Transformations for polyhedral process networks

Meijer, S.

Transformations for Polyhedral Process Networks

Sjoerd Meijer

Transformations for Polyhedral Process Networks

Proefschrift

ter verkrijging van

de graad van Doctor aan de Universiteit Leiden,

op gezag van Rector Magnificus prof.mr. P.F. van der Heijden,

volgens besluit van het College voor Promoties

te verdedigen op woensdag 8 december 2010

klokke 16:15 uur

door

Sjoerd Meijer

geboren te Leiderdorp

in 1979.

Samenstelling promotiecommissie:

promotor	Prof.dr. Ed F. Deprettere	Universiteit Leiden
co-promotor	Dr. Todor Stefanov	Universiteit Leiden
overige leden:	Prof.dr. Harry Wijshoff	Universiteit Leiden
	Prof.dr. Joost Kok	Universiteit Leiden
	Prof. Dr.-Ing. Jürgen Teich	Universität Erlangen-Nürnberg
	Prof.dr. Gerard Smit	Universiteit Twente
	Prof.dr. Henk Corporaal	Technische Universiteit Eindhoven

Transformations for Polyhedral Process Networks
Sjoerd Meijer. -
Thesis Universiteit Leiden. - With index, ref. - With summary in Dutch
ISBN 978-90-9025792-1

Contents

Chapter 1

Introduction

In 1965, Moore predicted that the number of transistors on a semiconductor and thus the overall chip performance would double every two years [56]. This has become known as Moore's law and due to the minitiurization of transistors, chip manufactures were able to produce faster, more powerful processors every year. Moore's law has proven to be correct for many years, but it was also clear that this trend had to come to an end at some point in time. Moore also stated that "no physical quantity can continue to change exponentially forever". Today, chip manufactures have to deal with electrical power leakage and heat dissipation as a result of packing more and more transistors into a smaller area. In addition, the minitiurization of transistors has reached its physical limits and it cannot further help in producing faster processors.

As a solution to produce more powerful processors, multi/many-core processor architectures were introduced. Multi/many-core processors consist of multiple processors, possibly of the same type, and are interconnected and integrated into a single chip. Hence, the name Multi-Processor Systems on-Chip (MPSoC). For example, mainstream consumer PCs nowadays come with dual/quad core processors, game consoles such as the `PlayStation 3` and its Cell processor have 9 cores [39], GPUs have 128 stream processors, and cell phones have many different compute and hardware components. Inspired by Moore's law, many people believe that the new trend is an exponential growth of the number of cores in processors. Processors, however, are only a small part of complex systems that are shipped to the market. Equally important is the entire software-stack that provides services to end-users and developers. A powerful processor is useless without good compilers, debuggers, simulators, operating systems, libraries, etc. So the programmability of a processor highly determines its success.

If we consider software compilers for single processors with a sequential execution model, then it is widely accepted that they do a reasonably good job in auto-

matically translating high-level program descriptions into low-level machine code. When the compiler technology for single processors matured, it raised the programming abstraction level and gave a boost to the productivity of developers and greatly improved maintainability and portability of program code. Both the hardware and compilers focused on exploiting Instruction Level Parallelism (ILP) as much as possible. Single processor architectures support ILP with superscalar, out-of-order, and instruction pipelining techniques implemented in hardware. For other architectures, such as VLIW [26] and EPIC [73] processors, it is the compiler's responsibility to find parallel instructions. Therefore, much research has been done in techniques such as automatic vectorization, software pipelining, and other scheduling techniques to overlap instructions (ILP) as much as possible.

While the programming of a single processor is already a difficult task, there is now another dimension of complexity with the introduction of Multi-Processor System on Chips (MPSoCs). The programming of these multi-processor systems is a difficult and time consuming process as it involves careful partitioning and assignment of program tasks to different processing elements of the MPSoC platform. A program task can for example be a function, i.e., a set of instructions, that reads function input arguments, performs some computations, and write the results to its function output arguments. Overlapping different program tasks by executing them in parallel at different processors of the MPSoC platform can result in significantly reduced execution times. This illustrates that besides Instruction Level Parallelism (ILP), that Task Level Parallelism (TLP) is an important factor that needs to be taken into account in programming MPSoC platforms. Exploiting TLP is difficult as the different program tasks need to synchronized and must also exchange data in a particular way, which makes the programming of MPSoC platforms more difficult than a single processor system. So the question is: how can MPSoC platforms be efficiently programmed using the available resources of the hardware platform?

If we roughly classify the different approaches to program Multi-Processor System on Chips (MPSoCs), we see that it is either the programmer's responsibility to create different program tasks, or a compiler oriented approach where program tasks are automatically extracted from sequential program specifications. Examples of the former approach are new programming languages (e.g., OpenCL [64], StreamIT [87]), language extensions (e.g., CUDA [59]), compiler pragma's (e.g., OpenMP), and libraries (e.g., Pthreads, MPI [27]). Examples of the latter are parallelizing compilers that extract program tasks or threads from sequential code (e.g., the Intel compiler [10], Pluto [13], SUIF [36], Polaris [12]). Parallelizing compilers is the subject of the work presented in this dissertation. The Leiden Embedded Research Center (LERC) has developed a tool-flow to program embedded Multi-Processor Systems on Chip (MPSoC) in a systematic and automated way. To be more specific, the goal is to make the programming more easy and to present a solution for the question raised

earlier: how to efficiently program an MPSoC. The LERC's solution relies on two basic principles: i) a parallel Model of Computation (MoC) must be used to specify an application, and ii) this parallel specification should be executed on a hardware platform that exactly matches the MoC.

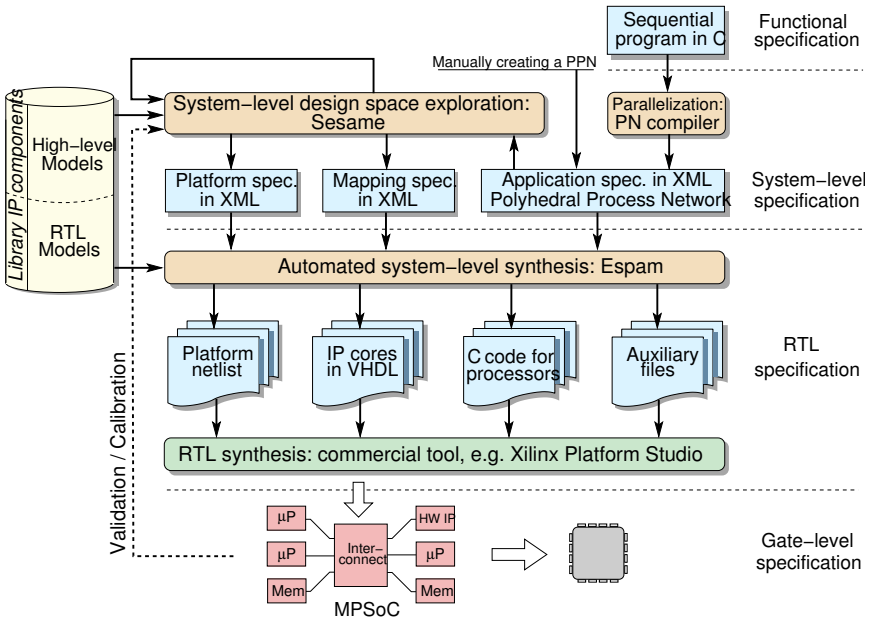

Figure 1.1: Daedalus tool-flow overview

The Daedalus tool-flow [61] that is being developed by LERC and shown in Figure 1.1, aims at providing a complete solution for system-level design of MPSoC platforms. It implements the two principles described above. From this tool-flow, let us consider first the functional specification of the application that a designer should provide. The first part of LERC's solution to make the programming of MPSoCs easier, relies on the fact that application developers find it more easy to specify an application as a sequential program as opposed to writing a parallel one. At the same time, we know that a parallel application specification can be mapped onto a parallel architecture more naturally than a sequential program. So, the idea is to combine the best of these worlds by deriving an equivalent parallel specification from sequential program specifications. This has resulted in the open-source `pn` compiler [95], that is part of the Daedalus tool-flow as shown in Figure 1.1. The `pn` compiler translates applications specified as Static Affine Nested-Loop Programs (SANLP), i.e., a subset of the C language as we discuss in Chapter 2, to Polyhedral Process Networks (PPNs) [8]. The PPN Model of Computation consists of autonomously running pro-

cesses with private memory and control that communicate over point-to-point FIFO channels using blocking FIFO read/write primitives (discussed in detail in Chapter 2).

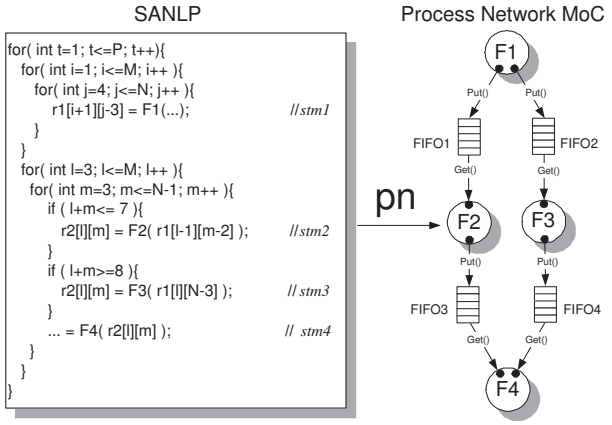

Figure 1.2: Compiling a Static Affine Nested-Loop Program (SANLP) to a Polyhedral Process Network

The derivation of a PPN from a static affine nested-loop programs is illustrated with an example in Figure 1.2. This example is taken from [89] and reveals how program statements are translated to processes and how array accesses are replaced by FIFO read/write statements. In Figure 1.2, a sequential program with 4 program statements is shown at the left-hand side. The statement's variable indexing functions are affine expressions in the loop iterators and static program parameters. The derived and functionally equivalent PPN for this code is shown at the right-hand side. Each program statement is translated to a process, and the array accesses have been replaced with read and write functions such that the processes only communicate data over FIFO channels.

Let us now consider the second design step of the Daedalus tool-flow, i.e., the translation from the system-level specification of the MPSoC platform to the RTL specification of the platform, as shown in Figure 1.1. The idea of the Daedalus tool-flow, is to generate a hardware platform that "natively" supports the execution of Polyhedral Process Networks (PPNs). That is, the ESPAM platform executes PPNs very efficiently because the operational semantics of the process network model of computation are supported with hardware components. For example, data communication and process synchronization of processes are realized by distributed memories, which can be organized as one or more FIFOs. Thus, blocking FIFO read/write primitives are hardware supported and make the processes to be self-scheduled very efficiently. Furthermore, the ESPAM platform allows processes to be assigned to independent

Instruction Set Architecture (ISA) components and/or IP-cores that must exist in the library of predefined IP components. The ESPAM tool automatically generates a hardware platform prototyped on an FPGA board based on 3 specifications as shown at the system specification level in Figure 1.1. The first specification is a high-level platform specification describing only the number of processing elements and the inter-connect of the platform. The second is an application specification in the form of a PPN that can be generated by the `pn` compiler, but can also be specified by hand. The third is a mapping specification describing how the processes of the PPN are assigned to the processing elements of the hardware platform. The ESPAM tool takes these 3 specifications as an input, and creates the corresponding RTL specification of the MPSoC platform and maps the PPN process threads onto IP-cores and/or programmable processors. Thus, we see that the Daedalus tool-flow enables designers to implement a sequential program specification onto a multi-processor system on chip in a systematic and automated way.

1.1 Problem Statement

The Y-chart approach is a very general iterative system-level design methodology [44]. Figure 1.3 illustrates this approach and captures the iterative process of getting

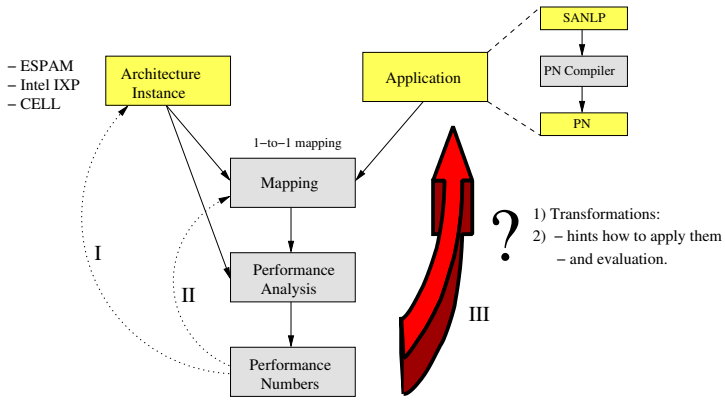

Figure 1.3: The Y-chart Approach

to a satisfactory design point. It takes an application specification and a platform specification. Then, after executing the application onto the platform, performance numbers are obtained for a particular design point. The performance of an application can be measured by considering the execution time or throughput of that application on a simulator or the real hardware platform. If the design point does not meet the

performance or resource constraints (i.e., the constraints on the number of tasks assigned to a processing element), then the platform, application and/or mapping can be adjusted accordingly. By iteratively changing some parameter values in this design methodology, the implementation should converge to, for example, the desired performance. Let us now project the different aspects of the Y-chart approach onto the Daedalus tool-flow. Recall that the Daedalus tool-flow (see Figure 1.1) takes the application, platform, and mapping specifications as an input, as shown in the Y-chart approach, and allows a designer to create and program an MPSoC platform. In addition, the Sesame tool [67, 88] that is integrated into Daedalus, can be used for design-space exploration at the system-level of abstraction. The Sesame tool, however, only explores different platform and mapping instances. These two design-space exploration aspects correspond to arrows `I` and `II` in the Y-chart approach, see Figure 1.3. The Daedalus tool-flow does *not* support the third exploration aspect, i.e., the exploration of different application instances as indicated with the bold arrow `III` in the Y-chart. Although some transformations have been defined to change a PPN application specification [79], i.e., to reduce/increase the number of processes in a PPN, the Daedalus tool-flow does not give any hints or tips to the designer how to apply these transformations in order to transform a PPN in the best possible way. Applying transformations as part of the tool-flow is the subject of this dissertation. It is crucial to assist the designer in applying the transformations in the best possible way since there are many possibilities to transform an application to meet the performance requirements or resource constraints. In this dissertation, we do not investigate different mapping strategies and always assume to have a 1-to-1 mapping of processes to processors. Thus, the grouping or splitting of tasks is not achieved by different mapping strategies, but by the `pn` compiler instead, i.e., we focus on the `pn` compiler that is used to derive PPNs from sequential program specifications. Although the `pn` compiler relieves the designer from the difficult and error-prone task of identifying and synchronizing different program partitions, it is not guaranteed that the performance/resource constraints are met. Recall that the `pn` compiler uses a partitioning strategy that creates a single thread for each program statement in the sequential code. As one program statement can be much more computationally intensive than others, the corresponding process network may be highly imbalanced not meeting the performance and resource constraints. Therefore, we formulate the first problem area as follows.

- **Issue I**: It is unlikely that all the designer's constraints are met in one translation step of the Daedalus tool-flow. That is, the Daedalus tool-flow can quickly generate a single design point, and can also explore different architecture and mapping instances by means of simulation. It, however, does not provide any compile-time infrastructure and hints/heuristics to transform and evaluate dif-

ferent application instances. Transforming application instances is crucial to meet the performance/resource constraints. Moreover, the compile-time hints are not only necessary to assist the designer in making the correct design decisions, but also to reduce the number of design points a designer should consider/evaluate. Therefore, the main research topic of this dissertation is to assist the designer in transforming a PPN specification to obtain a satisfactory design point as illustrated with the bold arrow `III` in Figure 1.3.

The first issue as discussed above addresses the program specification in the design process. A second addresses the target platform specification. The Daedalus tool-flow targets FPGA based platforms and creates an instance of the ESPAM execution platform. That is, an execution platform prototyped on an FPGA that matches the process network model of computation. However, such a specific platform may not always be available to a designer and we therefore formulate a second issue.

- **Issue II**: Currently, the Daedalus tool-flow aims at creating an MPSoC instance that exactly matches the process network model of computation on an FPGA based platform, but such a specific platform may not always be available. We want to investigate how to execute polyhedral process networks on programmable, off-the-shelf multi-processor platforms. This means that the different components of the process network model of computation must be mapped onto fixed hardware components of the target platform.

1.2 Contributions

To address the first issue as defined in Section 1.1, we define compile-time approaches to transform and thus optimize PPNs. These optimizations consist of compile-time guided application of transformations that restructure PPNs in a certain way. First, we briefly review the transformations as they have been defined in [78, 79] and then we present the contributions.

The first transformation is a process splitting transformation which increases the number of processes in a PPN, and the second is the process merging transformation which reduces the number of processes in a polyhedral process network:

1. The *process splitting* transformation is a transformation that copies program statements, comparable to the classical loop-unrolling transformation. As a result, the derived process network has multiple processes executing the same function possibly in parallel.

2. The *process merging* transformation achieves the opposite of the splitting transformation and groups, clusters, or merges several processes into one compound

process. The functions in the merged processes will be executed sequentially in the compound process.

Using these two transformations, an initial process network can be optimized to meet performance/resource constraints. The arbitrary PPN example shown in Figure 1.4, consists initially of 3 processes. Using the process merging transformation, processes *P2* and *P3* can be sequentialized into compound process *P23*. Thus, we say that less parallelism is exploited. By using the process splitting transformation, processes *P2* and *P3* can be split up to create extra copies. As a result, more processes can execute in parallel and thus we say that more parallelism is exploited.

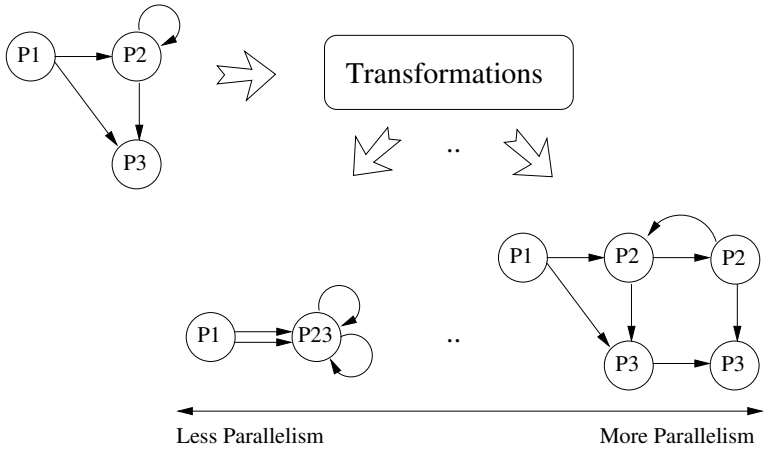

Figure 1.4: Deriving Different PPNs using Process Splitting and Merging Transformations

- **Contribution I [51, 53]:** our first contribution consists of compile-time solution approaches for process splitting and merging to assist the designer in achieving his performance/resource requirements:
  - *The process splitting transformation*: a process can be split up in many different ways and many factors influence the final performance results. We identify factors and define corresponding metrics that play a key role in the performance results, and show an analytical approach to calculate and evaluate them at compile-time. The analysis is performed locally on the process that is selected for splitting [51].
  - *The process merging transformation*: we define a throughput model for Polyhedral Process Networks (PPN). This allows the designer to evaluate

the throughput of different transformed networks derived from the same PPN. The designer, thus, can select the merging alternative with the best throughput. The throughput model is used for a global analysis of the entire network, as opposed to the splitting transformation, since the effects of the merging cannot be studied only by locally looking into the processes to be merged [53].

