to the last edge.

Initial tokens on an edge of an HSDF graph are also referred to as the *delay* of an edge. Here, "delay" refers to the temporal distance between the nodes in terms of iterations of the graph. For example, if an edge $(a \rightarrow b)$ has 2 initial tokens, then the firing of node b at iteration i depends on the token produced by node a at iteration $i-2$. The PPN model does not have a notion of initial tokens, which means we need to relate the delay between two HSDF nodes that are part of a cycle to the distance between processes in the PPN model.

A notion of dependence distances is available for SANLPs from which we derive PPNs. A dependence distance vector gives the difference between a target iteration vector and the source iteration vector of a dependence [Pug92, definition *d*]. For a PPN channel $(a \rightarrow b)$, the distance vector gives the difference between an iteration of process b that consumes a token and the iteration of process a that produced the token. In general, this difference may not be defined when the process iteration domains are different, which for example happens when the original statements are not located in the same loop nest. However, the PNGEN tool flow puts all processes in a common iteration space to compute buffer sizes. In this common iteration space, the dependence distance vector is defined for any pair of processes that are connected by a channel. We therefore employ the dependence distance in the common iteration space to assign initial tokens to feedback edges in the PPN modeling graph.

We cannot use the dependence distance directly, because of the following two reasons. First, the dependence distance is a vector for common iteration spaces consisting of more than one dimension. In contrast, the number of initial tokens of an HSDF graph should always be a scalar value. Second, a dependence distance may be *non-uniform*, that is, the dependence distances may vary for different pairs of iterations. In such cases, the dependence distance of a single dimension cannot be expressed using a constant integer only, but is expressed using iterators. In contrast, the number of initial tokens of an HSDF graph should always be a constant integer value.

To overcome both problems, we use a constant integral scalar approximation of a

dependence distance. The way in which PNGEN computes the buffer size (cf. Section 2.3.2) gives us a suitable approximation of the maximum dependence distance of a non-uniform dependence. For uniform dependence distances, the buffer size is an accurate measure of the dependence distance. For non-uniform dependence distances, the use of the buffer size introduces a source of inaccuracy in the PPN modeling graph.

The number of initial tokens d_c assigned to the last edge of a cycle is determined as follows. If the cycle is tight, that is, if in every iteration each process depends on the output of the previous iteration of its predecessor process, the processes execute sequentially without overlap between firings of different processes. In such a case, the dependence distance vector contains zeroes for all dimensions except the last for which it contains a one. That is, the dependence distance vector is of the form $[0, 0, \ldots, 1]$. The corresponding buffer size S_c is one, and we assign one initial token to the last edge of the cycle.

If the cycle is not tight, then overlapped execution between firings of different processes may occur. In such a case the dependence vector is different from the form described above. We assign $S_c + 1$ initial tokens to the last edge of a cycle which corresponds to the buffer size plus one additional initial token to accomodate overlapped execution. Currently, this is a known source of inaccuracy in the MCM modeling HSDF graphs derived from PPNs. Determining the number of initial tokens to assign to the last edge of a non-tight cycle is therefore subject of future investigation.

Feedforward channels

Feedforward channels connect a strongly connected component of a PPN to another strongly connected component. As such, the corresponding feedforward edges in an HSDF graph are not part of any cycle and thus would not affect the MCM value. In the HSDF model, edges have infinite capacity which implies that a feedforward edge indeed does not affect the MCM value of an HSDF graph. That is, a feedforward edge cannot reach a "full" state that would cause blocking writes decreasing throughput. In contrast to HSDF edges, PPN channels have a finite capacity which may cause blocking write conditions that decrease throughput.

To take the finite capacity of a channel into account, we add for each feedforward channel $(a \rightarrow b)$ a forward edge $e = (a \rightarrow b)$ and a *backedge* $(b \rightarrow a)$ [SB00, Section 10.4]. We assign zero initial tokens to the corresponding feedforward edge in the HSDF graph. The number of initial tokens $m_e \in \mathbb{N}$ on the backedge represents a particular buffer capacity. Empirically, we found that a value m_e corresponds to a buffer capacity

$$S_c = m_e + d_c - 2, \tag{4.4}$$

where d_c is the dependence distance approximation used in the discussion above on feedback channels. That is, the MCM computed using m_e matches the PPN period achieved with a buffer capacity of S_c tokens.

Bounding feedforward channel delays

According to the HSDF model, any positive number of initial tokens m on a backedge is allowed. This leads to an infinite number of possible buffer configurations. However, when m is below a certain value, a corresponding PPN buffer size may not exist due to the operational semantics of a PPN process. This gives a lower bound on m. Also, when m exceeds a certain value, the MCM is not affected anymore, which means that increasing the buffer size does not lead to a higher throughput. This gives an upper bound on m. Therefore, we can bound the design space by only considering the values that lie between the lower and upper bounds.

The lower bound on m for any edge in the PPN modeling graph is two, which is a consequence of the operational semantics of a PPN process. This lower bound of two can be explained as follows. In an HSDF graph, a token is kept on the edge until the consuming node has finished its firing. In a PPN graph, a token is transfered to a buffer internal to the process during the read stage. This effectively increases the buffer size by one. As such, a buffer size of one corresponds to a number of initial tokens $m = 2$.

The upper bound on m represents the point where increasing the buffer size does not yield a higher throughput. This corresponds to a value m for which the maximum cycle mean of the graph is determined by cycles of the original graph or selfloops, but not by a cycle introduced by the modeling of a feedforward edge. For an arbitrary feedforward edge $(a \rightarrow b)$, we choose m such that the resulting cycle mean value is less than or equal to the cycle mean of the selfloops of the nodes involved in the cycle:

$$\frac{t(a) + t(b)}{m} \leq \max\{t(a), t(b)\}.$$

For positive execution times t, this inequation holds if m equals the number of nodes in the cycle, which is two. However, other paths between a and b may exist which must be considered as well to avoid that they determine the maximum cycle mean. To ensure that none of the other paths between a and b determine the maximum cycle mean, we generalize the above inequation to a path consisting of n nodes:

$$\frac{\sum_{i=1}^{n} t(i)}{n} \leq \max_{i=1}^{n}\{t(i)\}. \tag{4.5}$$

Thus, the upper bound on m originating from a channel c equals the number of pro-

a) PPN containing feedforward edges. b) Corresponding PPN Modeling graph.

Figure 4.5: Handling feedforward edges in a PPN.

m_1	m_2	m_3	$MCM(G_{\text{4.5b}})$
1	1	1	90
1	1	2	60
1	1	3	60
1	2	1	90
1	2	2	60
1	2	3	60

m_1	m_2	m_3	$MCM(G_{\text{4.5b}})$
2	1	1	90
2	1	2	60
2	1	3	60
2	2	1	90
2	**2**	**2**	45
2	**2**	**3**	30

Table 4.1: MCM values for different numbers of initial tokens. Only for the configurations in boldface a valid PPN buffer size configuration exists.

cesses on the longest path connecting σ_c to δ_c.

In Figure 4.5a, we show a PPN containing three feedforward channels. In Figure 4.5b, we show the corresponding modeling graph. For each feedforward channel in the PPN, we have added a forward edge and a backedge in the modeling graph. The values m_1, m_2, and m_3 specify the amount of initial tokens assigned to the backedges. According to equation (4.5), the upper bound of m_1 and m_2 is two and the upper bound of m_3 is three. In Table 4.1, we show twelve possible combinations of m-values, deliberately assuming a lower bound of 1 for each m-value. This yields twelve different design points trading off buffer size against throughput. If we take the lower bound on m-values for PPN modeling graphs into account, any combination of m-values containing an m-value below two does not have a corresponding PPN buffer configuration. Hence, only for $(m_1, m_2, m_3) = (2, 2, 2)$ and $(m_1, m_2, m_3) = (2, 2, 3)$ an actual PPN buffer configuration exists. As such, for this example the buffer size design space is reduced to only two points.

Summary

The derivation of the more compact HSDF graph is summarized in Algorithm 4.1. The input is a PPN and a number Λ_p representing the execution time of a single firing

of each process p. The output is an HSDF graph that is intended only for throughput analysis by an MCM algorithm. Lines 1–4 in Algorithm 4.1 perform the conversion of PPN processes to HSDF nodes. Lines 5 and onwards in Algorithm 4.1 perform the conversion of PPN channels to HSDF edges, considering the three classes of channels.

To leverage Proposition 4.2, we assume that the "append edge" operations at lines 12 to 16 of Algorithm 4.1 prune any edges $e' = (\sigma'(c) \rightarrow \delta'(c))$ already present for which $d(e')$ is larger than $d()$ of the new edge being added. Here, $\sigma'(c)$ and $\delta'(c)$ give the HSDF node that corresponds to the PPN process given by σ_c and δ_c (cf. Definition 2.13).

Because there is a one-to-one correspondence between PPN processes and HSDF nodes, no exponential increase in the number of nodes occurs. An exponential increase of the number of edges in the HSDF graph is also avoided:

Proposition 4.3 (Number of Edges in PPN Modeling Graphs)**.** *The number of edges in a PPN modeling graph for a PPN* $(\mathcal{P}, \mathcal{E})$ *is at most* $|\mathcal{P}| + 2 \cdot |\mathcal{E}|$.

Proof. For each process $p \in \mathcal{P}$, a selfloop is added, resulting in $|\mathcal{P}|$ selfloops in the HSDF graph. For each channel $c \in \mathcal{E}$, no edge is added if c is a selfloop; at most one edge is added if c is a feedback edge; and at most two edges are added if c is a feedforward edge. Thus, if all channels in a PPN are feedforward edges, then at most $2 \cdot |\mathcal{E}|$ edges are added. □

Hence, any exponential increase in the number of nodes or edges is avoided in our approach.

4.5.4 Case Studies

Acyclic Example from Literature

We first examine an acyclic PPN that was also studied by Meijer et al. [MNS10, Fig. 7]. We show the sequential code and corresponding PPN in Figure 4.6a and 4.6b. The PPN consists of four processes and three channels. We use latency values $\{\Lambda_{P1} = \Lambda_{P2} = 61, \Lambda_{P3} = 126, \Lambda_C = 121\}$ which correspond to the process workloads used by Meijer et al. The authors found a system throughput of $\frac{1}{126}$ using their throughput estimation method.

The PPN modeling graph derived using Algorithm 4.1 is shown in Figure 4.6c. Each process has an iteration domain consisting of 1000 points. Therefore, the execution time of each HSDF node is set to 1000 times the corresponding latency Λ. We add a selfloop with one initial token to each node. All of the three channels of the PPN are feedforward channels. Therefore, we add for each channel both the forward edge

Algorithm 4.1 Derive a modeling graph from a PPN.

Input: PPN $G = (P, C)$, delays $\{\Lambda_p \mid p \in P\}$
Output: HSDF $H = (V, E, t, d)$

for all processes p in P **do**
  append node h to V with $t(h) = |D_p| \cdot \Lambda_p$
  append edge $e = (p \to p)$ to E with $d(e) = 1$
end for
for all channels c in C **do**
  if c is not a selfloop **then**
    if c is a feedback edge (i.e., part of an SCC) **then**
      $s = 0$
      if δ_c fires before σ_c **then**
        $s = S_c + \{1$ if a non-tight cycle containing c exists$\}$
      end if
      append edge $e = (\sigma'(c) \to \delta'(c))$ to E with $d(e) = s$
    else if c is a feedforward edge **then**
      append edge $e = (\sigma'(c) \to \delta'(c))$ to E with $d(e) = 0$
      $s = \max\{|p_i| + 1 \mid p_i$ is a path from σ_c to $\delta_c\}$
      append edge $b = (\delta'(c) \to \sigma'(c))$ to E with $d(b) = s$
    end if
  end if
end for
return H

and a backedge in the modeling graph. According to Equation (4.5), the number of initial tokens on each backedge equals two. The maximum cycle mean computation of the resulting modeling graph yields the following:

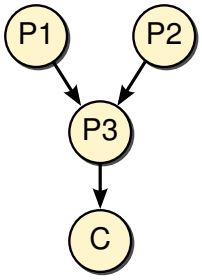
$$\begin{aligned} MCM(G_{4.6c}) &= \max\left\{ \frac{t(P1)}{1}, \frac{t(P2)}{1}, \frac{t(P3)}{1}, \frac{t(C)}{1}, \frac{t(P1)+t(P3)}{d(P1 \to P3)+d(P3 \to P1)}, \right. \\ &\qquad \left. \frac{t(P2)+t(P3)}{d(P2 \to P3)+d(P3 \to P2)}, \frac{t(P3)+t(C)}{d(P3 \to C)+d(C \to P3)} \right\} \\ &= \max\left\{ \frac{61000}{1}, \frac{61000}{1}, \frac{126000}{1}, \frac{121000}{1}, \frac{187000}{2}, \frac{187000}{2}, \frac{247000}{2} \right\} \\ &= 126000\,. \end{aligned}$$

Thus, a single iteration of the graph takes 126000 time units. In an iteration of the graph, sink process *C* fires 1000 times. Therefore, the period $T_C = \frac{126000}{1000} = 126$,

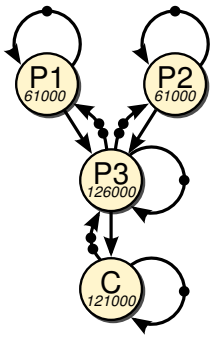
```
#define M 1000

for (i=0; i<M; i++) {
  P1(&a[i]);
  P2(&b[i]);
}
for (i=0; i<M; i++) {
  P3(a[i], b[i], &c[i]);
}
for (i=0; i<M; i++) {
  C(c[i]);
}
```

a) Sequential code.



b) PPN.



c) Modeling graph.

Figure 4.6: Throughput analysis on example from [MNS10].

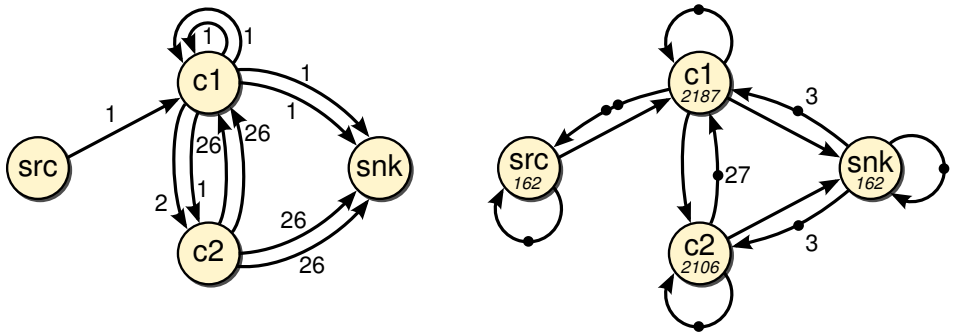
and the PPN's throughput equals $\frac{1}{126}$. This throughput value exactly matches the value found by Meijer et al.

Odd-even Transposition Sorting

With this example we illustrate the MCM analysis applied to a cyclic PPN. The odd-even transposition sorting is a parallel sorting algorithm which sorts an array of n elements. The algorithm consists of n comparator stages. In each odd-numbered stage, all even-indexed elements are compared with their odd-indexed neighbours and swapped if they are not in the correct order. In each even-numbered stage, all odd-indexed elements are compared with their even-indexed neighbours and swapped if necessary. In each stage, all $n/2$ pairs can be compared in parallel.

In Figure 4.7a, we show a PPN for the odd-even transposition sorting algorithm. The PPN consists of four processes. Source process *src* provides the data to be sorted. Processes *c1* and *c2* perform the compare-and-swap operations. Sink process *snk* consumes the sorted data.

In Figure 4.7b, we show the throughput modeling graph derived from the PPN. All four channels between *c1* and *c2* in Figure 4.7a are part of a strongly connected component. From dependence analysis we find that *c2* cannot fire before *c1* has fired. Thus, we put zero initial tokens on the forward edge connecting *c1* to *c2*. The dependence distance vectors for both edges are non-uniform. The upper bound on the dependence distance is 26. According to line 10 of Algorithm 4.1, we put 27 initial tokens on the feedback edge from *c2* to *c1*. All of the remaining edges are feedforward edges to which we assign two or three initial tokens according to Equation (4.5).



a) PPN. b) Modeling graph.

Figure 4.7: Throughput analysis of odd-even transposition sorting.

For brevity reasons we omit the full expansion of Equation (4.2) and only summarize the result. The maximum cycle mean is 2187 which originates from the selfloop of process *c1*. Since the *sink* process domain size is 54, the average period of the PPN is $\frac{2187}{54} = 40.5$ time units. This corresponds to the average period observed during simulation of the RTL.

4.6 Sequential Code Profiling

At the first level in Figure 4.1, we have the application specified as sequential C code. The previous performance estimation techniques discussed in this chapter required that a PPN was derived. We now investigate whether we can estimate the performance of a PPN directly from the sequential C code. This has led to a novel profiling-based method that works directly at the sequential source code level. Our novel method was inspired by the work of Kumar on measurement of parallelism in Fortran programs [Kum88].

In Section 4.6.1, we review some existing profiling techniques. In Section 4.6.2, we present the profiling primitives employed by our approach. In Sections 4.6.3 and 4.6.4, we present the two estimation types provided by our approach. In Section 4.6.5, we apply our profiling approach on a case study. In Section 4.6.6, we present how our profiling approach can model process splitting transformations without the need to actually apply the spltting transformations to the application code. In Section 4.6.7, we discuss the performance and memory overhead resulting from the profiling.

4.6.1 Related Work

Profiling is a well-established technique in which the behavior of a program is analyzed by closely monitoring the execution of the program. This monitoring is performed by extending a program with small *instrumentation* code fragments that collect statistics such as function invocation counts during program execution.

A popular free software tool to profile for example C and Fortran programs is *GNU gprof* [GKM82]. To profile a program, the compiler instruments the program with instrumentation code. The instrumentation code collects statistics when the instrumented program is running. After running the instrumented program, gprof processes the statistics into a call graph augmented with the execution time of each call. This allows the developer to determine in which parts of a program most of the execution time is spent. To obtain execution time information, gprof relies on statistical program counter sampling which is an inexact method. The execution times are only valid for the platform on which the profiling is performed, which makes gprof not useful for throughput assessments of programs implemented as PPNs on different, possibly heterogeneous platforms.

The *Valgrind* tool set provides tools for debugging and profiling program binaries [NS07]. Each Valgrind tool translates the individual machine instructions into an intermediate representation, instruments the intermediate representation, and translates the intermediate representation back into machine instructions. This allows for more accurate execution time estimates compared with statistical sampling methods, because each instruction is considered. However, as with gprof, the obtained execution times are only valid for the processor architecture for which the program was compiled. In our design flow, different parts of a program may execute on different processor architectures, such as ARM or MicroBlaze, or may be implemented as a LAURA processor. Such alternative heterogeneous architectures are currently not supported by Valgrind and we believe that adapting Valgrind to support such architectures would require a significant amount of effort.

Support for different processor architectures was a key design goal for the *TotalProf* profiler [GHC+09]. TotalProf processes the intermediate representation of an input program into virtual assembly that is captured in the LLVM intermediate representation. Different architectures are represented using different forms of the virtual assembly. The virtual assembly is instrumented with profiling statements and then taken through the code generator which generates executable code for the host machine. By targeting the host machine, fast execution of the instrumented virtual assembly code is obtained. By providing different architecture descriptions and interconnecting different TotalProf instances, TotalProf supports profiling of heterogeneous MPSoCs. TotalProf currently lacks support for the LAURA architecture, which

prevents us from using TotalProf for performance assessment of PPNs.

Another approach to achieve a high simulation speed when simulating MPSoCs, is by compiling the application code of a system for native execution on a general-purpose computer such as a desktop workstation [SHP12]. Such native simulation techniques eliminate the need for instruction set simulation or binary translation and thus avoid a significant amount of run-time overhead. Native simulation is often intended for debugging and verification of an MPSoC design. In contrast, our interest is mainly in performance assessment to a sufficient level of accuracy to make architectural tradeoffs.

Sackmann et al. presented a profiling-based method to parallelize sequential programs [SEJ11]. To discover the parts that may execute in parallel, the authors analyze call trees obtained using Valgrind. Each node in the call tree represents an invocation of a function. Two nodes are connected by an edge if a data dependence exists between the corresponding function invocations. If a pair of nodes is not connected by an edge, then the corresponding function invocations can execute in parallel. Unfortunately, the authors' approach does not guarantee that all data dependences are added to the graph. They rely on the user to ensure that all dependences are present in the call tree. Ensuring correctness of the call tree manually is tedious and error-prone, because call trees can be large even for small programs, and because a call tree is only valid for one execution of the program. Instead of partially relying on manual effort, we favor discovering the amount of parallelism in a fully automated way.

Kumar presented COMET (COncurrency MEasurement Tool) which measures the total parallelism in Fortran programs [Kum88]. The method assumes a hypothetical ideal parallel machine with unlimited resources and no scheduling, communication, and synchronization overhead. COMET takes a Fortran program and extends it with statements that monitor the execution of the program on the ideal parallel machine. The functionality of the original program is preserved in the extended program. By compiling and executing the extended program, statistics on the absolute amount of parallelism are collected.

At this level two approaches need to be mentioned: the SESAME [PEP06] and SPADE [LSWD01] approaches. They both work at the high level but do not use profiling. Instead they use traces to capture the workload of an application in terms of read, execute, and write events. Both SESAME and SPADE require deriving a PPN, which is what we want to avoid at the first level of Figure 4.1.

4.6.2 Sequential Code Instrumentation of Static Programs

In this section, we present *cprof* which is a novel method for PPN performance estimation that is inspired on COMET. Cprof can measure parallelism in an application

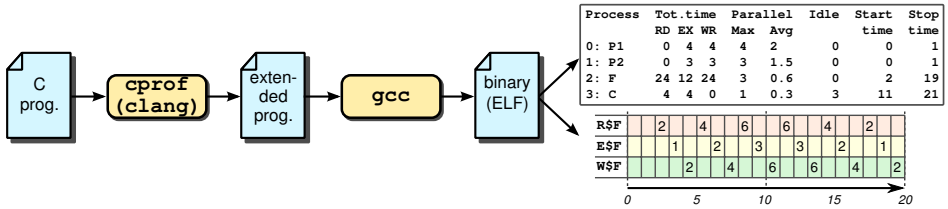


Figure 4.8: The cprof performance estimation technique.

without actually deriving a PPN, but assuming execution as a PPN on one of two machines. First, cprof can evaluate a PPN on an *ideal machine*. We consider an ideal machine to be a platform where each process iteration is mapped on a separate processor unit. The ideal machine lacks shared memory, as separate processor units communicate through separate point-to-point communication channels. Second, cprof can evaluate a PPN on a platform generated by ESPAM.

The ideal machine is used to measure the maximum degree of parallelism in an application, as we discuss in Section 4.6.3. The ESPAM platform is used to obtain an absolute throughput estimate for a PPN, as we discuss in Section 4.6.4.

Like COMET, cprof takes the original sequential program and produces an extended program containing additional profiling instrumentation statements. The cprof flow is depicted in Figure 4.8. We use the Clang/LLVM compiler infrastructure [LA04, Cla07] to automatically generate the extended program from a C program. The extended program is again a sequential program that should be compiled by a conventional C++ compiler and executed to obtain performance statistics. In contrast to COMET, which operates on programs written in the Fortran language, cprof operates on static affine nested loop programs (SANLPs) written in the C language. Another difference between COMET and cprof is the set of statements being instrumented. We only instrument the statements for which PNGEN constructs a process, because our goal is to obtain a performance estimate for a PPN execution. That is, cprof does not instrument control expressions and statements inside for-loop headers and if-conditions, since these are not translated into processes. In contrast, COMET instruments all statements, including loop-statements and if-statements. Instead of targeting only an ideal machine, cprof also targets a machine with a fixed amount of processing resources. This enables performance assessment of programs executing as a PPN on a user-defined platform.

Both COMET and cprof employ a global time scale. In this global time scale, COMET and cprof keep track of the timestamps at which the statements of a sequential program start and finish their executions. We instrument each statement such that the operational semantics of all processes are represented on a single global time

scale. The added instrumentation models the read, execute, and write stages of a PPN process that were described in Section 2.2.4. In the read stage, cprof determines at which time the actual statement can execute, based on the times at which all input data is available. In the execute stage, the statement execution finish time is determined. In the write stage, the times at which the output data of the statement is available are updated.

The main challenge is to determine the starting times of statement executions. We determine such starting times by taking into account the four aspects that were introduced in Section 4.2. To address the conditional synchronization, mutual exclusion, and basic calibration aspects, we employ instrumentation primitives. Since we only consider SANLPs in which control is static by definition and functions have a fixed latency, we do not need to address the conditional control flow aspect. The instrumentation primitives comprise shadow variables, control variables, and execution profiles. In the following paragraphs, we discuss each instrumentation primitive and explain how the first three performance aspects are addressed.

Shadow Variables

For each variable `v` in the original program, we add a *shadow variable* `$v` that holds the timestamp in which variable `v` is written. Similarly, for each array in the original program, we add an array of shadow variables which contains for each array element the timestamp at which a particular array element is written. A shadow variable is updated whenever the corresponding variable is written during the write stage. The new timestamp of a shadow variable `$v` is set to the timestamp at which the statement writing to `v` finishes writing `v`, plus an additional cost Λ_W modeling the write operation latency.

We use shadow variables to address the conditional synchronization aspect. Conditional synchronization in a PPN execution occurs when a process reads from one or more channels. As dictated by the operational semantics, a process function can only fire after all incoming channels have been read. That is, the firing of a process function may be postponed because of a blocking read operation on one of the incoming channels. In contrast to the other performance assessment techniques discussed in this chapter, the cprof technique does not explicitly model a program as a PPN consisting of processes and channels. Despite this, cprof is able to obtain accurate performance assessments that incorporate blocking read operations. To understand why we can analyze the performance of a PPN without actually deriving a PPN, we now explain how blocking read operations are taken into account, thereby addressing the conditional synchronization aspect.