- **Contribution II [52]:** we present a holistic approach to use both the process splitting and process merging transformation in combination. This is a necessity to obtain good performance results that cannot be achieved by using only one transformation. Our solution approach solves the problem of ordering the different transformations and the problem of identifying the most suitable processes to merge/split. We create a number of load-balanced compound processes equal to the number of tasks a designer wants to create that can, for example, be the available processing elements of the target platform. In the holistic approach, we use the results of *Contribution I* to decide how the processes can be best split up, and the throughput model can be used for evaluating the solutions.

- **Contribution III [50,58]:** to address the second issue presented in Section 1.1, i.e., the programming of standard and off-the-shelf MPSoC platforms, we present approaches to execute PPNs onto the Intel IXP Network Processor and the Cell Processor. Thus, we investigate how to efficiently realize FIFO communication using the provided communication infrastructures of these platforms.

1.3 Related Work

The research work presented in this dissertation contributes to the underlying theory of the Daedalus tool-flow [61], and hence it contributes to the the research area of tool-flows for systematic and automated application-to-platform mapping, which has been widely studied in the research community. As it is an extensive research area, we first give a brief overview of related tool-flows. Then, we describe in more detail the related work with respect to the specific contributions of this dissertation.

To start with the frameworks, the System-On-Chip Environment (SCE) [21] enables designers to go from a specification all the way down to a hardware/software implementation. The Program State Machine (PSM) is used as a model of computation, which brings together concepts of hierarchical concurrent finite-state machines, dataflow graphs and imperative programming languages in a single model of computation [28, 33]. Basically, it encapsulates basic algorithms written in C, providing the designer in this way with the flexibility to manually write C and to manually parti-

tion the code in a particular way using a data flow model. This is different from the Daedalus approach, as the designer only writes the sequential top-level application description. It is the responsibility of the `pn` compiler to partition the code and to derive a polyhedral process network. The functionality of the processes in the Daedalus tool-flow can be specified by the designer as sequential functions in C, similar to SCE, or as IP-cores from the component library.

A second related framework is `SystemCoDesigner`, which maps applications specified in SystemC onto a heterogeneous platform [42]. Similar to the SCE approach, it is the designer's responsibility to write an actor orientated application in SystemC, whereas the Daedalus tool-flow derives Polyhedral Process Networks (PPNs) from a sequential program. Similar to Daedalus, it allows to create a heterogeneous MPSoC by instantiating and connecting cores from a component library. In addition, actors in `SystemCoDesigner` can be implemented as a hardware accelerator using the Forte Cynthesizer. The high-level synthesis of processes to hardware is currently not (yet) supported by Daedalus. A research work in the context of the Daedalus tool-flow explored the VHDL synthesis of processes in a PPN using PICO [91], but it is not integrated into the Daedalus tool-flow and thus not available yet.

Two more frameworks that provide a complete environment for modeling applications, design space exploration, prototyping and synthesis of MPSoC platforms are Koski [41] and PeaCE [35]. The main difference between Daedalus and Koski is that the functionality of the system in Koski is described with an application model in an UML environment. And PeaCE, that is short for Ptolemy extension as a Codesign Environment, restricts itself to SDF graphs and finite state machines as the model of computation.

Next, we briefly discuss four frameworks that focus more on the software part of MPSoC platforms. MAPS is a framework for MPSoC application parallelization [15]. It provides a set of tools which guides the parallelization processes. In contrast to our analytical compile-time parallelization approach, MAPS parallelization is mainly based on profile information and manually written Kahn Process Network (KPN) specifications. It provides a source-to-source translation, i.e., the output code is threaded C code that can be compiled with other compilers to the target platform. MAMPS [45] is another tool-flow that maps SDF graphs onto MPSoC platforms. Besides the difference that they map SDFs, the work focuses on homogeneous MPSoCs consisting of MicroBlaze processors that are point-to-point connected. Daedalus supports heterogeneous platforms and interconnects such as crossbars and shared busses. On the other hand, MAMPS supports the mapping of multiple applications, while Daedalus currently supports only single application mapping. The Distributed Operation Layer (DOL) [84] is another framework for specifying and mapping parallel applications onto heterogeneous multiprocessor platforms. The target platform is a

fixed tiled multi-processor embedded system. As an application model, Kahn Process Networks (KPNs) are used that are specified manually by the designer. In the performance analysis, a technique is used based on real-time calculus, which has some similarities with our throughput model used to evaluate process merging transformation, i.e., the second contribution of this dissertation. We discuss this in more detail when we discuss the related work for the process merging transformation. In the design space exploration of DOL, mainly different mappings are evaluated, but different instances of the KPN application are not explored. As a last framework, we briefly discuss Metropolis [5]. It uses a pre-defined platform such that the system design problem is reduced to mapping the desired functions onto the given platform. Metropolis is a very general framework as it does not define any specific design tools, such as for example Daedalus. Instead, based on a meta-model with formal semantics, it allows designers to simulate, formally analyze, and synthesize complex systems.

Next, we discuss the related work with respect to the specific contributions of this dissertation, i.e., the process network transformations and the mapping of PPNs onto programmable MPSoCs.

Our **process splitting** transformation is related to the loop unrolling transformation used in compiler design [57]. The relation is that both transformations aim at enhancing parallelism in a sequential program. However, loop unrolling enhances instruction level parallelism by copying a loop body several times and re-indexing the variables in the body, thus creating more parallel instructions and reducing the loop control overhead. In contrast, our splitting transformation enhances task-level parallelism by copying a program statement a number of times such that these copies can be encapsulated in concurrent processes. In [77], splitting and re-timing transformations are described for improving block schedules for Homogeneous Synchronous Data Flow (HSDF) graphs by exploiting inter-iteration parallelism. This is related to our splitting transformation in the sense that the latter also facilitate the exploitation of inter-iteration parallelism available in a SANLP when such program is converted to a set of PPN specifications. In [66], Parhi and Messerschmitt describe a splitting transformation developed to be applied on iterative data-flow programs. This transformation is similar to our splitting in that both transformations increase the number of tasks in a program and exploit the hidden concurrency for static programs. The main difference between our work and the work presented in [66, 77] however, is that we have devised an approach to evaluate the quality achieved by applying the transformations when targeting a particular MPSoC platform. We show in this dissertation, that there are several factors that must be taken into account when deciding what transformation to apply in order to improve the system performance. In contrast, in [77] the transformations are applied on the HSDF graph corresponding to an application where no information about the target implementation platform is con-

sidered. In [83], Teich and Thiele propose an approach to partition affine dependence algorithms for mapping onto reduced/fixed size processor arrays. Their approach is based on two transformations called *Expand* and *Reduce*. This relates to our work in the sense that process splitting transformations are also an approach to partition algorithms. However, there are two important differences. First, the result of the partitioning, i.e., the generated PPNs are suitable for mapping onto heterogeneous multi-processor platforms. Second, by using our process splitting transformations we do a *reverse* partitioning compared to the approach of Teich and Thiele. They start with a dependence graph (DG) representation of an algorithm which is the partitioning of an algorithm. Then they apply tiling (grouping) on the DG representation to obtain a desired partitioning in which less parallelism is exploited. In contrast, we start with a SANLP, derive a PPN, and by applying process splitting we partition the computational workload onto several processes. That is, in the proposed approach we take into account the characteristics of a particular MPSoC target platform and evaluate the quality of different (possible) transformations, thereby obtaining a desired partitioning in which more parallelism is exploited.

When we look at the **process merging** transformation, then we see that many related research works focus on the merging of tasks or processes, which is called clustering in the domain of Synchronous Data Flow (SDF) graphs [47]. These works, however, mainly deal with the code generation of clustered or grouped tasks itself [9, 23]. We analyze and model networks with a given compound process and schedule to compare different PPN instances by defining and using a throughput model, see Chapter 4. There are other works on throughput computation, but they are developed for SDF and CSDF models [30, 55], which are less expressive models than the PPN model we use. Besides the difference in the models of computation, there is also a difference in the analysis. That is, in [30] two approaches are presented to calculate the throughput of SDFGs based on either the conversion of SDF to Homogeneous SDF or on state space exploration. In both cases, the disadvantage is that the number of actor or states, respectively, can explode. The advantage, however, is that cyclic graphs can also be analyzed, while our approach is restricted to acyclic process networks. Another work also investigated the trade-offs in buffer requirements and throughput constraints for SDFs [80], and in a follow up also for cyclo-static dataflow graphs [81]. The analysis, again, relies on state-space exploration techniques, but it does investigate buffer requirements that we omitted in our throughput model. The reason is that we assume buffer sizes that give maximum performance, which are calculated by the `pn` compiler. Another main difference with these works is that we use the throughput model for evaluating and comparing the process splitting and merging transformations, while the throughput models for (C)SDF graphs focus only on buffer sizes and throughput. Thus, they do not investigate any transformations. Another analytical model for analyzing embedded real-time systems is network calculus [46] and an ex-

tension of this which is called real-time calculus [16, 85]. The analysis is based on the minimum and maximum number of events that arrive in a time interval, which are called the arrival curves. In a similar way, service curves are defined, which represent upper and lower bounds of the available resources in an interval. Based on given traces of event streams, timing properties, on-chip memory requirements, and the load on different platform components can be analyzed. This is different from our approach as we only analyze the throughput of the process network given the workload of each process. Thus, our approach does not require to have the event stream of the system, which may be difficult to obtain. In the network calculus, however, the minimum and maximum arrival of events are propagated and thus also the dynamic behavior is captured. In our approach, we calculate an average throughput and thus the dynamic throughput behavior of processes is not captured. It makes, however, our throughput model simple and very efficient to compare different network instances and process merging transformations. As a consequence, however, our approach does not analyze the memory requirements/constraints. While the network calculus does analyze the memory requirements, it can suffer from some inaccuracies when the bounds on the event streams are not tight. Finally, an approach is presented in [22] to automatically synthesize a multiprocessor architecture for process networks under particular mapping and performance constraints. This is different from our work as the process networks are not analyzed and transformed.

The second contribution of this dissertation deals with a **holistic approach to combine process splitting and merging transformations**, which is most closely related to the work in [31] that aims at exploiting coarse-grained task, data and pipeline parallelism in stream programs. The StreamIt [87] compiler derives stream graphs which are mapped on the Raw architecture and has optimizations for filter fusion and fission [32], comparable to our process merging and splitting transformations. In their approach, they start to fuse filters until a certain point and then perform fission on this coarsed-grain task to create more data-level parallelism. The fusion is performed as long as the result of each fusion is stateless. We show in Chapter 5 that processes with state (self-edges) and networks with cycles can also be fissed and that performance gains are possible, which is not considered in [31]. A second difference is that we derive process networks from sequential programs written in C and not in a language, such as the StreamIt language, that has constructs to specify filters and FIFO communication and each kernel has a single input and single output channel. The processes in our polyhedral process networks can have multiple input/output channels and can read/write all or a subset of these channels. In [14], another approach is shown for mapping stream programs onto heterogeneous multiprocessor systems. A partitioning algorithm is presented that takes as input a graph, and outputs a mapping to fuse kernels to tasks. In an iterative manner, tasks are merged, kernels are moved from bottleneck processors, and tasks are created. Similar to the StreamIt approach, an

annotated version of the C programming language is used, and only stateless kernels are split for greater parallelism. Besides the average load of each kernel on each processor, similar to the workload of our processes, an additional parameter is required to be obtained from run-time analysis. That is, the average date rate on each stream that must be obtained from a profile.

In [68], the scheduling of Synchronous DataFlow (SDF) graphs [47] to parallel targets focused on partitioning and scheduling techniques that exploit task and pipeline parallelism. To schedule a SDF graph, a precedence graph is first constructed, which exposes the available data level parallelism. Then, to limit the explosion of nodes, clustering is applied and thus composite nodes are created. A fundamental difference with our work is that workloads are not taken into account in the clustering as we discuss in Chapter 5. And in addition to this, polyhedral process networks are more expressive than SDFs as FIFO channels can be read/written in a way that are described by (parameterized) polytopes. Thus, FIFO reads/writes can occur in some patterns, similar to the cyclo-static dataflow graphs (CSDF) [11], with the difference that the cycles in PPNs can be very large as they are derived from nested-loop programs. The R-Stream compiler [54] is a proprietary high level compiler for stream programs. It also uses the polyhedral model to partition code and data for a parametric parallel machine. The work focuses on the re-scheduling of computations (e.g., modulo scheduling) and placing explicit communications (e.g., DMA calls) to automatically put a multi-buffering scheme in place. Thus, the focus is on scheduling at the level of statement instances, and not on tasks/processes that can contain many statement instances as in our case.

The third contribution of this dissertation investigates **the mapping of polyhedral process networks onto programmable MPSoC platforms** such as the Intel IXP network processor and the Cell processor. We have developed source-to-source translation tool-flows to generate compilable source-code for the different components of PPNs. i.e, the processes and FIFO channels as we discuss in Chapter 6. To program the IXP, some high-level programming models have been developed. This basically means that the developer can use some higher-level languages and abstractions, e.g., the possibility to compose a number of operations that work on streams of data, and that assembly language is not a developer's only option. NP-Click [75] is one example as it offers an abstraction of the underlying hardware. Another effort for improving the programming of an IXP, is the μL programming language and the μC compiler by Network Speed Technologies [29, 82]. The difference with our approach is that both NP-Click and μL programming language, obviously, focus very much on internet packet handling, while we are interested in a programming model that supports the class of stream-based applications. Intel on the other hand, has developed an auto-partitioning C compiler as described in [49], which is therefore more closely related to our approach. An input application is specified as a set of

sequential C programs, which are called packet processing stages (PPSes). These PPSes closely correspond to the Communicating Sequential Processes (CSP) model of computation [37]. However, to express a program in PPSes is the responsibility of a programmer. In contrast, the `pn` compiler automatically generates PPNs from applications written as static affine nested loop programs [95].

Regarding the Cell processor, a great number of research works have been published since its introduction: ranging from case-studies and application specific implementations, to frameworks that deal with parallelization and mapping of applications onto the Cell. One model-based project that is similar to our approach in programming the Cell BE platform is the architecture-independent stream-oriented language StreamIt, which shares some properties with the Synchronous DataFlow (SDF) model of computation. The Multicore Streaming Layer (MSL) [99] framework realizes the StreamIt language on the Cell BE platform thereby focussing on automatic management and optimization of communication between cores. All data transfers are explicitly controlled by a static scheduler. This is different from our approach, since we use the PPN model of computation where the processes synchronize and communicate data over FIFO channels using blocking read/write primitives in absence of a global scheduler. A PPN is therefore self-scheduled, which can have as an advantage that there is no central scheduler that can become the bottleneck of the system. On the other hand, the blocking FIFO communication is software implemented, which makes it expensive communication primitives to use. As a last difference, and already discussed in this section, the SDF MoC used by StreamIt is less expressive Model of Computation (MoC) than our PPN MoC. Besides frameworks that support the parallel execution of applications, there are also communication libraries that focus more on the low-level communication infrastructure of the Cell, such as for example the Cell Messaging Layer [65]. It presents a similar idea as in our approach, i.e., a receiver initiated communication scheme as we will discuss in Section 6.1. However, the library offers just low-level send and receive primitives without focusing on the realization of more complex communication schemes such as FIFO reads/writes.

1.4 Outline

The remaining part of this dissertation is organized as follows.

In Chapter 2, we first introduce the basic terminology and show with a simple running example how polyhedral process networks are derived from sequential static affine nested-loop programs.

In Chapter 3, we present the first process network transformation, i.e., the process splitting transformation. We define the metrics that play an important role in process

splitting and give a solution approach how these can be evaluated at compile-time to select the best partitioning.

In Chapter 4, we discuss the second transformation, i.e., the process merging transformation. In order to evaluate which merging is the best, we define a throughput model for process networks such that the throughput for a given PPN can be calculated and evaluated.

In Chapter 5, we present a holistic approach to transform PPNs using the process splitting and merging transformations in combination. We show that it is necessary to use both transformations to achieve the best performance results that cannot be achieved using one transformation only.

In Chapter 6, we present approaches to realize FIFO communication for executing polyhedral process networks on the Intel IXP network and the Cell BE processors. Both platforms are instances of programmable MPSoCs platform, but each with their own characteristics. While the IXP has hardware support for FIFO communication to some extent, the CELL must implement FIFO communication completely in software.

Finally, we conclude this dissertation in Chapter 7 with a summary of the presented research work along with some concluding remarks.

Chapter 2

Background

In this chapter, we give the definitions and notations that are used throughout the rest of this dissertation, i.e., we review some basic mathematical notations and definitions as discussed in for example [72, 74]. We thereby focus on polyhedra and the polyhedral model that are used by compiler optimizations to efficiently analyze and transform input programs. Then, we define the input programs, i.e., the class of applications, that can be analyzed with this polyhedral model and show an example of a Polyhedral Process Network (PPN). We discuss the structure and properties of PPNs, which is necessary to understand the chapters that deal with analyzing and transforming PPNs.

2.1 Polyhedra

The **scalar product** or **inner product** of two vectors **a** and **b**, denoted by $\mathbf{a} \cdot \mathbf{b}$, is defined as $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T\mathbf{b} = \sum_{i=1}^{n} a_i b_i$, where $a = (a_1, .., a_n)$ and $b = (b_1, .., b_n)$ are column vectors. Note that $\mathbf{a} \cdot \mathbf{b} = \mathbf{0}$ iff vectors **a** and **b** are orthogonal or $\mathbf{a} = \mathbf{b} = 0$.

Given a non-zero vector **y** in $\mathbb{R}^n$ and a constant α, the following sets of points are defined:

- A **hyperplane** $H = \{\mathbf{x} \mid \mathbf{x} \cdot \mathbf{y} = \alpha\}$.
- A **closed half-space** $H = \{\mathbf{x} \mid \mathbf{x} \cdot \mathbf{y} \geq \alpha\}$.
- An **open half-space** $H = \{\mathbf{x} \mid \mathbf{x} \cdot \mathbf{y} > \alpha\}$.

An affine hyperplane is a $(d-1)$-dimensional hyperplane in a d-dimensional space, and thus divides the space in exactly two parts. A line, for example, is an affine

hyperplane in a 2-dimensional space, but not in a 3-dimensional space. We will use hyperplanes to define a polyhedron, but also in the process splitting transformation to partition processes in PPNs (see Chapter 3).

A **rational polyhedron** $\mathcal{P}$ is a subset of $\mathbb{Q}^d$ bounded by a finite number of closed half-spaces, i.e.,

$$\mathcal{P} = \{\mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} \geq \mathbf{b}\} \tag{2.1}$$

where A is an integral $m \times d$ matrix, and b is an integral vector of size m.

A **polytope** is a bounded polyhedron.

Figure 2.1 shows two 2-dimensional spaces with a number of closed half-spaces defining two polyhedra. The purpose of this example is to show the difference between a polyhedron and polytope. In Figure 2.1 A), a polyhedron is shown that is defined by only two constraints. As a result, the polyhedron is unbounded because there are no constraints on the maximum values that the points can have. In contrast, Figure 2.1 B) shows 4 lines/constraints that encapsulate all points within the grey area, which makes it an example of a bounded polyhedron, i.e. a polytope.

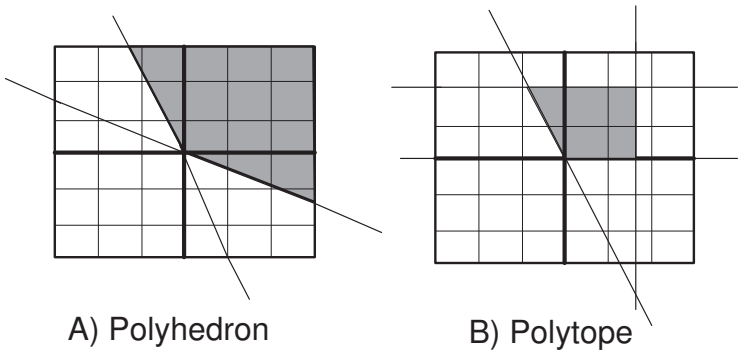

Figure 2.1: Polyhedron vs. Polytope

Polyhedra can also depend on a vector of parameters, denoted by $\mathbf{p}$, and we therefore define a parameterized polyhedron, denoted by $\mathcal{P}(\mathbf{p})$.

A parameterized polyhedron $\mathcal{P}(\mathbf{p})$ is a polyhedron whose closed half-spaces are affinely dependent on a vector of parameters $\mathbf{p} \in \mathbb{Q}^d$, i.e.,

$$\mathcal{P}(\mathbf{p}) = \{\mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} \geq B\mathbf{p} + \mathbf{b}\} \tag{2.2}$$

where A is an integral $m \times d$ matrix, B is an integral $m \times n$ matrix, and b is an integral vector of size m.

We use polyhedra to model all iterations of a program statement in nested-loop programs. That is, we extract and use the polyhedral model to efficiently analyze and transform input programs, which we further discuss in Sections 2.4 and 2.5. In Section 2.2, we first discuss how different points in a set can be compared and ranked using Parametric Integer Linear Programming (PILP) techniques.

2.2 Lexicographic Order

In program analysis, many problems can be formulated as a Parametric Integer Linear Programming (PILP) problem. An example of such a problem is to find the first, or last, array element accessed by a program statement in a nested-loop. Thus, parametric integer programming [24], [74] is used to find exact solutions and feasible points ranked according to a lexicographic order. In program analysis of nested-loop programs, we are dealing with sets of integer vectors defined by linear inequalities. If we consider a set S as an example, then recall from Section 2.1 that it is defined as $S = \{x \in \mathbb{Z}^d \mid Ax \geq b\}$ with $A \in \mathbb{Z}^{m \times d}$ and $b \in \mathbb{Z}^d$. Then, parametric integer linear programming is used to find the minimum or maximum point in set S. And two points $a \in \mathbb{Z}^n$ and $b \in \mathbb{Z}^n$ in set S can be compared by using the lexicographic order.

We say that **a** is **lexicographically smaller** than **b**, denoted by $\mathbf{a} \prec \mathbf{b}$, if for the first position i in which both vectors are different, we have $a(i) < b(i)$. This is expressed as a set of equalities and inequalities as:

$$\mathbf{a} \prec \mathbf{b} \equiv \bigvee_{i=1}^{n} \left(a(i) < b(i) \wedge \bigwedge_{j=1}^{i-1} a(j) = b(j) \right) \tag{2.3}$$

Let us take as an example a set S with 5 elements: $S = \{(1,1), (1,2), (2,1), (2,2), (2,3)\}$. Using Formula 2.3, we see that $(1,1)$ is lexicographical smaller than $(1,2)$, denoted by $(1,1) \prec (1,2)$, because $(1 = 1 \wedge 1 < 2)$. Similarly, we see that $(1,1)$ is lexicographical smaller than $(2,3)$, i.e., $(1,1) \prec (2,3)$, because comparing the first component of both points gives $(1 < 2)$. Element $(1,1)$ is the smallest element of set S and we define it as the lexicographical minimum element, denoted by $lexmin$. Similarly, we also define the lexicographical maximum point as the largest element, denoted by $lexmax$. For set S, element $(2,3)$ is the largest element. The problem of finding the lexicographical minimum/maximum point within a set of linear constraints can be solved with PILP. The example set S as we have defined it above

can also be represented by a set of constraints, i.e., $S = \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 2 \wedge 1 \leq j \leq 3\}$, and the ILP problem (no parameters are used in this example) can be subsequently formulated as shown in Table 2.1.

Objective:	$lexmin\{(i,j)\}$
Subject to:	$1 \leq i \leq 2$
	$1 \leq j \leq 3$

Table 2.1: Constraint system

The solution to find the minimum point for a given convex domain is based on the dual simplex algorithm [48] that is implemented in open-source libraries such as isl [93], Parma Polyhedral Library [4], and Piplib [24]. On a very high-level, the idea of the PIP algorithm and dual simplex method, is to find a minimum real point for a given convex set. Then, iteratively, new constraints are added not removing any integer points from the set. These libraries will thus find $(1,1)$ as the lexicographical minimum, and $(2,3)$ as the lexicographical maximum point.

Using the lexicographical order, it is also possible to rank an iteration point in polyhedra.

Definition 1 *The **rank** of a point $p \in \mathcal{P}$, is a number $n \in \mathbb{Z}$ denoting all points that are lexicographical smaller than p.*

For example, let us consider point $(i = 1, j = 3)$ of the `filter` function call statement in Figure 2.3 B). To rank this point, we use the lexigraphical order to determine all points that precede $(1,3)$. Therefore, we first consider all points that are smaller in the first component of point $(1,3)$, i.e., $i < 1$. The points that satisfy this constraint, corresponds to all points within the top most and largest grey box in Figure 2.3 B); for all these points $i = 0$. In addition, we consider the points that have the same value in the first component, but which have a smaller value in the second component, i.e., $i = 1 \wedge j < 3$. This corresponds to all points within the second and smallest grey box in Figure 2.3 B). Thus, the rank of point $(1,3)$, corresponds to the number of elements in the set $(i < 1 \ \vee \ (i = 1 \wedge j < 3))$, i.e., all greyed points in Figure 2.3 B). If we assume $N = 100$, then the rank of $(1,3)$ is $100+2 = 102$, which is thus obtained by counting the number of points in a set. Counting the number of points in (parametric) polyhedra, i.e., the enumeration of (parametric) polyhedra, is a research field in itself. The basic idea is to derive a quasi-polynomial that describes the number of integer points in a polytope $\mathcal{P}$. For an in-depth discussion, the reader is referred to, for example, the works [18], [97]. In this dissertation, we use that work which is implemented in the polyhedral library PolyLib [98]. Thus, when we want

to know the cardinality, or the number of points, of a set S, which we denote by $|S|$, then we use the counting functions from these libraries.

2.3 Static Affine Nested-Loop Programs

In Section 2.5, we consider parallel application specifications that are functionally equivalent to sequential program specifications that are static affine nested-loop programs. These are the subject of this section.

Definition 2 *A **static affine nested loop program (SANLP)** is a program where each program statements is enclosed by one or more loops and if-statements, and where:*

- *loops have a constant step size;*
- *loops have bounds that are affine expressions of the enclosing loop iterators, static program parameters, and constants;*
- *if-statements have affine conditions in terms of the loop iterators, static program parameters, and constants;*
- *index expressions of array references are affine constructs of the enclosing loop iterators, static program parameters, and constants;*
- *data flow between statements in the loop is explicit, which prohibits that two statements that contain function calls communicate through shared variables invisible to the compiler.*

An example of a static affine nested-loop program is shown in Figure 2.2.

```
 1  #parameter  10 <= N <= 100;

 2  for (i=0; i<= 2*N; i++)
 3    for (j=0; j<= 4*N; j++)
 4       a[i][j] = read_data ();          // statement S0

 5  for (i=0; i<= N; i++) {
 6    for (j=i; j<= N; j++) {
 7      if (i+j <= N-1) {
 8        a[i][j] = filter(a[2*i][4*j]);  // statement S1
 9      }
10      write_data(a[i][j]);              // statement S2
11    }
12  }
```

Figure 2.2: Example code of a SANLP

A static program parameter `N` is defined in `line 1`. This static parameter indicates that `N` can take a value between 10 and 100 which, however, cannot change at run-time. Using static parameters is very useful because an equivalent parallel specification, such as a PPN, needs to be derived only once, even if some requirements of the application change. Loops need not necessarily be perfect nests. That is, the program statements can appear at any level of the nested-loop, and thus not necessarily at the innermost loop level. Furthermore, the program statements can be guarded by if-statements, as shown in `line 7`. However, the conditions in these if-statements can only be affine combinations of loop iterators, static program parameters, and constants, and thus cannot have data dependent behavior. The functions in `line 4, 8, 10` read and write data only through arrays, and not for example through shared variables, or pointers to the arrays not visible to the compiler. In other words, the data flow is made explicit by reading/writing data only through affine array accesses.

The polyhedral model is an appealing model to represent and manipulate loop nest structures and their program statements in static affine-nested loop programs, as shown in for example [69], [63], [70]. Program parts that can be modeled with the polyhedral model are called *static control parts (SCoPs)* in the compiler community [76]. To be more precise, a SCoP is defined as a single-entry-single-exit region of the control-flow where loops bounds and conditional predicates are affine functions enclosing loop counters and invariant parameters. Once the polyhedral model is extracted from a SANLP or SCoP, see Section 2.4, data dependence analysis and loop restructuring transformations such as loop fusion, loop fission and strip-mining can be efficiently implemented using existing tools (e.g., PolyLib [98], the Parma Polyhedral Library, and Cloog [7]). The reason is that the iteration domain of a program statement, i.e., all iterations of that statements, are represented by a single geometrical object - a polyhedron. This polyhedron can be analyzed with PILP techniques as presented in Section 2.2.

Although the polyhedral model does impose some restrictions on the input program, in many application domains it is natural to express time critical parts of the applications in the form of a SANLP. Examples are DSP and audio/video stream-based applications in consumer electronics, modeling and simulation applications in high performance computing, molecular biology, radio astronomy, medical imaging, and high energy physics. Therefore, the polyhedral model is highly relevant because it enables efficient code restructuring and analysis in many program code parts.

2.4 Extracting the Polyhedral Model from SANLPs

The polyhedral model is a description of all program statements and their iteration points in Static Affine Nested-Loop Programs (SANLPs) with polyhedra. We refer to all iteration points of a program statement as the iteration domain, which in the program code (i.e., the SANLP) is defined by the enclosing loops of the program statements. Since the iteration points of a program statement are executed in a particular order, the polyhedral objects that model these iterations are ordered as well, i.e., the polyhedral model that we use for our program analysis consist of:

- polyhedra that define the iteration domains of program statements,
- a lexicographical ordering (see Section 2.2) of the points within the polyhedra.
- and data access functions for array references, which map a point from the iteration domain to a point in the data space that is accessed by the array references, i.e., the affine index expression as discussed in Section 2.3.

In the polyhedral model that we extract from SANLPs, an iteration vector is associated with each program statement. The dimension of the vector is equal to the number of loops that enclose the statement. The i-th component of the vector corresponds to the value of a loop iterator at depth i. Thus, the iteration domain of a statement is given by a set of linear inequalities defining a polyhedron in an d-dimensional domain, where d corresponds to the dimension of the iteration vector, i.e., the depth of the enclosing loop nest. In fact, the polyhedral model of the iteration domain of a statement is just a set of linear equalities and inequalities. Here is an example.

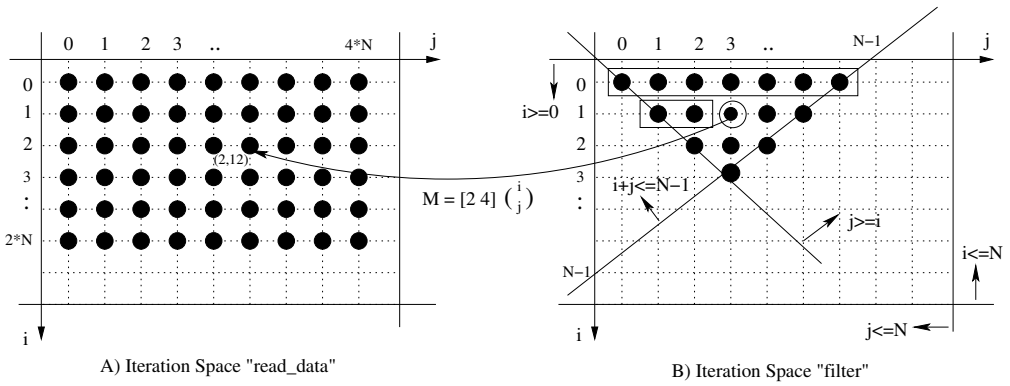

Figure 2.3: Iteration Space of $read\_data$ and $filter$ Function Call Statements

Figure 2.3 shows the two iteration domains of the `read_data` and `filter` function call statements from Figure 2.2. Let us focus on the iteration domain of the

`filter` function call statement shown in Figure 2.3 B). For brevity, we refer to this statement as $S1$. Since statement $S1$ is enclosed by two for-loops i and j, its iteration domain is 2-dimensional, and is referred to as D_{S1}. The lower/upper bounds of the enclosing loops are the first constraints that we take into account when defining the iteration domain of $S1$. Loop i starts at 0 and has maximum value of N, which translates to the following 2 constraints: $i \geq 0$ and $i \leq N$. Loop j has an initial value equal to i and has a maximum value of N, which translates to another two constraints: $j \geq i$ and $j \leq N$. In addition to the constraints imposed by the lower/upper bounds of loops, the execution of program statement $S1$ is guarded by an if-statement, which imposes another restriction on the iteration domain, i.e, only iteration points smaller than $i + j \leq N - 1$ are executed. Figure 2.3 B) shows 5 different lines in a 2-dimensional domain, which correspond to the 5 constraints imposed by the upper/lower bounds of the loops and the if-statement as we have described above. Thus, the constraints restrict the iterations points that are executed by $S1$, and the iteration points actually executed by $S1$ are denoted by the solid dots in Figure 2.3 B), i.e., they form a triangle. These iteration points are executed in the order from top to bottom and from left to right. We have extracted all constraints on the execution of $S1$ to define its iteration domain D_{S1} in the polytope representation: $D_{S1}(N) = \{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i \leq N \wedge i \leq j \leq N \wedge i + j \leq N - 1\}$. All executions of program statement $S1$ are in this way represented with one geometrical object, i.e., a polytope. Once an iteration domain has been extracted for a statement, it can be efficiently further analyzed and transformed using polyhedral analysis and tools. For example, the number of integer points of an iteration domain can be counted [18], [96] which is useful for loop optimizations [20] and data cache analysis [19]. Another application is the (re)scheduling of iterations and subsequently the code generation of iteration domains [7].

2.5 Polyhedral Process Networks

Extracting the polyhedral model for SANLPs as discussed in Section 2.4, enables exact data-flow analysis of scalar and array references. This exact data flow analysis uses the PILP techniques as discussed in Section 2.2 and is the most fundamental step in deriving PPNs in a fully analytical way from SANLPs as described in [89, 90, 95]. For an in-depth discussion on the derivation of PPNs, the reader is referred to these works. In this section, we only discuss the different properties of PPNs, and show the corresponding PPN for the code example in Figure 2.2.

In the partitioning strategy of the `pn` compiler [95], one autonomous process with local control and memories is created for each program statement. Subsequently, the control for the FIFO communication is automatically derived. We refer to pro-

cess networks derived by the `pn` compiler as *polyhedral process networks* (PPNs). The reason is that they are functionally equivalent to Static Affine Nested Loop Programs (SANLPs), the processes are structured in a particular way, and the execution of processes and FIFO reads/writes are described by polyhedra. Polyhedral process networks are, therefore, a special case of Kahn Process Networks (KPNs) [40], because Kahn Process Networks is a simple, yet powerful model of computation that only specifies how processes synchronize and communicate. Thus, the KPN model of computation does *not* impose any restrictions on, for example, the internal structure of processes and only defines that processes use a blocking FIFO read primitive and have unbounded FIFO buffers. However, as already mentioned above, the processes in PPNs are internally structured in a particular way. That is, in each execution of a process, we can distinguish a *Read* phase (R), an *Execute* phase (E), and a *Write* phase (W). To be more specific, a process consists of:

1. a list of *input port domains* to read all the function input arguments from the corresponding input FIFO channels,

2. a *function* that processes the input arguments and produces function output arguments, and

3. a list of *output port domains* to write the function output arguments to the corresponding output FIFO channels.

There can be two exceptions: *source* and *sink* processes. The former only generates data and does not read any data from other processes. The latter only collects data and does not write any data to other processes. However, source/sink processes can have incoming/outgoing channels, but then these channels are self-channels and data is read/written from/to itself. We illustrate the structure of the processes in a PPN with an example shown in Figure 2.4. This PPN is derived from the SANLP shown in Figure 2.2, where we have set the parameter N to 100. Since that SANLP consists of 3 statements $S0$, $S1$ and $S2$, the corresponding PPN consists of 3 processes $P0$, $P1$ and $P2$.

It can be seen that process $P0$ is a source process because it does not read data from other processes, and that process $P2$ is a sink process because it does not write data to other processes. Process $P1$, on the other hand, first reads data from FIFO channel $F1$, processes it by executing function `filter`, and writes the result to its outgoing FIFO channel $F3$. Thus, it clearly shows the different read, execute, write phases as also indicated with the letters R, E, and W in Figure 2.4. Furthermore, we see that each process executes a particular function that corresponds to a function from the SANLP.