The SANLP class of C programs that we consider are written assuming a sequential

execution model in which statements execute one after another. Only one statement executes at any point in time. The execution order is given by the textual order of statements in the program and control flow statements such as for-loops. For a given program, many alternative execution orders may exist which all yield the same functional meaning of the program as long as all *data dependence* relations are respected [Ban97]. A program translated to a PPN employs such an alternative execution order, in which processes execute concurrently. Each process fires as soon as its input data is available. Thus, the performance of a PPN is determined by the availability times of data. The availability of data is captured by the data dependence relations of a sequential program. Three different types of data dependence relations may exist in a sequential program: flow dependences, anti-dependences, and output dependences [PW86]. We now explain how cprof handles each dependence type using shadow variables, such that an accurate performance assessment of a PPN derived from the sequential program is obtained.

Flow dependences, or read-after-write dependences, occur when a statement reads a variable v written by a previous statement. During the write operation, the timestamp at which the write occurs is stored in the shadow variable of v. Upon reading v, the timestamp in the shadow variable of v is taken as the time at which v is available. By definition, the read operation occurs after the write operation, meaning the flow dependence is correctly modeled.

Anti-dependences, or write-after-read dependences, occur when a statement writes data to a variable v that was read by a previous statement. Upon reading v, the timestamp in the shadow variable of v is taken as the time at which v is available. This shadow variable is then overwritten by the write operation, which by definition occurs after the read operation. However, in both COMET and cprof, the instrumentation for the write operation does not take into account the time at which v was actually read. Thus, the anti-dependence is not modeled.

Output dependences, or write-after-write dependences, occur when two statements write data to the same variable v. The timestamp of the first write operation is stored in the shadow variable of v. The timestamp of the second write operation overwrites the previous timestamp in the shadow variable, without taking into account the timestamp of the first write operation. Only the timestamp of the last write is kept, meaning the output dependence is not modeled.

In summary, COMET and cprof only model flow dependences, and ignore anti-dependences and output dependences.[1] This means that cprof cannot model the performance of a SANLP on a general purpose processor accurately if the program con-

[1] An extension to incorporate anti-dependences and output dependences in COMET was described by Kumar as well [Kum88, Section VI], but this extension has not been incorporated in cprof as anti-dependences and output dependences are not relevant in the PPN context [Tur07, Chapter 3].

tains anti-dependences or output dependences. However, the purpose of cprof is to model performance of a SANLP assuming execution as a PPN. A key property of the PPN MoC is that only flow dependences affect its performance. Anti-dependences and output dependences in a sequential program do not affect the performance of a PPN, such that these can be safely ignored by cprof, as we now explain.

Anti-dependences in a sequential program do not affect the performance of a PPN derived from the sequential program, because each produced data token has a private storage location that is preserved until the token is consumed. Multiple data tokens representing different values for the same variable of the sequential program may exist simultaneously. A property of the PPN model is that the storage location of a data token is never written again after consumption [Tur07, Chapter 3]. This means anti-dependences do not occur in a PPN and thus do not affect performance.

Output dependences in a sequential program do not affect the performance of a PPN derived from the sequential program, because a data token is only produced on a channel if the data token is guaranteed to be consumed. This means output dependences do not occur in a PPN, and thus output dependences do not affect the performance.

We have explained that flow dependences are correctly modeled by shadow variables in cprof. We use shadow variables to address the conditional synchronization aspect. The conditional synchronization aspect follows from the operational semantics of a PPN process. These operational semantics dictate that a process is allowed to fire only when all input data is present. In cprof, this means that the maximum value of the shadow variables of the inputs of the statement represents the correct time of firing.

Once a statement starts executing in cprof, no delays or stalls occur until the statement has finished writing its output variables. This implies that cprof uses non-blocking write semantics, assuming that channel sizes are always sufficiently large to store the produced data. Finite channel sizes affecting throughput via a blocking write mechanism are currently not supported by cprof. To consider finite channel sizes, we should explicitly model each channel resource. This requires derivation of a PPN from the sequential program, because the channels are not explicitly present in the sequential program. Modeling finite channels sizes is therefore a subject of future research.

Control Variables

For each statement `s` for which PNGEN constructs a process, we add a *control variable* `C$s` that holds the earliest time at which statement `s` can execute. This earliest time is determined by the availability of the input data to statement `s`. The statement

executes only when all input data is available, resembling the operational semantics of a PPN process. As such, we ensure that data dependence relations are not violated, and thus address the conditional synchronization aspect. Control variables also address the mutual exclusion aspect, which will be detailed further in Sections 4.6.3 and 4.6.4.

A control variable is updated during the read stage by considering the shadow variables of the variables read by statement `s`. A statement `s` can execute after all variables read by statement `s` have been written. Thus, a control variable `C$s` is set to the maximum value of all shadow variables that are read by statement `s`. Taking the maximum value of all shadow variables effectively delays the statement to the timestamp at which all data is available, resembling a blocking read operation in terms of the PPN operational semantics. The statement only executes when all data is available, which is in accordance with the PPN semantics.

Statement Execution Profile

For each statement `s`, we also add three one-dimensional arrays `R$s`, `E$s`, and `W$s` which together constitute the *statement execution profile* of `s`. In the statement execution profile, we collect the read, execute, and write behavior of the statement over time. The statement execution profile collect at a high level the operational behavior of a process. For example, `W$s[23] = 2` means that two write operations are in progress at time 23. All array elements are initialized to zero. Array `R$s` is updated after reading a statement input. Array `E$s` is updated after executing the statement. Array `W$s` is updated after writing a statement output. An update to any of the three arrays involves incrementing the array elements in an interval $[t_s, t_f)$ by one. Here, t_s is the starting time and $t_f = t_s + \Lambda$ is the finish time of an operation with latency Λ.

After executing the instrumented program, we can extract the following information for each process from the statement execution profiles:

- The total time spent on read operations, statement executions, and write operations is obtained by summing all elements in the corresponding `R$s`, `E$s`, and `W$s` arrays.
- The process start time $s(p)$, which is the first time at which a process can fire, equals the index of first non-zero element in `R$s`. For statements that do not consume any input data, this equals zero.
- The process finish time $f(p)$, which is the time at which the process has finished all iterations, equals the index of the last element in `W$s`. For statements that do not produce any output data, we instead take the index of the last element in `E$s`.

- The number of idle cycles, which is the number of time units in the interval $[s(p), f(p))$ in which each of the Rs, Es, and W$s arrays contains a zero.
- The maximum number of statement executions that are in progress simultaneously is obtained by finding the maximum value in E$s.
- The number of process iterations that are in progress simultaneously at a given time t is given by the *flat execution profile*, which we define as

$$\texttt{R\$s}[t] + \texttt{E\$s}[t] + \texttt{W\$s}[t]. \tag{4.6}$$

The average process period can be computed from the process start and finish times as

$$T_p = \frac{f(p) - s(p)}{D_p}. \tag{4.7}$$

We then compute the throughput of a process by taking the reciprocal of T_p. The throughput of the entire PPN is given by the throughput of the sink process, according to Definition 4.2.

Global Execution Profile

From all statement execution profiles, we compute the *global execution profile* G$ as follows:

$$\texttt{G\$}[\texttt{k}] = \sum_{i=0}^{|P|-1} \texttt{R\$}i[\texttt{k}] + \texttt{E\$}i[\texttt{k}] + \texttt{W\$}i[\texttt{k}], \tag{4.8}$$

$$0 \leq \texttt{k} < \max\{\forall p \in P \mid f(p)\}.$$

That is, we sum for each timestamp k the statement execution profiles of all processes. The global execution profile resembles the *PROFILE* array in COMET. However, the global execution profile includes read and write operations, which are not included in COMET. The global execution profile provides information about the behavior of the application as a whole. We can extract the following information from the global execution profile:

- The *PPN execution time*, which equals the number of elements in G$.
- The *maximum degree of parallelism*, which is the maximum number of simultaneously active processes at any time, equals the maximum element in G$.
- The *average degree of parallelism* is obtained by dividing the sum of all elements in G$ by the number of elements in G$.

Execution Times

To address the basic calibration aspect, we rely on user-provided performance data for the process functions. In particular, cprof uses the Λ and II values from Definition 3.1 to characterize the latency and initiation interval of each process function. This allows cprof to handle both pipelined (e.g., LAURA) processing resources and non-pipelined (e.g., programmable) processing resources. In addition, cprof assumes fixed latencies Λ_R and Λ_W for read and write operations that resemble communication over PPN channels. The particular use of the Λ and II values is discussed in Section 4.6.5.

Currently, cprof assumes that Λ and II are constant for all invocations of a process function. This is the main source of inaccuracy, because this is not always the case. For example, a division function may have a multi-cycle latency in general, but may have a one-cycle latency if the divider equals one. Including such latency characteristics should be possible in cprof, because cprof allows a fully functional execution of the original program. However, our main interest lies in a simple and fast performance estimation approach, and thus detailed dynamic performance models are currently beyond the scope of cprof.

4.6.3 Maximum Degree of Parallelism

To gain insight in the amount of parallelism in a given application specification, we instrument an input program in a way that models execution on a hypothetical *ideal machine*. This gives the *maximum degree of parallelism*, which represents an upper bound on the number of processing resources required to execute the PPN without processing resource contention. Adding more processing resources does not result in a further speedup.

The selection between an ideal machine with an infinite number of processor resources and a real machine with a finite number of processor resources depends on whether or not the mutual exclusion aspect is addressed. Mutual exclusion can be enforced when a control variable `C$s` is updated during the read stage by taking into account the time at which a processor is available. On both the ideal and real machine, the conditional synchronization aspect needs to be addressed, because a statement cannot start until all input data is available. On the ideal machine, the availability of the input data is the only condition for the statement to execute, which means we do not take the mutual exclusion aspect into account. Thus, each control variable `C$s` is set to the maximum value of all shadow variables representing inputs to `s`, which implies `s` executes as soon as all its input data is available. After execution of the instrumented program on the ideal machine, the information from the global execution profile `G$` can be used to draw conclusions about the application performance.

The consequence of using the ideal machine is that the PPN execution time equals the minimum time needed for the PPN execution. By definition, the ideal machine has sufficient processing resources to avoid processing resource contention. A PPN execution time of 1 time unit means that the PPN can fire all iterations of all processes simultaneously. A PPN execution time larger than 1 time unit means that parts of the PPN are inherently sequential.

Related to the maximum degree of parallelism is the *average degree of parallelism*. This represents an upper bound on the speedup that can be obtained with an unbounded number of processors [EZL89] compared to execution of a PPN on a single processor.

4.6.4 Absolute Throughput Estimation

By instrumenting an input program such that the mutual exclusion aspect is taken into account, we can obtain an absolute throughput estimate of the execution of a PPN derived from the input program. This is possible because each statement of a sequential program corresponds to a process in a PPN and because the operational semantics of a process are well-defined (cf. Section 2.2.4). On a realistic execution platform with a finite number of processing resources, an additional condition for the availability of a suitable processing resource needs to be considered before a statement can be executed. This mutual exclusion aspect was not addressed by COMET, because COMET assumes only the ideal machine. If we assume that all executions of a statement are mapped onto the same processing resource, then control variable `C$s` should at least equal the time `C$s` $+ II$ at which the processing resource can initiate a new execution. Instead of taking the maximum only over all shadow variables as described in Section 4.6.3, we now take the maximum over all shadow variables and `C$s` $+ II$.

4.6.5 Case Study

In this case study section, we first apply the absolute throughput estimation (cf. Section 4.6.4) to the program shown in Figure 4.9. Next, we show how to determine the maximum degree of parallelism (cf. Section 4.6.3) of this program.

Case Study: Absolute Throughput

In Figure 4.10, we show the cprof instrumentation for the program code shown in Figure 4.9. We assume that all executions of statement `func1` are mapped onto a single processor. We furthermore assume that a read operation takes Λ_R time units;

```
1  for (i=0; i<9; i++) {
2    source(&x[i], &b[i]);          // Statement 0
3    func1(x[i], &a[i]);            // Statement 1
4    sink(a[i], b[i]);              // Statement 2
5  }
```

Figure 4.9: Example program code on which we illustrate cprof.

that a write operation takes Λ_W time units; that the latency of `func1` is given by $\Lambda_{\texttt{func1}}$; and that the initiation interval of `func1` is given by $II_{\texttt{func1}}$.

At line 17 of Figure 4.10, we determine the starting time of statement 1 using the expression `max(C$1, $x[i])` which takes the maximum from control variable `C$1` and shadow variable `$x[i]`. The control variable in the `max`-expression ensures that the statement executes after the previous execution of the statement has finished, which means the processor resource is available. The shadow variable in the `max`-expression ensures that the statement executes when the variable `$x[i]` has been written. By adding Λ_R, we delay execution of the statement to incorporate a read operation delay of Λ_R time units. At line 18, we add the read operation to the read execution profile. At line 21, the original statement is executed. For the applications that we consider, actual execution of the statements is not required to obtain throughput assessments. By omitting the actual execution of statements, the throughput assessment can be performed in less time. At line 22, we set the finish time of the statement and at line 23 we add the actual execution of the statement to the execution profile. At line 24, we update control variable `C$1` such that the next execution of the statement starts at least after a full initiation interval. At line 27, we set the time at which `a[i]` is written. At line 28, we add the write operation of `a[i]` to the write execution profile.

At line 31, we show the instrumentation for statement 2 which takes two inputs `a[i]` and `b[i]`. Both `a[i]` and `b[i]` have to be available before statement 2 can execute, so we consider the shadow variables of both input variables to determine the starting time of statement 2. By including both in the `max`-expression, we take the conditional synchronization aspect into account.

After executing the instrumented code, the execution profiles are obtained. In Figure 4.11, we show the statement execution profiles for all three statements, assuming $\Lambda_R = \Lambda_W = 1$, $\Lambda_{\texttt{func1}} = 2$, $\Lambda_{\texttt{source}} = \Lambda_{\texttt{sink}} = 1$ and $II_{\texttt{source}} = II_{\texttt{func1}} = II_{\texttt{sink}} = 1$. The first read operation of `source` starts at time 0, because the statement does not depend on any input data. The first read operation of `func1` starts at time 2, which means the process `func1`'s start time equals 2. The last write operation of `func1` finishes at time 38, which means process `func1`'s finish time equals 38. Using Equation (4.7) we find that the average period $T_{\texttt{func1}} = 4$, which can be verified

```
1  for (i=0; i<9; i++) {
2    // Read stage (void, no input arguments to statement 0)
3
4    // Execution stage
5    source(&x[i], &b[i]);            // Original statement 0
6    done = C$0 + Λsource
7    for (t=C$0; t<=done; t++) E$0[t]++;
8    C$0 += IIsource;
9
10   // Write stage
11   $x[i] = done + ΛW;
12   for (t=done; t<=done+ΛW; t++) W$0[t]++;
13   $b[i] = $x[i] + ΛW;
14   for (t=done; t<=done+ΛW; t++) W$0[t]++;
15   ------------------------------------------------------------
16   // Read stage
17   C$1 = max(C$1, $x[i]) + ΛR;
18   for (t=C$1-ΛR; t<=C$1; t++) R$1[t]++;
19
20   // Execution stage
21   func1(x[i], &a[i]);              // Original statement 1
22   done = C$1 + Λfunc1
23   for (t=C$1; t<=done; t++) E$1[t]++;
24   C$1 += IIfunc1;
25
26   // Write stage
27   $a[i] = done + ΛW;
28   for (t=done; t<=done+ΛW; t++) W$1[t]++;
29   ------------------------------------------------------------
30   // Read stage
31   C$2 = max(C$2, $a[i], $b[i]) + 2*ΛR;
32   for (t=C$2-2*ΛR; t<=C$2; t++) R$2[t]++;
33
34   // Execution stage
35   sink(a[i], b[i]);                // Original statement 2
36   done = C$2 + Λsink
37   for (t=C$2; t<=done; t++) E$2[t]++;
38   C$2 += IIsink;
39
40   // Write stage (void, no output arguments to statement 2)
41 }
```

Figure 4.10: Instrumentation by cprof for the program shown in Figure 4.9.

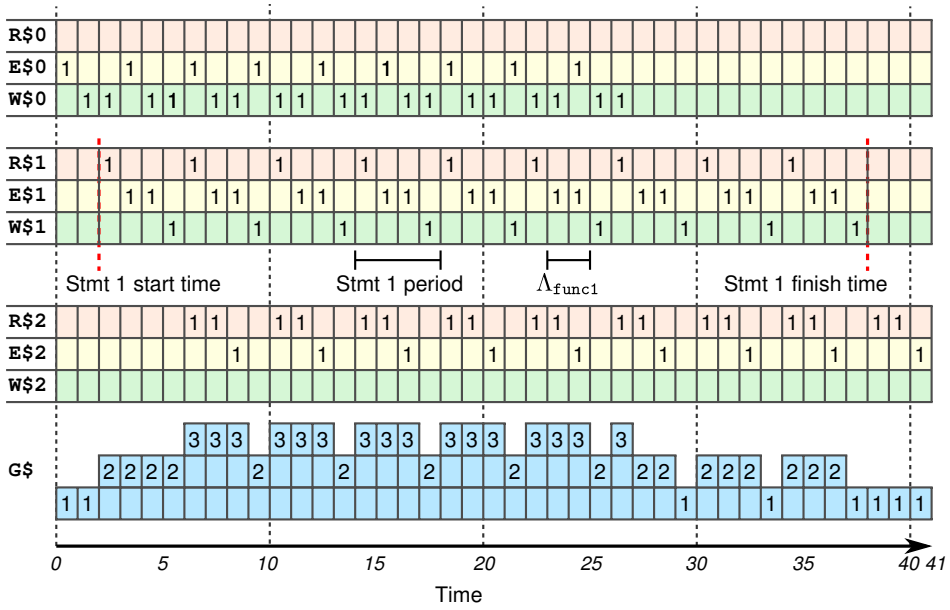


Figure 4.11: Execution profiles obtained by executing the instrumented code of Figure 4.10. Empty cells in the `R$`, `E$`, and `W$` profiles represent '0'.

visually in Figure 4.11.

The first read operation of `sink` starts at time 6, because the first execution of `sink` depends on variables `a[0]` and `b[0]`. Variable `a[0]` is written by `func1` at time 5, and variable `b[0]` is written by `source` at time 2. The first execution of `sink` can thus only start after time 5 at which both `a[0]` and `b[0]` are available. The number of idle cycles for the `sink` statement is eight, since there are eight time units in the interval $[6, 41)$ in which $\texttt{R\$2} = \texttt{E\$2} = \texttt{W\$2} = 0$. This means the `sink` statement does not receive data at a fast enough rate. The number of idle cycles for the other two statements is zero, which means they fully utilize their processing resources.

In the bottom part of Figure 4.11 we show the global execution profile `G$` that is obtained using Equation (4.8). From the global execution profile, we can observe that that a full execution of the PPN takes 41 time units. Furthermore, we can observe that at most three operations are active simultaneously. Thus, the maximum degree of parallelism in this execution equals three. Summing all elements in `G$` gives a total amount of work equal to 90 units. The average degree of parallelism in this execution is $\frac{90}{41} \approx 2.1$. This means that on average, approximately two processes are active.

Case Study: Maximum Degree of Parallelism

To find an upper bound on the throughput of the application, we are interested in the amount of parallelism inherent in the application. To reveal the amount of parallelism in the entire application we instrument the code as described in Section 4.6.3. This requires only a small change to the instrumentation code that updates the control variables. For example, the newly instrumented code for statement 1 only differs in one place from the code shown in Figure 4.10. At line 17, we now assign `$x[i] +` Λ_R to the control variable. That is, we ignore the previous value of `C$1` such that the statement is executed as soon as the input data is available. As a result, each statement execution is performed on its own processing resource, which mimics execution on an ideal machine.

For the input program of Figure 4.9, execution on an ideal machine results in the execution profiles shown in Figure 4.12. All nine iterations of each statement execute in parallel. For example, the process derived from statement 0 executes nine instances of its function at time 0. The two output arguments of each of the nine statement executions are available at time 1, since $\Lambda_{\texttt{source}} = 1$. This results in eighteen write operations at time 1.

From the global execution profile shown in the bottom part of Figure 4.12, we can observe that execution on the ideal machine takes eight time units. This is a lower bound on the execution time of a PPN derived from the input program under the given latency values. Furthermore, we can observe that at most eighteen operations execute

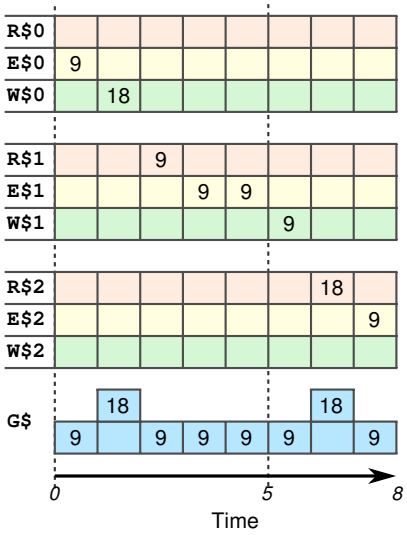


Figure 4.12: Execution profiles obtained by profiling the code of Figure 4.9 on an ideal machine. Empty cells in the `R$`, `E$`, and `W$` profiles represent '0'.

in parallel. Thus, to attain the minimum execution time of eight time units, a system with eighteen processors is required. The average degree of parallelism equals $\frac{90}{8} = 11.25$. This means that on average, 11.25 operations are in progress. Typically, the average degree of parallelism provides a design point which delivers a performance close to the maximum achievable performance, at a substantially reduced number of processing resources [EZL89]. With 11 or 12 processors, the minimum execution time becomes 10 time units, which is 25% above the minimum execution time. The reduction in the number of processing resources is 33–39%.

4.6.6 Transformation Performance Estimation

In the previous sections, we have distinguished two modes of operation of cprof. In Section 4.6.3, we presented how to determine the maximum degree of parallelism, where each iteration of a process is executed on a separate processing resource. In Section 4.6.4, we presented the absolute throughput estimation mode, where all iterations of a process are executed on the same processing resource. These two modes represent the two extremal design points of the possible assignments of iterations to processing resources. Many alternative design points exist between both extremes, which can be obtained by varying the assignment of iterations to processing resources. These two extremal design points are essential information for the

```
1  for (i=0; i<9; i++) {
2    pr$ = i % N;                        // For modulo unfold
3    pr$ = i / ((9-0)/N);                // For plane cut
4
5    // Read stage
6    C$1[pr$] = max(C$1[pr$], $x[i]) + Λ_R;
7    for (t=C$1[pr$]-Λ_R; t<=C$1[pr$]; t++) R$1[t]++;
8
9    // Execution stage
10   func1(x[i], &a[i]);                 // The original statement
11   done = C$1[pr$] + Λ_func1
12   for (t=C$1[pr$]; t<=done; t++) E$1[t]++;
13   C$1[pr$] += II_func1;
14
15   // Write stage
16   $a[i] = done + Λ_W;
17   for (t=done; t<=done+Λ_W; t++) W$1[t]++;
18 }
```

Figure 4.13: Instrumented code for statement 1 of Figure 4.9 to analyze splitting transformations.

designer. At this stage, the designer knows whether he can satisfy a performance constraint at all from the maximum degree of parallelism. However, this extremal design point has a very high implementation cost, as it assumes execution on an ideal machine. A realistic design point is provided by the absolute throughput estimate. Using splitting transformations (cf. Section 5.1.1, the designer can evaluate intermediate design points with higher performance, eventually satisfying his constraints. Before starting this exploration, a designer already knows if his performance constraint can be satisfied.

A convenient way to obtain the alternative design points is through process splitting transformations that resemble loop unfolding transformations [Muc97, SKD02]. Such splitting transformations are covered in detail in Chapter 5.

In this section, we present how the performance of transformed PPNs can be analyzed using cprof, without the need to actually apply the splitting transformation on the program code. This allows a designer to quickly evaluate different design points, and then select the design point that best matches the design requirements. Then, the designer has to apply only those splitting transformations that result in the selected design point to obtain the desired implementation. To model a splitting transformation with factor N, we generalize the mutual exclusion aspect to N processors.

In Figure 4.13, we show the instrumented code for statement 1 of Figure 4.9. This

code differs from the original instrumented code shown in Figure 4.10, to enable performance estimation of splitting transformations. The differences with the original instrumentation are underlined in Figure 4.13. The main difference with the original instrumentation is that statement 1's control variable `C$1` is changed into an array of N elements, where N is the splitting factor. The control variable array is indexed using a processing resource selection variable `pr$`. At the start of each iteration, this variable is set to the identifier of the processing resource to which the iteration is assigned.

As detailed further in Section 5.1.1, a process iteration domain can be split in different ways. We distinguish between modulo unfolding and plane cutting transformations. The assignment to `pr$` depends on the chosen transformation. At line 2 in Figure 4.13, we assign `i%N` to analyze a modulo unfolding transformation. At line 3 in Figure 4.13, we assign `i / ((9-0)/N` to analyze a plane cutting transformation. Setting the `pr$` variable effectively selects the control variable that is used for the iteration. The instrumentation statements for the read, execute, and write stages then work on this control variable according to the method described in Section 4.6.4.

In Figure 4.14, we show the execution profiles obtained after instrumenting the code of Figure 4.10 such that a modulo unfolding transformation on `func1` is modeled. We assume $N = 3$, which results in three partitions of the `func1` statement. For each partition, we maintain separate statement execution profiles, to which we append an $\_n$ suffix, with $0 \leq n \leq 2$ identifying the partition. The execution time is reduced to 33 time units, compared to 41 time units for the untransformed case. The number of idle cycles for the `sink` statement is reduced from 8 to zero, which means it now receives data at a fast enough rate. As a result, the processing resource executing `sink` is now fully utilized. However, the three processing resources executing `func1` are now underutilized, because each exhibits 10 idle cycles.

4.6.7 Instrumentation Overhead

A program instrumented by cprof exhibits two forms of overhead: performance degradation and an increased memory footprint. Performance degradation is caused by the instrumentation statements that update the shadow variables, control variables, and execution profiles. These profiling primitives are updated at each statement execution using a few inexpensive addition instructions. However, these instructions result in significant performance degradation when dealing with large function latencies Λ, because the number of instrumentation instructions depends on the latency.