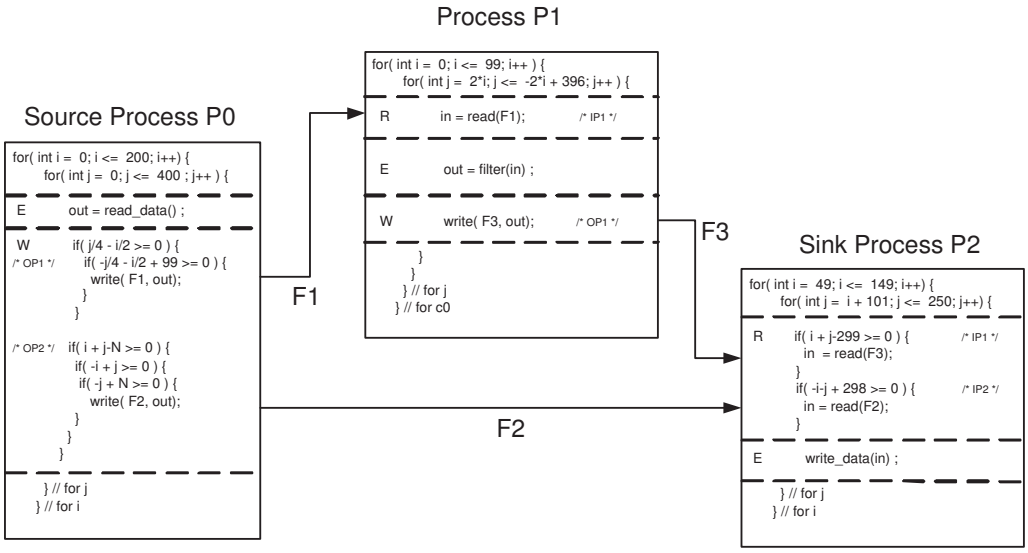

Figure 2.4: Derived Polyhedral Process Network (PPN) Model and its Representation in Executable Program Code

Definition 3 *A **process function** represents the computational part of a process. It corresponds to a function call statement in the sequential application that is a pure function without side-effects which only reads/writes through its input/output arguments.*

For the PPN in Figure 2.4, the process function of $P0$ is `read_data`, and we see that `filter` and `write_data` are the process functions of processes $P1$ and $P2$, respectively. These process functions are important for the process splitting and merging transformations that we present in Chapters 3 and 4, because the goal of both transformations is to create a more load-balanced PPN. Therefore, it is important to know the cost for executing the process function once, and we refer to this as the process workload.

Definition 4 *The **process workload** of a process P_i, denoted by W_{P_i}, represent the total number of required time units to execute the process function once, provided that i) all its input data is available (i.e., the reading phase is ignored), and ii) the time to write the output data is excluded (i.e., the writing phase is ignored).*

To give an example for the PPN shown in Figure 2.4, one can think of the `read_data` process function as a very light-weight process function that only reads data from a

memory location, i.e., the process workload W_{P0} is very small as executing that function is completed in a few clock cycles. The `filter` process function can be considered to be a more coarse-grain function as some actual computation is performed on the data. Thus, the process workload W_{P1} is much larger than W_{P0}. Similar to the `read_data` function, the `write_data` process workload W_{P2} is very small as only data is written back to some memory location and not any computations on the data is performed. The process workload does not include the time required to read/write the data before/after executing the process function.

Definition 5 *A **process iteration** of a process P_i is defined as a single execution of the process function, where first all input data is read from incoming FIFO channels (i.e., the read phase), the process function is executed (i.e., the execute phase), and subsequently all output data is written to outgoing FIFO channels (i.e., the write phase).*

All iterations of a process are described by a process iteration domain.

Definition 6 *The **process iteration domain** of a process P_i, denoted by D_{P_i}, is defined as all process iterations of process P_i and is described by a set of equalities and inequalities, i.e., a polytope.*

Thus, the process iteration domain is described with a polytope as we have discussed in Section 2.1. For process $P1$, for example, the process iteration domain is defined as $D_{P1} = \{(i, j) \in \mathbb{Z}^2 \mid 0 \leq i \leq 99 \wedge 2i \leq j \leq -2i + 396\}$. This corresponds to the control part, i.e., the two for-loops, of process $P1$ as can be seen in Figure 2.4.

Definition 7 *The **process iteration domain size**, denoted by $|D_{P_i}|$, represents the total number of iterations of process P_i.*

The process iteration domain size is obtained by counting the number of integer points in a polytope, which is supported by the polyhedral library PolyLib [98] as also indicated in Section 2.2. Calculating a process iteration domain sizes, is obviously the first prerequisite to estimate the total execution time of a process. It is used to evaluate the process splitting transformation and is further discussed in Chapter 3.

Finally, we give 3 definitions that are related to the communication in polyhedral process networks. First, we consider input port domains that implement the control to read data from FIFO channels, then we define a mapping function that specifies an iteration point where data is produced, and finally we define an output port domain that specifies a set of points that generate data for a particular input port.

Definition 8 *The n-th **input port domain** of process P_i, denoted by $IP^n_{P_i}$, is defined as a subset of the process iteration domain where data is read from the n-th incoming FIFO channel, i.e., $IP^n_{P_i} \subseteq D_{P_i}$.*

Consider process $P1$ in Figure 2.4, which has one input port domain IP^1_{P1}. This input port domain is read at each process iteration, which means that the input port domain contains the same points as the process iteration domain, i.e., $IP^1_{P1} = D_{P1} = \{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i \leq 99 \wedge 2i \leq j \leq -2i + 396\}$. But when we look at process $P2$, for example, then we see that the input data is sometimes read from IP^1_{P2} and in other cases from IP^2_{P2}, i.e. they are a subset of the process iteration domain. To be more specific, we see that $IP^1_{P2} = \{(i,j) \in \mathbb{Z}^2 \mid i + j - 299 \geq 0\} \cap D_{P_2}$, and that $IP^2_{P2} = \{(i,j) \in \mathbb{Z}^2 \mid -i + j + 298 \geq 0\} \cap D_{P_2}$, where $D_{P_2} = \{(i,j) \in \mathbb{Z}^2 \mid 49 \leq i \leq 149 \ \wedge \ i + 101 \leq j \leq 250\}$. Note that these two input ports are mutually exclusive. The reason is that the `write_data` process function has only one input argument and process $P2$ needs only one input token per process iteration from one of these two input ports.

In a similar way, we define an output port domain which represents the process iterations for which writing data to a particular FIFO channel occurs.

Definition 9 *The n-th **output port domain** of a process P_i, denote by $OP^n_{P_k}$, is defined as the subset where data is written to the n-th outgoing FIFO channel, i.e., $OP^n_{P_k} \subseteq D_{P_k}$.*

When we first consider process $P1$ from Figure 2.4 again, we see that it has one output port, which is active at each process iteration, i.e., $OP^1_{P1} = D_{P1}$. A more complicated example is the first output port domain OP^1_{P0} of process $P0$. It can be seen that it is active only at particular iterations, i.e., $OP^1_{PO} = \{(i,j) \in \mathbb{Z}^2 \mid j/4 - i/2 \geq 0 \wedge -j/4 - i/2 + 99 \geq 0\} \cap D_{P_0}$, where $D_{P_0} = \{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i \leq 200 \ \wedge \ 0 \leq j \leq 400\}$. The reason that divisions of j and i by 4 and 2 appear in the constraints, is the result of consumer process $P1$ reading data in a particular pattern, i.e., array a is accessed with $2 * i$ and $4 * j$, see `line 8` in Figure 2.2.

We have defined input and output port domains that specify at which process iterations data is read/written. Thus, producer/consumer pairs of processes are connected via the output port domain of the producer with the input port domain of the consumer. Besides the port domains that specify when data is read/written, there is mapping function that specifies the relation between the input and output port domains, i.e., we define an affine mapping function M which maps the consumer process iterations to the producer iterations where the data is produced:

Definition 10 *We define an affine **mapping function** M^k as a function that maps the process iteration points from the k-th input port domain of a consumer process P_i to the process iteration points of the corresponding producer process P_j, i.e., $OP^l_{P_j} = M^k(IP^k_{P_i})$.*

An example is given in Figure 2.3. Let us consider process iteration $(1,3)$ of the

consumer process that executes function `filter` as shown in Figure 2.3 B). Since data is read from $a[2 * i][4 * j]$, process iteration $(1, 3)$ reads data that is produced at iteration $(2 * 1,\ 4 * 3) = (2, 12)$. This point is marked in the producer iteration domain as shown in Figure 2.3 A). Thus, we have a mapping function $M : \begin{pmatrix} i_p \\ j_p \end{pmatrix} = [2\ 4] \begin{pmatrix} i_c \\ j_c \end{pmatrix}$, where $\begin{pmatrix} i_p \\ j_p \end{pmatrix} \in OP^1_{P_0}$ and $\begin{pmatrix} i_c \\ j_c \end{pmatrix} \in IP^1_{P_1}$. This mapping function is also shown in Figure 2.3, and maps consumer iteration $(i_c = 1, j_c = 3)$ to its corresponding producer iteration $(i_p = 2, j_p = 12)$.

2.6 Validity of Transformations

In this section, we briefly review the validity of the process transformations presented in Chapters 3, 4, and 5. That is, we indicate that we can always apply the process splitting and merging transformations in a valid way for any given PPN. We first look at the validity of statement reordering transformations for sequential input programs, because the same constraints apply for process transformations in PPNs. Therefore, we discuss the different types of data dependencies that can exist between program statements that should be respected when the program code is transformed, which ensures that the transformed program code is input/output equivalent with the original program code.

As data dependence analysis is such a crucial step in program transformations, it is extensively researched and discussed in the literature [2, 6, 43, 57]. The goal of data dependence analysis is to find dependent program statements that read/write data from/to the same memory location. Three different data dependence relations can be identified. A flow or *true* dependence exist between two program statements A and B when A produces data that is read by B. This is denoted by $A\ \delta^f\ B$. The two other dependence relations are *anti* and *output* dependencies. In case of an anti-dependence, data is first read by a statement A and then written by statement B, which is denoted by $A\ \delta^a\ B$. An output-dependence exists when two statements A and B write to the same memory location, which is denoted by $A\ \delta^o\ B$. These data dependencies can also exists between different executions of statements within a loop-nest. If that is the case, then we say that the dependence is *loop-carried*.

As already mentioned, the data dependence information is used in optimizations and transformations to ensure correct behavior of the transformed program code. Transforming the program code is valid as long as the data dependencies are not changed. In our analysis to derive PPNs from static affine nested loop programs, we use *exact* array data flow analysis [25, 71]. This means that dependencies between statements are represented by exact dependency relations in the form of an affine combination of loop iterators and program parameters. Thus, the dependencies are

not abstracted with, for example, direction or distance vectors. An example of an exact dependence relation is the mapping function shown in Figure 2.3. It maps an iteration point of the consumer iteration space to an iteration point of the consumer where the data is produced. Taking into account these exact data dependence relations between consumer and producer statements, makes it possible to apply the process splitting and merging transformations in a valid way for any given PPN. That is, for the splitting transformation, more processes are introduced and the dependencies are recalculated to ensure that the processes communicate data in the proper way. For the process merging transformation, the executions of different processes are merged into a single process. The polyhedra that describe the process iterations of different processes, are merged using the work and code generator described in [7]. The merging is done pairwise for two given processes and the validity is checked using the exact dependence relations as described in [92, 94].

Chapter 3

Process Splitting Transformations

In this chapter, we present an approach how the process splitting transformation, introduced in Chapter 1, can be applied to transform a Polyhedral Process Network in order to select and obtain the best performance results from different splitting alternatives. Recall that the Polyhedral Process Network (PPN) model of computation is used as a programming model in the Daedalus framework [62] to help with the difficult task of programming and mapping applications onto Multi-Processor Systems on Chip. PPNs are automatically derived from sequential nested-loop programs by using the `pn` compiler [95] as we have illustrated with an example in Chapter 2. In the derived parallel PPN specification, the following partitioning strategy is used: each process in the PPN corresponds to a function call statement in the sequential program.

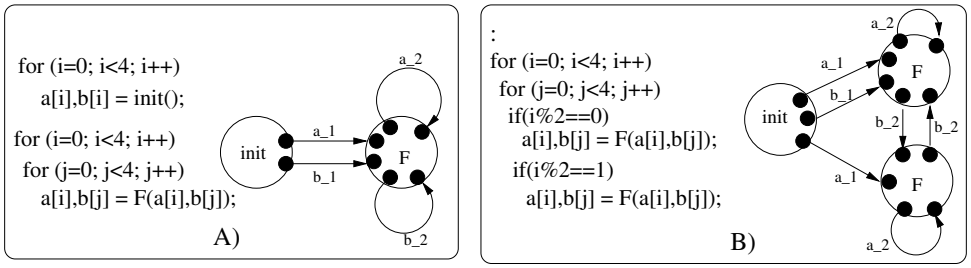

Figure 3.1: Polyhedral Process Networks

Figure 3.1 A) shows a PPN consisting of two process with 4 FIFO channels, and also the nested loop program from which this PPN is derived. Deriving the network using the `pn` partitioning strategy, as described above, does not necessarily lead to

optimal performance results as the network may not be well balanced. Therefore, process partitioning transformations can distribute the workload of a single process over multiple processes to better balance the network. We can achieve this, for example, as shown in Figure 3.1 B). The function call statement `F` is duplicated and assigned to odd and even iterations of the outer loop iterator. The corresponding network has now two processes executing the `F` function resulting in a more balanced network. In [79], a number of algorithmic transformations have been presented which a designer can apply on the source code to balance the network. However, no hints are given to the designer when a particular transformation can be applied to minimize, for example, the execution time. So, a number of transformations have been defined, but the designer does not know when to apply which transformation. In our motivating examples (Section 3.2) we show that it is not straightforward to select the best transformation for the best performance results. In order to select the best partitioning transformation, the different alternatives must be evaluated and metrics are required to do so. This chapter, therefore, deals with:

1. Definition of evaluation metrics;

2. Calculation of the metric values using an analytical framework;

3. A compile-time evaluation approach to select a particular transformation based on the metric values.

We show results for 3 different applications with different properties mapped onto the Cell processor [39] and the ESPAM platform prototyped on a Xilinx Vertex 2 FPGA [61].

3.1 Process Splitting: Definitions, Notations, and Examples

First, it is important to note that process splitting is a general term referring to transformations duplicating program code to obtain more processes. In Figure 3.1 B), we have shown one example of process splitting, but there are many other possibilities to duplicate the program code. In [79], a number of parametric transformations have been presented that can be used to split up processes. Two of these splitting transformations are the modulo unfolding and the plane-cut transformation:

Notation 1: we refer to the **modulo unfolding** transformation as `unfold(I,U)`, where parameters `I` and `U` are respectively the iteration vector of the function of a process and the vector of unfolding factors for each loop iterator.

Notation 2: we refer to the **plane-cut** transformation as `planecut(I,P)` where parameter `I` is the iteration vector and parameter `P` is a set of affine hyperplanes (see Section 2.1).

Definition 11 *A **process partition**, or partition in short, is a new instance of an original process that is created by applying a process splitting transformation* `unfold` *or* `planecut`*. Thus, the different process partitions execute the same function, possibly, in parallel.*

In the remainder of this chapter, we focus on the unfolding and plane-cutting transformations. In [79], some more (algorithmic) transformation techniques have been presented. An example is the skewing transformation, which re-times the process iterations. However, only the unfolding and plane-cutting split-up a process, i.e., assign process iterations to different partitions. To illustrate the difference in the unfolding and plane-cutting transformations, we consider the example shown in Figure 3.2.

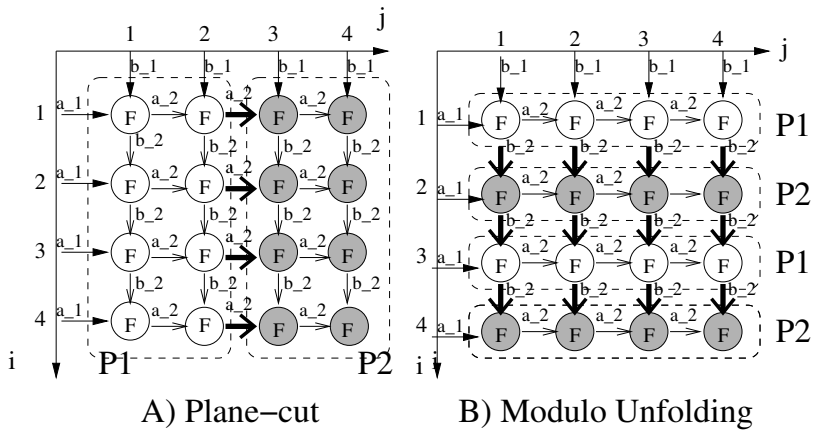

Figure 3.2: Examples of Process Splitting Transformations

Figure 3.2 shows the dependency graph of the program depicted in Figure 3.1 A) and is characterized by a two dimensional process iteration domain and horizontal and vertical dependencies. Loop iterator i corresponds to the outer loop and iterator j corresponds to the inner loop such that the lexigraphical order is from top to bottom and from left to right. The arrows denote dependencies. The dependency graphs are annotated with two possible partitionings which are the result of applying transformations. The plane-cut transformation `planecut({i,j},{j=2})` has been applied in Figure 3.2 A) such that partition $P1$ executes all points with $j \leq 2$ (the white iteration points) and $P2$ executes all points with $j \geq 3$ (grey points). Another partitioning

is shown in Figure 3.2 B) which corresponds to the modulo unfolding transformation presented in Figure 3.1 B) and is formally specified as `unfold({i,j},{2,0})`. All even i iterations are assigned to $P2$, and all odd i iteration points are assigned to $P1$. The plane-cut and unfolding transformations and partitions differ in terms of the amount of inter-process communication (as indicated with the bold arrows) and initial delay of the partitions. In the plane cutting example in Figure 3.2 A), inter-process communication occurs 4 times and the first iteration point of $P2$, i.e., point $(1, 3)$, must wait for 2 iterations $(1, 1)$ and $(1, 2)$ of $P1$ before it can start executing. In the modulo unfolding partitioning in Figure 3.2 B), $P2$ starts after 1 iteration of $P1$, but then 12 inter-process data transfers are performed. This makes clear that different transformations lead to different behavior of the partitioned processes.

To give a more elaborate example of the internal structure of processes, we consider the processes in Figure 3.3. It shows one of the unfolded `F` processes and source process `init` from Figure 3.1 B).

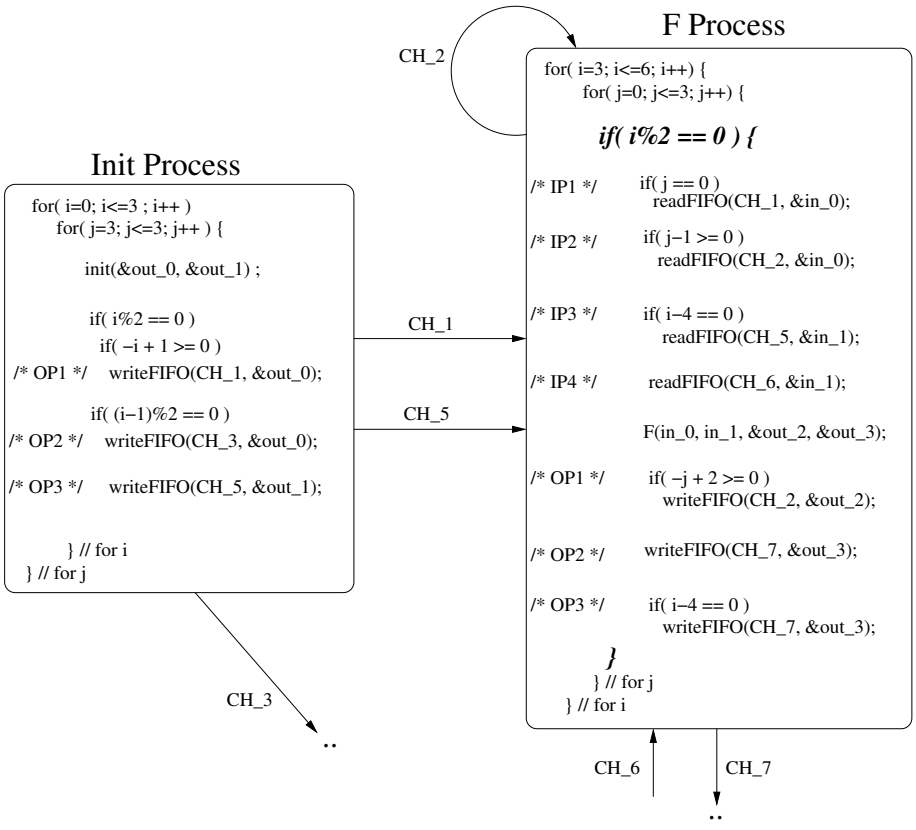

Figure 3.3: Structure of Unfolded Process F

It can be seen that process F has one so called *self-channel* or *self-edge*, i.e., channel CH_2 from/to process F. Self-channels are important in determining how to split-up processes as will be discussed later. Furthermore, it can be seen that the splitting transformation introduces a control statement inside the process (i.e., the bold modulo statement) to partition and ensure that an iteration point is executed by one partition only, and not by two partitions for example.

3.2 Challenges of Applying the Process Splitting Transformation

In this section we show performance results for two applications. These two motivating examples show that the question which transformation to apply contains many subtle parts, based on the interplay of many factors which may not be evident at first sight. This makes it difficult to select the proper process splitting transformation.

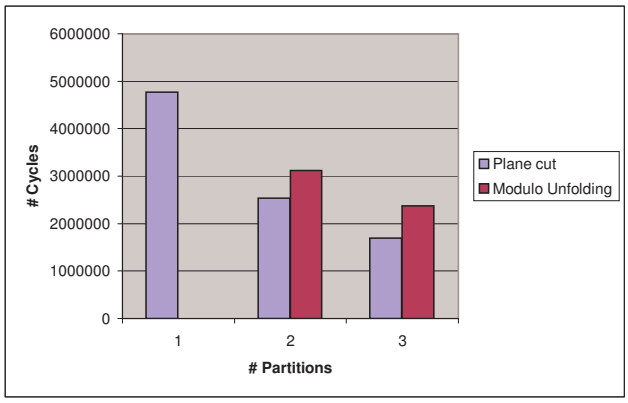

Figure 3.4: Results of Different Splittings on the ESPAM platform

The first bar in Figure 3.4 corresponds to the performance result for the unmodified application and its derived PPN in Figure 3.1 A) mapped on the ESPAM platform [60, 61]. The application is executed in 4.8 million cycles. Then, the network is balanced by applying the modulo unfolding and plane-cut transformations and thus two partitions are created for function call statement F. The second bar corresponds to the plane cut transformation and the third bar to the two times unfolded version shown in Figure 3.1 B). The fourth and fifth bars display results for creating three partitions using the same transformations. It can be seen that the plane-cut transformation is better than the modulo unfolding: 2.5 million vs. 3.1 million cycles for creating 2 partitions and 1.8 million vs. 2.2 million cycles for creating 3 partitions.

These results are surprising as the initial producer delay for the plane-cut is larger than for the modulo unfolding, but still the plane-cut transformation leads to better performance results. In this example, the number of intra and inter-process communication is not important as the cost for intra and inter-process communication are the same on the ESPAM platform. Therefore, the measured performance results can only be explained by a non-constant cost for the communication when different transformations are applied, which involves a FIFO read/write primitive and a control part when to read/write (the function workload cannot change and is constant). We observe that by introducing modulo statements, the communication (the control part) becomes more costly as the modulo expressions will appear in the definitions of the input/output ports. An example is the bold modulo statement in the `F` process in Figure 3.3. The modulo statement is introduced as a result of the transformation and is evaluated every iteration. In general, the if-conditions for reading/writing from/to FIFO channels are more expensive as more complex expressions must be evaluated.

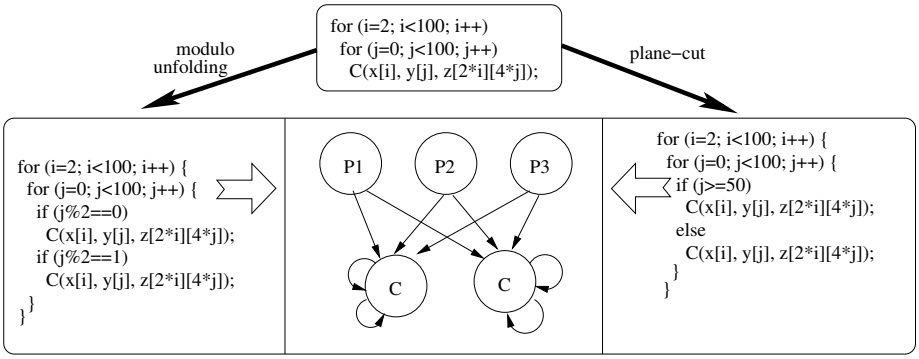

Figure 3.5: Modulo unfolding vs. Plane-cut

Another application is shown in Figure 3.5. The initial application source-code at the top (the producer processes $P1$, $P2$, and $P3$ are omitted for the sake of brevity) is transformed by unfolding the inner loop two times: `unfold({i,j},{0,2})`, and a plane-cut on the inner loop: `planecut({i,j},{j=50})`. The PPN is topologically the same for both transformations, but internally the processes are different. In Figure 3.6, the performance results for the initial network and both transformed networks are shown. The first bar corresponds to the initial network and it shows that the application requires 22 million cycles to finish its execution. The second and third bar correspond to the plane-cut and modulo unfolding and require, respectively, 17 million and 15 million cycles. We observe that the plane-cut method is slightly worse compared to the modulo unfolding. Although there are no dependencies between the two processes executing function `C` (see Figure 3.5), the consumer processes `C` in

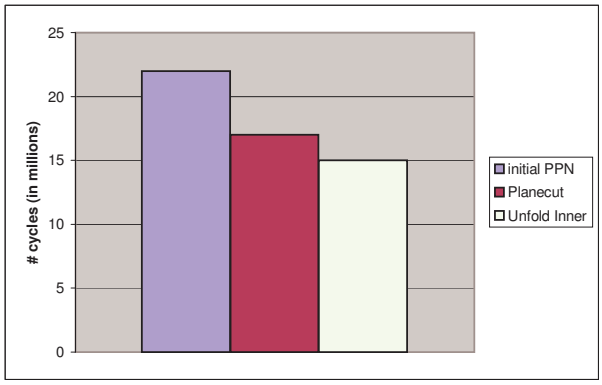

Figure 3.6: Measured Performance Results of the PPNs in Figure 3.5 on ESPAM

the plane-cut example must wait more iterations before the producer processes generate the first data compared to the modulo unfolding example (this is discussed in detail in Section 3.3.2 and in the case-studies in Section 3.5). From this example we learn that it is not enough to consider only inter-process communication and initial delay caused by other partitions, but also the delay caused by all other producers. In Section 3.3, we define the metrics that should be taken into account in applying and evaluating different transformations.

Problem Statement

There are many possibilities to partition processes as we have shown in this section. Different partitioning strategies have a significant impact on performance results and thus selecting the best partitioning strategy is crucial in achieving the best possible results. Figure 3.4 and 3.6, for example, show that it is not straightforward to select the best partitioning candidate. The challenge is to find a compile-time solution to predict the best possible partitioning and thus minimize the execution time. Therefore, one should be able to answer the following two questions:

- Given the two parameterized transformations `unfold(I,U)` and `planecut(I,P)`, which transformation should one apply for a given process to be split-up?

- For a chosen transformation, what should the parameter values be? For the `unfold` transformation, for example, one should choose one or more loop iterators to unfold and corresponding unfolding factors.

3.3 Partitioning Metrics

A process P_i has a process iteration domain D_{P_i} and is transformed by transformation H into n disjoint partitions $H(D_{P_i}) = \{D_{P_i^1}, .., D_{P_i^n}\}$. Different partitioning transformations result in partitions with different properties and in this section we discuss six metrics we have identified to evaluate different partitionings. The metrics we discuss are *i*) computation costs, *ii*) communication costs, *iii*) initial delays, *iv*) production period, *v*) data transfers, *vi*) additional control overhead.

3.3.1 Computation and Communication Costs

In each process iteration, a function is executed as illustrated in Figure 3.3 (function `F`). The complexity of this function can vary from a simple multiply-accumulate operation in a matrix multiplication kernel to a coarse grain task such as a DCT in a JPEG encoder application. The complexity of this function contributes, among other factors, to the delay at which data is produced. In determining the total execution time of a process P_i, the workload, i.e., the **computation cost**, of a process function is taken into account and is denoted by W_{P_i} (see also Section 2.5). An accurate costs estimation is thus crucial for selecting the best possible partitioning strategy and inaccurate estimations can lead to wrong decisions. We consider the function cost as an input parameter for our algorithm that can be obtained by running the function once on the target platform. We consider the function cost to be a constant value, see Section 3.6 for a discussion on this. Besides the execution of a function, a process reads from a number of input channels to get all function input arguments at each iteration. Similarly, it writes the result to a number of output channels. The FIFO read/write primitives can be supported by hardware (e.g., the ESPAM platform), or must be supported with a software implementation (e.g., the CELL). Clearly, the **communication cost** of data communication depends on the target platform and can influence the partitioning significantly. With a software implementation of FIFOs, for example, data communication can easily become more costly than the computation itself. The ratio of computation and communication is an important metric to evaluate different partitionings. To the costs for inter-process communication we refer as C_{inter} and for intra-process communication we use C_{intra}. These are constant costs to transfer a single token from a producer to a consumer process and are obtained by checking/measuring the costs for the read/write primitives on the target platforms. The reader is referred to Section 3.6 for a more in-depth discussion on using constant values for the cost of process functions and FIFO communication.

3.3.2 Initial Delay

A partition may not directly start executing its first iteration as a result of dependencies. In that case, a producer process, or another partition, is responsible for generating the required initial data.

Definition 12 *We define the **initial delay** as the number of iterations a producer executes before it generates the first data for a partition, and we denote it by* $Y(D_{P_i^n})$ *for a partition* $D_{P_i^n}$.

For example, the second partition $P2$ in Figure 3.2 A) must wait 2 iterations for producer $P1$ before it can start its execution and in Figure 3.2 B) the second partition can start after 1 iteration. For each partition $D_{P_i^n}$ we calculate the initial delay, which may be caused by a producer process or another partition. Each partition has a number of input ports and we determine the lexicographical minimum point of each function input argument. This point corresponds to the iteration point where data is read for the first time with respect to that function argument. Figure 3.7 shows the function call statement `F` from Figure 3.3. It has two input arguments `in0` and `in1`. At different iterations, argument `in0` is read from input ports $IP1$ or $IP2$, and the second argument from input ports $IP3$ or $IP4$.

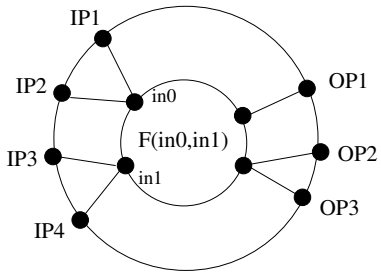

Figure 3.7: Function Input Arguments and its Delay Calculation

For each input argument, we determine the first read action by considering the lexicographical minimum point of all associated input ports. For the example above, we calculate the minimum of $IP1$ and $IP2$, and then we do the same for $IP3$ and $IP4$. In general, when there are x input arguments with y input ports associated to the first function argument and z ports to the last argument, we calculate the producer points

as follows:

$$\begin{aligned} p_1 &= M(a), \;\; where \;\; a = lexmin(\bigcup_{j=1}^{y} IP_j) \\ &\vdots \qquad\qquad\qquad \vdots \\ p_x &= M(b), \;\; where \;\; b = lexmin(\bigcup_{j=1}^{z} IP_j) \end{aligned} \tag{3.1}$$

We apply the mapping function M (see Section 2.5) of each input port to obtain all producer points p_t where $1 \leq t \leq x$. The initial data is generated at these producer iteration points, which means that the consumer is waiting for all preceding producer iteration points to receive its initial data. Now, to calculate this initial delay, the `rank` function (see Section 2.2) is applied to a producer point returning the number of preceding iterations for a given iteration point. We calculate this offset, the initial delay Y_t, for all producer points $p_t \in D_{P_t}$ of the last partition D_{C^n} as follows:

$$Y_t(D_{C^n}) = \begin{cases} rank(p_t, D_{P_t}) & if P_t \neq C^n \\ rank(p_t, D_{P_t}) + \sum_{x=0}^{n-1} Y(D_{C^x}) & otherwise \end{cases} \tag{3.2}$$

It shows that if the producer P_t and consumer C^n are different processes, then the offset is calculated based only on the number of iterations of the producer process. If the producer point belongs to the same process but to a different partition, then the delay of the preceding partitions $Y(D_{C^x})$ are taken into account. The initial time $T_{C^n}^{init}$ a consumer C^n is waiting for initial data, is determined by the slowest producer. To calculate this time, we consider all $Y_t(D_{C^n})$ values as defined above. These values are multiplied by the estimated time $T_{P_t}^{iter}$ required for one process iteration, which we define with Formula 3.9 in Section 3.4, of the corresponding producer and the maximum value is taken:

$$T_{C^n}^{init} = max_t\{Y_t(D_{C^n}) \cdot T_{P_t}^{iter}\} \tag{3.3}$$

3.3.3 Production Period

The calculation of the initial delay is not enough to accurately estimate the execution time of a partition. For example, a producer can generate data for a consumer at its first iteration, but then it may take a number of iterations before it generates new data. This illustrates that the **production period** of a producer process is another import metric.

Definition 13 *The **production period** of a process is the number of process iterations between two consecutive data productions.*

A more elaborate example is given in Figure 3.8. Both the circles and crosses denote process iteration points. The circles indicate that data is produced for a particular consumer at that point, and the crosses indicate that no data is produced. A consumer

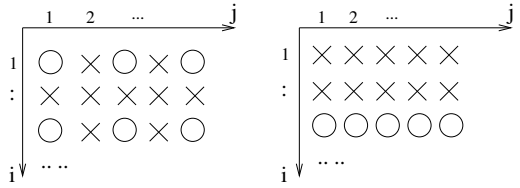

Figure 3.8: Production Period Examples

process receiving data from these two producers is waiting 1 iteration for producer $P1$ and 10 iterations for $P2$ to generate the initial data so that the consumer can start executing. After this initial delay, producer $P2$ is producing data at each iteration, while $P1$ is producing data in either 2 or 5 iterations. We define the average production period d_{P_i} as the average number of iterations that is required to generate new data by producer P_i.

Definition 14 *The **average production period**, denoted by d_{P_i}, is calculated by dividing the total number of iteration points of a producer process P_i by the total number of generated data tokens:*

$$d_{P_i} = \frac{|D_{P_i}|}{|M(IP_C)|} \tag{3.4}$$

where IP_C is the input port domain of consumer process C, M is the mapping function which is used to obtain the producer iteration points for this input port domain, and D_{P_i} is the process iteration domain of the producer process P_i.

To illustrate the production period, we consider the example in Figure 3.8 and assume the iteration domain consist of 3 rows and 5 columns. The production period is $\frac{15}{6} = 2.5$ and $\frac{15}{5} = 3$ for producer $P1$ and $P2$, respectively. The time $T_{P_i}^{period}$ required to generate new data is the average production period multiplied by the required time $T_{P_i}^{iter}$ that is needed for one process iteration of a producer:

$$T_{P_i}^{period} = d_{P_i} \cdot T_{P_i}^{iter} \tag{3.5}$$

In Section 3.4, we explain how the time $T_{P_i}^{iter}$ for a process iteration is calculated.

3.3.4 Data Transfers

Different partitionings can lead to a different number of inter- and intra-process **data transfers** which is denoted by DT. A data transfer occurs when data is read/written to/from a FIFO channel. We already considered the example in Figure 3.2 A), where the plane-cut results in $4 + 4 = 8$ data transfers (the bold arrows) from one process to the other process and 40 transfers to/from the same process. In Figure 3.2 B), the partitioning strategy results in $12 + 12 = 24$ inter-process data transfers and $12 + 12 = 24$ intra-process data transfers. The number of data transfers is important. For the examples in Figure 3.2, it is clear that the plane-cut is better than the modulo unfolding if inter-process communication is costly, because there are only 8 inter-process communication compared to 24 transfers for the modulo unfolding transformation.

For a process P_i, we calculate the number of intra and inter process data transfers by considering all input/output port domains of this process and check, in the polyhedral process network, if the corresponding output/input port domains belong to the same process P_i. If this is the case, then we classify the input/output port and corresponding channel as intra-process communication, and inter-process communication otherwise. We compute the number of intra and inter process data transfers as follows:

$$\begin{aligned} DT^{Rd}_{inter} &= \sum_i |M^i(IP_i)| \\ DT^{Rd}_{intra} &= \sum_j |M^j(IP_j)| \\ DT^{Wr}_{inter} &= \sum_k |OP_k| \\ DT^{Wr}_{intra} &= \sum_l |OP_l| \end{aligned} \tag{3.6}$$

Equation 3.6 shows that the size of all input port domains determine the total number of intra/inter process data transfers for data that is read. In a similar way, we define data that is written as inter/intra process data transfers by considering the output port domains.

3.3.5 Additional Control Overhead

The process partitioning transformations are equivalent to source-code transformations as already indicated and also described in [79]. In Figure 3.1 B), a function call

statement is duplicated and assigned to even/odd iterations of the outer loop iterator. We have shown in Figure 3.3, that the control for reading/writing from/to FIFO channels becomes more complex as a result of the transformation. This **additional control overhead** can change the computation-communication ratio. If this is not taken into account, then execution times cannot be accurately estimated leading to incorrect predictions which transformation is better. It is very difficult however, to predict this additional control overhead as the nesting level of the if-statements are different for each application and transformation. As a result, costs for the control overhead cannot be accurately estimated at compile-time. Furthermore, it is not feasible to ask the designer to provide the costs as there may be many ports to be checked. However, there are cases when the control overhead can be safely ignored. The additional control can only change significantly the computation-communication ratio if the computational process workload is small. With coarse grain tasks, the additional control will not change significantly this ratio and it is not necessary to take this into account in the cost function. Another approach to avoid the additional control overhead is a manual modification of the generated code. In case of the modulo unfolding for example, the introduced modulo statements can be manually removed from the generated code by adjusting the loop step-size and corresponding conditions in the input/output port domains. The conditions for the plane-cut are usually much simpler and thus can be ignored in many cases. In our approach we consider examples with compute intensive tasks and change manually the generated code to remove the additional control overhead.