The memory footprint of the instrumented program may easily be twice the memory footprint of the original uninstrumented program. This large memory footprint is mainly caused by two instrumentation primitives: shadow variables and statement

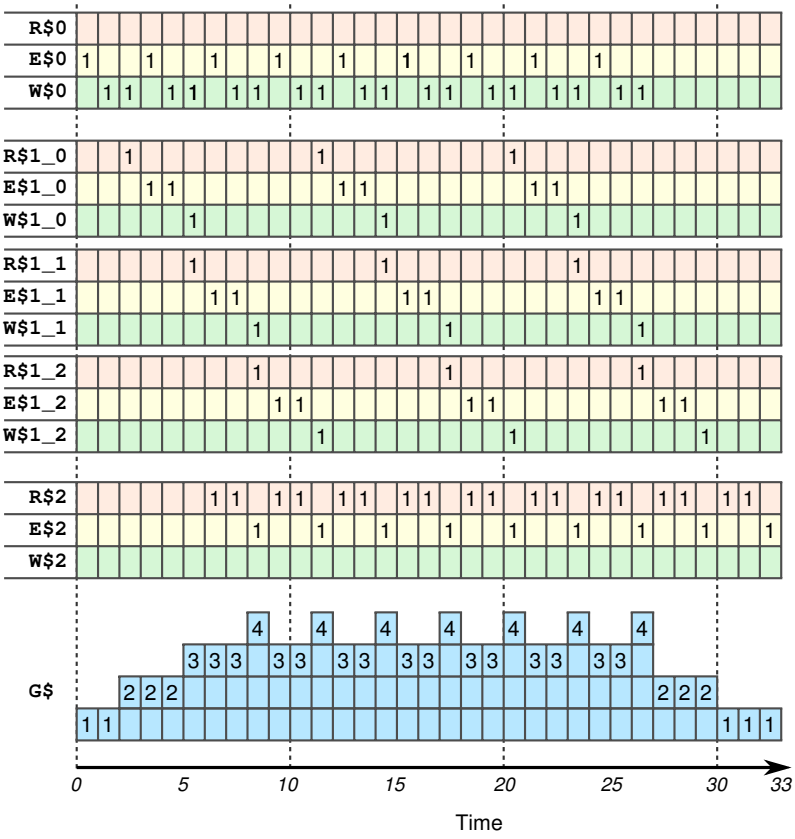


Figure 4.14: Execution profiles obtained by executing the code of Figure 4.10, with statement 1 subject to a modulo unfolding transformation with splitting factor $N = 3$.

Aspect	RTL sim.	SystemC	MCM	cprof
Analytical	✘	✘	✔	✘
Functional validation	✔	✔	✘	✔
Runtime	min-hrs	minutes	seconds	seconds
Accuracy	very high	high	medium	high
Effort	medium	high	small	small
Buffer sizes	✔	✔	✔	✘
Reordering	✔	✘	✘	✔
Interconnect type	✔	✔	✘	✘
Intra-process overlap	✔	✔	✘	✔

Table 4.2: Characteristics of different estimation methods.

execution profiles. For each scalar variable or array element a shadow variable is instantiated. Thus, programs that contain many variables or large arrays result in programs with many shadow variables, increasing the memory footprint. The statement execution profiles mainly affect the memory footprint depending on the execution time. Thus, the memory footprint increases as the execution time of a program increases.

4.7 Comparison

We have discussed four performance estimation methods in the previous sections. How do these four methods compare to each other? In this section, we compare nine different aspects for the four estimation methods in Table 4.2. Below, we discuss each aspect in more detail.

Only the MCM method is *analytical*. This means it does not rely on actual execution of (a simulation model of) the PPN, but computes a performance estimate by analytical means. The other three methods rely on execution of the PPN. That is, each firing of each process is simulated during the estimation. As such, a *functional validation* can be performed with little additional effort, which allows a designer to verify functional correctness.

The *running time* of each method varies from minutes or hours for RTL simulation, to seconds for the MCM and cprof methods. The difference in running times is caused by the difference in the level of detail of the estimation method. For example, the RTL simulation method works at the level of logical gates and registers, while the cprof method works at the level of process firings. For most systems, the number of logical

gates and registers is a few orders of magnitude higher than the number of processes. As such, the RTL simulation method needs to update more simulation primitives per time step than the cprof method, which leads to a longer running time. Because the MCM method is analytical, its running time is independent of the process domain sizes and latencies. In contrast, the non-analytical methods are dependent on these factors, since they simulate every time unit of the system execution.

The *accuracy* of the RTL simulation method is very high, since the method uses the same RTL code that is used for implementation of the system. The accuracy of the SystemC and cprof methods is lower than the accuracy of the RTL simulation method, because low-level details of for example communication delays are omitted in the SystemC and cprof methods. Nevertheless, both approaches have a comparable accuracy, because they use the same characterization of process execution times. The MCM method only has high accuracy for PPNs with uniform dependence distances and without reordering communication. For PPNs with non-uniform dependence distances or reordering communication, the MCM method's accuracy decreases.

The *effort* for the designer to obtain a throughput estimate varies significantly between the four methods. The generation of an RTL simulation project is highly automated in ESPAM's ISE backend. Because the RTL simulation uses the same RTL that is also used for synthesis, no custom simulation models have to be developed. Still, some effort is required from the designer, such as integrating custom IP cores into the system. A SystemC simulation requires considerably more effort, in particular if the system contains custom processing or communication components for which no SystemC model exists. In such a case, the designer has to develop a SystemC model for the unsupported components before a SystemC simulation can be performed. The MCM and cprof methods are both fully automated and require no effort from the designer.

The cprof method currently does not take the finite *buffer sizes* of a PPN into account, as explained in Section 4.6.2. The other three methods do take buffer sizes into account, such that a blocking write resulting from a full FIFO buffer may result in a smaller throughput estimate.

Reordering channels are currently only supported by the RTL simulation and cprof methods. Reordering support for the RTL simulation method is provided by the synthesizable reordering buffer presented in Section 3.6. Reordering support for the SystemC simulation method requires development of a SystemC reordering buffer model. Reordering support for the MCM method requires further investigation. Reordering support for the cprof method is provided because tokens are not stored in a channel model, but are stored in shared random access memory instead.

The MCM and cprof methods assume fixed-latency communication between processes and do not take the *interconnect type* into account. In contrast, SystemC and

Name	Type	#processes	#channels (OO)	cyclic
mns10	kernel	4	3 (0)	no
grid	kernel	4	5 (0)	self
oddeven-sort	kernel	4	13 (0)	yes
dv97ex4	kernel	4	7 (2)	self
qr	kernel	8	15 (0)	yes
mmm	kernel	8	10 (0)	self
mvt	kernel	7	11 (1)	self
sobel	kernel	5	15 (0)	no
mp3dec	application	28	58 (0)	yes
mrvd-qrd	application	43	110 (0)	yes
mjpeg-enc	application	6	6 (0)	no
H.264dec	application	11	24 (0)	yes

Table 4.3: Characteristics of benchmarks used in experiments.

RTL simulation may include any type of communication component, such as a FIFO buffer or Network-on-Chip (NoC). However, the designer needs to develop a SystemC model to use a new interconnect type in a SystemC simulation.

The RTL simulation, SystemC simulation, and cprof methods support *intra-process overlapped execution* of subsequent iterations of a process. The MCM analysis method does not support overlapped execution of subsequent iterations, because it assumes sequential execution of all iterations when determining the execution times of the throughput modeling graph nodes.

4.8 Experimental Results

To assess the feasibility and accuracy of the four performance estimation methods presented in this chapter, we have performed experiments on twelve different applications. The first eight applications are small kernels, whereas the remaining six applications perform a larger amount of work per process. In Table 4.3, we list for each application the number of processes; the total number of channels; the number of out-of-order channels; and whether the PPN is cyclic, acyclic with selfloops, or truly acyclic. For example, application *dv97ex4* consists of 4 processes, and 7 channels of which two are out-of-order, and contains selfloops but no cycles involving multiple nodes.

Not all aspects are covered by each of the four throughput estimation methods. To

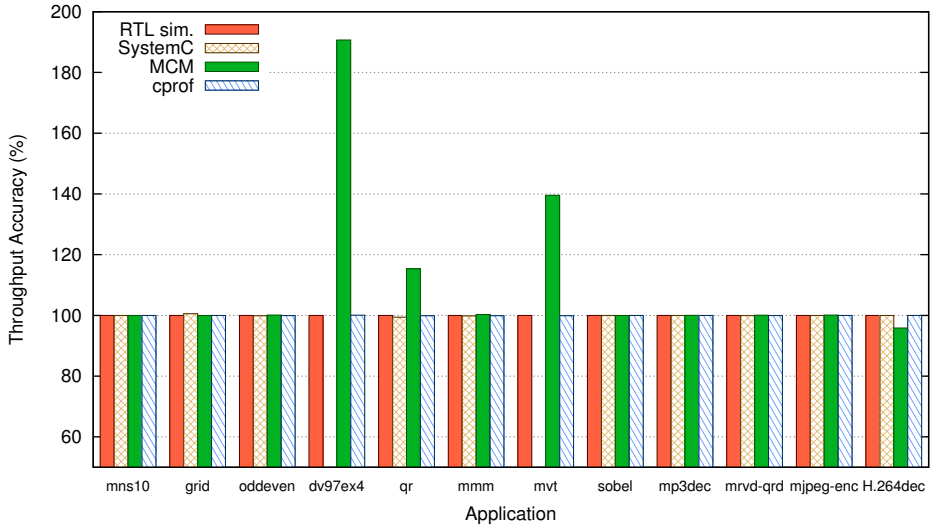


Figure 4.15: Accuracy of throughput estimation methods.

enable a comparison across the four methods, we make the following four assumptions. First, we assume buffer sizes are large enough to avoid performance penalties due to blocking write operations on full channels. That is, increasing any buffer size by any amount does not result in a higher throughput of the PPN. Second, we assume a fixed latency Λ_f for each firing of a function f. Third, we explicitly exclude overlapped execution between iterations of a process by setting each function II_f to $\Lambda_f + c$, where c equals the number of cycles to read and write a single token. As such, no overlap occurs between the read, execute, and write stages of a process. Fourth, we assume that all read operations of an iteration happen in parallel in one cycle according to the LAURA execution model. Similarly, we assume that all write operations of an iteration happen in parallel in one cycle.

4.8.1 Accuracy

We show the accuracy of each of the four methods for our set of twelve applications in Figure 4.15. On the vertical axis, we show the percentual deviation from the actual throughput value. We assume that RTL simulation gives a fully accurate assessment, and thus use the RTL simulation as the baseline for comparing accuracy of the SystemC simulation, MCM, and cprof methods.

As can be seen in Figure 4.15, only the accuracy of the MCM method exhibits significant deviations. The inaccuracy of *qr* and *H.264dec* is caused by incorporating

non-uniform dependences in the MCM modeling graph. The inaccuracy of *dv97ex4* and *mvt* is caused by out-of-order communication in the application. We cannot define tight bounds on the inaccuracy of the MCM method, nor whether the method overestimates or underestimates the actual throughput.

The SystemC and cprof methods deliver highly accurate results for all applications. The difference in reported throughput with RTL simulation is at most on the order of tens of clock cycles, which can be attributed to (re)initialization of components. SystemC simulations are missing for the *dv97ex4* and *mvt* applications, because ESPAM's current SystemC backend does not support reordering communication.

Using any of the four methods described in this chapter, the period of a PPN can be obtained. This period is expressed as a number of clock cycles. However, to obtain the absolute execution time of a PPN period, the number of clock cycles should be multiplied by the clock cycle length. This clock cycle length depends on factors such as combinational path lengths and routing delays. These factors are known only after place-and-route of the PPN's RTL implementation. Thus, none of the four methods allow obtaining throughput assessments expressed in absolute time.

4.8.2 Running Time

We have measured running times of the different estimation methods on an Intel Core 2 Duo system running at a 2400 MHz clock frequency and having 4 GB of RAM available. The running time for RTL simulation includes scripted Xilinx ISE 13.1 project creation, simulation model compilation using Xilinx Fuse 0.40d, and simulation model execution. The running time for SystemC simulation includes compilation and execution of the simulation model. The running time for MCM analysis includes generation of the model and execution of the SDF^3 MCM analysis tool [SGB06]. The running time for cprof includes generation, compilation, and execution of the instrumented code. For all estimation methods, we disable generation of waveforms or traces, to eliminate tracing overhead from the results.

We show the running times of each of the four methods for our set of twelve applications in Table 4.4. The running times for RTL simulation vary from tens of seconds for applications with small function latencies and small domains, to hours or even days for applications with large function latencies or large domains such as *mjpeg-enc*. The running times for SystemC simulation vary from seconds to minutes, making SystemC simulation a few orders of magnitude faster than RTL simulation. The running times for the MCM method are in all but two cases well below one second. Exceptions are *mp3dec* and *mrvd-qrd*, where the large number of edges and cycles in the PPN results in a large number of cycle means to be computed. The running times for cprof are in most cases below one second. Exceptions are *mjpeg-enc*

Application	**RTL sim.** (s)	**SystemC** (s)	**MCM** (s)	**cprof** (s)
mns10	46	2.6	0.1	0.3
grid	34	2.7	0.1	0.3
oddeven-sort	36	2.7	0.1	0.3
dv97ex4	46	n/a (OO)	0.1	0.3
qr	38	3.0	0.1	0.3
mmm	37	2.9	0.1	0.3
mvt	35	n/a (OO)	0.1	0.3
sobel	320	33	0.1	0.3
mp3dec	63	1.2	1.2	0.4
mrvd-qrd	64	4.6	140	0.4
mjpeg-enc	248433	220	0.1	6.4
H.264dec	13771	91	0.1	2.7

Table 4.4: Running times of different estimation methods.

and *H.264dec*, which have large function latencies on the order of thousands of clock cycles. These long latencies result in many updates to the execution profiles, which increases the total running time of cprof. In contrast, the running time of the MCM method does not depend on actual function latency values due to the analytical nature of the MCM method.

4.9 Conclusion and Summary

In this chapter, we have evaluated four different performance estimation methods for PPNs. The first is RTL simulation, which is often not attractive or feasible due the amount of time required to obtain a performance estimate for a given system. The second is SystemC simulation, which yields accurate results in significantly less time compared to RTL simulation. The third is a novel analytical approach for PPNs based on MCM analysis. Our MCM method is able to deliver accurate results for a subset of PPNs. However, we cannot define tight bounds on the inaccuracy of the MCM method, nor whether the method overestimates or underestimates the actual throughput. This model is theoretically attractive and gives insight in the behavior of a PPN, but is impractical because of the lack of accuracy bounds. The fourth is a novel profiling-based approach for PPNs, named cprof. This allows one to obtain accurate results, often in less than one second, without deriving a PPN. Moreover, cprof also allows assessment of the amount of parallelism in the application, and allows early

performance assessment of transformed versions of the applications without the need to actually transform the application.

Each performance estimation method works at a different level of the design phase, providing different tradeoffs between estimation time, effort, and accuracy. In particular the cprof method that works at the sequential code provides a fast, robust, and scalable performance assessment method. Given the characterization using Definition 3.1, cprof can deliver a very accurate performance estimate of a possibly heterogeneous system. As a result, the designer can perform the design iteration depicted in Figure 1.3 in significantly less time.

CHAPTER 5

APPLICATION TRANSFORMATION

In the previous chapter, we have presented different methods for fast performance evaluation of applications modeled as a PPN. In this chapter, we present how the results of such evaluations can be used to obtain alternative application instances. These alternative application instances are functionally equivalent, but differ in performance and resource cost characteristics. In Section 5.1, we discuss four transformations that we consider to automatically obtain alternative application instances from a given application specification. In Section 5.2, we present heuristics to select when and with which parameters the four transformations should be applied. In Section 5.3, we summarize this chapter.

5.1 Transformations

In the Daedalus design flow, the application is specified as a sequential program. By default, a single PPN process is constructed for each function call in the sequential program. The PPN obtained can easily be transformed in another PPN by transforming the sequential program such that a functionally equivalent PPN with different performance and resource cost characteristics is obtained. In this section, we consider the following four transformations: splitting (Section 5.1.1), merging (Section 5.1.2), stream multiplexing (Section 5.1.3), and scheduling (Section 5.1.4).

5.1.1 Splitting

To increase the amount of potential parallelism in an application modeled as a PPN, a designer can increase the number of processes. An established way to achieve this is by applying a process splitting transformation [SKD02, MNS09].

Definition 5.1 (Process Splitting Transformation)**.**
A *splitting transformation* with factor N on a PPN process p generates N copies of p that are identified as $p_0, p_1, \ldots, p_{N-1}$. These copies are referred to as the *partitions* of p. The original process iteration domain D_p is split into disjoint *subdomains* D_{p_0}, $D_{p_1}, \ldots, D_{p_{N-1}}$. The original process p is removed from the PPN.

Each partition executes the same function as the original process p, but for different iterations. The splitting transformation resembles a loop unrolling transformation in which a loop body is duplicated a number of times, or a loop splitting transformation in which a loop is split into multiple loops that each iterate over a subset of the iteration points of the original loop [Muc97, Chapter 17]. The original purpose of these loop transformations in a compiler is to increase instruction-level parallelism, whereas the purpose of process splitting in this dissertation is to increase the amount of coarser-grained task-level parallelism in an application. The process iteration domain of a process can be split using different systematic approaches to obtain different distributions of the points in the original process iteration domain [Ste04, SKD02]. In this chapter, we consider two different systematic approaches: modulo unfolding and plane cutting.

Definition 5.2 (Modulo Unfolding)**.**
A *modulo unfolding* splitting transformation, specified as $unfold(p, d, N)$, splits a process p into N partitions on dimension d. The process iteration domain of each instance p_i becomes

$$D_{p_i} = \{\mathbf{x} \mid \mathbf{x} \in D_p \ \wedge \ \mathbf{x}_d \bmod N = i\}.$$

Our $unfold$ transformation is defined for a single dimension d, whereas the UNFOLD procedure of Stefanov et al. is defined for multiple dimensions [Ste04, Chapter 3.3]. This merely serves to simplify our definition of $unfold$, motivated by our observation that unfolding transformations are often applied on a single dimension only. The behavior of Stefanov's UNFOLD procedure can always be obtained by applying $unfold$ on the partitions created by a previous $unfold$ transformation.

Definition 5.3 (Plane Cutting)**.**
A *plane cutting* splitting transformation, specified as $planecut(p, H)$, splits a process p using a set of hyperplanes $H = h_0, h_1, \ldots, h_{|H|-1}$. The hyperplanes divide process domain D_p into N subdomains, where N depends on the actual hyperplanes specified. For each subdomain x, a partition p_x with domain $D_{p_x} = x$ is created.

Process domains and dependence relations exhibit a regular structure when they are derived from static affine nested loop programs that have repetitive and regular behavior. As a result, the hyperplanes cutting such domains are closely related to

```
1  for (i = 0; i < 4; i++)
2    P1(&x[i]);
3
4  for (i = 0; i < 3; i++)
5    P2(&(y[i]) );
6
7  for (i = 0; i < 4; i++)
8    for (j = 0; j < 3; j++)
9      F(x[i], y[j], &x[i], &y[j]);
10
11 for (i = 0; i < 4; i++)
12   C(x[i]);
```

Figure 5.1: Sequential C code on which we demonstrate transformations.

each other. Therefore, we present an alternative way of specifying a plane cutting transformation, by specifying a single hyperplane h and a factor N instead of a set of hyperplanes H:

Definition 5.4 (Plane Cutting (alternative))**.**
Alternatively, a plane cutting transformation specified as $planecut(p, h, N)$ splits a process p into N instances using hyperplanes parallel to hyperplane h. A set of parallel hyperplanes H that divide D_p into N subdomains with comparable cardinalities are obtained by searching as explained by de Zwijger [Zwi12, Algorithm 1]. The process iteration domain of each instance p_i becomes

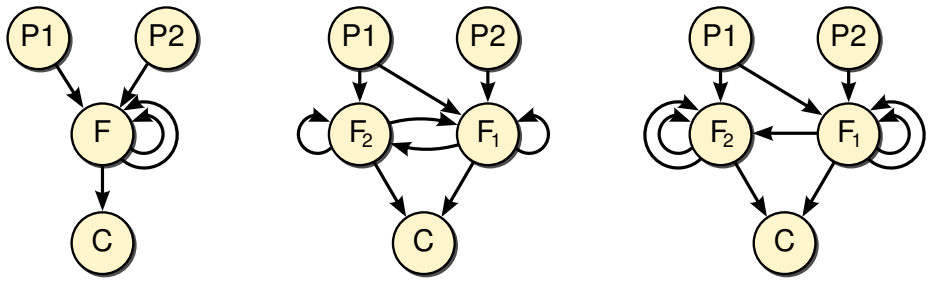
$$D_{p_i} = \{\mathbf{x} \mid \mathbf{x} \in D_p \ \wedge \ h_i \leq \mathbf{x} < h_{i+1}\}.$$

Examples

In Figure 5.2a, we show the PPN derived from the C program shown in Figure 5.1. In Figure 5.2b and 5.2c, we show the PPNs after applying modulo unfolding and plane cutting transformations on process F. We assume splitting factor $N = 2$ for both transformations, such that two partitions F_1 and F_2 are obtained.

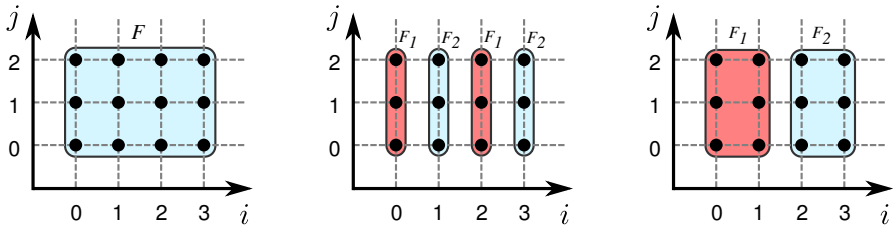
The original domain of process F is shown in Figure 5.3a. It consists of twelve points, corresponding to the twelve iterations of the for-loops at lines 7–8 in Figure 5.1. In Figure 5.3b, we show the two subdomains obtained after applying a modulo unfolding transformation $unfold(F, i, 2)$. The subdomain of F_1 consists of the six points in the original domain D_F for which $i \bmod 2 = 0$. The subdomain of F_2 consists of the remaining six points in the original domain D_F for which $i \bmod 2 = 1$.

In Figure 5.3c, we show the two subdomains obtained after applying a plane cutting transformation $planecut(F, \{i = 2\}, 2)$. The subdomain of F_1 consists of the six



a) Original. b) After modulo unfolding. c) After plane cutting.

Figure 5.2: A PPN and two transformed instances of the same PPN, obtained by splitting process F by a factor of two on its outermost dimension.



a) Original. b) Modulo unfolding. c) Plane cutting.

Figure 5.3: Process domains obtained after splitting by a factor of two.

points in the original domain D_F for which $i < 2$. The subdomain of F_2 consists of the remaining six points in the original domain D_F for which $i \geq 2$.

Position in Tool Flow

Splitting transformations not only affect the process being split, but also processes and channels adjacent to this process. For example, in the transformed PPNs shown in Figure 5.2b and 5.2c, process *P1* has two outgoing channels, whereas it has only one outgoing channel in the original PPN shown in Figure 5.2a. The precise implications for the adjacent processes and channels depend on the applied transformations. In the example shown in Figure 5.2, modulo unfolding results in one selfloop on process F_1, whereas plane cutting results in two selfloops on process F_1.

If we would apply the *unfold* and *planecut* operations on the PPN, then we should also update the adjacent processes and channels accordingly. Instead, we apply the *unfold* and *planecut* operations on the intermediate PDG that does not yet contain dependence information in the PNGEN tool flow [Zwi12]. Different from the ap-

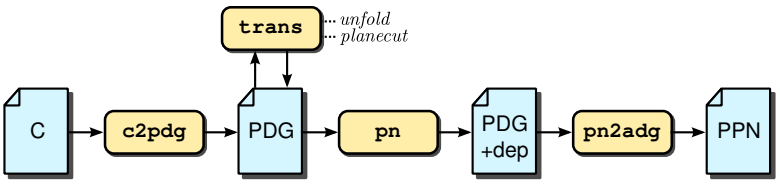


Figure 5.4: Application of splitting transformations in the PNGEN tool flow of Figure 2.9.

proach of Stefanov et al. [SKD02] that operates on the sequential code, we operate directly on polytopes. This is depicted by the `trans` block in Figure 5.4. Thus, we effectively apply transformations on an intermediate representation of the input program. The advantage of this approach is that the transformation only needs to operate on the process being split and its partitions. Adjacent processes and channels that result from the transformation are updated naturally by the PN tool, without any additional development effort needed for the `trans` block or the PN tool.

5.1.2 Merging

The splitting transformations discussed above increase the number of processes in a PPN. Assuming a separate processing resource is instantiated for each process, splitting transformations increase resource costs of PPN implementations. Complementary to splitting, the merging transformation combines processes into a single process, thereby decreasing resource costs.

Definition 5.5 (Process Merging Transformation)**.**
Application of a *merging transformation* on a set of PPN processes P results in a new *compound process* p_c which executes all firings originally executed by the processes in P. The original processes in P are removed from the PPN.

The domain cardinalities of the different processes in P are not necessarily equal. Thus, some processes in P should fire more often than others. Moreover, data dependence relations may exist between processes in P. The firings of these processes in the compound process should be scheduled such that these data dependence relations are not violated. We use the schedule computed by PNGEN to schedule the firings of the merged processes in the compound process, because this schedule includes all firings of all processes and respects data dependence relations. We refer to the work of Stefanov for further details on the merging transformation [Ste04, Chapter 3.6].

The merging transformation has been implemented in the ESPAM tool, where it can be applied by assigning multiple processes to the same processing resource in the

mapping specification. However, ESPAM only supports merging for programmable processing resources such as MicroBlaze processors. Merging LAURA processors is not supported in the current version of ESPAM. A workaround is possible if the compound process domain can be expressed as a convex polyhedral set. In such a case, the merged processes should be replaced by a compound process at the source code level. This form of merging is applied in Chapter 6.