3.4 Compile-time Selection of Splitting Transformation

In this section, we present a solution approach and analytical model to predict, at compile-time, which transformation should be applied to obtain the best performance results. To compare different transformations, we estimate the execution time of a transformation.

Definition 15 *The **execution time of a transformation**, denoted by $T_{transformation}$, is defined as the estimated total execution time, i.e., the time required to execute all process iterations of the last processes partition which is obtained after applying the process splitting transformation.*

One solution to evaluate the different splitting transformations is simply to evaluate all possibilities. This is possible, because we define in this section a compile-time model that allows a designer to estimate the execution of a transformation. However, evaluating all possibilities is not a very attractive solution as the number of possibilities to check and evaluate can be large. Here we present an approach that does

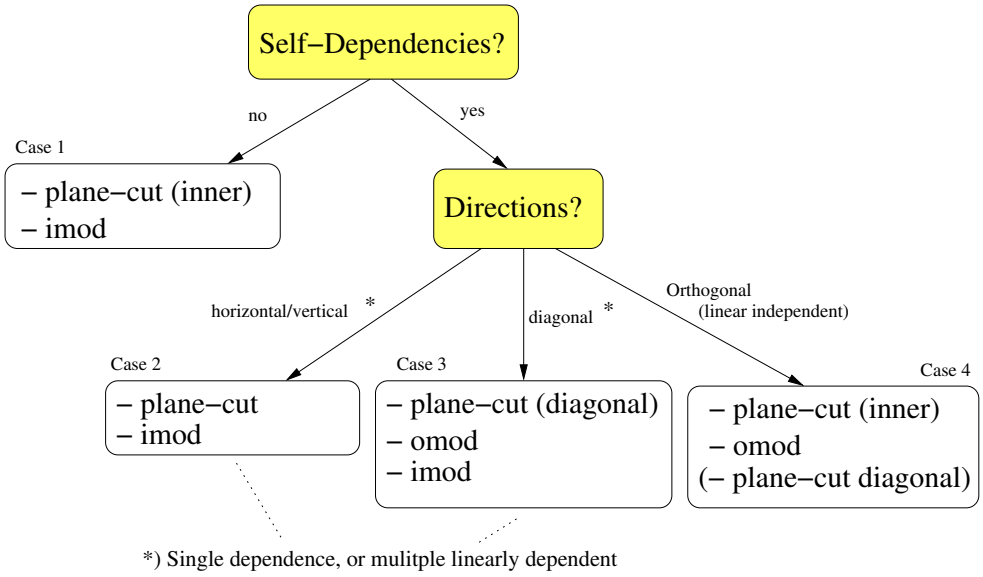

Figure 3.9: Decision Tree

not require to evaluate all possible transformations, i.e., some transformations can be excluded beforehand. To achieve this, the decision to evaluate and apply a particular transformation for a given process is made using the decision tree shown in Figure 3.9. The transformations listed in the leaf nodes of the decision tree are considered, the corresponding execution times $T_{transformation}$ are calculated using the analytical model, and the minimum value is selected. There are 5 possibilities to apply a process splitting transformation: a horizontal, vertical, diagonal plane-cut, and modulo unfolding on the inner- and outermost loop. Thus, the advantage of using the decision tree is that some possibilities do not need to be evaluated.

To balance the network, the designer starts with selecting the most computationally intensive process which will be split-up using the unfolding or plane-cut transformation. Following the decision tree, inter-process communication is avoided as much as possible by analyzing the self-dependencies of that process. If there are no self-dependencies at all before the partitioning, then a partitioning cannot introduce inter-process communication. If a single self-dependency exists, then inter-process communication can be introduced by a transformation if the transformation is not chosen carefully. Thus, the idea of the decision-tree is to avoid inter-process communication as much as possible by creating partitions that ”follow the directions” of these dependencies. In other words, producer-consumer pairs are clustered into

the same partition, and not assigned to different partitions, such that the communication remains local. For example, if there exists a single horizontal dependency in a 2-dimensional process iteration domain, then vertical partitions will introduce inter-process communication, while horizontal partitions will not. For multiple dependencies that are orthogonal to each other, a partitioning with inter-process communication cannot be avoided. These cases are captured in the decision tree shown in Figure 3.9 and we discuss each of these cases in more detail. Please note that we illustrate below our approach with 2-dimensional process iteration domains, while the approach also works for processes with n-dimensional domains where $n > 2$. This is shown with a case-study in Section 3.5.2. For higher dimensional iteration domains (i.e., $n > 2$), the principle of the decision tree in Figure 3.9 remains the same, only the space spanned by the dependencies are different. Consider, for example, `case 2` of the decision tree shown in Figure 3.9. A horizontal dependency in a 2-dimensional domain is a line, while in the a 3-dimensional domain it can also be a plane. Thus, independent partitions can be created as long as the dependencies do not span the entire iteration domain.

Case 1

The first branch in the tree checks if there are any self-dependencies. If not, then only the plane-cut and modulo unfolding on the inner most loop iterator (indicated by *imod* in Figure 3.9) are compared. Thus, `case 1` is the easiest case because inter-process communication cannot be introduced by the splitting transformation since the process does not have any self-dependencies. In this case, the most important factor is the initial delay which we illustrate with the example shown in Figure 3.10.

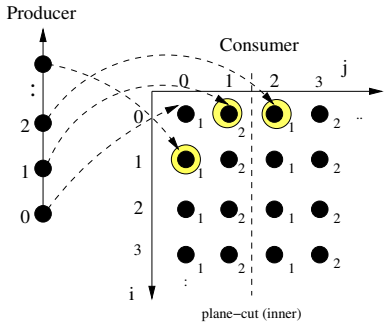

Figure 3.10: Decision Tree: Case 1

Recall from Section 3.3.2 , that the initial delay represents the number of process iterations, which a producer process needs to execute before it generates the first

input data for a consumer process. This initial delay is the reason that only the plane-cutting on the inner-loop j and the modulo unfolding on the inner-loop are compared in `case 1`. For other transformations, such as modulo unfolding on the outer-loop i, the initial delay will always be larger. To illustrate this, take into account that the lexicographical order of the Consumer process iterations is from from top to bottom and from left to right, in Figure 3.10. Process iteration $(i = 0, j = 2)$ is, therefore, the *first* process iteration to be executed by the *second* process partition after applying the plane-cut transformation. Similarly, process iteration $(i = 1, j = 0)$ is the first iteration to be executed by the second partition after applying modulo unfolding on the outer-loop i, and iteration $(i = 0, j = 1)$ is the first for the unfolding transformation on the inner loop j. When data is produced in the same order as it is consumed, then it should be clear that iteration $(1, 0)$ must always wait more iterations than iterations $(0, 1)$ and $(0, 2)$ before its input data is generated by the producer. Hence, unfolding on the outer-loop i is not considered. The plane-cut is the preferred transformation to apply, because the introduced overhead of the transformation is less than modulo unfolding on the inner-loop j. However, the initial delay can be much larger and therefore the plane-cut and modulo unfolding (inner) are the two transformations that are evaluated and compared at compile-time.

Case 2 & Case 3

In case the selected process has self-dependencies, then the dependency directions are analyzed. We have identified 3 different cases as shown in Figure 3.9. For `case 2` and `case 3`, inter-process communication can still be avoided: i.e., when the process has a horizontal/vertical self-dependency, or a diagonal self-dependency. For these cases, the dependent iterations are assigned to the same partition and the communication remains local to each partition.

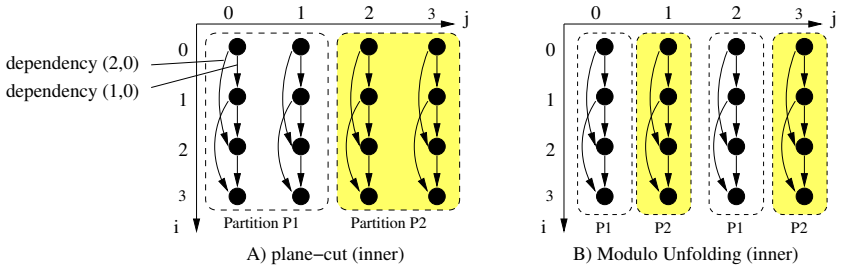

Figure 3.11: Decision Tree: Case 2

The reason to consider a single self-dependency and multiple linearly dependent

self-dependencies as one case, is because inter-process communication can be avoided. Therefore, it is crucial that the multiple self-dependencies are linearly dependent, which is illustrated with the example in Figure 3.11. Intuitively, the idea is to split-up a process in such a way that the plane-cut or modulo unfolding follows "the same direction" as the linearly dependent self-dependencies. Figure 3.11 shows such an example with two different dependencies: one in the direction of $(i+1, j+0)$, or in short $(1, 0)$, and the other one in the direction of $(2, 0)$. These dependencies allow a partitioning that creates independent partitions, with the dependent iterations assigned to a same partition. This is illustrated with the plane-cut transformation shown in Figure 3.11 A), and the modulo unfolding on the inner loop j shown in Figure 3.11 B). It is clear that for these cases there is no difference if there is only one self-dependency, or multiple linearly dependent: the partitions will be free of any inter-process communication. In `case 2`, the modulo unfolding on the outer loop is not considered because the initial delay will always be significantly larger than the other two partitionings and therefore it will never be better. The best transformation is obtained by evaluating the execution times of the plane-cut and modulo unfolding on the inner loop iterator. While Figure 3.11 shows two processes with *vertical* self-dependencies, another possibility are *horizontal* self-dependencies, i.e., in the direction $(i+0, j+1)$. We do not further elaborate on this case as the analysis is the same as for the vertical dependencies.

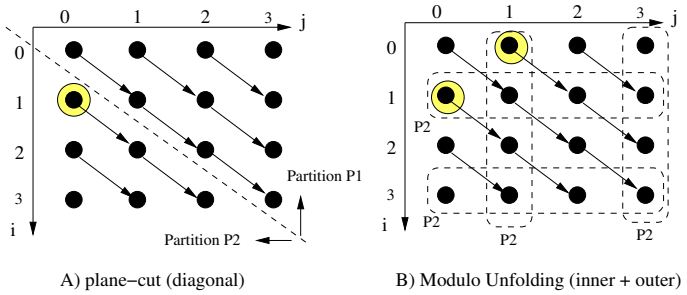

Figure 3.12: Decision Tree: Case 3

For diagonal self-dependencies, i.e., `case 3` of the decision tree, different splitting transformations should be evaluated compared to the horizontal/vertical dependencies. Figure 3.12 shows an example of a diagonal self-dependency. In this case, it is clear that a diagonal plane-cut results in partitions that do not need to communicate, as shown in Figure 3.12 A). On the other hand, the initial delay can be quite large. The *first* iteration of the *second* partition corresponds to iteration $(1, 0)$. If a producer processes first generates data for all points on the first line with $i = 0$, then the second partition cannot directly start executing. In that case, a modulo unfolding on the

inner/outer loop as shown in Figure 3.12, will have much smaller initial delays: the first iterations of the second partition correspond to iterations $(0, 1)$ and $(1, 0)$ for the modulo unfolding on the inner and outer loop, respectively. Note that in this example, the first iterations of the second partition for the diagonal plane-cut and unfolding on the outermost loop i are the same, i.e., iteration $(1, 0)$, but this does not need to be the case in general. Although the modulo unfolding can have a smaller initial delay than the plane-cut transformation, the different partitions must synchronize and communicate data, which is not the case for the plane-cut. The transformation that results in the best performance results, therefore, depends on the costs for FIFO communication and the process workload, and thus the plane-cut and modulo unfolding transformations should be evaluated and compared.

Case 4

When a process has multiple linearly independent self-dependencies, it is not possible to create partitions without any inter-process communication. This corresponds to `case 4` of the decision tree. For example, when a process with a 2-dimensional process iteration domain and 2 self-dependencies that are perpendicular, i.e., they are orthogonal as shown in Figure 3.13 A), any process splitting transformation will result in inter-process communication between the different partitions.

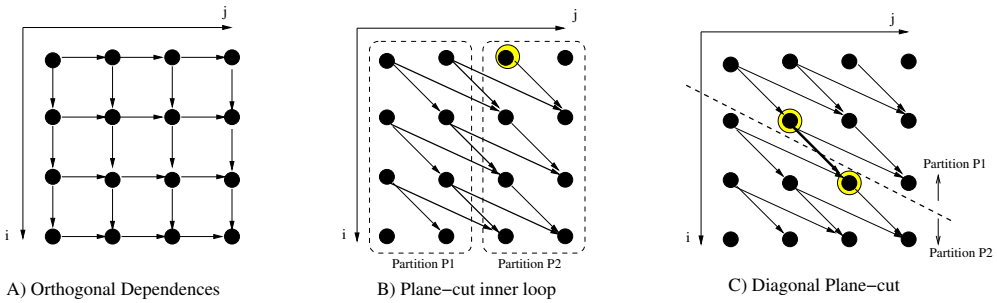

Figure 3.13: Decision Tree, Case 4: Linear Independent Self-Dependencies

In Figure 3.13 A), a 2-dimensional process iteration domain is shown where the arrows denote dependencies, i.e., the dependencies are orthogonal to each other. The lexicographical order of the iteration points is from top to bottom and from left to right, (i.e., i is the outer loop and j the inner loop). Thus, for dependencies that are orthogonal to each other, unfolding on the inner most loop is not considered because this transformation leads to sequential execution of the partitions. In addition to the unfolding on the inner most loop, we also do not consider the diagonal plane-cut. The reason is that the delay for the iteration points at the diagonal of the second partition,

is always much larger than the initial delay for the plane-cut on the inner loop and the modulo unfolding on the outer loop. Therefore, the plane-cut transformation on the inner loop must be compared with unfolding the outer loop (refered to as *omod*).

While orthogonal dependencies are one example of linearly independent dependencies, there are many other possibilities for two dependencies to be linearly independent. An example is shown in Figure 3.13 B). Although the dependencies are not orthogonal, they are linearly independent and a plane-cut on the inner loop $j = 2$, as shown in Figure 3.13 B), would result in 9 inter-process communications. A diagonal plane-cut, however, as shown in Figure 3.13 C), would result in only 1 inter-process communication. Furthermore, we see that for both plane-cuts, that there is no initial delay. However, there is a small delay for the first synchronization point in the diagonal plane-cut, i.e., the two highlighted iteration points in Figure 3.13 C). That is, the synchronization point is the 5th iteration point of *partition 1*, and the consumer point is the 4th iteration of *partition 2*. This means that *partition 2* is waiting 1 iteration for *partition 1* to receive its data, which does not occur in the plane-cut on the inner loop. Despite this small delay, the diagonal plane-cut can possibly be better than a plane-cut on the inner loop, depending on the costs for communication and the workload of the process function, because it has less inter-process communications. Therefore, the diagonal plane-cut, plane-cut on the inner loop, and modulo unfolding should be evaluated and compared.

Calculating the Execution Time of a Transformation

Now we present how the execution time of a transformation can be estimated and thus how transformations can be evaluated and compared. The execution time of a transformation is calculated by summing the initial time $T^{init}_{P^n_i}$ the last partition is waiting for data and the time $T^{exec}_{P^n_i}$ required for executing that last partition P^n_i:

$$T_{transformation} = T^{init}_{P^n_i} + T^{exec}_{P^n_i} \tag{3.7}$$

The initial delay $T^{init}_{P^n_i}$ is defined in Formula (3.3) and represents the maximum time before the first initial data is produced by producer processes. The execution time $T^{exec}_{P^n_i}$ for a partitioning is defined and calculated as follows:

$$T^{exec}_{P^n_i} = |D_{P^n_i}| \cdot max(T_{avg\_period}, T^{iter}_{P^n_i}) \tag{3.8}$$

In this formula, $T^{iter}_{P^n_i}$ is the execution time that is required to execute a single iteration of the last partition. The costs for executing a single process iteration includes reading all the process function input arguments, execution of the process function, and writing of the result(s) to the output port(s). If this time is less than the time

required by a producer to generate data, then the execution of an iteration is dominated by the producer process. For this reason, we check if $T_{avg\_period} \geq T^{iter}_{P^n_i}$ and use this time, if necessary, multiplied by the number of process iteration points in the domain to calculate the execution time $T^{exec}_{P^n_i}$. The time required to execute a single iteration $T^{iter}_{P^n_i}$ in this formula is approximated by considering the workload $W_{P^n_i}$ of the partition P^n_i, and the average time for inter- and intra-process data transfers:

$$T^{iter}_{P^n_i} = W_{P^n_i} + \frac{DT^{Rd}_{inter}}{|D_{P^n_i}|} \cdot C^{Rd}_{inter} + \frac{DT^{Rd}_{intra}}{|D_{P^n_i}|} \cdot C^{Rd}_{intra} + \\ + \frac{DT^{Wr}_{inter}}{|D_{P^n_i}|} \cdot C^{Wr}_{inter} + \frac{DT^{Wr}_{intra}}{|D_{P^n_i}|} \cdot C^{Wr}_{intra} \tag{3.9}$$

where C^{Rd}_{inter}, C^{Rd}_{intra}, C^{Wr}_{inter}, C^{Wr}_{intra} are the costs for reading and writing data for inter and intra-process communication as defined in Section 3.3.1. DT^{Rd}_{inter}, DT^{Rd}_{intra}, DT^{Wr}_{inter}, and DT^{Wr}_{intra} are, respectively, the total number of inter and intra process data transfers as defined in Formula 3.6.

If the computation of a process is not dominated by its own execution $T^{iter}_{P^n_i}$, but by the producer(s) and its large production period(s), then the average period T_{avg_period} from the producers is used to calculate the execution time of a single iteration. T_{avg_period} in Formula (3.8) corresponds to the execution time a partition is waiting for data considering its producer process. The average time is approximated taking into account the number of tokens transfered between a producer-partition pair with respect to the total number of data transfers. This number is used as a weight for the production period of a producer. The average period T_{avg_period} is calculated by summing the production period multiplied by the weight factor for all n producers:

$$T_{avg\_period} = \sum_{i=1}^{n} T^{period}_{P_i} \cdot \frac{|OP_i|}{\sum_{j=1}^{n} |OP_j|} \tag{3.10}$$

where $T^{period}_{P_i}$ corresponds to the production period as defined in Formula (3.5).