5.1.3 Stream Multiplexing

For acyclic PPNs, the splitting transformations discussed above enable a designer to increase the throughput of a PPN. However, these splitting transformations may not yield any throughput increase for PPNs containing one or more cycles. This happens when the processes involved in a cycle depend on the output of a previous firing of its predecessor process, also known as a *recurrence* or *feedback*. As a result, the processes in a cycle may fire entirely sequentially, thereby preventing overlapped execution among the processes. Since the processes spend most of their time waiting for data in a blocking read state, their processing resources are idle for a considerable amount of time. A common solution to make use of these idle times is to process independent data streams. This can be done using a stream multiplexing transformation:

Definition 5.6 (Stream Multiplexing Transformation)**.**
Applying a *stream multiplexing transformation* with a factor N to a process p extends process domain D_p with an innermost dimension containing N points. For each value of N, process p operates on data that is not accessed for other values of N. This transformation is applied on all processes involved in a cycle of a PPN.

A stream multiplexing transformation neither increases nor decreases the latency or throughput of a single execution of the PPN. Only when multiple executions of the PPN are considered, the average period at which PPN executions finish is decreased, yielding an increase in throughput.

The stream multiplexing transformation resembles the software pipelining technique for programmable processors in which instructions from subsequent iterations of a loop are executed in an overlapping fashion [PD76, Lam88]. However, software pipelining works at the level of individual instructions, whereas our stream multiplexing transformation works at the level of coarser-grained tasks. Another difference is that software pipelining operates on the iterations of a given loop, whereas the stream multiplexing transformation introduces a new loop. Generation of a software pipelined loop for a programmable processor requires a sophisticated scheduling algorithm such as modulo scheduling. In contrast, applying the stream multiplexing

```
1 src(&v);
2 for (i = 0; i < 3; i++) {
3   P1(v, &v);
4   P2(v, &v);
5   P3(v, &v);
6 }
```

Figure 5.5: Sequential C code resulting in a PPN containing a feedback loop.

transformation on a PPN does not require any scheduling algorithm because of the self-scheduling semantics of the PPN model.

A technique closely related to stream multiplexing is C-slowing The C-slowing technique is often used in conjunction with register retiming to improve throughput of synchronous digital circuits [LRS93]. C-slowing replaces each register in a circuit with a sequence of C registers, such that C independent data streams can be in processed in an overlapped fashion. Retiming then tries to balance combinational path lengths by moving these registers through the combinational logic. As a result, the clock frequency and throughput may increase, at the expense of a higher latency caused by the additional registers. The C-slowing technique is closely related to the stream multiplexing transformation, as both add independent streams to overcome feedback in a design. However, the main purpose of C-slowing is to increase the clock frequency of a circuit, whereas the main purpose of stream multiplexing is to increase throughput of multiple executions of a PPN.

Zissules et al. conducted a case study on a QR decomposition algorithm for which they increased the number of independent streams [ZKD04]. This was done in an ad-hoc fashion, whereas our stream multiplexing provides a more systematic approach to accomplish the same goal.

Example

In Figure 5.5, we show a C program for which the corresponding PPN, shown in Figure 5.7a, contains a feedback loop involving *P1*, *P2*, and *P3*. In each execution of the PPN, processes *P1*, *P2*, and *P3* fire in sequence three times. Because each firing of these processes requires the output of the previous process through variable v, no overlapped execution occurs. This is depicted in Figure 5.8a.

In Figure 5.8b, we depict the firings of *P1*, *P2*, and *P3* after applying stream multiplexing with a second independent data set. That is, process *P1* starts operating on the first data "set" v_1 at time $t = 0$, and process *P1* starts operating on the second data set v_2 at time $t = 2$. As a result, two executions of the PPN complete in only

```
1  for (t = 0; t < F; t++)
2    src(&v[t]);
3  for (i = 0; i < 3; i++) {
4    for (t = 0; t < F; t++)
5      P1(v[t], &v[t]);
6    for (t = 0; t < F; t++)
7      P2(v[t], &v[t]);
8    for (t = 0; t < F; t++)
9      P3(v[t], &v[t]);
10 }
```

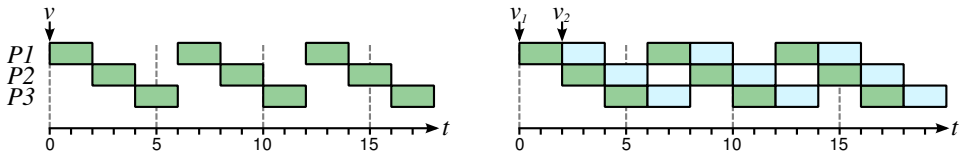
Figure 5.6: Applying stream multiplexing by a factor F to the program of Figure 5.5.

a) Original PPN. b) Corresponding PPN modeling graph.

Figure 5.7: PPN and PPN modeling graph derived from the C code in Figure 5.5.

slightly more time than needed for a single execution of the PPN. In Figure 5.6, we show the equivalent C program implementing a stream multiplexing transformation by a factor F. The transformation consists of applying a scalar expansion on all variables and adding a loop of F iterations. The scalar expanded variables are indexed using the iterator of the newly added loop.

After applying a stream multiplexing transformation of a factor two, each process is still idle for one third of the time, as shown by the gaps between the filled boxes in Figure 5.8b. This means applying a stream multiplexing transformation of a factor three would still not increase the latency of a single execution of the PPN but increase throughput of multiple executions. After stream multiplexing by a factor three, no processes are idle, which means three is the maximum stream multiplexing factor that does not increase latency for the given PPN. A stream multiplexing factor of four or higher would increase the latency of a single execution, because at time $t = 6$ process *P1* would start processing the fourth data set v_4, while output from *P3* belonging to the first data set is also available for processing by *P1*. In such a case, the splitting transformations can be considered to further increase throughput, because the cycle no longer limits throughput.



a) Without stream multiplexing. b) With stream multiplexing by a factor 2.

Figure 5.8: Flat execution profiles for the C code of Figure 5.5 and Figure 5.6.

5.1.4 Scheduling

In the previous chapters, we have assumed that each process in a PPN traverses its process iteration domain in the lexicographical order. Depending on the presence of data dependence relations between iterations, alternative execution orders of the points in the process iteration domain may exist that preserve all data dependence relations. Some of these alternative execution orders may yield a higher throughput when the iterations are executed in a pipeline fashion on for example a LAURA processor, which we show in an example below. We change the order in which the points of a process iteration domain are executed by applying a scheduling transformation.

Definition 5.7 (Process Scheduling)**.**
A *process scheduling* transformation on a process p, specified as $schedule(p)$, modifies the execution order of iterations such that independent iterations are grouped together and executed in sequence.

We distinguish between two types of schedules in a PPN: local schedules and global schedules. A *local schedule* defines the execution order of different iterations of an individual process. A *global schedule* defines the firing order of the different processes in a PPN. The scheduling transformation solely affects the local schedules of processes.

Motivating Example

We illustrate the process scheduling transformation using the PPN shown in Figure 5.2a, which was derived from the C code shown in Figure 5.1. Data dependences require that iteration $(0, 0)$ executes before iterations $(0, 1)$ and $(1, 0)$. Similarly, $(0, 1)$ should execute before $(0, 2)$.

When executing the iterations according to the original lexicographical order, we do not achieve the highest degree of overlapped execution. When implementing F using a P-stage pipeline and following the lexicographical order, execution of the first four iterations takes $3P + 1$ clock cycles, as depicted in Figure 5.9. However,

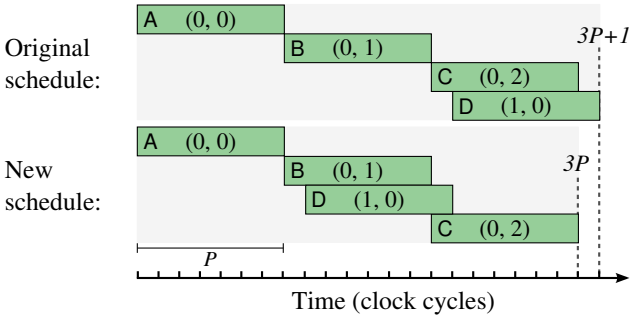


Figure 5.9: Pipeline behavior for two different schedules of the PPN shown in Figure 5.2a.

if we execute the first four iterations in the order $(0,0)$, $(0,1)$, $(1,0)$, $(0,2)$, we still respect data dependences but execution takes only $3P$ cycles. Although in this simplified scenario the gain is only one clock cycle, we have observed that changing the iteration execution order may increase throughput up to $2.7\times$ for applications such as QR decomposition [HK12].

Previous works have found that applying a skewing transformation to source code and then converting the transformed source code to a PPN may increase throughput of the PPN [SKD02, ZKD04, HK09]. A skewing transformation on the appropriate loop results in the same throughput increase for our motivating example. Thus, skewing is an effective way to increase overlapped execution, and consequently, improve pipeline utilization. However, identifying the skewing transformation parameters, such as the loop to skew, requires thorough studying of the application. Therefore, we present an automated approach to find an alternative execution order of process iterations that yields better pipeline utilization.

Scheduling PPN Processes

When applying a scheduling transformation, we use affine schedules to compactly define an execution order on the points of a process iteration domain:

Definition 5.8 (Affine Schedule)**.**
An *affine schedule* is a polyhedral map that assigns a positive time stamp to each point $\mathbf{i}$ of a process iteration domain. In this chapter, we denote an s-dimensional affine schedule as [1]

$$\theta = \left\{ \mathbf{i} \rightarrow \mathbf{i'} \mid \mathbf{i'} = H \cdot \mathbf{i} + \mathbf{h} \right\},$$

where $\mathbf{i}$ is an n-dimensional iteration vector, $\mathbf{i'}$ is an s-dimensional time stamp vector, H is an $n \times s$ matrix, and $\mathbf{h}$ is a vector of size s.

For a given iteration $\mathbf{i} \in D_p$, computing $\theta(\mathbf{i})$ yields a time stamp at which iteration $\mathbf{i}$ can execute. These time stamps should not be interpreted as absolute time, but rather as a partial order on the iterations in D_p. We assume that execution of an iteration takes one time unit and that sufficient processing resources are available to execute all iterations with the same time stamp simultaneously. Two affine schedules θ_p and θ_q for two dependent processes p and q are *valid* if for all pairs of dependent write and read operations $(\mathbf{w}, \mathbf{r})$, the schedules enforce that the write operation is executed before the read operation. That is, when a write operation $\mathbf{w}$ of p produces data for a read operation $\mathbf{r}$ of q, then $\theta_p(\mathbf{w}) \prec \theta_q(\mathbf{r})$ should hold. In the remainder of this chapter, we only consider valid schedules.

As an example, consider the affine schedule

$$\theta = \{(i, j) \rightarrow i + j\}. \tag{5.1}$$

For iteration $(1, 2)$, the schedule yields 3 which means the iteration can execute at time 3. For iteration $(2, 1)$, the schedule also yields 3. This means that both iterations can execute at the same time and, assuming that the schedule is valid, that both iterations can execute in parallel.

If a schedule is multidimensional, that is, $s > 1$, then execution times are ordered according to the lexicographical order. For example, a schedule

$$\theta = \{(i, j) \rightarrow (i + j, j)\} \tag{5.2}$$

yields $\theta(1, 2) = (3, 2)$ and $\theta(2, 1) = (3, 1)$. Because $(3, 1) \prec (3, 2)$, iteration $(2, 1)$ should execute before iteration $(1, 2)$.

A PPN process traverses its process domain in a sequential fashion according to the lexicographical order, which is a total order. That is, for any two iterations $\mathbf{i_1}$ and $\mathbf{i_2}$, the lexicographical order defines which iteration is executed first. To comply with the

[1] In literature, e.g., [Fea92a], an affine schedule is often denoted alternatively as

$$\theta(\mathbf{i}) = H \cdot \mathbf{i} + \mathbf{h},$$

where $\mathbf{i}$, H, and h follow those of Definition 5.8.

PPN process semantics, we should consider only those affine schedules that define a total order on the iterations of a process domain. The one-dimensional schedule of Equation (5.1) yields the same time stamp for different iterations, which implies it does not define a total order. Extending this schedule to the two-dimensional schedule of Equation (5.2) results in a schedule that yields a unique time stamp for each possible pair of positive values (i, j). This property allows us to use the two-dimensional schedule of Equation (5.2) in a process scheduling transformation. A schedule is said to be *bijective* if it assigns a unique time stamp to each distinct iteration vector.

To *apply* a scheduling transformation on a process p, we modify the process domain D_p to reflect the order given by an affine schedule θ_p. That is, we transform the process domain D_p into a new domain D'_p. For bijective schedules, each point in D_p has exactly one corresponding point in D'_p. The new domain D'_p is obtained by polyhedral map application of the schedule to the process domain:

$$D'_p = \theta_p(D_p). \tag{5.3}$$

The resulting domain D'_p is again traversed according to the lexicographical order.

Example Application of a Schedule

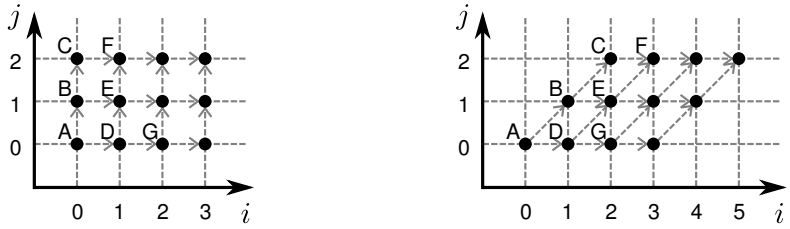
We illustrate application of a schedule using the PPN shown in Figure 5.2a. We apply a new schedule on process *F* of this PPN. The domain of this process is extracted from the for-loops in the sequential code of Figure 5.1 as

$$D_F = \left\{ (i,j) \in \mathbb{Z}^2 \;\middle|\; \begin{array}{l} 0 \leq i \leq 3 \wedge \\ 0 \leq j \leq 2 \end{array} \right\}.$$

By applying the two-dimensional schedule of Equation (5.2), we obtain a new domain

$$D'_F = \left\{ (i',j') \in \mathbb{Z}^2 \;\middle|\; \begin{array}{l} j' \leq i' \leq j' + 3 \wedge \\ 0 \leq j' \leq 2 \end{array} \right\}.$$

In Figure 5.10, we show the original and the transformed process domains. Both domains have the same cardinality because each point shown in Figure 5.10a has a counterpart in Figure 5.10b that can be obtained by applying the schedule to the point. To indicate the correspondence between points in the original and transformed domains, we have labeled seven points with a letter. For example, the counterpart of point $(1, 1)$ labeled 'E' is $(2, 1)$. The same labels are used in Figure 5.9. Traversal of the original domain according to the lexicographical order results in the execution order A, B, C, D, ... Traversal of the transformed domain according to the lexicographical order results in the execution order A, D, B, ..., C, ... This corresponds



a) Original. b) After scheduling.

Figure 5.10: Process domains of F of Figure 5.2a before and after application of the schedule in Equation (5.2). A data dependence from one iteration point of F to another is indicated by an arrow.

to the new schedule depicted in the bottom part of Figure 5.9 in which 'D' is moved forwards to execute in a pipeline fashion with 'B'. Thus, by applying the schedule of Equation (5.2), we achieve the desired overlapped execution. Below, we discuss how to obtain a schedule for a given PPN.

Determining a Schedule

The chosen schedule affects the degree of overlapped execution between process iterations that is achievable by a scheduling transformation. Finding a schedule that maximizes overlapped execution is a non-trivial optimization problem. A natural way to overlap execution of process iterations is to perform loop parallelization. This is a well-studied field in compiler technology, in which various loop parallelization algorithms have been proposed. Existing algorithms differ in the way they represent the data dependence relations of nested loop programs. For example, Allen and Kennedy's algorithm [AK87], Wolf and Lam's algorithm [WL91], and Darte and Vivien's algorithm [DV97] take as input an approximation of the dependence graph. Such an approximation restricts the ability of the algorithms to reveal all available parallelism [DRV01]. On the other hand, Feautrier's algorithm [Fea92a, Fea92b] takes the exact dependence graph as input and is therefore more powerful than the other algorithms. Also, Feautrier's algorithm will find the optimal schedule if it can be expressed as an affine mapping of the iteration space. Lim and Lam's algorithm takes a similar input representation as Feautrier's algorithm, but maximizes parallelism while minimizing the number of synchronizations [LL98].

Feautrier's algorithm is employed by e.g. the MMAlpha tool [GQR03] to generate hardware from algorithms specified in the Alpha language. Feautrier's algorithm has a high computational complexity, which motivated Feautrier to apply the algo-

rithm to sets of communicating regular processes [Fea06]. Unfortunately, Feautrier does not elaborate on the implications of the new schedule for the communication channels between processes. Later in this section, we show that these implications cannot always be ignored. Another way to address the computational complexity of Feautrier's algorithm and control flow overhead of the resulting schedules is by limiting the possible schedule coefficients [PBCC08]. This results in simpler schedules, at the expense of more scheduling dimensions, which may counteract the benefits of simpler schedules.

Applying Feautrier's Scheduling Algorithm to PPNs

Feautrier's multidimensional scheduling algorithm takes as input a *Generalized Dependence Graph* (*GDG*) represented as $G = (V, E, \mathcal{D}, \mathcal{R})$, where

- V is a set of vertexes representing statements,
- E is a set of edges representing data dependences,
- $\mathcal{D}$ is a set containing a polyhedral set for each vertex, and
- $\mathcal{R}$ is a set containing a polyhedral map for each edge.

Given a GDG, the algorithm constructs a multidimensional schedule for each statement in a greedy fashion. In each step, the algorithm constructs a linear program to find an affine function with minimum latency that satisfies as many dependence relations as possible. The dependences that are not satisfied are considered in a subsequent recursive step. Each recursive step leads to a new dimension in the schedule being constructed. The algorithm terminates when all dependences are satisfied, or when no affine schedule can be found.

We are interested in Feautrier's algorithm for two reasons. First, Feautrier's algorithm finds the *optimal* schedule if it can be expressed in the affine form of Definition 5.8. That is, no other affine schedule exists that yields a lower execution latency. This implies that Feautrier's algorithm includes all transformations that can be expressed using an affine mapping of an iteration domain, such as loop interchange or loop skewing [Fea92b, Viv02]. Second, we do not have to perform any additional analysis to run Feautrier's algorithm on a PPN because all input needed for Feautrier's algorithm is already made available by the exact data dependence analysis step of PNGEN.

To apply Feautrier's scheduling algorithm to a PPN, we relate statements to processes and dependences to channels. That is, for each process p, we add a vertex representing p to V and we add the process domain D_p to $\mathcal{D}$. For each channel c, we add an edge representing e to E and we add the channel relation M_c to $\mathcal{R}$. Feautrier's algorithm computes an affine schedule for each vertex. We apply the schedule of

each vertex to the corresponding process domain according to Equation (5.3). As a result, all processes of a PPN execute their iterations in the optimal order found by Feautrier's algorithm, potentially increasing overlapped execution.

Extension to a Bijective Schedule

The schedule returned by Feautrier's algorithm is not necessarily a bijective schedule. In fact, the schedule is only bijective if no overlap between any pair of iterations is possible, which occurs only if an application is entirely sequential. When two iterations may execute in parallel, then the schedule yields the same time stamp for both iterations. To comply with the PPN process semantics, we should extend the schedule with one or more dimensions such that for each iteration the schedule yields a unique time stamp.

We use the default algorithm of `isl` [Ver08] to extend the schedule found by Feautrier's algorithm to a bijective schedule. This default algorithm minimizes the dependence distance over the dependences, using an approach similar to Pluto's [BBK+08]. For our running example schedule of Equation (5.1), extending the schedule using `isl` gives the schedule of Equation (5.2) in which a second dimension containing j has been added.

Impact of Scheduling

The schedule computed by Feautrier's algorithm does not necessarily enforce in-order communication of data between processes. Thus, after applying the schedule, the order in which tokens are produced by the producer process may be different from the order in which tokens are consumed by the consumer process, and vice versa. This requires us to perform a reordering test [TKD07] on each channel after applying a scheduling transformation. Some channels may be classified as out-of-order after scheduling, and thus these should be implemented using a reordering buffer to preserve the functional behavior of the original application.

Existing reordering buffer designs were shown to have a considerable negative impact on both performance and resource usage [TKD03]. To avoid counteracting the performance benefits of a better schedule because of possible reordering communication, we use the reordering buffer that was presented in Section 3.6. Read and write operations on this reordering buffer take only one clock cycle. This means that replacing a FIFO buffer with a reordering buffer increases resource usage, but does not introduce additional delay cycles. As a result, we avoid counteracting the benefits of a better schedule.

```
1 for (i=0; i<=5; i++) {
2   for (j=max(0,i-3); j<=min(2,i); j++) {
3     F(x[i], y[j], &x[i], &y[j]);
4   }
5 }
```

Figure 5.11: Code to traverse the transformed iteration space of Figure 5.10b in the lexicographical order.

Another consequence of the scheduling transformation is the complexity of the evaluation logic blocks of a LAURA processor. The complexity of iterating through a rescheduled domain typically increases. To illustrate this using our running example, we show the code which iterates over the rescheduled process domain in the right part of Figure 5.11. This code is more complicated than the code iterating over the original process domain, because loop bounds of the `j`-loop are now `max` and `min` expressions involving `i`. This increases the combinational path lengths in the RTL for the evaluation logic blocks of the read and write units shown in Figure 2.12, affecting both resource usage and the maximum achievable clock frequency. From experiments, we found that control overhead induced by a scheduling transformation may reduce the clock frequency by 50%, potentially negating the benefits of increased overlapped execution. To avoid that control overhead counteracts the benefits of a better schedule, a designer may for example choose to consider the evaluation logic optimization techniques described in Section 3.5.2.

5.2 Transformation Efficacy Analysis

In the previous section, we have discussed four different transformations that can be applied on a PPN. Many combinations of these transformations are possible to obtain design points that provide different tradeoffs between circuit area and performance. Deciding which transformations to apply to obtain a particular design point is not trivial. In this section, we present how the results of PPN throughput analysis can be applied to assess the efficacy of transformations. That is, for the transformations that we consider, we discuss the conditions when a particular transformation should be applied to obtain a particular change on PPN throughput.

5.2.1 Splitting

Throughput of a PPN can often be increased by applying one of the splitting transformations discussed in Section 5.1.1 on a process. To apply a splitting transforma-

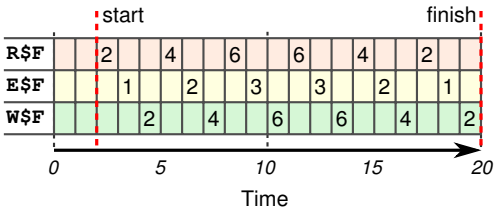


Figure 5.12: Statement execution profile for function F of the program shown in Figure 5.1. Empty cells represent zero.

tion, the designer should select a splitting method such as modulo unfolding or plane cutting, and the splitting factor. Selection of the splitting method was discussed extensively by Meijer et al. [MNS09]. However, Meijer's algorithm still requires the designer to specify a splitting factor. We therefore present heuristics to find a splitting factor in this section.

An obvious upper bound on the splitting factor of a process is the cardinality of the process domain. If the splitting factor for a process is chosen higher than the domain cardinality, then some partitions resulting from the splitting transformation contain zero iterations, meaning that a lower splitting factor would suffice as well.

Maximum Iteration Overlap

Another upper bound on the splitting factor of a process is the maximum *iteration overlap*. We define iteration overlap as the number of process iterations that can execute simultaneously at a given time, assuming a sufficient number of processing resources is available. The maximum iteration overlap thus represents the maximum number of process iterations that can execute simultaneously during the entire execution of the PPN. We propose two different methods to obtain this upper bound: by profiling or by analytical means.

Profiling-based Determination of Maximum Iteration Overlap

For the profiling-based method we employ cprof to obtain the maximum iteration overlap. We profile the sequential application code on a hypothetical ideal machine with an infinite number of processing resources as described in Section 4.6.3. We can extract the iteration overlap at a given time t from the statement execution profile of a process using Equation (4.6). By ranging t between the process start and finish times we obtain the maximum iteration overlap.

In Figure 5.12, we show an execution profile obtained using cprof for function F of the program shown in Figure 5.1. The first iteration $(0, 0)$ of the process derived from

the function call to F starts at time 2. Since F takes two input arguments, two read operations are executed which execute in parallel on the ideal machine. At time 3, the function executes and at time 4, the two output arguments are written. Considering the entire execution profile E$F in the range $[2, 20)$, at most three iterations of F execute in parallel. Thus, the maximum iteration overlap for the program of Figure 5.1 equals three.

Analytical Determination of Maximum Iteration Overlap

For the analytical method we employ Feautrier's multi-dimensional scheduling algorithm. Recall from Section 5.1.4 that for each process iteration $\mathbf{i}$, we can use Feautrier's algorithm to compute the earliest timestamp $\mathbf{t} = \theta_p(\mathbf{i})$ at which $\mathbf{i}$ can execute. This earliest timestamp is solely determined by the data dependences of the application. Feautrier's algorithm assumes no processing resource contention occurs, resembling an ideal machine. For iterations that execute in parallel, the schedule yields the same timestamp. To find out the maximum iteration overlap for a given schedule, we compute the maximum number of iterations executing at the same timestamp.

The iterations executing in parallel at a given timestamp $\mathbf{t}$ are given by the inverse of the schedule

$$\theta_p^{-1}(\mathbf{t}), \quad \text{where } \mathbf{t} \text{ is in the range of } \theta_p(\mathbf{i}), \forall \mathbf{i} \in D_p.$$

That is, we only consider timestamps $\mathbf{t}$ at which one or more points in the domain execute. A piecewise quasipolynomial that gives the number of iterations executing in parallel at a time $\mathbf{t}$ can be found by computing the cardinality using the barvinok library. The upper bound on this piecewise quasipolynomial represents the maximum number of iterations executing in parallel, and is given by

$$\max \left| \theta_p^{-1}(\mathbf{t}) \right|. \tag{5.4}$$

This upper bound can be found using the barvinok library.