3.5 Case-Studies

In this section we present 3 different applications. The first application is an application with a single diagonal dependency for the compute process, the second application is a matrix multiplication, and the third is an application with four different producers and (initial) delays. We map the applications on the ESPAM platform [60,61] prototyped on a Xilinx Virtex 2 FPGA and the CELL processor [34]. For programming the Xilinx Virtex 2 Pro FPGA, we use the Daedalus tool-flow [62] to implement

a multi-processor system on chip. Each process from the network is mapped onto a MicroBlaze softcore processor and the processes are point-to-point connected. The FIFO channels are implemented using FSL channel components provided by Xilinx. We measured that writing/reading to/from FIFOs is completed in just 10 clock cycles. The second platform is the CELL BE processor and we use the code generator presented in [58] to map applications on the Cell processor of a `Playstation 3` console. We map the compute processes to different SPEs and source/sink processes to the PPU. The FIFO channels are implemented in local memories of both the producer and consumer process. Synchronization with signals/mailboxes ensures mutual exclusive access, which makes the read/write primitives much more expensive compared to the ESPAM platform. In these case-studies, we will not exhaustively explore all cases and transformations. Instead, we focus on `case 3` and `case 4` of the decision tree shown in Figure 3.9, because they are the most interesting from the dependencies point of view. For these two cases, we experiment with different intial delays, production periods, and inter-process communication. For each experiment, we show our approach applied on different transformations to verify that our model correctly captures these differences and thus predicts correctly the execution times.

3.5.1 Single Diagonal Dependence

In this experiment we consider a kernel as also used in [25]. This example is used to check if we can correctly predict which transformation is better by using the analytical model as we have defined in Section 3.4. The application is characterized by a compute process with a two dimensional iteration domain and a single diagonal self-dependency as shown in Figure 3.14. The application has three statements $S1$, $S2$, and $S3$ and the corresponding iteration domains and dependencies are shown in Figure 3.14 as well. In this example, a triangular assignment of process iterations to partitions using a diagonal plane-cut results in two partitions $P1$ and $P2$ free of any inter-process communication. The second partition $P2$ does not have any initial delay with respect to the first partition $P1$, but it does have a relatively large initial delay with respect to producer $S1$, i.e., 6 process iterations of $S1$, see Figure 3.14. The modulo assignment on the other hand, as also illustrated in Figure 3.14, would introduce many inter-process communications, but it has a small initial delay of only 2 iterations with respect to partition $P1$. With this experiment, we investigate if the model captures well the trade-off of having inter-process communication at low costs, or a case without any inter-process communication but with a relatively large initial delay. For testing purposes only, the iteration domains, compared to Figure 3.14, have been increased in the experiments to 20 iterations points for producer $S1$, and a 2-dimensional iteration domains of 10×10 for the compute process $S2$.

To evaluate and determine the transformation to be applied for this example, the

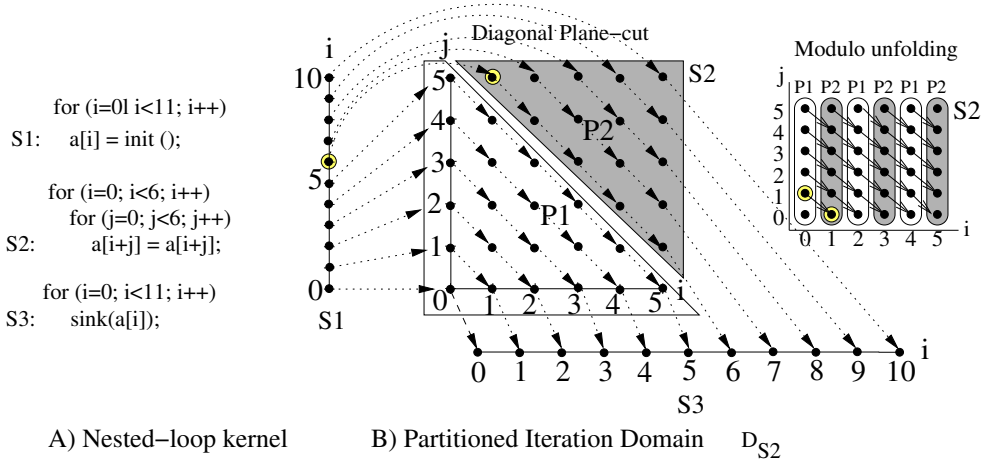

Figure 3.14: Nested-loop Program and Partitioned Dependence Graph

decision tree is checked as presented in Section 3.4. There is a self-dependency for compute process $S2$, so the right branch is taken and the dependency directions are analyzed. It is a single diagonal self-dependency and thus the decision tree indicates that we should consider the transformations in `case 3`, i.e., the transformations plane-cut and modulo unfolding on the inner and outer loop must be evaluated using Formula 3.7.

Communication	**Cost**
$C^{Rd}_{inter} : PPE \leftrightarrow SPE_i$	4000
$C^{Rd}_{inter} : SPE_i \leftrightarrow SPE_j$	160
$C^{Rd}_{intra} : SPE_i \leftrightarrow SPE_i$	10
$C^{Wr}_{inter} = C^{Wr}_{intra}$	10

Table 3.1: Communication Costs on the Cell

Table 3.1 shows the costs for communication on the Cell platform. It can be seen that there are two different costs for C^{Rd}_{inter}, because inter-process communication in the Cell can occur between the PPE and an SPE (the cost is 4000 cycles), but also between different SPEs (the cost is 160 cycles). Reading data from the same SPE, and also the writing of data, costs 10 cycles. There is no difference in the costs for writing data via inter/intra process communication, because data is always written to a local FIFO buffer of a producer process.

The two partitions $P1$ and $P2$ have a process workload of $W_{P1} = W_{P2} = 5000$.

The producer $S1$ does not have any workload such that $W_{S1} = 0$. These computation costs are shown in Table 3.2.

Computation	Cost
$W_{P1} = W_{P2}$	5000
W_{S1}	0

Table 3.2: Computation Costs on the Cell

Next, we consider the specific metric values of the *second partition* $P2$ for the different process splitting transformations as shown in Table 3.3.

Metric	planecut	unfold (outer)	unfold (inner)
Prod. Delays $Y_{S1}(D_{P2})$,$Y_{P1}(D_{P2})$	11, 0	0, 3	2, 0
Production Periods $d_{S1},\ d_{P1}$	$\frac{20}{10}(S1)$	$\frac{50}{45}(P1)$, $\frac{20}{5}(S1)$	$\frac{50}{45}(P1)$, $\frac{20}{5}(S1)$
DT^{Rd}_{inter}	9	45 + 5 = 50	45 + 5 = 50
DT^{Rd}_{intra}	36	0	0
DT^{Wr}_{inter}	9	45 + 5 = 50	45 + 5 = 50
DT^{Wr}_{intra}	36	0	0

Table 3.3: Partition P2 and its Metric Values

The first row shows that the plane-cut transformation has an initial delay of 11 iteration caused by producer $S1$. The modulo transformation on the outer loop has an initial delay of 3 iterations: the second partition $P2$ needs to wait 2 iterations for the first partition $P1$, which on its turn needs to wait 1 iteration for producer $S1$. The modulo transformation on the inner loop has an initial delay of 2 iterations, which is caused only by only one process, i.e., producer $S1$. For the plane-cut experiment, 10 data tokens are read from $S1$, which produces 20 tokens in total. Therefore, the production period is $\frac{20}{10}$. Furthermore, 9 tokens are read/written via inter-process communication, and 36 tokens are read/written via intra-process communication. For both the unfolding transformations, 50 tokens are read via inter-process communication and 0 tokens via intra-process communication. The writing of tokens is performed with 50 tokens via inter-process communication, and 0 tokens via intra-process communication. We use these metric values to calculate the execution time of the modulo unfolding transformation T_{omod} using the model defined in Formula 3.7 as follows:

$$
\begin{aligned}
T_{omod} &= T^{init}_{P2} + T^{exec}_{P2} = 11108 + 305450 = 316558 \\
T^{iter}_{P2} &= 5000 + \tfrac{45}{50} \cdot 160 + \tfrac{5}{50} \cdot 4000 + \tfrac{50}{50} \cdot 10 = 5554 \\
T^{period}_{S1} &= \tfrac{20}{5} \cdot 10 = 40
\end{aligned}
$$

$T_{P1}^{period} = \frac{50}{45} \cdot 5554 = 6788$
$T_{avg\_period} = \frac{5}{50} \cdot T_{S1}^{period} + \frac{45}{50} \cdot T_{P1}^{period} = \frac{5}{50} \cdot 40 + \frac{45}{50} \cdot 6788 = 6109$
$T_{P2}^{exec} = 50 \cdot max(5554, 6109) = 305450$
$T_{P2}^{init} = d_{P1} \cdot T_{P1}^{iter} = 2 \cdot 5554 = 11108$

If we do the same for the plane-cut and unfolding on the inner loop, then we obtain $T_{plane} = 301248$ and $T_{imod} = 304736$. Thus, we find that $T_{plane} < T_{imod} < T_{omod}$ which indicates that the plane-cut transformation can be applied best because its estimated execution time is smaller compared to the other 2 transformations. In other words, our solution approach finds that the plane-cut transformation must be applied to obtain the best performance results. This compile-time hint is correct according to the measured performance results shown in Figure 3.15.

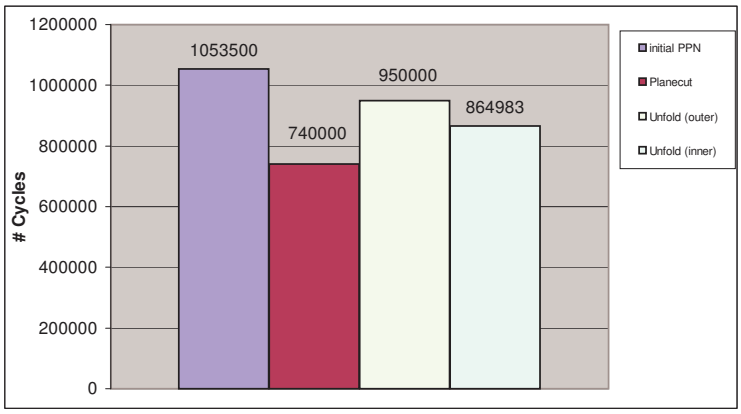

Figure 3.15: Diagonal Dependencies: Measured Performance Results on the Cell

The first bar in Figure 3.15 shows the result for the initial PPN on the Cell. The application executes in just over 1 million cycles. The second, third and fourth bar show the measured performance results for the plane-cut, and modulo unfolding on the outer and inner loop, respectively. We observe that the plane-cut is better than the 2 modulo unfolding transformations, which corresponds to the compile-time hints as calculated above. The purpose of calculating the execution time is not to estimate the real absolute performance results as close as possible, but to capture the trend of the transformations instead. The difference of the calculated execution times and the measured performance results on the Cell, for example, can be explained by the initialization and termination of SPE threads.

For the ESPAM platform we perform the same calculations and predictions. The metrics are different only for the computation and communication costs. These costs

are both shown in Table 3.4, i.e., the process workload of the compute process is 5000 cycles, and the cost for reading/writing data through inter- and intra-processes communication is 10 cycles. Note that the costs for all communication types are the same on the ESPAM platform, whereas on the Cell they are different and more expensive.

Metric	Cost
Workload W_{P2}	5000
Comm. Costs: $C^{Rd}_{inter}, C^{Wr}_{inter}$	10
Comm. Costs: $C^{Rd}_{intra}, C^{Wr}_{intra}$	10

Table 3.4: Workload and Communication Costs on ESPAM

Using the metric values in Table 3.3 and 3.4, we calculate and predict the execution time for the three transformations on the ESPAM platform in the same way as we have shown above. We find that $T_{omod} \approx 252240$, $T_{plane} \approx 276200$, and $T_{imod} \approx 251220$ and observe that $T_{imod} < T_{omod} < T_{plane}$. Thus, the prediction is that the modulo unfolding transformation on the inner loop is better than the plane-cut and unfolding on the outer loop. The measured performance results shown in Figure 3.16 illustrate that this predictions are correct.

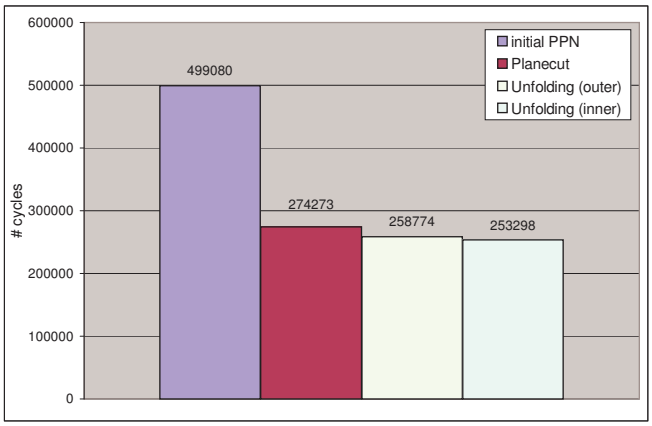

Figure 3.16: Diagonal Dependencies: Measured Performance Results on ESPAM

The first bar shows the measured performance results for the initial PPN, the second bar corresponds to the plane-cut transformation, and the third and fourth bars correspond to the results for the modulo unfolding on the outer and inner loop, respectively. It can be seen that the differences in the measured performance for the

different transformations are very small, as also predicted by the estimated execution times. Despite these small differences, the predictions are correct and unfolding on the inner loop results in the best performance results. We see that the plane-cut transformation gives the worst performance results on the ESPAM platform, while it is the best alternative on the Cell. From this experiment, we conclude that the analytical model captures well the fact that the initial delay can be the dominating factor even if there is inter-process communication, i.e., for the ESPAM platform the communication costs are cheap thereby making the initial delay the crucial factor.

Note that on the ESPAM platform the estimated execution times approximate very well the actual measured execution times. For the Cell platform, the estimated execution times are less than the measured execution times for this particular experiment, because we do not take into account the overhead in SPE thread creation, synchronization, and termination, and the absolute execution times are small. For PPNs with large execution times, this overhead will not be significant and, thus, the estimated execution times will approximate better the performance results as we show in the next experiments.

3.5.2 Matrix Multiplication with Multiple Dependencies

We consider a matrix multiplication kernel implemented with a 3 dimensional loop nest structure. A single plane and its dependencies are already shown in Figure 3.2. The matrix application is an extension of this as there are a number of these planes with dependencies from each point in a plane to the same point in the next plane. The matrix multiplication application is considered because both transformations will lead to a great number of inter- and intra-process communication, such that the same transformation may have a completely different impact on the Cell than on the ESPAM platform. We verify that the analytical model and solution approach correctly predicts this behavior. The initial PPN consists of 4 processes. Processes $P1$, $P2$, $P3$ initialize, respectively, the matrix where the result is stored and the two matrices that are multiplied. Process $P4$ is the compute process and with the plane-cut and unfolding transformations we create a second process $P4'$. We consider compute process $P4$, check the decision tree in Figure 3.9 and see that there are multiple self-dependencies for this process; the horizontal and vertical dependencies are orthogonal to each other, i.e., `case 4` of the decision tree. Thus, the transformations *plane-cut on the inner loop*, and *unfolding on the outermost loop* should be evaluated. Note that we do not evaluate the diagonal plane-cut, which is taken into account when the dependencies are linearly independent and not orthogonal, see the discussion on `case4` in Section 3.4. If we experiment with a kernel of $200 \times 200 \times 200$ iterations and apply the plane-cut transformation on the inner loop, then the first 100 iterations of the inner loop are assigned to the first partition and the remaining 100 to the sec-

Metric	planecut	unfold (outer)
$Y_{P1}(D_{P4'})$, $Y_{P2}(D_{P4'})$, $Y_{P3}(D_{P4'})$, $Y_{P4}(D_{P4'})$	0, 100, 100, 100	200, 200, 0, 1
Production Periods $d_{P1}, d_{P2}, d_{P3}, d_{P4}$	0, 2, 2, 100	2, 2, 0, 1
DT^{Rd}_{inter}	$40 \cdot 10^3$	$4 \cdot 10^6$
DT^{Rd}_{intra}	$12 \cdot 10^6$	$8 \cdot 10^6$
DT^{Wr}_{inter}	0	$4 \cdot 10^6$
DT^{Wr}_{intra}	$12 \cdot 10^6$	$8 \cdot 10^6$

Table 3.5: Partition $P4'$ and its Metric Values on the Cell

ond. As a result, the initial delay of the second partition is 100 iterations. In the modulo unfolding all iterations of the outer loop $i\%2 = 1$ are assigned to the first partition, and $i\%2 = 0$ to the second. As a result, the delay is 1 for the second partition. The metric values for this example are shown in Table 3.5, and it can be seen that there is a great number of inter and intra process data transfers.

Now we compute the time for both transformations by using these values in the formulas as we have presented before. We do no repeat all intermediate steps to calculate these numbers, but just give the final outcome. Note that the costs for FIFO communication is the same as in the previous experiment, see Table 3.1. The workload is also the same, i.e., 5000 cycles for the compute process(es).

The analytical model gives as a result that $T_{plane} \approx 20.4{\cdot}10^9$ and $T_{omod} \approx 21.4{\cdot}10^9$. Because the estimated time for the plane-cut transformation is less than the modulo unfolding, we conclude that the plane-cut transformation results in better performance results. As can be seen in Figure 3.17, the analytical model predicts correctly that the measured performance results on the Cell platform for the plane-cut transformation is better than the unfolding transformation. The first bar corresponds to the initial Polyhedral Process Network, which needs more than 4000 million cycles to finish its execution. The plane-cut transformed network is finished in 20071 million cycles and the unfolding transformation in 20445 million cycles.

Now we follow the same steps and predict the results for the ESPAM platform. Recall that the costs for communication and computation on the ESPAM platform is 10 clock cycles for both intra and inter process communication. The workload of the compute process(es) is 5000 cycles, and the process iteration domain is $20 \times 20 \times 20$. Thus, the total number of process iterations is 8000. After splitting the compute process, 4000 process iterations are executed by one partition, and the other 4000 process iterations by the other partition. We calculate the values and we obtain $T_{plane} \approx 20.29 \cdot 10^6$ and $T_{omod} \approx 20.24 \cdot 10^6$. Since the communication costs on the ESPAM platform are very cheap and the same for intra or inter process data transfers, we observe that the initial delay of a partition (i.e., $Y_{P4}(D_{P4'})$, see Table 3.5) is the

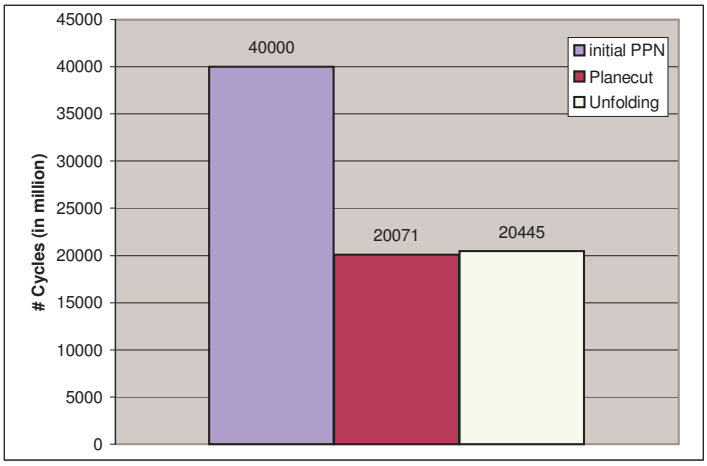

Figure 3.17: Measured Performance Results of Matrix Multiplication on the Cell

determining factor in this experiment. The analytical model predicts that the modulo unfolding transformation leads to better performance results. Figure 3.18, indeed, shows that for the measured performance results, the unfolding is better than the plane-cut.