We illustrate the analytical method using the schedule of Equation (5.1) for the function call to F of the program shown in Figure 5.1. The iterations executing at a timestamp t are given by the inverse polyhedral map

$$\theta^{-1}(t) = \{t \rightarrow (i, t - i) \mid 0 \leq i \leq 3 \wedge i \leq t \leq i + 2\},$$

which can be obtained using isl. For example, computing $\theta^{-1}(1)$ tells us that iterations $(0, 1)$ and $(1, 0)$ can execute in parallel at $t = 1$. This can be verified by looking at Figure 5.10a: iterations B and D only depend on A, since B and D only have an incoming arrow from A. Thus, once A has been executed, both B and D can execute.

The number of iterations that can execute at a given time t is given by the piecewise quasipolynomial

$$|\theta^{-1}(t)| = \begin{cases} t+1 & \text{if } 0 \leq t \leq 2, \\ 6-t & \text{if } 3 \leq t \leq 5, \\ 0 & \text{otherwise.} \end{cases}$$

The upper bound of this piecewise quasipolynomial equals 3, implying that at most three iterations can execute in parallel in the program of Figure 5.1. This is in agreement with the value found by means of profiling-based determination of maximum iteration overlap: at most three iterations execute simultaneously, as shown by profile `E$F` in Figure 5.12.

Average Iteration Overlap

Using both the profiling-based and analytical approaches discussed above, we found that at most three iterations execute in parallel in the program of Figure 5.1. Thus, an upper bound on the splitting factor is three. However, only during two out of six occasions, three iterations actually execute in parallel, and in four out of six occasions, a third processor would be idle.

Using the maximum iteration overlap as a splitting factor then results in a system in which some processors are used only during these few points in time. This may result in a high area overhead while a slightly lower throughput could be achieved with significantly less processors. Therefore, the average number of process iterations executing simultaneously may provide a better tradeoff between throughput and resource cost, as proposed by Eager et al. [EZL89]. We propose two different methods to obtain the average iteration overlap: by profiling or by analytical means.

Profiling-based Determination of Average Iteration Overlap

To determine the average iteration overlap by profiling, we again leverage cprof's application analysis method presented in Section 4.6.3. We extract the average iteration overlap from the statement execution profile of a process by dividing the process domain cardinality by the number of non-zero entries in `E$`.

Using Figure 5.12, we find the average iteration overlap for function `F` in the program of Figure 5.1. The process domain of `F` consists of twelve points. The execution profile `E$F` consists of six non-zero entries. Thus, the average iteration overlap is $\frac{12}{6} = 2$.

Analytical Determination of Average Iteration Overlap

To determine the average iteration overlap analytically, we again leverage Feautrier's algorithm. Instead of computing the maximum number of iterations using Equation (5.4), we compute

$$\frac{\sum_{\mathbf{t}\in\Theta_p} \left|\theta_p^{-1}(\mathbf{t})\right|}{|\Theta_p|}, \quad \text{where} \quad \Theta_p = \{\theta_p(\mathbf{i}) \mid \mathbf{i} \in D_p\}\,. \tag{5.5}$$

That is, we evaluate the piecewise quasipolynomial at every timestamp and sum these evaluations, which can be done using the `barvinok` library [Ver03a]. We then divide by the total number of timestamps to obtain the number of iterations executing in parallel on average.

For function F in the program of Figure 5.1, Equation (5.5) evaluates to

$$\frac{1+2+3+3+2+1}{6} = 2.$$

Thus, the average iteration overlap is two.

Depending on design constraints, different upper bounds on the process splitting factor may be considered. If maximum performance is required regardless of resource cost, then the maximum iteration overlap should be used as an upper bound. If a less expensive solution is required, then the average iteration overlap provides an upper bound that provides a good balance between resource cost and performance, as shown by Eager et al. [EZL89].

5.2.2 Merging

Meijer et al. investigated applying the merging transformation on programmable processors such as the MicroBlaze [MNS10]. In this section, we investigate application of the merging transformation on LAURA processors. In the general case of LAURA processor merging, resource cost savings are limited, because the IP cores implementing each process' functionality should still be provided. These IP cores often account for the greater part of the LAURA processor cost. However, when LAURA processors execute the same function, then a merging transformation can reduce resource cost by *resource-sharing* the IP core among the processes in the compound process.

The processes merged onto the same LAURA processor compete for the same IP core of the LAURA processor. This may cause a decrease in throughput if at least one of these processes is in the critical path. Therefore, two LAURA processors should only be merged if they do not execute at the same time. This can be determined by

inspecting statement execution profiles obtained from cprof. For example, assume the arrays `E$1_0`, `E$1_1`, and `E$1_2` in Figure 4.14 represent the execution profiles of three separate LAURA processors. That is, these arrays indicate when the IP core of the LAURA processor is active during the execute stage of a process iteration. At most one of the three `E$`-arrays contains a one at any time, meaning that at most one of the three LAURA processors is active at any time. Therefore, we conclude that merging these three LAURA processors into a single LAURA processor would not affect throughput.

5.2.3 Stream Multiplexing

The stream multiplexing transformation aims to increase throughput of multiple executions of a PPN containing a feedback loop. A stream multiplexing transformation can still be beneficial when the cycle mean of the feedback loop cannot be decreased by other transformations of processes in the feedback loop, such as for example a splitting transformation, or by replacement of a programmable processor with a dedicated hardware IP core. We first identify two conditions when a stream multiplexing transformation can be beneficial. We then discuss how to determine the maximum stream multiplexing factor such that the latency of a single PPN execution is not increased.

Efficacy Conditions

A first condition is that a complete execution of the PPN is independent of the previous execution of the PPN, to enable interleaving of multiple executions. This is often the case for the streaming applications that we consider, as the PPN often works on discrete and independent units of the incoming data stream such as video frames.

A second condition is that the PPN should have a feedback loop that limits throughput of a single execution of a PPN. Such a feedback loop can be detected using the MCM analysis technique presented in Section 4.5. Computing the cycle means of a PPN reveals which parts of a PPN prevent meeting a target throughput τ. The cycle means that are greater than $T = \frac{1}{\tau}$ represent parts of the PPN that prevent meeting throughput τ. The cycle means are the result from three different classes of cycles that occur in the PPN modeling graph:

1. cycles involving only one process;
2. cycles resulting from feedforward edges; and
3. cycles resulting from feedback edges.

These three cycle classes stem from the three channel classes identified for the construction of edges in the PPN modeling graph that is discussed in Section 4.5.3.

Cycle	Cycle mean	Class	Feedback loop?
src → *src*	3	first	no
P1 → *P1*	9	first	no
P2 → *P2*	18	first	no
P3 → *P3*	24	first	no
src → *P1* → *src*	6	second	no
P1 → *P2* → *P3* → *P1*	51	third	yes

Table 5.1: Cycles in the PPN modeling graph for a PPN derived from Figure 5.5.

Cycles of the first class always originate from the selfloop added to the PPN modeling graph to eliminate auto-concurrency of a process. If the corresponding cycle mean is greater than T, then a period of T time units cannot be achieved due to sequential execution of all process iterations on a single processing resource. A limitation of the MCM analysis technique is that pipelined execution of multiple process iterations cannot be captured, because the MCM analysis technique does not incorporate the II value of a process. The actual execution time of a process may be lower than reported by the MCM technique if $II < \Lambda$ such that pipelined execution of process iterations is possible. The actual execution time of a pipelined process depends on the presence of selfloops in the original PPN. Such selfloops represent a feedback loop in which an iteration depends on the output of a previous iteration. We therefore consider a cycle of the first class as a feedback loop if the original PPN contains a selfloop for the process in the cycle.

Cycles of the second class originate from the backedges added to model finite buffer sizes. If the corresponding cycle mean is greater than T then the cycle represents a buffer whose size is too small to sustain period T. These cycle means can be prevented from being the maximum cycle mean by enlarging buffer sizes such that they do not affect performance. In Section 4.5.3, we have described how initial tokens on backedges can be chosen such that cycles of this second class never have the maximum cycle mean. We therefore ignore cycles of the second class when analyzing the PPN for feedback loops.

Cycles of the third class originate from cycles present in the original PPN. If the corresponding cycle mean is greater than T then the cycle represents a bottleneck inherent in the application. We therefore always consider a cycle of the third class as a feedback loop.

As an example, we consider the PPN shown in Figure 5.7a. The PPN modeling graph constructed from this PPN is shown in Figure 5.7b. The cycles in the PPN

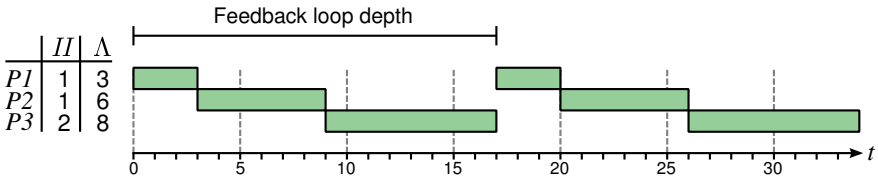


Figure 5.13: Flat execution profiles for the code of Figure 5.5.

modeling graph are listed in Table 5.1. All four cycles of the first class are not considered as a feedback loop, because the original PPN does not contain a selfloop for any of the processes in these cycles. As discussed above, cycles of the second class are never considered as a feedback loop. The PPN modeling graph contains one cycle of the third class, which is considered a feedback loop. We therefore conclude that a stream multiplexing might be beneficial and proceed to determine the maximum stream multiplexing factor.

Maximum Stream Multiplexing Factor

The maximum stream multiplexing factor is the maximum number of PPN executions that can be interleaved without increasing the latency of a single PPN execution. We illustrate how the maximum factor can be found using the flat execution profiles shown in Figure 5.13. These flat execution profiles are obtained by profiling the code of Figure 5.5 using cprof with the II and Λ values for each process shown in the left part of Figure 5.13. The factor is determined by the depth of the feedback loop and the II of the process functions involved in the feedback loop. We represent a feedback loop as a set of PPN channels $C \subseteq \mathcal{E}$.

The depth of a feedback loop C is the number of clock cycles since the start of the first process in the feedback loop until the next firing of the first process in the feedback loop. The feedback loop depth can be determined from the flat execution profiles obtained using cprof. For the flat execution profiles shown in Figure 5.13, we find that the feedback loop depth is 17 clock cycles. Alternatively, the feedback loop depth can be determined by analysis of the PPN. The dependence distance of a channel $(a \rightarrow b)$ represents the distance between process a and b as an iteration count. We use the channel size as a scalar approximation of the dependence distance, as motivated in Section 4.5.3. The sum of the dependence distances of the channels in the feedback loop gives the feedback loop depth expressed as an iteration count. To obtain the feedback loop depth $\mathit{depth}(C)$ expressed in terms of clock cycles, we multiply the size of each channel $c \in C$ with the latency of the process that writes to

c:

$$depth(C) = \sum_{c \in C} S_c \cdot \Lambda_{\sigma(c)},$$

with S_c being the size of channel c.

After determining the feedback loop depth, we compute the number of PPN executions that can be interleaved by dividing the feedback loop depth by the maximum II of all processes in the feedback loop. For the example of Figure 5.13, the maximum II of all processes is two because of process *P3*. Dividing the maximum feedback loop depth by the maximum II gives the number of independent executions of the feedback loop that can be interleaved. For the example of Figure 5.13, dividing 17 by 2 gives 8.5, which we round down to eight complete executions. Thus, a stream multiplexing transformation with a factor of eight can be applied to increase the throughput of multiple executions of the PPN, without increasing the latency of a single execution of the PPN.

5.2.4 Scheduling

Processes containing deeply pipelined IP cores may suffer from pipeline underutilization which limits throughput. Such underutilization is caused by a data dependency of the current iteration on a previous iteration that is still in the pipeline. Using the scheduling transformation presented in Section 5.1.4, the distance between dependent iterations can be altered, such that a higher throughput may be obtained. However, a scheduling transformation only increases throughput under certain circumstances, while it increases the control overhead of a LAURA processor in many cases. We therefore identify the following four criteria to assess the efficacy of a scheduling transformation on a process.

1. The purpose of a scheduling transformation is to increase pipeline utilization. Thus, the processor onto which a process is mapped should allow pipelined execution of process iterations. In terms of our implementation model of Definition 3.1, this means that $II < \Lambda$.
2. The process should have sufficient "room" for overlapped execution. Applying a scheduling transformation to a process which inherently executes its iterations in a fully sequential fashion will not improve performance.
3. The process should exhibit significant idling because of data dependences, causing the pipeline to be underutilized. Applying a scheduling transformation to a process that already yields full pipeline utilization will not improve performance.
4. The control overhead resulting from the new schedule should not cancel out

the performance gain of the new schedule.

Criterium 1 implies that the scheduling transformation is only effective for processes mapped onto LAURA processors. Overlapped, pipelined execution of process iterations is not possible on the programmable processors supported by ESPAM, such as the MicroBlaze, because their single-threaded instruction pipeline is too short to allow overlapped execution of process iterations.

Criterium 2 requires analysis of the application. The maximum iteration overlap that was introduced in Section 5.2.1 gives an upper bound on the number of iterations that can execute in an overlapped fashion. A maximum iteration overlap of one means that none of the iterations may execute in a partially overlapped fashion because the application is inherently sequential. In such a case, a scheduling transformation cannot improve overlapped execution, and thus should not be applied.

Criterium 3 can be evaluated in two ways: by analyzing the application code using cprof (cf. Section 4.6), or by analyzing a scheduled version of the application code using cprof. The first method is less accurate than the second, but is easier to perform because no changes to the application code have to be made.

To get a rough assessment of whether a scheduling transformation improves overlapped execution using the first method, we evaluate the original application code using cprof on both a real machine and an ideal machine. We assume a pipeline depth of four, that is, $\Lambda_{\mathrm{F}} = 4$ and $II_{\mathrm{F}} = 1$, meaning that up to four iterations can be active simultaneously. In Figure 5.14, we show the flat execution profile for the program of Figure 5.1 on the real and ideal machine. We observe that on the real machine, only one iteration is active for most of the time. On the ideal machine, on average two iterations are active. In both cases, the pipeline is underutilized, because a maximum iteration overlap of four dictated by the pipeline depth is not achieved. A scheduling transformation increases the average utilization from one to two simultaneously active iterations. We have verified using RTL simulation that a scheduling transformation on the program of Figure 5.1 indeed increases overlapped execution. As another example, consider the flat execution profiles of a 1D Jacobi kernel [BBK+08] in Figure 5.15. On the real machine, on average 7 iterations are active simultaneously. On the ideal machine, 29 iterations are active simultaneously. Although more overlapped execution occurs on the ideal machine, the average iteration overlap of seven on the real machine is already sufficient to keep the five-stage pipelined IP core of the application fully utilized. We have verified using RTL simulation that a scheduling transformation does not increase overlapped execution of the Jacobi application.

Alternatively, to get a more accurate assessment of the impact of scheduling on throughput using the second method, we evaluate a scheduled version of the application code using cprof. The scheduled application code can be obtained in a semi-

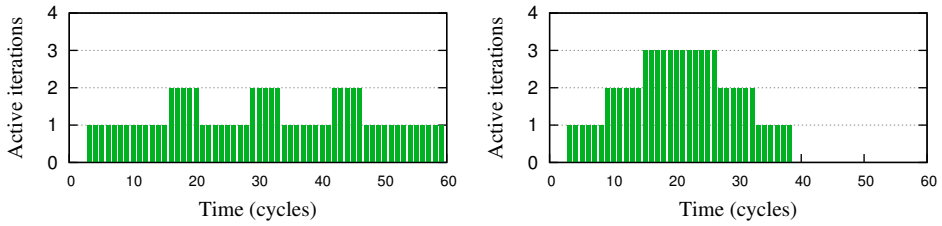


Figure 5.14: Flat execution profile for function `F` of the program shown in Figure 5.1.

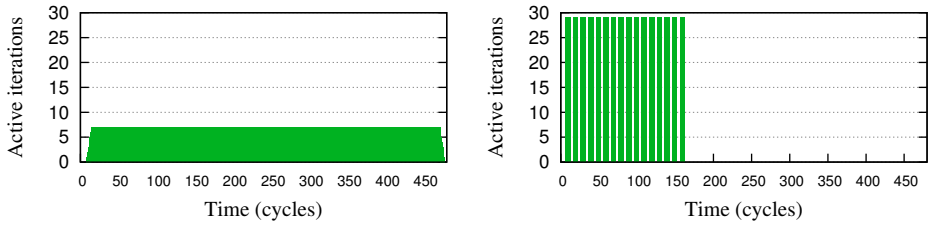


Figure 5.15: Flat execution profile for the Jacobi application.

automated way using for example CLooG [Bas04] or `isl` [Ver08]. By comparing the execution time of the original application code with the execution time of the scheduled application code, we quantify the effect of a scheduling transformation. For the example of Figure 5.14, we measure a decrease in execution time of 29%. For the example of Figure 5.15, we measure an increase in execution time of 56%, which means the scheduling transformation degrades performance. As a result of the analysis, we chose to apply the scheduling transformation for the example of Figure 5.14, but not for the example of Figure 5.15.

Criterium 4 is difficult to address at compile time, because the effects of the new schedule on control overhead are not known until time-consuming low-level synthesis and place-and-route steps have been performed. To avoid time-consuming synthesis steps, we use a heuristic to quickly determine if a particular schedule is likely to result in significant control overhead. A non-unit coefficient in a schedule leads to "gaps" in the transformed domain. For example, consider the polyhedral map of Equation (2.1) which has a coefficient of two for j_1. By applying this polyhedral map to the polyhedral set of Figure 2.2b, we obtain the transformed polyhedral set shown in Figure 2.3. Because of the non-unit coefficient, the transformed polyhedral set contains gaps in dimension j. To handle such gaps in the LAURA processor,

a division by the coefficient is required in the evaluation logic blocks. This is not a problem for coefficients that are a power of two, since division by such values can easily be implemented in RTL using bit shifts. For coefficients that are not a power of two, the resulting division may severely limit the maximum achievable clock frequency. Therefore, when Feautrier's algorithm computes a schedule with coefficients that are not a power of two, a scheduling transformation is not likely to yield higher throughput.

5.3 Conclusion and Summary

We have discussed four PPN transformations in this chapter: process splitting, process merging, stream multiplexing, and scheduling. We have presented how each of these transformations can be applied to a PPN in an automated fashion in the Daedalus tool flow. This enables a designer to quickly obtain functionally equivalent implementations of the same application that differ in performance and resource cost aspects.

Deciding when to apply any subset of the discussed transformations to obtain an implementation meeting a particular performance requirement is a nontrivial task for a designer. We leverage two techniques introduced in Chapter 4 to guide the designer in selecting the appropriate transformations and transformation parameters: the analytical MCM analysis technique and the profiling-based cprof technique. This enables a designer to systematically obtain an implementation that best matches a performance constraint.

CHAPTER 6

INDUSTRIAL CASE STUDY

In this chapter, we study the design process of an industrially relevant sphere decoder application used in wireless mobile communications. We take a sequential C specification of this application as a starting point. Our goal is to automatically obtain an RTL implementation in VHDL or Verilog from the sequential C specification. We compare two tool flows to achieve this goal: the commercial AutoESL high-level synthesis tool [Xil11] [1] and the open-source Daedalus system-level design tool flow [Lei08]. AutoESL is a state-of-the-art high-level synthesis environment that combines heuristics with designer input to obtain design points that satisfy design constraints. We want to compare the Daedalus-based approach discussed in this dissertation with the AutoESL approach to gain insight in the effectiveness of our approach.

We introduce the application in Section 6.1. We review a reference implementation of the application in Section 6.2. We describe an implementation using AutoESL in Section 6.3, and describe an implementation using Daedalus in Section 6.4. We compare the different implementations in Section 6.5 and conclude in Section 6.6.

6.1 Sphere Decoding

The application that we study in this chapter implements part of the *WiMAX* standard [FK08]. *WiMAX* (*Worldwide Interoperability for Microwave Access*), based on the IEEE 802.16e-2005 standard, refers to a new generation of (mobile) wireless broadband access networks. WiMAX employs *Multiple Input, Multiple Output*

[1] AutoESL is currently known as Xilinx Vivado HLS [Xil13].

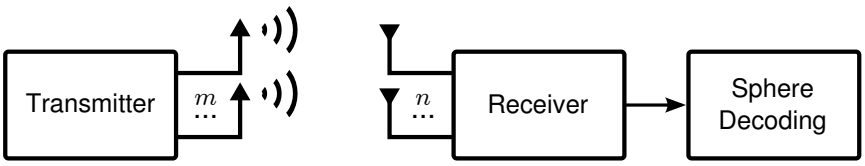


Figure 6.1: An $m \times n$ MIMO system that uses sphere decoding to reconstruct the transmitted symbols.

(*MIMO*) antenna configurations, meaning that both the transmitter and the receiver use multiple antennas, as illustrated in Figure 6.1. All transmitter antennas transmit at the same frequency, but each antenna transmits data from a different data stream. This results in multiple parallel data streams that share the same frequency channel, referred to as *spatial multiplexing*. Spatial multiplexing increases bandwidth efficiency, but comes at the cost of increased computational demands at the receiver side, where advanced techniques are required to separate the different data streams.

Different techniques exist to separate data streams at the receiver side. Decoding the data from the different antennas using a *Maximum Likelihood* (*ML*) detector yields the optimal *Bit Error Rate* (*BER*) performance [BBW+05]. However, the computional complexity of an ML detector grows exponentially with the number of antennas and the choice of modulation scheme, making an ML detector implementation cost-prohibitive for high-data rate systems with large numbers of antennas. Alternatively, channel decoding can be realized using a *sphere decoder*, whose implementation is less expensive while still achieving a BER performance comparable to that of an ML detector [ACDR09]. The actual sphere decoding step is preceded by a channel preprocessing step, which prepares a *channel matrix* that characterizes the MIMO antenna system. In this chapter, we focus on the channel preprocessor of the sphere decoder system that was described in [DTD+09]. The considered sphere decoder system implements a receiver for the most demanding case of the IEEE 802.16e-2005 standard, namely a 64-QAM system with 4 transmitter and 4 receiver antennas.

In Figure 6.2, we show the block diagram of the sphere decoder system that we consider. Before the actual sphere detecting takes place, the channel matrix preprocessor prepares the channel matrix. Inside the channel matrix preprocessor, channel estimation [BSE04] is used to determine the complex-valued 4×4 channel matrix. To improve BER performance, channel reordering is applied to this matrix. The resulting matrix is reorganized into an 8×8 real-valued matrix by the Modified Real-Valued Decomposition (*M-RVD*) block. This real-valued matrix is then converted to an upper-triangular matrix using QR Decomposition (*QRD*). Next, the sphere detector is applied to produce a stream of detected QAM symbols. Subsequent decoding

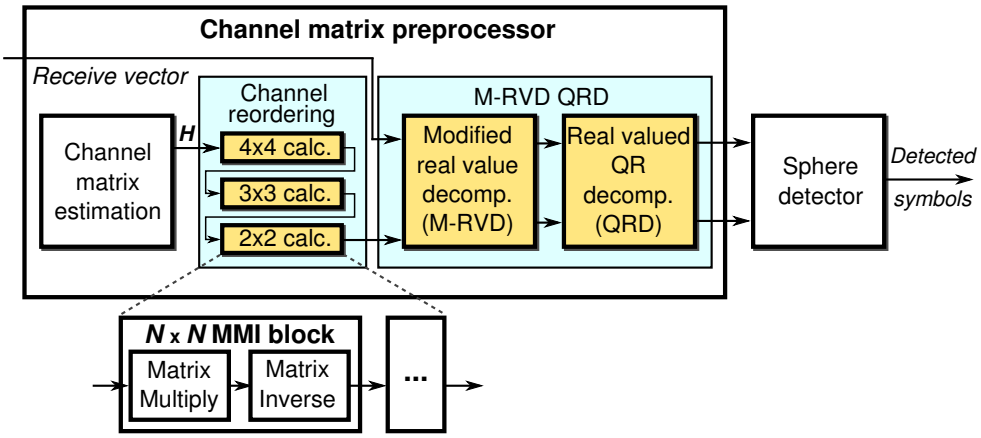


Figure 6.2: Sphere decoder block diagram.

of these symbols then yields the original transmitted bits.

In this chapter, we focus on the Modified Real Value Decomposition (M-RVD) and QR Decomposition (QRD) blocks of the sphere decoder. These two blocks are combined into a single block that we refer to as the *M-RVD QRD* block. Implementing these blocks to meet the application throughput requirements, while minimizing resource usage and latency through the receiver is a challenging design task because of the presence of recurrences in the application.

6.2 Reference Implementation

As a reference implementation, we consider the sphere detector described by Dick et al. [DTD+09]. This reference implementation has been implemented in Xilinx System Generator which is a high-level block-based design tool. The reference implementation is essentially a manually built structural RTL design, containing explicit instantiation of memory and computation primitives and explicit control structures.

The reference implementation targets a mid-speed grade Xilinx Virtex-5 FPGA with a clock frequency of 225 MHz. To conform to the WiMAX throughput targets, the design processes 360 data subcarriers in 102.9 μs. The channel matrix is recomputed for every data subcarrier, which implies the channel matrix preprocessor needs to process a new matrix every

$$\frac{102.9\ \mu\text{s}\ /\ 360}{1/225\ \text{MHz}} \approx 64 \text{ clock cycles.} \tag{6.1}$$

```
1  for (j=0; j<8; j++)
2    for (m=0; m<8; m++)
3      for (t=0; t<15; t++) {
4        X[j][0][t] = diagonal(X[j][0][t], ...);
5        for (n=1; n<8; n++) {
6          if (n < 7-m)
7            R[m][n-1][t] = offdiagonal(R[m][n][t], ...);
8        }
9      }
```

Figure 6.3: Top-level structure of the 8×8 M-RVD QRD C code. Additional code for the time division multiplexing refactoring is underlined.

To meet this high throughput requirement, all blocks in Figure 6.2 operate in a pipeline fashion, which is common for wireless receiver applications. The matrix elements are represented using 18-bit fixed point data types throughout the design. Data is communicated from one block to the next using FIFO buffers and double buffered and dual-ported memories, implementing a streaming system [NV08]. Each block operates on only a few kilobytes of data at a time, which means the sizes of the communication memories are relatively small. Therefore, all memories are implemented using on-chip block memory primitives, that is, no external memory is required for inter-block communication.

The QR decompositions used in the M-RVD QRD block are based on Givens Rotations [SM93]. This method consists of two stages, which we refer to as the *diagonal* and *off-diagonal* cells. The diagonal cell computes an angle such that the leading matrix element is rotated to zero. That angle is subsequently used by the off-diagonal cells to apply the rotation to the remaining nonzero elements of the same matrix row. The top-level structure of the M-RVD QRD C code is shown in Figure 6.3.

6.3 AutoESL

In this section, we describe implementing the M-RVD QRD block of the sphere decoder using the *AutoESL* tool. AutoESL (formerly known as AutoPilot) has been developed since 2006 by AutoESL Design Technologies, Inc. as a commercialization of the xPilot tool from UCLA [CFH+06], and was acquired by Xilinx in 2011 [Xil11]. AutoESL accepts code written in a synthesizable subset of the C, C++, or SystemC language as input. We focus on C++ design entry, with the goal of leveraging C++ template classes to represent arbitrary precision integer types and template functions to represent parameterized components. For the remainder of this

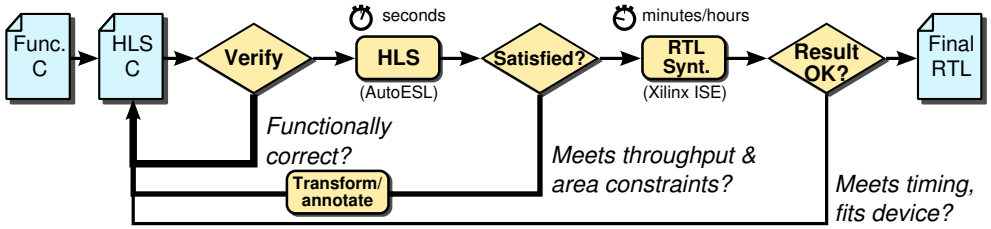


Figure 6.4: High-Level Synthesis design flow.

section, we refer to the high-level language code as "C code" without elaborating on these details.

6.3.1 Design Flow

The overall design process we have followed is shown in Figure 6.4. We start from a functional specification in the C language and a corresponding test bench. The C specification is a reimplementation of the MATLAB model that was used for the reference implementation made with System Generator. By using the test bench and a representative set of test vectors, the C specification is then repeatedly refactored to reflect the desired architecture, while preserving the functionality. This refactoring process makes use of two different interpretations of the C specification. The *functional* interpretation represents the conventional semantics of the C code, describing the sequential and functional behavior. The *architectural* interpretation represents the HLS semantics of the C code, describing the RTL architecture at a high level. The designer makes sure that the functional interpretation of the refactored C code is still identical to that of the original C code, while the architectural interpretation is changed to satisfy non-functional requirements like resource cost and throughput. Manipulation of the architectural interpretation focuses on the coarse-grained architectural aspects, such as memory porting, parallelism, and resource sharing. Fine-grained architectural aspects, such as RTL pipelining details, are handled automatically by the HLS tool by means of predefined characterization data of the target FPGA device.

The throughput resulting from the architectural interpretation can be analyzed statically or dynamically as an output of the HLS compilation. Resource cost estimates are reported after HLS compilation as well. If the various cost and performance metrics satisfy the design requirements, the resulting RTL is synthesized using platform-specific low level synthesis tools. Since HLS tools do not have precise knowledge about e.g. routing delays, metrics reported by the HLS tool typically differ to some

extent from the actual timing characteristics and resource costs obtained after RTL synthesis.

At all times in the development process, the source code of the design is fully functional and can be verified using the C test bench using a regular C compiler and debugger. This is very different from a traditional RTL design flow, where a fully functional version of the design source code becomes available only after weeks or even months of labor. This RTL source code is developed independently of the original reference code, thereby requiring an extensive validation phase. In contrast, obtaining functionally correct design source code that successfully passes through the HLS tool is only a matter of days or even hours. This means an early functionally correct RTL implementation can be obtained quickly, although it is unlikely to already meet resource cost and throughput constraints.

6.3.2 Design Entry

Modern HLS tools like AutoESL and PICO (semi-)automatically leverage a wide range of compiler optimization techniques such as common subexpression elimination and loop unrolling, and computer architecture techniques such as pipelining and resource sharing to improve cost and performance aspects of a design. For some of these techniques, the effectiveness is highly dependent on the structure of the application. Therefore, the decision when and how to apply a particular technique often has to be made by the designer. Some techniques can be applied or controlled with a tool pragma, while other techniques must be reflected in the way the algorithm is described. In this section we describe the techniques applied for the M-RVD QRD block of the sphere decoder application. In particular, we have applied a combination of time division multiplexing, loop unrolling, array partitioning, and case-specific optimizations. All of these techniques have been applied by modifying C code only such that a different architectural interpretation is obtained, while the functional interpretation is preserved.

Time Division Multiplexing

For designs without feedback, an HLS tool is generally able to instantiate registers freely to increase clock frequency and throughput. However, in pipelines that are part of a feedback loop, registers cannot be inserted freely without introducing pipeline stalls. Hence, feedback loops, also known as *recurrences*, in a design are the key limiter of throughput [Pap91]. For example, Figure 6.3 shows the high level structure of the 8×8 M-RVD QRD loop nest. Although there are several recurrences in the application, the critical recurrence in this code occurs when the result

`X[j][0]` of the `diagonal` function call is used as an argument to the next `diagonal` call. Synthesis of the `diagonal` function results in a 14-stage pipeline. As a result, each `diagonal` call has to wait 14 cycles until the result of the previous call becomes available, which means the pipeline is highly underutilized. To accommodate the recurrence without introducing pipeline stalls, we use the wait cycles to process independent data streams, by applying *Time Division Multiplexing* (*TDM*) over 15 datasets. The underlined parts in Figure 6.3 explicitly reflect time division multiplexing or *c-slowing* [LRS93] over separate datasets through the inner `t`-loop.

We observe several characteristics of this design. First, the code accurately reflects the order in which data is processed in the reference design. Second, the TDM refactoring is expressed entirely at the C code level. This means it can be seamlessly ported to any HLS tool that supports the used C constructs, such as multi-dimensional arrays. Third, the number of datasets to iterate over, that is, the TDM depth, cannot be determined without knowing the sizes of the critical recurrences. Although AutoESL does not compute the number of datasets automatically, the HLS process does analyze the source code for recurrences and reports to the designer where recurrences are not satisfied. The designer can use this information in a subsequent AutoESL run. In the sphere decoder application, since 360 *independent* data subcarriers have to be processed for each frame, TDM is a straightforward way to handle the critical recurrence while incurring small increases in resource cost and latency. The resource cost increase stems from additional buffering for the fifteen time multiplexed data subcarriers. The processing latency of the M-RVD QRD block for a single data subcarrier is 945 clock cycles, or $4.2\mu s$.

Loop Unrolling

Application throughput constraints translate directly or indirectly into parallelism requirements on the RTL architecture. For example, the code in Figure 6.3 processes a block of 15 subcarriers. As shown in Equation (6.1), every 64 cycles a new subcarrier must be processed to meet application throughput requirements. As a result, the loop nest in Figure 6.3 must start executing a new block of 15 subcarriers every $15 \times 64 = 960$ cycles. Because the outer loops together comprise 960 iterations, this implies that the body of the `t` loop must be pipelined with an initiation interval II_t of 1. As a result, the inner `n` loop must be unrolled to perform all off-diagonal computations in parallel, which is possible in this application since the calls to `offdiagonal` in the inner loop are independent. We specify the pipelining and unrolling as `pragma` directives to AutoESL, thereby minimizing rewriting of the code and preserving code readability and maintainability. These pragmas are shown in Figure 6.5. AutoESL currently requires unrolled loops to have constant loop bounds, hence the need to ex-

```
1  #pragma AP ARRAY_PARTITION variable=R complete dim=2 partition
2  for (j=0; j<8; j++)
3    for (m=0; m<8; m++)
4      for (t=0; t<15; t++) {
5        #pragma AP PIPELINE ii=1
6        X[j][0][t] = diagonal(X[j][0][t], ...);
7        for (n=1; n<8; n++) {
8          #pragma AP UNROLL
9          if (n < 7-m)
10           R[m][n-1][t] = offdiagonal(R[m][n][t], ...);
11       }
12     }
```

Figure 6.5: Applying loop unrolling (line 8), pipelining (line 5), and array partitioning (line 1) to the M-RVD QRD C code.

plicitly move the conditional statement into the loop body. During the HLS process, AutoESL automatically attempts to compute the number of cycles the loop nest takes to execute, taking into account constant loop bounds and pipeline latencies. This enables a designer to quickly interpret the achieved throughput.

Array Partitioning

After unrolling, the seven off-diagonal cells need to be fed with new data every clock cycle. One of the data sources is a three-dimensional array `R` that is mapped onto a block memory primitive of the FPGA. These block memory primitives have only two memory access ports, which means at most two accesses to array `R` can take place every clock cycle. However, every clock cycle seven different elements need to be read from `R`, since the loop iterator `n` of the unrolled loop appears in the array index expression. This means shortage of memory ports now limits throughput. To overcome this problem, we apply array partitioning to partition the array into subarrays [CJLZ09], again directed by `pragma` directives. We show such a pragma on line 1 of Figure 6.5 to partition the second dimension of array `R`. Each subarray is then mapped onto a separate block memory primitive, effectively providing two memory ports dedicated to each subarray and thereby solving the array bandwidth limitation. Again, memory port limitations are analyzed during the HLS process and AutoESL reports when shortage of memory ports prevents achieving the requested pipelining.

```
1 template <int Wa, int Wb, int Wc>
2 ap_int<Wa+Wb> MADD(ap_int<Wa> a, ap_int<Wb> b, ap_int<Wc> c) {
3   #pragma AP INLINE self off
4   #pragma AP LATENCY max=3
5   #pragma AP INTERFACE ap_none port=return register
6   return a*b+c;
7 }
```

Figure 6.6: C++ code for the MADD function.

Case-specific Optimizations

As an example of a case-specific optimization we consider a non-obvious source of multiplications in the C language, namely multi-dimensional array accesses. Since an array is eventually mapped to a memory with a single-dimensional address space, the multi-dimensional array index has to be converted into a linear address. For example, consider an $M \times N$ array defined in C as `a[M][N]`. The address of array element `a[i][j]` is computed with the expression $i \cdot N + j$. The cost of evaluating this expression varies greatly with the value of N. For example, when $M = 8 \wedge N = 15$, computing the address requires a multiplication by fifteen, which cannot be implemented using only a single shift operation because it is not a power of two. When the array dimensions are interchanged, thus $M = 15 \wedge N = 8$, the multiplication by fifteen is replaced by a multiplication by eight which can be implemented using a single shift operation.

Function and Class Templates

The M-RVD QRD block is specified entirely in the C++ language. To illustrate how function and class templates from C++ can be used, we show the code of the Multiply/Add (MADD) function which is part of a library used by the diagonal cells of the M-RVD QRD block. We provide the C++ code of this function in Figure 6.6. Throughout the design we use arbitrary precision integer (`ap_int`) data types. To allow effective use of library functions, we have designed these functions to support different argument bit widths using C++ templates, as illustrated in line 1 of Figure 6.6 for the MADD function.

Resource Sharing

In many embedded signal processing applications, maximizing throughput is often not as important as minimizing resource usage for a given throughput. In these cases,

effective resource sharing is an important design goal. Some resource sharing is implicit when a loop is pipelined rather than unrolled, since consecutive iterations of the loop execute on the same datapath generated from the body of the loop. In this section, we focus on achieving *additional* resource sharing.

AutoESL employs heuristics to decide which function calls are inlined. The MADD in Figure 6.6 was inlined in our design study. When an inlined function is called in two different places, the entire implementation of this function appears twice in the RTL. To enable sharing of resources in such cases by AutoESL, we disable inlining using the pragma in line 3 of Figure 6.6.

User-Influence on the Generated RTL

AutoESL provides means to influence aspects of the RTL at the source code level. The use of such means turned out to be inevitable to obtain a design competitive with hand-written RTL for the M-RVD QRD block. Because AutoESL's default timing characterization prevented timing closure of RTL resulting from multiplications in the C code, we have enforced the correct characterization by means of the pragmas shown in lines 4 and 5 in Figure 6.6. Line 4 enforces a latency of three clock cycles and line 5 enforces an output register of the MADD RTL block. Such pragmas allow a designer to "correct" suboptimal decisions of the HLS tool for a particular part of the design. The need for such manual corrections should diminish over time as HLS tools are further improved.

6.3.3 Design Productivity

To compare design times of the HLS and reference implementations, we have reconstructed the approximate amount of working time on the designs. Design times for the reference implementation have been estimated by the original implementors [DTD+09] as 4.5 weeks. Design times for the HLS implementation have been extracted from source code version control logs as 5 weeks. We observe that the design times to reach an optimized implementation are approximately the same for the HLS implementation and the reference implementation. However, the RTL design flow yields only a single design point, while the HLS design flow yields many design points with different performance and cost tradeoffs.

The effects of the refactoring-based design process for the M-RVD QRD block can be seen in Figure 6.7. On the left vertical axis, we show the overall application throughput determined from static clock cycle count analysis of AutoESL, combined with post-place and route timing closure information. On the right vertical axis, we show the corresponding post-place and route LUT and flipflop usage. For comparison, the

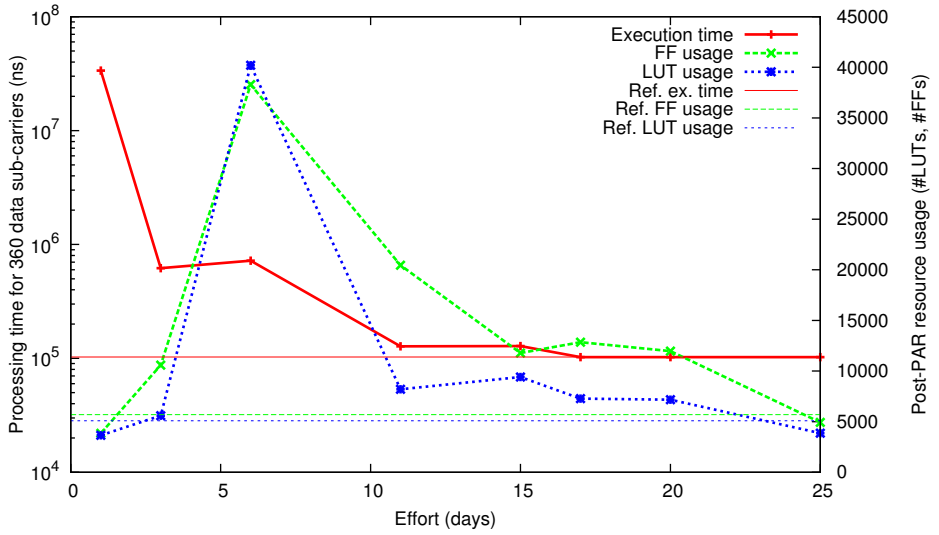


Figure 6.7: Performance and resource usage of the M-RVD QRD block plotted as a function of development time.

horizontal lines represent the target application throughput and resource usage of the reference implementation. After obtaining a "clean" algorithmic C model, it took about a day to get the code through AutoESL for the first time. This required rewriting of several nonsynthesizable constructs such as non-analyzable pointers. This first implementation exploits little parallelism as it executes almost entirely sequentially. By performing continual refactoring, the throughput and cost are improved with full functional verification at each refactoring step. The bulk of the architectural refactoring was completed in about ten working days. The remaining time has been spent tuning the design to reduce resource usage and to improve timing closure in place and route. Below we summarize the design process for the M-RVD QRD block.

- **Day 1:** The C code is accepted by the HLS tool, and a functional hardware implementation is already available. However, the total processing time is off by two orders of magnitude.
- **Day 3:** After becoming more familiar with the tool and applying basic refactoring techniques such as enabling pipelining using a single C preprocessor pragma, the processing time is reduced significantly.
- **Day 6:** Because of code restructuring such as loop unrolling, the resource usage increases considerably. This limits the achievable clock frequency, effectively increasing the processing time again. However, the design source code

is now in a shape that enables further optimizations.

- **Day 11:** C integer data types are replaced by fixed point data types and optimized primitive blocks (e.g., the MADD function) are introduced in the design. This significantly reduces resource usage. A pipeline *II* of one is now feasible. However, during RTL synthesis the design does not achieve timing closure, which means throughput constraints can still not be met.
- **Day 17:** By optimizing the code of the data paths (e.g., by applying the case-specific optimization described in Section 6.3.2), latencies and resource usage are further reduced. As a result, the RTL now achieves timing closure, so at this point an implementation meeting throughput requirements is available.
- **Day 25:** Further optimizations including algorithmic optimizations have been applied to reduce the resource usage of the design.

In this work we had the advantage that the reference implementation was already available to us. Thus, we knew the high-level application architecture that had to be constructed to meet throughput requirements. Hence, we have been fully concentrating on getting a similar architecture out of the HLS tool initially. After obtaining a design point meeting throughput requirements, the goal was changed to reducing resource cost to the level of the reference implementation. Again, we had the advantage of knowing detailed resource cost statistics for the reference implementation, thus by comparing with the HLS implementation we knew what parts could be optimized further. Many different design points can be implemented using HLS in a short amount of time, as each design point in Figure 6.7 is a fully functional design with different performance and cost aspects. On the other hand, the RTL design process has yielded only one design point in approximately the same amount of time.

6.4 Daedalus

We have implemented the M-RVD QRD block of the sphere decoder also using the Daedalus tool flow. We have followed the same refactoring-based design process as with the AutoESL design, with the only difference that the "HLS" step of Figure 6.4 now consists of running Daedalus instead of AutoESL. We have started from the same C code that was also used as starting point for the AutoESL design. Obtaining an RTL implementation from C code consists of two steps, as depicted in Figure 1.1. The first step is to convert the M-RVD QRD C code into a PPN specification using PNGEN, which we describe in Section 6.4.1. The second step is to synthesize an RTL implementation from the PPN specification using ESPAM, which we describe in Section 6.4.2.

6.4.1 Design Entry

We distinguish between two classes of refactorings in a Daedalus design flow. Refactorings of the first class transform the source code into a form suitable for Daedalus. Refactorings of second class are similar to the architectural refactorings in the AutoESL design flow. That is, they serve to alter the architectural interpretation of the source code. The first two refactorings discussed below are of the first class, whereas the remaining refactorings are of the second class.

Compatibility Restructurings

PNGEN requires that the sequential C code is a static affine nested loop program (cf. Section 2.3). The sequential C code for the M-RVD QRD block already conforms to this requirement, such that meeting the SANLP requirement requires no effort. To ease integration of IP cores in LAURA, we rewrite the sequential code such that besides for- and if-statements, only function calls and plain copy assignment statements are exposed in the top level function given to PNGEN. This means other statements containing arithmetic operations have to be embedded into function calls. For example, the following statement

```
in_diag = -mat_im[i];
```

is rewritten as

```
negate(mat_im[i], &in_diag);
```

with `negate` being a new function that writes the negation of its first argument to its second argument.

Introduction of Source and Sink Processes

Each PPN should have at least one source and one sink process that represent the input and output interfaces of the system. In a physical implementation, these source and sink processes exclusively communicate with the environment. For example, a source process may represent a video capture device, whereas a sink process may represent a display device. The remaining non-source and non-sink processes perform the actual data processing.

In the current implementation of PNGEN, source and sink processes have to be explicitly specified in the C input, using function calls that have only output arguments and only input arguments, respectively. Any arguments to the top level function, such as `im` on line 1 of Figure 6.8, are currently ignored by PNGEN. The M-RVD QRD reference code communicates input and output data via array arguments of the

```
1  void mrvdqrd(int im[4][4][15], ...) {
2    #pragma AP ARRAY_PARTITION variable=im complete dim=1 partition
3    ...
4  }
```

Figure 6.8: Input arguments to the top-level M-RVD QRD function.

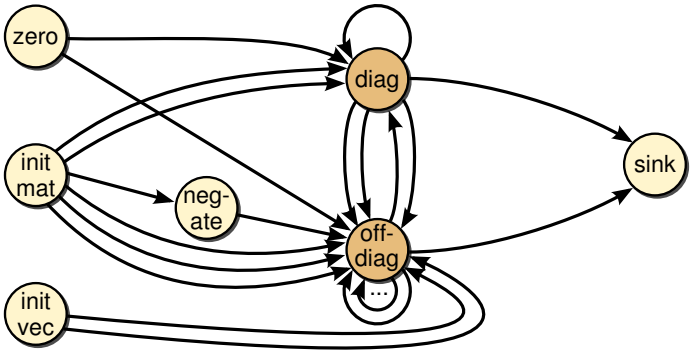


Figure 6.9: Initial PPN for the M-RVD QRD block.

top level function, as shown in Figure 6.8. This means we have to translate the top level function argument list into source and sink processes. For every input argument, we introduce a function and a loop nest such that all elements of the array are written exactly once. We illustrate this for the `im` input argument in Figure 6.11. The input argument is removed from the function header and defined as a local variable. The order in which the elements are written should match the order in which the elements are read for the first time by any subsequent processes. This ensures communication can be implemented using regular FIFO channels instead of more expensive reordering buffers. In a similar way, for every output argument, we introduce a function and a loop nest such that all elements of the array are read exactly once.

We are able to reuse the original test bench by making additional modifications to the C code. First, we modify the test bench to read and write test vectors from and to global variables. Next, we make the source and sink processes stateful by introducing an internal counter that is incremented upon every invocation of the particular function. Using this counter, the corresponding array elements are read from or written to the global test vectors. Although these changes assist us in verifying the functionality after each C code transformation, they have no implications for the final hardware implementation, since the source and sink processes are typically replaced by the interfaces they represent.

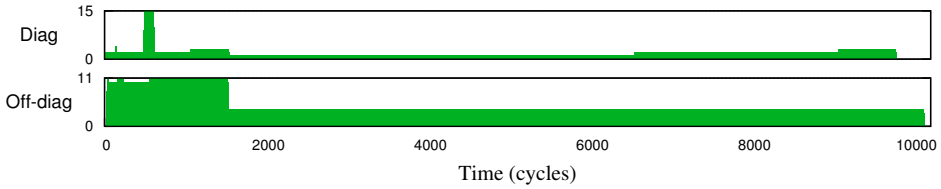


Figure 6.10: Flat execution profiles of diagonal and off-diagonal cell resources.

Initial PPN

After the source code refactorings discussed so far, a first PPN can be obtained using PNGEN, which is shown in Figure 6.9 This initial PPN contains one diagonal cell and one off-diagonal cell, since the diagonal and off-diagonal cell calls appear only once in the C code of Figure 6.3. An architecture with only one off-diagonal cell cannot achieve the throughput demanded by the application requirements. This can be observed using the flat execution profiles shown in Figure 6.10 obtained using cprof. The execution time for a single execution of the PPN is about 10000 clock cycles, which is more than ten times the desired execution time of 960 clock cycles.

Splitting

The AutoESL and reference implementations contain one diagonal cell and eight off-diagonal cells to meet the throughput requirements of the sphere decoder application. We have applied loop unrolling to the innermost loop to obtain eight off-diagonal cells in the AutoESL design. HLS tools such as AutoESL provide a pragma to unroll a loop while keeping the code compact and maintainable. To obtain the same architecture, we apply a plane cutting transformation to the PPN with a factor 8 on the innermost dimension n of the `offdiagonal` process, specified as:

$$planecut(\texttt{offdiagonal}, n, 8)$$

This results in eight `offdiagonal` processes, which resembles the architecture of the reference implementation.

After splitting the off-diagonal cell, the source processes also have to be split to ensure all eight off-diagonal cells receive data at a fast enough rate. This is similar to partitioning an input array in the HLS context, effectively increasing the bandwidth of that array. For example, for an input array `im`, representing the imaginary components of the complex-valued channel matrix, we apply a pragma in AutoESL to partition this array. This is shown by the pragma in Figure 6.8. The pragma splits `im` into four distinct subarrays `im_0[4][15]`, `im_1[4][15]`, `im_2[4][15]`, and `im_3[4][15]`. We have removed the input and output arguments to the top-level function by introducing

```
1  void mrvdqrd() {
2    int im[4][4][15];
3    for (j=0; j<4; j++)
4      for (m=0; m<4; m++)
5        for (c=0; c<15; c++)
6          initmatrix( &im[j][m][c] );
7  }
```

Figure 6.11: Source process for input argument `im` of Figure 6.8.

source and sink processes as discussed above. The code in Figure 6.11 implements a source process for input matrix `im`. After splitting the off-diagonal cell, we need to split the `initmatrix` process as well to make sure data from `mat_im` is delivered at a fast enough rate. This is done by applying a plane cutting transformation with a factor 4 on the `j` dimension of `initmatrix`. As a result, we obtain four `initmatrix` processes that deliver data to four out of eight off-diagonal cells. The other four off-diagonal cells require the real components of the complex-valued channel matrix, which is stored in an array `re`. We apply the same plane cutting transformation to the source process for this array.

A similar relation exists between sink process splitting in PPNs and output array partitioning in HLS, to ensure that data produced by the off-diagonal cells is consumed at a fast enough rate.

After the splitting transformations, we again use cprof to evaluate the performance of the new PPN. The technique described in Section 4.6.6 allows us to evaluate the splitting transformations at the sequential code level, without the need to apply the transformations to the sequential code. The resulting flat execution profiles are shown in Figure 6.12. The execution time is now reduced to about 960 clock cycles, which means the PPN meets the application throughput requirements.

Process Merging

Similar to the AutoESL and the reference implementations, the PPN implementation now consists of one processing resource for the diagonal and eight processing resources for the off-diagonal cell computations. This allows the PPN to meet the throughput demands of the application. We now ask ourselves if we can reduce the resource cost of the implementation while satisfying the throughput constraints. For this purpose, we analyze the utilization of the eight off-diagonal cell LAURA processors using the flat execution profiles obtained by cprof. The number of simultaneously active iterations on each off-diagonal cell LAURA processor over time is

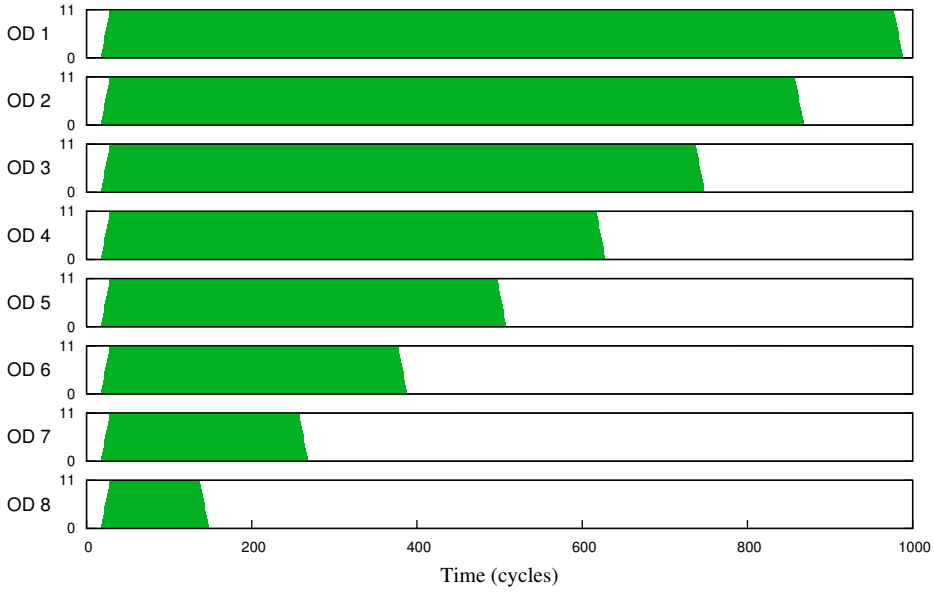


Figure 6.12: Flat execution profiles of off-diagonal cell resources after splitting by a factor eight.

```
1  for (n=1; n<8; n++) {
2    if (n < 7-m) {
3      if (n == 1)            pr$ = 1;
4      if (n == 2)            pr$ = 2;
5      if (n == 3 || n == 8) pr$ = 3;
6      if (n == 4 || n == 7) pr$ = 4;
7      if (n == 5 || n == 6) pr$ = 5;
8
9      // Offdiagonal cell profiling instrumentation ...
10   }
11 }
```

Figure 6.13: Evaluating merging transformations using cprof.

shown in Figure 6.12. The maximum number of simultaneously active iterations on a processor is given by the processor's pipeline depth, which is 11 cycles. From Figure 6.12, we observe that the utilization of the first off-diagonal cell processor (OD 1) is almost 100%, because the pipeline is fully occupied by eleven iterations for most of the time. On the other hand, the last off-diagonal cell processor (OD 8) is active for only $\frac{1}{8}$ of the time, which means the utilization is approximately 12%.

Our goal is to merge the processors with low utilization, such that resource cost is reduced while throughput is not affected. Off-diagonal cell 1 determines the overall throughput, as it has the longest execution time according to Figure 6.12. By looking at this figure, we expect that merging off-diagonal cells 3 and 8 should lead to a combined execution time that is still shorter than the execution time of OD 1. A similar expection holds for merging OD 4 and 7, and OD 5 and 6. This would lead to an implementation with only five off-diagonal cell processors instead of eight.

To evaluate whether this merging transformation is beneficial, we consider the two conditions for a merging transformation described in Section 5.2.2. The first condition is that the processes that are merged execute the same function. This condition is met, since each process executes the same off-diagonal cell function. The second condition is that the overall throughput should not be affected by the merging. We use cprof to assess if this condition is met. We leverage the technique of Section 4.6.6 to evaluate the merging transformation at the sequential code level. Recall that this technique employs a variable `pr$` which selects the processing resource on which an iteration executes. We assign iterations of the `n`-loop to `pr$` according to the merging transformation, as shown in Figure 6.13. For example, off-diagonal cell 1 is not merged with any other off-diagonal cell, and is therefore assigned exclusively to processing resource 1 on line 3. Offdiagonal cells 3 and 8 are merged, and are therefore both assigned to processing resource 3 on line 5. The resulting flat execution profiles

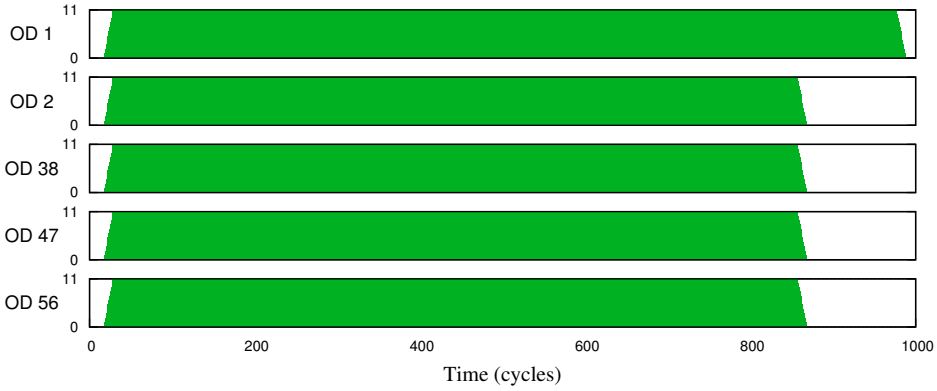


Figure 6.14: Flat execution profiles of off-diagonal cell resources after merging cells 3 & 8, 4 & 7, and 5 & 6.

are shown in Figure 6.14. The total execution time of the merged version is identical to the execution time of the unmerged version. We therefore conclude that the proposed merging transformation would be worthwhile to apply for further evaluation at the implementation level.

6.4.2 Synthesis

After obtaining a process network with the desired throughput characteristics, we generate an RTL implementation using the ESPAM tool. Because the Daedalus tool flow does not provide means to synthesize data paths, we have reused the IP cores for the diagonal and off-diagonal cell functions from the AutoESL implementation. These IP cores can easily be integrated into the execute units of the generated LAURA processors that implement the processes. Since the source and sink processes represent interfaces of the application, we do not synthesize LAURA processors for these processes. The interface to the PPN consists of the FIFO buffers that connect the interior processes of the PPN to the source and sink processes.

To achieve the highest clock frequency currently possible for LAURA processors, we have used the optimization described in Section 3.5.1. Despite this optimization, the RTL for the eight-off-diagonal-cell implementation achieves a clock frequency of 176 MHz, whereas 225 MHz is required to meet the application throughput demands.

The alternative implementation with only five off-diagonal cells was not implementable by ESPAM, because merging of LAURA processors is not supported in the general case as explained in Section 5.1.2. An alternative with seven off-diagonal

Design	LUT	FF	DSP	BRAM	Fmax
SysGen [DTD+09]	5082	5699	30	19	225
AutoESL [CLN+11, NNH+11]	3862	4931	30	19	225
Daedalus-8OD	6506	5235	30	38	176
Daedalus-7OD	6672	5309	27	70	172
Daedalus-5OD			21		

Table 6.1: M-RVD QRD post-place-and-route implementation statistics.

cells, obtained by merging the last two off-diagonal cells, was implementable, because the compound process has a convex polyhedral set as its domain.

6.5 Comparison

In Table 6.5, we compare resource usage statistics for the M-RVD QRD block of a reference RTL design [DTD+09], the AutoESL design, and the Daedalus design. To obtain accurate comparisons, we have reimplemented the reference design using the Xilinx ISE 12.1 tools targeting a Virtex-5 VLX110T-2 FPGA. The AutoESL design has been developed using AutoESL AutoPilot 2010.07.ft and has also been implemented using ISE 12.1 targeting the same FPGA. The Daedalus designs have been developed using PNGEN 0.10-93-g73a41d1 and ESPAM 2011.10, and have been implemented using ISE 12.1 targeting the same FPGA. Verification of the RTL was performed using a manually written testbench in VHDL that used the same test vectors as the testbench for the SysGen and AutoESL designs.

The SysGen, AutoESL, and Daedalus-8OD designs all employ the same architecture containing one diagonal and eight off-diagonal cells. This is reflected in the DSP resource cost, which is the same for all three designs. The AutoESL design has lower LUT and FF cost mainly because the off-diagonal cell was more optimized than the off-diagonal cell of the SysGen design. The Daedalus-8OD design has higher LUT cost than the SysGen design because of the logic implementing the LAURA processors and channels. The Daedalus-8OD design has lower FF cost than the SysGen design, because the Daedalus-8OD design was not optimized for the target clock frequency of 225 MHz, and thus lacks careful insertion of more FF primitives to meet the target clock frequency. The Daedalus-8OD design requires twice the amount of block memory (BRAM) primitives as the SysGen and AutoESL design, to allow sufficiently large channel sizes that do not degrade the throughput. The Daedalus-7OD design contains only seven off-diagonal cells, which is reflected in a saving of three

DSP primitives. This comes at the expense of slightly higher LUT and FF cost, and almost a doubling in BRAM cost. The increase in BRAM cost is caused by larger buffer sizes needed to avoid blocking writes that degrade throughput. The Daedalus-5OD design was not implementable, as explained in Section 6.4.2. We know that an off-diagonal cell requires three DSP primitives. Eliminating three off-diagonal cells thus leads to a reduction of nine DSP primitives. Estimating the other cost characteristics of the Daedalus-5OD is not trivial, because these characteristics depend on the interplay of many factors. Therefore, these characteristics are left empty in Table 6.5. We did not succeed in obtaining an architecture with less than eight off-diagonal cells using AutoESL. Attempts to express such architectures in the C code resulted in implementations that did not satisfy the throughput requirements.

In Section 6.3.3, we have compared the design times of the SysGen and AutoESL designs. Comparing the design times of the Daedalus and AutoESL designs is difficult for the following three reasons. First, the AutoESL design time includes time needed to study the application and the SysGen reference design. Second, the blocks implementing the diagonal and off-diagonal cells were already available during the Daedalus design, whereas these had to be developed and optimized during the AutoESL design. Third, we needed to debug and adapt the Daedalus tools, as the application revealed corner cases that were not correctly handled by the tools. A design time estimate would thus be blurred because of these three reasons. Making an educated guess nonetheless, we expect that we could reproduce the architecture of the SysGen and AutoESL designs using Daedalus in about two weeks.

6.6 Conclusion and Summary

We were able to achieve an RTL implementation from sequential C code for an industrially relevant application using both the commercial AutoESL and academic Daedalus tools. The AutoESL design was competitive to the manually built reference implementation. The architecture employed by the AutoESL and reference designs could be replicated using Daedalus, although the Daedalus design did exhibit higher resource cost and a lower clock frequency. We attribute this to Daedalus being a primarily a research environment, in which the limited development power is invested in research aspects rather than competition with commercial products. We expect that more competitive designs can be obtained using Daedalus with additional engineering effort, as we do not see fundamental limitations.

The use of synthesis techniques and optimizations presented in Chapter 3, the cprof analysis technique presented in Chapter 4, and the transformations presented in Chapter 5 proved essential in obtaining the architecture of the sphere decoder reference

design. Moreover, the cprof technique allowed us to quickly evaluate performance of alternative application instances at the sequential code level. We therefore conclude that the work presented in this dissertation are essential contributions to handle industrially relevant applications in Daedalus.

CHAPTER 7

CONCLUSIONS

In this dissertation, we presented techniques that allow a designer to implement MP-SoCs using the Daedalus system-level design methodology, while taking into account design constraints on system performance. The techniques presented in this dissertation leverage the Daedalus methodology to provide a forward synthesis flow that bridges the specification and implementation gaps. However, the Daedalus methodology did not yet provide a satisfactory solution to satisfy the performance constraints of a designer. In the conventional forward synthesis flow, the designer knows only after a time-consuming forward synthesis step if performance constraints are met. Instead, the designer should obtain feedback faster, possibly at the expense of reduced accuracy, allowing him to avoid a time-consuming forward synthesis step if he knows a design will not satisfy his constraints. We identified three central research problems in Section 1.2. We presented techniques to address these three central problems in Chapters 3, 4, and 5.

The first central problem we addressed was the synthesis problem. We found that the current forward synthesis flow lacked support for RTL implementations for particular classes of input programs and application characteristics that the PNGEN compiler could already process. Our solution to this problem in Chapter 3 consists of four contributions. The first contribution is a characterization of function implementations, which allows us to reason about performance of systems. The second contribution incorporates novel optimizations that were performed by the PNGEN tool, but which were not yet incorporated in the generated RTL architecture, into ESPAM. This allows us to handle a broader class of input programs. The third contribution comprises optimizations for the LAURA processor model's evaluation logic blocks. These optimizations involve pipelining of expression data paths and storage of compile-time

evaluated expressions in ROMs. Pipelining of the evaluation logic blocks enables a LAURA processor to run at a higher clock frequency, which may be required to meet application design constraints. The use of ROMs enables a LAURA processor to handle more complex domains that may result from transformations. The fourth contribution consists of a novel reordering buffer design. This allows Daedalus to generate RTL implementations for applications that exhibit out-of-order communication. The reordering buffer was designed such that replacing a regular FIFO with a reordering buffer does not increase the latency in cycles of read and write operations to the buffer. As a result, transformations that introduce out-of-order communication no longer cause an increased communication latency. The reordering buffer thus enables performance gains of such transformations.

The second central problem we addressed was the performance estimation problem. We found that no applicable performance estimation methods existed that could handle polyhedral process networks implemented using LAURA processors. Estimating the performance of pipelined execution of process iterations was lacking. Such performance estimations are essential to reason about design constraints on system performance. We have investigated and presented performance estimation techniques at four different levels in the Daedalus design flow in Chapter 4. The first performance estimation technique is RTL simulation, which works on the RTL implementation that is the final output of Daedalus. Instead of prototyping this RTL implementation on an FPGA, we simulate the RTL that implements the system. We found that RTL simulation is not feasible for systems containing programmable processors, because of long simulation times. The second performance estimation technique is SystemC simulation, which works at the mapped model of the system. SystemC simulation is faster than RTL simulation, but less accurate. The third performance estimation technique is MCM analysis, which works on the parallel model of the application. The MCM analysis technique is analytical, which has the advantage that estimation time does not depend on the application workload. This leads to performance estimation times that are shorter than SystemC or RTL simulation times. However, we cannot define tight bounds on the inaccuracy of the MCM method, nor whether the method overestimates or underestimates the actual throughput. This model is theoretically attractive and gives insight in the behavior of a PPN, but is impractical because of the lack of accuracy bounds. The fourth performance estimation technique is a novel profiling-based approach for PPNs, named cprof, which works directly on the sequential code. This allows one to obtain accurate results, often in less than one second, without deriving a PPN.

The third central problem we addressed was the transformation problem. We found that it is not trivial for a designer to select a set of transformations and transformation parameters such that a design constraint on performance is met. We have pre-

sented four PPN transformations (i.e., splitting, merging, stream multiplexing, and scheduling) in Chapter 5. For each transformation, we analyzed factors that affect the efficacy of the transformation. This aids the designer to select the appropriate transformations needed to satisfy performance constraints. The first transformation is splitting, which duplicates a process such that throughput may be increased at the expense of increased resource cost. We have proposed analytical and profiling-based strategies to select the splitting factor. The second transformation is merging, which combines multiple processes such that resource cost is reduced, potentially at the expense of decreased throughput. We have identified a special case in which merging of LAURA processors can reduce resource cost while not affecting throughput. The third transformation is stream multiplexing, which increases throughput of multiple PPN executions. We have identified criteria to assess when a stream multiplexing transformation is beneficial, and have presented how to select the stream multiplexing factor such that the latency of a single PPN execution is not affected. The fourth transformation is scheduling, which reorders iterations of a process to increase pipeline utilization. We have identified criteria to assess when a scheduling transformation can be applied to achieve improved pipeline utilization and, consequently, higher throughput.

To validate our solutions to the three central problems, we have conducted a case study using an industrially relevant application used in wireless communication receivers. We compare the extended Daedalus tool flow with the commercial AutoESL high-level synthesis tool in Chapter 6. Specifically, we have focused on a channel matrix preprocessor subblock of a sphere decoder. A manually crafted RTL reference design was available to us. Using a continuous refactoring-based design flow, we were able to replicate the architecture of the reference design using both AutoESL and Daedalus. Refactorings in the AutoESL flow consist primarily of pragma annotations. Refactorings in the Daedalus flow consist primarily of source code restructurings and transformations discussed in Chapter 5. We were able to meet the tight performance design constraint using AutoESL, but not using Daedalus as low-level clock frequency aspects have not been engineered out in the Daedalus tools. Nonetheless, we were able to replicate the architecture of the reference design using Daedalus, which means Daedalus can handle industrially relevant applications. The cprof technique and transformations presented in this dissertation proved essential to obtain the desired architecture of the application in the Daedalus design flow in a short amount of time. Moreover, the cprof technique allowed evaluating alternative design points at the sequential code level.

By addressing the three research problems, we have established a powerful system-level design flow capable of solving industrially relevant design problems, as the designer knows if his design will satisfy his performance constraints. This makes

it worthwhile to explore various transformations of the system, still at the sequential code level. When finding a satisfactory design point, the designer commits to the time-consuming forward synthesis flow, knowing that the design will satisfy his performance constraints.

SAMENVATTING

Deze dissertatie beschrijft methodes die het ontwerpproces van applicatiespecifieke multiprocessorsystemen vereenvoudigen voor de ontwerper. Dit ontwerpproces wordt in toenemende mate ingewikkelder, vanwege toenemende complexiteit, toenemende vraag naar rekenkracht en tijdsdruk van de markt. Om het ontwerpproces voor toekomstige generaties van multiprocessorsystemen behapbaar te houden, dient het ontwerpproces op systeemniveau plaats te vinden. Hiertoe gebruiken wij de Daedalus methodologie. We introduceren de Daedalus methodologie in Hoofdstuk 1. De Daedalus methodologie voorziet in een voorwaarts ontwerpproces van systeemniveau naar FPGA implementatie. Echter, de Daedalus methodologie voorziet nog niet in een ontwerpproces waarin een FPGA implementatie met door de ontwerper bepaalde prestatiekenmerken wordt afgeleid vanuit een systeemniveau ontwerp. Deze dissertatie levert daartoe een bijdrage.

De Daedalus methodologie maakt gebruik van het Polyhedrale Proces Netwerk (PPN) model, welke een belangrijke rol speelt in het vervolg van de dissertatie. We introduceren het PPN model alsmede enkele gerelateerde modellen in Hoofdstuk 2.

In Hoofdstuk 3 presenteren we uitbreidingen op bestaande technieken om PPN's te implementeren in FPGA-technologie. Deze uitbreidingen zijn noodzakelijk om industrieel relevante applicaties te kunnen implementeren in de Daedalus methodologie.

In Hoofdstuk 4 behandelen we vier verschillende methodes om de prestaties van een applicatie gemodeleerd als PPN te bepalen. De eerste techniek omvat simulatie op registerniveau. De tweede techniek omvat simulatie op systeemniveau middels SystemC modelering. De derde techniek omvat het analytisch bepalen van het maximale lus gemiddelde. De vierde techniek omvat instrumentatie van de sequentiële code.

In Hoofdstuk 5 behandelen we vier transformaties om de prestaties van PPNs te beïnvloeden. De eerste transformatie omvat het splitsen van een proces. De tweede transformatie omvat het samenvoegen van twee processen. De derde transformatie omvat het verhogen van de doorvoer middels het toevoegen van onafhankelijke datastromen. De vierde transformatie omvat het verhogen van de doorvoer middels het wijzigingen van de volgorde waarin procesiteraties worden uitgevoerd. We geven

richtlijnen om aan hand van applicatie analyse transformaties and transformatieparameters te selecteren zodat door de ontwerper bepaalde prestatiekenmerken behaald worden.

In Hoofdstuk 6 behandelen we het ontwerpproces van een industrieel relevante applicatie gebruikt in draadloze communicatie. We vergelijken een ontwerpproces middels een commerciële C-naar-RTL methode met de Daedalus methode. Met beide methodes zijn we in staat om de gewenste architectuur te verkrijgen.

Het werk beschreven in deze dissertatie omvat een systeemniveau ontwerpproces waarbij in een vroeg stadium rekening gehouden wordt met prestatie-eisen van het te ontwerpen systeem. Hiermee kan een designer beoordelen of een gegeven prestatie-eis haalbaar is, alvorens een tijdrovend syntheseproces op te starten.

CURRICULUM VITAE

Sven van Haastregt was born in Rijpwetering, the Netherlands, in 1985. He obtained his gymnasium diploma at Visser 't Hooft Lyceum, in Leiden, the Netherlands, in June 2003. In September 2003, Sven enrolled in the Computer Science track at Leiden University, the Netherlands. He received his BSc degree (with honors) in September 2006, and the MSc degree (with honors) in August 2008. After obtaining the MSc degree, Sven started his PhD research at the Leiden Embedded Research Center (LERC), part of the Leiden Institute of Advanced Computer Science (LIACS), Leiden University. He was involved in the MEDEA SoftSoC project on metamodeling of Hardware-dependent Software (HdS) and conducted research on high-level synthesis and performance estimation and optimization of polyhedral process networks. During his research, Sven (co-)authored various peer-reviewed conference and journal articles. He also contributed to the open-source PNGEN and ESPAM tools, and initiated the PNTOOLS software. From January 2010 to April 2010, Sven was a guest researcher at Xilinx Research Labs, in San Jose, California, United States of America. Besides his work as a researcher, Sven was involved as a teaching assistant in the Compiler Construction, Operating Systems, and Challenges in Computer Science Seminar courses which are part of the Computer Science bachelor program of Leiden University. The research culminated in the writing of this PhD dissertation in 2013. As of March 2013, Sven is working as a software engineer at ARM.

ACKNOWLEDGMENTS

Back in 2006, when I was still an MSc. student, Sjoerd Meijer lured me into the Leiden Embedded Research Center (LERC). Unknowingly at the time, Sjoerd laid the foundation towards this dissertation, and for this I am very grateful. In addition, I would like to thank all present and former members of LERC for providing an inspiring working atmosphere. In particular I want to mention Todor Stefanov and fellow PhDs Teddy Zhai, Mohamed Bamakhrama, and Ana Balevic for the many interesting discussions that we had. I also acknowledge all other colleagues from the Leiden Institute of Advanced Computer Science (LIACS). In particular I want to mention André Deutz, Nies Huijsmans, Rudy van Vliet, and Hendrik Jan Hoogeboom, with whom I have enjoyed teaching courses on operating systems, compiler construction, and Android programming.

Over time I have supervised several BSc. and MSc. students who worked towards their graduation in the LERC group. I want to thank all of them, and in particular Eyal Halm and Wouter de Zwijger for their significant research contributions.

Sven Verdoolaege introduced me to the polyhedral model and developed a vast amount of software to tame this model. Upon reporting a bug or feature request, he often replied within a few hours with a source code patch that addressed my issue. This level of support is rare and I am therefore very grateful to him.

In 2008, Vinod Kathail and his team at Synfora introduced me to the world of high-level synthesis. In retrospect, this was an important first step towards this dissertation. Early 2010 I had the privilege of staying at Xilinx Research Labs in San Jose, CA, USA for a few months. This stay was made possible by Kees Vissers. I want to thank him for giving me this chance to work with state-of-the-art high-level synthesis tools, and for his support during my stay. I also want to thank Stephen Neuendorffer for sharing his extensive knowledge of high-level synthesis and FPGAs, and for reviewing my thesis and providing very detailed and useful feedback. I want to thank all colleagues from Xilinx with whom I worked, either directly or indirectly, in particular Juanjo Noguera, Chris Dick, and Ivo Bolsens.

This work was partly funded through the MEDEA+ SoftSoC project 2A714 about

Hardware-dependent Software for Systems on Chip. I would like to thank all persons with whom I have worked together in this project, in particular Ruud Derwig from Synopsys, Razvan Nane from TU Delft, Nikolay Kavaldjiev and Giuseppe Garcea from Compaan Design, and Mohammad Al Hissi from LERC.

Finally, I want to thank my parents, brother, family and friends, and all others who have shown interest in this work.

BIBLIOGRAPHY

[ACDR09] K. Amiri, J. Cavallaro, C. Dick, and R. M. Rao. A High Throughput Configurable SDR Detector for Multi-user MIMO Wireless Systems. *Journal of Signal Processing Systems*, April 2009.

[AK87] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. on Programming Languages and Systems*, 9:pp. 491–542, 1987.

[Bal13] A. Balevic. *Exploiting Multi-Level Parallelism in Streaming Applications for Heterogeneous Platforms with GPUs*. Ph.D. thesis, Leiden University, 2013.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer, 1988.

[Ban97] U. Banerjee. *Dependence Analysis*. Loop Transformations for Restructuring Compilers. Kluwer, 1997.

[Bas04] C. Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Parallel Architecture and Compilation Techniques (PACT)*, pp. 7–16. September 2004.

[BBK+08] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Proc. of Compiler Construction (CC)*, pp. 132–146. March–April 2008.

[BBS09] T. Bijlsma, M. J. G. Bekooij, and G. J. M. Smit. Inter-Task Communication via Overlapping Read and Write Windows for Deadlock-Free Execution of Cyclic Task Graphs. In *Proc. of Systems, Architectures, Modeling and Simulation*, SAMOS'09, pp. 140–148. 2009.

[BBW+05] A. Burg, M. Borgmann, M. Wenk, M. Zellweger, W. Fichtner, and H. Bolcskei. VLSI Implementation of MIMO Detection using the Sphere Decoding Algorithm. *IEEE Solid-State Circuits*, 40(7):pp. 1566–1577, 2005.

[BEF+94] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, *et al.* Polaris: Improving the Effectiveness of Parallelizing Compilers. In *Languages and Compilers for Parallel Computing*, pp. 141–154. 1994.

[BELP96] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow. *IEEE Trans. on Signal Processing*, 44(2):pp. 397–408, 1996.

[BHLM94] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *Journal of Computer Simulation*, 4:pp. 155–182, April 1994.

[BM10] B. Bailey and G. Martin. *ESL Models and Their Application: Electronic System Level Design and Verification in Practice*. Springer, 2010.

[BSE04] A. Bahasi, B. Saltzberg, and M. Ergen. *Multi-Carrier Digital Communications Theory and Applications of OFDM*. Springer, 2004.

[BZNS12] M. Bamakhrama, J. Zhai, H. Nikolov, and T. Stefanov. A Methodology for Automated Design of Hard-Real-Time Embedded Streaming Systems. In *Design, Autom., Test Eur. (DATE)*, pp. 941–946. March 2012.

[Cal11] Calypto Design Systems. Catapult C. `http://www.calypto.com/catapult_c_synthesis.php`, 2011.

[CFGV09] P. Clauss, F. Fernández, D. Garbervetsky, and S. Verdoolaege. Symbolic Polynomial Maximization over Convex Sets and its Application to Memory Requirement Estimation. *IEEE Trans. on VLSI Systems*, 17(8):pp. 983–996, August 2009.

[CFH+06] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Platform-Based Behavior-Level and System-Level Synthesis. In *Proc. of the Int. SoC Conference*. September 2006.

[CGMT09] P. Coussy, D. Gajski, M. Meredith, and A. Takach. An Introduction to High-Level Synthesis. *IEEE Des. Test*, 26:pp. 8–17, July 2009.

[Che04] J. Chelikowski. Introduction: Silicon in all its Forms. In P. Siffert and E. F. Krimmel, editors, *Silicon: Evolution and Future of a Technology*. Springer, 2004.

[CJLZ09] J. Cong, W. Jiang, B. Liu, and Y. Zou. Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization. In *Intl. Conf. on Computer-Aided Design (ICCAD)*. November 2009.

[Cla07] Clang team. Clang: a C Language Family Frontend for LLVM. `http://clang.llvm.org/`, 2007.

[CLN+11] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE TCAD*, 30:pp. 473–491, April 2011.

[CM08] P. Coussy and A. Morawiec, editors. *High-Level Synthesis – From Algorithm to Digital Circuit*. Springer, 2008.

[DBC+07] H. Devos, K. Beyls, M. Christiaens, J. van Campenhout, E. D'Hollander, and D. Stroobandt. Finding and Applying Loop Transformations for Generating Optimized FPGA Implementations. *Trans. on High Perf. Embedded Architectures and Compilers I*, 4050:pp. 159–178, July 2007.

[Dec03] J. Decaluwe. MyHDL. `http://www.myhdl.org/`, 2003.

[DG98] A. Dasdan and R. Gupta. Faster Maximum and Minimum Mean Cycle Algorithms for System Performance Analysis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):pp. 889–899, October 1998.

[DGP+08] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, *et al.* System-on-Chip Environment: A SpecC-Based Framework for Heterogeneous MPSoC Design. *EURASIP Journal on Embedded Systems*, 2008(1), 2008.

[DHRT07] H. Dutta, F. Hannig, H. Ruckdeschel, and J. Teich. Efficient Control Generation for Mapping Nested Loop Programs onto Processor Arrays. *Journal of Systems Architecture*, 53(5–6):pp. 300–309, 2007.

[DRV01] A. Darte, Y. Robert, and F. Vivien. Loop Parallelization Algorithms. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pp. 141–172. 2001.

[DTD+09] C. Dick, M. Trajkovic, S. Denic, D. Vuletic, R. Rao, *et al.* FPGA Implementation of a Near-ML Sphere Detector for 802.16e Broadband Wireless Systems. In *Proc. of the SDR'09 Conference*. December 2009.

[DTZ+05] S. Derrien, A. Turjan, C. Zissulescu, B. Kienhuis, and E. Deprettere. Deriving Efficient Control in Process Networks with Compaan/Laura. *International Journal of Embedded Systems*, 2005.

[DV97] A. Darte and F. Vivien. Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs. *Intl. Journal of Parallel Programming*, 25(6):pp. 447–496, December 1997.

[EJL+03] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, *et al.* Taming Heterogeneity—the Ptolemy Approach. *Proceedings of the IEEE*, 91(2), January 2003.

[EZL89] D. Eager, J. Zahorjan, and E. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Trans. Computers*, 38(3):pp. 408–423, 1989.

[Fea88] P. Feautrier. Parametric Integer Programming. *RAIRO Recherche Opérationnelle*, 22(3):pp. 243–268, 1988.

[Fea91] P. Feautrier. Dataflow Analysis of Scalar and Array References. *Intl. Journal of Parallel Programming*, 20(1):pp. 23–53, 1991.

[Fea92a] P. Feautrier. Some Efficient Solutions to the Affine Scheduling Problem, Part I: One-Dimensional Time. *Intl. Journal of Parallel Programming*, 21(5):pp. 313–348, October 1992.

[Fea92b] P. Feautrier. Some Efficient Solutions to the Affine Scheduling Problem, Part II: Multi-Dimensional Time. *Intl. Journal of Parallel Programming*, 21(6):pp. 389–420, December 1992.

[Fea96] P. Feautrier. Automatic Parallelization in the Polytope Model. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pp. 79–103. Springer-Verlag, 1996.

[Fea06] P. Feautrier. Scalable and Structured Scheduling. *Intl. Journal of Parallel Programming*, 34(5):pp. 459–487, October 2006.

[Fin10] M. Fingeroff. *High-Level Synthesis: Blue Book*. Xlibris Corp., 2010.

[FK08] K. Fazel and S. Kaiser. *Multi-Carrier and Spread Spectrum Systems*. Wiley, 2008.

[GAGS09] D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer, 2009.

[GCD92] R. Gupta, C. Coelho, and G. DeMicheli. Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components. In *Design Automation Conf. (DAC)*, pp. 225–230. June 1992.

[Gei09] M. Geilen. Reduction Techniques for Synchronous Dataflow Graphs. In *Design Automation Conference (DAC)*, pp. 911–916. July 2009.

[Gem96] A. van Gemund. *Performance Modeling of Parallel Systems*. Ph.D. thesis, Delft University of Technology, April 1996.

[GGS+06] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, *et al.* Throughput Analysis of Synchronous Data Flow Graphs. In *Conference on Application of Concurrency to System Design (ACSD)*, pp. 25–36. June 2006.

[GHC+09] L. Gao, J. Huang, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr. TotalProf: a Fast and Accurate Retargetable Source Code Profiler. In *Proc. of CODES+ISSS*, pp. 305–314. October 2009.

[GKM82] S. Graham, P. Kessler, and M. McKusick. gprof: a Call Graph Execution Profiler. In *Symposium on Compiler Construction*, pp. 120–126. 1982.

[GL97] M. Griebl and C. Lengauer. The Loop Parallelizer LooPo. In D. Sehr *et al.*, editors, *Languages and Compilers for Parallel Computing*, volume 1239 of *LNCS*, pp. 603–604. Springer, 1997.

[GNB08] Z. Guo, W. Najjar, and B. Buyukkurt. Efficient Hardware Code Generation for FPGAs. *ACM Trans. Archit. Code Optim.*, 5(1):pp. 6:1–6:26, May 2008.

[GQR03] A.-C. Guillou, P. Quinton, and T. Risset. Hardware Synthesis for Multi-Dimensional Time. In *Application-Specific Systems, Architectures and Processors (ASAP)*, pp. 40–50. 2003.

[Gri04] M. Gries. Methods for Evaluating and Covering the Design Space during Early Design Development. *Integration, the VLSI Journal*, 38(2):pp. 131–183, 2004.

[GTA06] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs.

In *Arch. Support for Prog. Lang. and Operating Syst. (ASPLOS)*, pp. 151–162. 2006.

[GZA+11] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet. Polly – Polyhedral Optimization in LLVM. In *IMPACT Workshop*. April 2011.

[HH96] J. Hennessy and M. Heinrich. Hardware/Software Codesign of Processors: Concepts and Examples. In G. D. Micheli and M. Sami, editors, *Hardware/Software Codesign*, volume 310 of *NATO ASI*, pp. 29–44. Kluwer, 1996.

[HHK10] S. van Haastregt, E. Halm, and B. Kienhuis. Cost Modeling and Cycle-Accurate Co-Simulation of Heterogeneous Multiprocessor Systems. In *Design, Automation and Test in Europe (DATE'10)*, pp. 1297–1300. March 2010.

[HK09] S. van Haastregt and B. Kienhuis. Automated Synthesis of Streaming C Applications to Process Networks in Hardware. In *Design, Automation and Test in Europe (DATE'09)*, pp. 890–893. April 2009.

[HK12] S. van Haastregt and B. Kienhuis. Enabling Automatic Pipeline Utilization Improvement in Polyhedral Process Network Implementations. In *Appl.-specific Systems, Architectures and Proc. (ASAP'12)*. July 2012.

[HKL+08] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo. PeaCE: A Hardware-Software Codesign Environment for Multimedia Embedded Systems. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):pp. 24:1–24:25, May 2008.

[HNVK11] S. van Haastregt, S. Neuendorffer, K. Vissers, and B. Kienhuis. High Level Synthesis for FPGAs Applied to a Sphere Decoder Channel Preprocessor. In *Symposium on Field Programmable Gate Arrays (FPGA'11)*, p. 278. February–March 2011.

[Hoa85] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HRDT08] F. Hannig, H. Ruckdeschel, H. Dutta, and J. Teich. PARO: Synthesis of Hardware Accelerators for Multi-Dimensional Dataflow-Intensive Applications. In *Applied Reconf. Computing (ARC)*, pp. 284–289. 2008.

[HZ+10] S. van Haastregt, J. Zhai, *et al.* pntools. `http://daedalus.liacs.nl/pntools`, 2010.

[IP95] K. Ito and K. K. Parhi. Determining the Minimum Iteration Period of an Algorithm. *VLSI Signal Processing*, 11(3):pp. 229–244, 1995.

[JS05] A. Jantsch and I. Sander. Models of Computation and Languages for Embedded System Design. *IEE Proc. on Computers and Digital Techniques*, 152(2):pp. 114–129, March 2005.

[Kah74] G. Kahn. The Semantics of a Simple Language For Parallel Programming. In *Proc. of the IFIP Congress*. North-Holland Publishing, 1974.

[KDWV02] B. Kienhuis, E. Deprettere, P. v. d. Wolf, and K. Vissers. A Methodology to Design Programmable Embedded Systems – The Y-Chart Approach. In *Systems, Architectures, Modeling, Sim. (SAMOS)*, pp. 18–37. 2002.

[KES+00] E. de Kock, G. Essink, W. Smits, P. van der Wolf, J.-Y. Brunel, *et al.* YAPI: Application Modeling for Signal Processing Systems. In *Design Automation Conference (DAC'00)*, pp. 402–405. June 2000.

[Khr08] Khronos Group. OpenCL – The Open Standard for Parallel Programming of Heterogeneous Systems. `http://www.khronos.org/opencl/`, 2008.

[KKO+06] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, *et al.* UML-based Multiprocessor SoC Design Framework. *ACM Trans. Embed. Comput. Syst.*, 5(2):pp. 281–320, 2006.

[KSS+09] J. Keinert, M. Streubuehr, T. Schlichter, J. Falk, J. Gladigau, *et al.* SystemCoDesigner—an Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):pp. 1–23, 2009.

[Kum88] M. Kumar. Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications. *IEEE Trans. on Computers*, 37(9):pp. 1088–1098, September 1988.

[LA04] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization (CGO)*, pp. 75–88. March 2004.

[Lam88] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Programming Language Design and Implementation (PLDI)*, pp. 318–328. June 1988.

[LC10] R. Leupers and J. Castrillon. MPSoC Programming using the MAPS Compiler. In *Asia/South Pacific Design Automation Conference (ASP-DAC)*, pp. 897–902. January 2010.

[Lei08] Leiden Embedded Research Center. Daedalus home. `http://daedalus.liacs.nl/`, 2008.

[LL98] A. Lim and M. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Partitions. *Parallel Computing*, 24(3-4):pp. 445–476, 1998.

[LM87] E. Lee and D. Messerschmitt. Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing. *IEEE Trans. on Computers*, C-36(1):pp. 24–35, January 1987.

[LM98] W. Luk and S. McKeever. Pebble: A Language for Parametrised and Reconfigurable Hardware Design. In *Field-Programmable Logic and Applications (FPL)*, pp. 9–18. 1998.

[LRS93] C. Leiserson, F. Rose, and J. Saxe. Optimizing Synchronous Circuitry by Retiming. In *Third Caltech Conference On VLSI*. 1993.

[LS11] E. Lee and S. Seshia. *Introduction to Embedded Systems – A Cyber-Physical Systems Approach*. Lee and Seshia, 2011.

[LSV98] E. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 17(12):pp. 1217–1229, 1998.

[LSWD01] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere. System Level Design with Spade: an M-JPEG Case Study. In *Int. Conference on Computer Aided Design (ICCAD'01)*, pp. 31–38. 2001.

[Mar06] G. Martin. Overview of the MPSoC Design Challenge. In *Design Automation Conference (DAC)*, pp. 274–279. July 2006.

[Mar11] P. Marwedel. *Embedded System Design*. Springer, 2011.

[MBBM07] A. Moonen, M. Bekooij, R. v. d. Berg, and J. v. Meerbergen. Practical and Accurate Throughput Analysis with the Cyclo Static Dataflow Model. In *Proc. of MASCOTS*, pp. 238–245. 2007.

[MBGS10] O. Moreira, T. Basten, M. Geilen, and S. Stuijk. Buffer Sizing for Rate-Optimal Single-Rate Data-Flow Scheduling Revisited. *IEEE Trans. Comput.*, 59(2):pp. 188–201, 2010.

[Mei10] S. Meijer. *Transformations for Polyhedral Process Networks*. Ph.D. thesis, Leiden University, 2010.

[MK88] G. D. Micheli and D. Ku. HERCULES – A System for High-Level Synthesis. In *Design Automation Conference (DAC)*, pp. 483–488. 1988.

[MKMT90] G. D. Micheli, D. Ku, F. Mailhot, and T. Truong. The Olympus Synthesis System. *IEEE Design & Test*, 7(5):pp. 37–53, 1990.

[MNS09] S. Meijer, H. Nikolov, and T. Stefanov. On Compile-time Evaluation of Process Partitioning Transformations for Kahn Process Networks. In *Proc. of CODES+ISSS*, pp. 31–40. October 2009.

[MNS10] S. Meijer, H. Nikolov, and T. Stefanov. Throughput Modeling to Evaluate Process Merging Transformations in Polyhedral Process Networks. In *Design, Automation, and Test in Europe (DATE)*. March 2010.

[MPC88] M. McFarland, A. Parker, and R. Camposano. Tutorial on High-Level Synthesis. In *Design Automation Conf. (DAC)*, pp. 330–336. 1988.

[Muc97] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[MVBG+12] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt. An Overview of Today's High-Level Synthesis Tools. *Design Automation for Embedded Systems*, August 2012.

[Nad12] D. Nadezhkin. *Parallelizing Dynamic Sequential Programs using Polyhedral Process Networks*. Ph.D. thesis, Leiden University, 2012.

[NC10] R. Nikhil and K. Czeck. *BSV by Example*. Bluespec, Inc., 2010.

[NHS+11] R. Nane, S. van Haastregt, T. Stefanov, B. Kienhuis, V. M. Sima, and K. Bertels. IP-XACT Extensions for Reconfigurable Computing. In *Application-specific Systems, Architectures and Processors (ASAP'11)*, pp. 215–218. September 2011.

[Nik09] H. Nikolov. *System-Level Design Methodology for Streaming Multi-Processor Embedded Systems*. Ph.D. thesis, Leiden University, 2009.

[NNH+10] J. Noguera, S. Neuendorffer, S. van Haastregt, J. Barba, K. Vissers, and C. Dick. Sphere Detector for 802.16E Broadband Wireless Systems Implementation on FPGAs using High-level Synthesis Tools.

In *Conference on Software Defined Radio (SDR'10)*. November–December 2010.

[NNH+11] J. Noguera, S. Neuendorffer, S. van Haastregt, J. Barba, K. Vissers, and C. Dick. Implementation of Sphere Decoder for MIMO-OFDM on FPGAs using High-Level Synthesis Tools. *Analog Integr. Circuits Signal Process.*, 69(2-3):pp. 119–129, December 2011.

[NRD+09] H. Nikolov, A. Rao, E. Deprettere, S. Nandy, and R. Narayan. A H.264 Decoder: A Design Style Comparison Case Study. In *Asilomar Conf. on Signals, Systems, and Computers*, pp. 236–242. November 2009.

[NS07] N. Nethercote and J. Seward. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Programming Language Design and Implementation (PLDI)*, pp. 89–100. 2007.

[NSD08a] H. Nikolov, T. Stefanov, and E. Deprettere. Automated Integration of Dedicated Hardwired IP Cores in Heterogeneous MPSoCs Designed with ESPAM. *EURASIP Journal on Embedded Systems*, 2008.

[NSD08b] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and Automated Multi-processor System Design, Programming, and Implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(3):pp. 542–555, March 2008.

[NV08] S. Neuendorffer and K. Vissers. Streaming systems in FPGAs. In *Systems, Architectures, Modeling, and Simulation (SAMOS)*. July 2008.

[Ope97] OpenMP ARB. The OpenMP API Specification for Parallel Programming. `http://www.openmp.org/`, 1997.

[Pag96] I. Page. Constructing Hardware-Software Systems from a Single Description. *Journal of VLSI Signal Processing*, 12(1):pp. 87–107, 1996.

[Pap91] M. Papaefthymiou. Understanding Retiming Through Maximum Average-Weight Cycles. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 338–348. 1991.

[PBCC08] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative Optimization in the Polyhedral Model: part II, Multidimensional Time. In *Programming Language Design and Implementation*, pp. 90–100. 2008.

[PCB+06] S. Pop, A. Cohen, C. Bastoul, S. Girbal, P. Jouvelot, *et al.* GRAPHITE: Loop Optimizations Based on the Polyhedral Model for GCC. In *Proc. of GCC Developper's Summit*, pp. 179–198. June 2006.

[PD76] J. Patel and E. Davidson. Improving the Throughput of a Pipeline by Insertion of Delays. In *Symposium on Computer Architecture (ISCA)*, pp. 159–164. January 1976.

[PEP06] A. Pimentel, C. Erbas, and S. Polstra. A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels. *IEEE Trans. on Computers*, 55(11):pp. 99–112, February 2006.

[PGS+09] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu. FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs. In *Appl. Specific Proc. (SASP)*, pp. 35–42. July 2009.

[PK89] P. Paulin and J. Knight. Scheduling and Binding Algorithms for High-Level Synthesis. In *Design Automation Conf. (DAC)*, pp. 1–6. 1989.

[PP12] R. Piscitelli and A. Pimentel. Design Space Pruning through Hybrid Analysis in System-level Design Space Exploration. In *Design, Automation and Test in Europe (DATE)*, pp. 781–786. March 2012.

[Pug91] W. Pugh. Uniform Techniques for Loop Optimization. In *International Conference on Supercomputing (ICS)*, pp. 341–352. ACM, 1991.

[Pug92] W. Pugh. Definitions of Dependence Distance. *ACM Lett. Program. Lang. Syst.*, 1(3):pp. 261–265, September 1992.

[PW86] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):pp. 1184–1201, December 1986.

[Qui84] P. Quinton. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. In *ISCA*, pp. 208–214. 1984.

[RDK00] E. Rijpkema, E. Deprettere, and B. Kienhuis. Deriving Process Networks from Nested Loop Algorithms. *Parallel Processing Letters*, 10(2/3):pp. 165–176, 2000.

[Rij02] E. Rijpkema. *Modeling Task Level Parallelism in Piece-wise Regular Programs*. Ph.D. thesis, Leiden University, 2002.

[Row94] J. A. Rowson. Hardware/Software Co-Simulation. In *Design Automation Conference (DAC)*, pp. 439–440. 1994.

[SB00] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

[SD03] T. Stefanov and E. Deprettere. Deriving Process Networks from Weakly Dynamic Applications in System-Level Design. In *Hardware/software codesign (CODES)*, pp. 90–96. October 2003.

[SEJ11] M. Sackmann, P. Ebraert, and D. Janssens. A Model-Driven Approach for Software Parallelization. In *MoDELS Workshop*, pp. 276–290. 2011.

[SGB06] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design (ACSD)*, pp. 276–278. June 2006.

[SGB08] S. Stuijk, M. Geilen, and T. Basten. Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs. *IEEE Trans. Comput.*, 57(10):pp. 1331–1345, October 2008.

[SHP12] H. Shen, M. Hamayun, and F. Pétrot. Native Simulation of MPSoC Using Hardware-Assisted Virtualization. *IEEE Trans. on CAD of Integrated Circuits and Systems (TCAD)*, 31(7):pp. 1074–1087, 2012.

[SKD02] T. Stefanov, B. Kienhuis, and E. Deprettere. Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances. In *HW/SW codesign (CODES)*, pp. 7–12. May 2002.

[SM93] T. Shepherd and J. McWhirter. *Systolic Adaptive Beamforming*, volume 25 of *Springer Series in Information Sc.* Springer-Verlag, 1993.

[Spe97] R. Spence. *The Acquisition of Insight*. Royal College of Art, 1997.

[Ste04] T. Stefanov. *Converting Weakly Dynamic Programs to Equivalent Process Network Specifications*. Ph.D. thesis, Leiden University, 2004.

[Syn10] Synopsys. SynphonyC. `http://www.synopsys.com/Systems/BlockDesign/HLS`, 2010.

[Sys05] SystemC. IEEE Std 1666, 2005.

[SZT+04] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System Design using Kahn Process Networks: The Compaan/Laura Approach. In *Design, Automation and Test in Europe (DATE'04)*, pp. 340–345. February 2004.

[TCL05] T. Todman, J. G. F. Coutinho, and W. Luk. Customisable Hardware Compilation. *The Journal of Supercomputing*, 32(2):pp. 119–137, 2005.

[TKD03] A. Turjan, B. Kienhuis, and E. Deprettere. Realizations of the Extended Linearization Model. In *Domain-Specific Embedded Multiprocessors*, chapter 9, pp. 171–191. Marcel Dekker, Inc., 2003.

[TKD07] A. Turjan, B. Kienhuis, and E. Deprettere. Classifying Interprocess Communication in Process Network Representation of Nested-Loop Programs. *ACM Trans. Embed. Comput. Syst.*, 6(2), May 2007.

[TS09] L. Thiele and N. Stoimenov. Modular Performance Analysis of Cyclic Dataflow Graphs. In *EMSOFT'09*, pp. 127–136. October 2009.

[Tur07] A. Turjan. *Compiling Nested Loop Programs to Process Networks*. Ph.D. thesis, Leiden University, 2007.

[Ver03a] S. Verdoolaege. Barvinok. `http://freecode.com/projects/barvinok`, 2003.

[Ver03b] S. Verdoolaege. Integer Set Analysis tool set. `http://repo.or.cz/w/isa.git`, 2003.

[Ver08] S. Verdoolaege. Integer Set Library. `http://repo.or.cz/w/isl.git`, 2008.

[Ver10] S. Verdoolaege. Polyhedral Process Networks. In S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, editors, *Handbook on Signal Processing Systems*, pp. 931–965. Springer, 2010.

[Viv02] F. Vivien. On the Optimality of Feautrier's Scheduling Algorithm. In *Proc. of Euro-Par Conference*, pp. 299–308. Springer-Verlag, 2002.

[VNS07] S. Verdoolaege, H. Nikolov, and T. Stefanov. PN: a Tool for Improved Derivation of Process Networks. *EURASIP Journal on Embedded Systems*, 2007.

[VSB+07] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting Integer Points in Parametric Polytopes using Barvinok's Rational Functions. *Algorithmica*, 48(1):pp. 37–66, May 2007.

[WL91] M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. Parallel Distributed Systems*, 2:pp. 452–471, October 1991.

[Xil02] Xilinx, Inc. System Generator. `http://www.xilinx.com/tools/sysgen`, 2002.

[Xil11] Xilinx, Inc. AutoESL. `http://www.xilinx.com/tools/autoesl.htm`, 2011.

[Xil13] Xilinx, Inc. Vivado HLS. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design`, 2013.

[YBK+07] Y. Yankova, K. Bertels, G. Kuzmanov, G. Gaydadjiev, Y. Lu, and S. Vassiliadis. DWARV: DelftWorkBench Automated Reconfigurable VHDL Generator. In *Field Progr. Logic and Appl. (FPL)*, pp. 697–701. 2007.

[ZDG97] J. Zhu, R. Dömer, and D. Gajski. Syntax and Semantics of the SpecC Language. In *Proc. of the SASIMI Workshop*, pp. 75–82. 1997.

[ZI08] C. Zissulescu-Ianculescu. *Synthesis of a Parallel Data Stream Processor from Data Flow Process Networks*. Ph.D. thesis, Leiden University, 2008.

[ZKD04] C. Zissulescu, B. Kienhuis, and E. Deprettere. Increasing Pipelined IP core Utilization in Process Networks using Exploration. In *Field-Programmable Logic and Applications (FPL)*, pp. 690–699. 2004.

[ZNS11] J. Zhai, H. Nikolov, and T. Stefanov. Modeling Adaptive Streaming Applications with Parameterized Polyhedral Process Networks. In *Design Automation Conference (DAC)*, pp. 116–121. 2011.

[ZSKD03] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *Field Programmable Logic and Applications (FPL)*, pp. 911–920. 2003.

[ZTKD02] C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. Solving Out of Order Communication using CAM Memory; an Implementation. In *Circuits, Systems and Signal Processing (ProRISC)*. 2002.

[Zwi12] W. de Zwijger. *Increasing Explicit Parallelism in Polyhedral Process Networks using Transformations*. Master's thesis, LIACS, Leiden University, June 2012.

INDEX