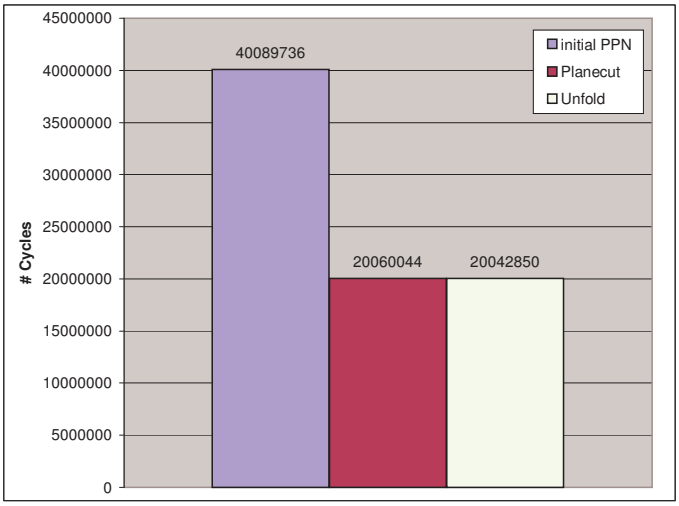

Figure 3.18: Measured Performance Results of Matrix Multiplication on ESPAM

The first bar shows the results of the matrix multiplication mapped as a Polyhe-

dral Process Network onto the ESPAM platform. It is finished in a bit more than 40 million clock cycles. The second bar shows the result for the plane-cut transformation, which is finished in 20060044 cycles. The third bar corresponds to the modulo unfolding and we see that the unfolding transformation is slightly better than the plane-cut, i.e., it is finished in 20042850 cycles.

3.5.3 Four Producers with Delays

In this experiment, we investigate the effects of production periods on different transformations. The production period of one producer process is chosen to be much larger than the other producers. The experiment has been setup in this way, to see if the analytical model under these conditions still correctly predicts the trend. The Polyhedral Process Network (PPN) used in this experiments is derived from the nested loop program below:

```
for (i=2; i<100; i++)
  for (j=0; j<100; j++)
    x[i], y[j] = C(x[i], y[j], z[2*i][4*j], w[i][j]);
```

At each iteration, function `C` is executed and data is read from different arrays. Arrays x and y are read at each iteration and also new values are written into it. Thus, there are two (orthogonal) self-dependencies for this function call statement. The third input argument array z is indexed with expressions $2 * i$ and $4 * j$. Consecutive read accesses at the consumer process, map to iteration points at the producer process which are not consecutive. For example, iterations $(2, 0)$ and $(2, 1)$ of the consumer map to iterations $(4, 0)$ and $(4, 4)$ at the producer. In this way, we model a producer process with a production period that is different from the other processes. The fourth input argument is array w, which is written and read at each iteration of the producer and consumer. Furthermore, the first iteration of i starts at 2, such that there is an initial delay for each of the producers. The corresponding PPN is shown in Figure 3.19 A). It consists of 4 producer processes $P1, P2, P3, P4$ and a single consumer C.

To determine which transformation is better, the decision tree (see Figure 3.9) indicates that the transformations plane-cut on the inner loop and unfolding on the outer loop must be compared, i.e., it is `case 4`, as the dependencies are orthogonal in this example. The networks for the unfolding and plane-cut transformations are shown in Figure 3.19 B) and C), respectively. It can be seen in Figure 3.19 C) that, for the plane-cut transformation, the second partition $C2$ receives data from processes $P1, P2, P4, C1$. The first iteration to be executed by the second partition $C2$ is iteration point $(2, 50)$. Producer process $P1$ generates data for this point at iteration $(4, 200)$ as a result of index expressions $2*i$ and $4*j$ at the consumer $C2$. Therefore, the initial delay is $4 * 400 + 200 = 1800$ iterations with regards to producer process

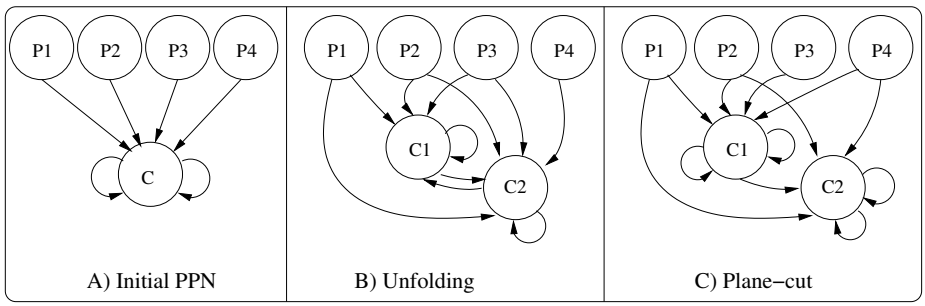

Figure 3.19: Consumer(s) with 4 Producers

$P1$. To calculate the production period, we find that producer $P1$ executes 80.000 iterations and that consumer $C2$ reads 4900 tokens from it. Therefore, the production period is $\frac{80000}{4900} \approx 16$ iterations. For the other producer process, the initial delays and production periods are calculated in a similar way and are also shown in Table 3.6. For the unfolding transformation, we see in Figure 3.19 B) that partition $C2$ depends on 5 producers. To give an example of the initial delay calculation for this transformation, we consider the first iteration point $(3, 0)$ of partition $C2$. This point is mapped to iteration point $(6, 0)$ of the producer $P1$, and hence the initial delay is $6 * 400 + 1 = 2401$. The other delays are 1201, 4, 1 and 1 iterations with respect to the remaining 4 producer processes, which is also shown in Table 3.6.

Metric	**planecut**	**unfold (outer)**
$Y_{P1}(D_{C2}), .., Y_{P4}(D_{C2}), Y_{C1}(D_{C2})$	1800, 850, 0, 3, 3	2401, 1201, 4,1,1
$d_{P1},\ d_{P2},\ d_{P3},\ d_{P4},\ d_{C1}$	16, 16, 0, 2, 50	16, 16, 2, 1, 2
DT^{Rd}_{inter}	98	4800
DT^{Rd}_{intra}	9652	4851
DT^{Wr}_{inter}	0	4800
DT^{Wr}_{intra}	9652	4851

Table 3.6: Partition C2 and its Metric Values on the Cell

The communication costs and the process workload are the same as in the previous experiments, i.e., the communication costs are shown in Table 3.3 and the workload is 5000 cycles for the compute process. If we use these metric values to calculate and predict the execution times of the transformed PPNs, we obtain that $T_{plane} \approx 39$ million cycles and $T_{omod} \approx 37$ million cycles.

The measured performance results on the Cell platform confirm that the compile-time hint is correct. The first bar in Figure 3.20 shows that the PPN is finished in

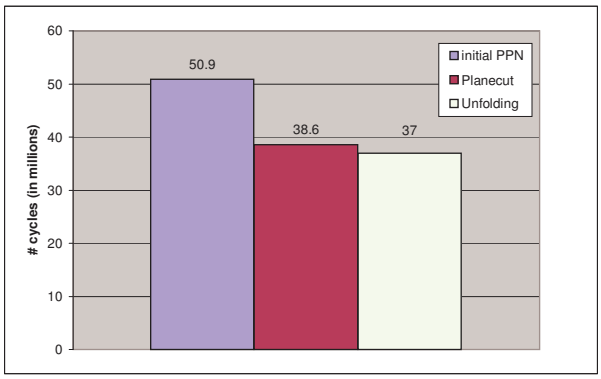

Figure 3.20: Measured Performance Results on the Cell

50.9 million cycles. The second bar corresponds to the plane-cut transformation and is finished in 38.6 million cycles, and the third bar corresponds to the unfolding transformation which is finished in 37 million cycles. We observe that, indeed, the unfolding transformation is better compared to the plane-cut transformation.

If we want to predict which transformation is better for the ESPAM platform, we repeat all steps. The only difference are the metric values for writing/reading to/from FIFO channels, which are shown in Table 3.4. If we compute the execution time for both transformations, we find $T_{plane} \approx 27.8$ million cycles and $T_{omod} \approx 25.6$ million cycles . This prediction indicates that the unfolding transformation should be applied to minimize the execution time.

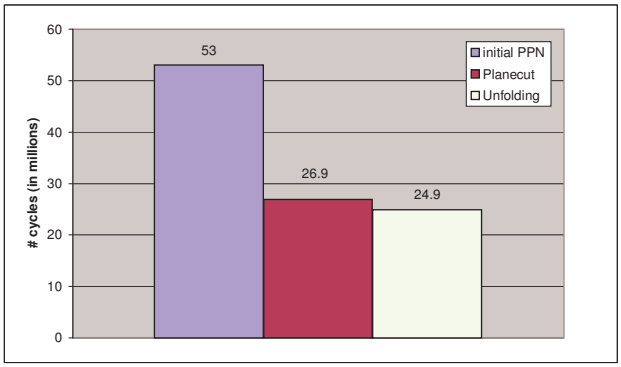

Figure 3.21: Measured Performance Results on ESPAM Platform

The measured performance results on the ESPAM platform are shown in Figure 3.21. The initial polyhedral process network is finished in 53 million cycles, the plane-cut

transformed network in 26.9 million cycles, and the unfolded network in 24.9 million cycles. This confirms the prediction that the unfolding transformation leads to better performance results than the plane-cut.

3.6 Discussion and Summary

We have presented a compile-time approach to select a particular splitting transformation in order to achieve the best possible performance results. We defined the metrics that are required to make such a decision, showed how the metric values can be calculated, and presented a solution approach that uses these metric values to evaluate the different transformations to give hints to the designer. With the experiments, we have shown that our model correctly predicts which transformation can be applied best. In order to correctly predict which transformation is better, the designer needs to provide the following parameters: the workload of all functions, the costs for FIFO reading/writing on the target platform, and on which process the process splitting should be applied. A designer may therefore still have the following questions:

1. Which process should be split-up for the best performance results?
2. What if the process workload is not constant?
3. What if the cost for FIFO reading/writing is not constant?

The first two questions are related, because the process splitting transformation has the largest positive impact when it is applied on the process with the largest workload, i.e., the computationally most intensive process. To obtain the process workload, the designer has to run the functions on the target platform, or generate a profile of the application. Thus, not only the workload is obtained, but also a first indication which process can possibly be the bottleneck process of the system. For simple polyhedral process networks, i.e., if they behave like SDF graphs [47] and always read/write from/to the FIFO channels, the workload is enough to identify the bottleneck processes. However, when the process network has complicated communication patterns, it becomes very difficult to identify a single bottleneck process. The reason is that different processes can dominate the throughput at different stages of the execution of the application. This could imply that the designer needs to apply splitting on different processes in order to obtain a balanced PPN that meets the performance requirements, i.e, following the Y-chart approach, and in an iterative way, splitting can be applied consecutively on different processes. In Chapter 5, we show an example of different processes that dominate the throughput at different stages of the execution of the PPN. Moreover, an approach is presented how to apply the process splitting and merging transformation in combination that relieves the designer from

the task to select a particular process. In this approach, the results of this chapter are used to decide how a process must be split up. Thus, question 1 posed here is solved as discussed in Chapter 5.

Besides selecting the best process on which the splitting transformation should be applied, a designer can have process functions with *non-constant* execution times. In the experiments discussed in Section 3.5, the workload is constant because the functions internally do not have any branches. In other words, the process workload consists of one sequence of instructions, without any branches with a varying number of instructions. Executing such functions will always require the same number of time units, i.e., it is constant. However, if a function does have branches then the execution time of that process can vary depending on which branches are taken. To model the workload of a process in this case, two options are possible: to take the worst-case execution time of the function, or to calculate an average value. It should be clear, however, that the model becomes less precise regardless whatever option the designer chooses as a solution to set the workload. The main question is: will this result in incorrect predictions what transformation should be applied? We have not investigated this with experiments, but it is not difficult to imagine that this can actually happen. If the error in the workload is significant, then the wrong value can be chosen in calculating the execution time of one process iteration as shown in Formula 3.8. On the other hand, if an imprecise workload value is used, then it is used in all evaluations of the different splitting transformations. So, in the end the trend may still be correct, but as already mentioned above, this has not been investigated. The reason is that we consider a class of applications, i.e., streaming applications, that does not expose this behavior in its process functions. Typically, data is streamed in and a series of computations are performed on the data before data is written back. In the unlikely case the process functions have some branches, then these different branches have similar computational workload.

Similar to the process workload, the costs for FIFO communication has also been modeled with a constant value. The problem is that imprecise cost estimations make evaluating the model less precise. The communication costs can have non-constant values when the platform interconnect, for example, is designed to provide a “best effort” service, instead of a “guaranteed service”. We assumed the latter and thus created platform instances that provide constant costs for FIFO communication. For embedded platforms this is a realistic assumption, because these platforms should be predictable and analyzable. In the ESPAM platform for example, the FIFO communication is implemented with hardware components and the processors can be point-to-point connected. In this case, the costs for FIFO communication is truly constant. However, if a crossbar is chosen as the interconnect for the different processors, then the FIFO costs are not constant anymore as it depends on the number of requests and the arbitration scheme of the crossbar. In [38], a performance model is introduced

for different crossbar configurations, which can serve as a basis to model the FIFO costs, but we did not investigate this in the experiments. The other platform used in the experiments is the Cell platform, which uses the so called Element Interconnect Bus (EIB) [3] to connect the different processing elements. It is a bus consisting of 4 data rings and a shared command bus and multiple data transfers can be in process concurrently on each ring. We implemented FIFO communication on this provided communication infrastructure [58] and modeled the costs with a constant value. This could be inaccurate as a FIFO transfer on the CELL consists of 3 parts, i.e., 2 signals and 1 DMA transfers, and thus 3 factors influence the actual time for performing one data transfer. However, when we measure the costs for FIFO reading/writing on the real hardware, they are almost constant. Apparently, all request can be processed and no delays occur in processing them, i.e., the bus is not saturated with requests, and the costs for FIFO reading/writing are nearly constant. We were therefore able to also correctly predict the performance results for the different process splittings on the CELL platform.

Chapter 4

Process Merging Transformations

Recall from Chapter 3 that the partitioning strategy of the `pn` compiler may not necessarily result in PPNs that meet the performance/resource requirements. To meet the performance requirements, a designer can apply the process splitting transformation as discussed in Chapter 3. In this chapter, we introduce the process merging transformation that reduces the number of processes in a PPN. The process merging transformation is not only useful to meet the performance constraints, but also allows a designer to achieve the same performance using fewer processes in some cases. We show that many solutions exist to merge different processes in a PPN with great differences in performance results. Thus, it is not trivial to select the best merging solution. We address this issue in this chapter by presenting a compile-time solution to evaluate different merging alternatives.

4.1 Process Merging: Definitions

The process merging transformation reduces the number of processes in a PPN by sequentializing n processes in a single compound process.

Definition 16 *The **process merging** transformations takes n processes $P_1, .., P_n$ and sequentializes them into one compound process $P_{1..n}$.*

Definition 17 *A **compound process** is formed by merging n processes and executes in a sequential way the functions of the processes that are merged.*

A compound process has, therefore, the following properties:

- Per iteration of the compound process, process functions of $P_1, .., P_n$ are executed sequentially.

- The process iteration domain sizes of $P_1, .., P_n$ can be different. Then, the different process functions are executed sequentially per compound process iteration for a number of overlapping process iterations. In the remaining compound process iterations, where the process iterations do not overlap, only the process function(s) is executed of the process that has the largest number of process iterations.

- If there exists a dependency between the processes, then the `pn` compiler calculates a safe offset between the process functions in the compound process.

As a result of using the process merging transformation, less processes need to be mapped on the platform's processing elements, at the price of possibly having less processes running in parallel. A designer needs to apply the process merging transformation in case *i)* the number of processes is larger than the number of processing elements, or *ii)* the network is not well balanced and therefore the same overall performance can be achieved using less resources. For both cases, the problem is that many different options exist to merge two or more processes. The total number of options to merge different processes for a PPN with *n* processes is $\sum_{i=2}^{n} \left(\begin{array}{c} n \\ i \end{array} \right)$. To give an example for a PPN with 5 processes there are $\left(\begin{array}{c} 5 \\ 2 \end{array} \right) + \left(\begin{array}{c} 5 \\ 3 \end{array} \right) + \left(\begin{array}{c} 5 \\ 4 \end{array} \right) + \left(\begin{array}{c} 5 \\ 5 \end{array} \right) = 26$ different options to merge 2, 3, 4, or 5 processes. The challenge is how to find the best solution from all these options. To solve this problem, an analytical throughput modeling framework for Polyhedral Process Networks (PPNs) is defined in this chapter. The throughput model is used to evaluate the throughput of different process mergings in order to select the best option which gives a system throughput as close as possible to the initial PPN.

4.2 Challenges of Applying the Process Merging Transformation

With 3 motivating examples we show that selecting the best merging option is not a straightforward task as it depends on the inter-play of many factors which may not be evident at first sight. The first factor to be considered is the *workload* of a process. Recall from Chapter 2, that the workload W_{P_i} of a process P_i denotes the number of time units that are required to execute a function, i.e., the pure computational workload, excluding the communication. Figure 4.1 shows a PPN consisting of 6 processes. It is annotated with the process workload and shows the number of readings/writings from/to each FIFO channel. Process $P2$, for example, has a workload of 10 time units and a single token is read/written from/to a FIFO channel per process iteration, which is denoted by "$[1]$" and can be repeated (possibly) in-

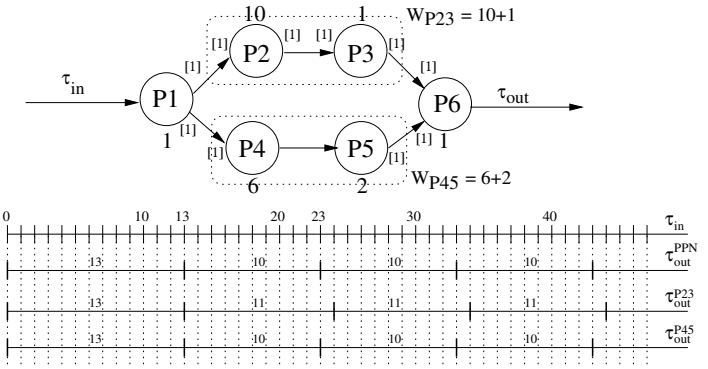

Figure 4.1: Process Workload Influencing the System Throughput

finitely many times. The network has two datapaths $DP1 = (P1, P2, P3, P6)$ and $DP2 = (P1, P4, P5, P6)$ that transfer an equal amount of tokens. We observe that process $P2$ determines the system throughput, which is illustrated with the time lines at the bottom of Figure 4.1. The first time line shows the rate τ_{in} at which tokens arrive at the network, i.e., each time unit. The second time line shows the system throughput of the initial PPN, denoted by τ_{out}^{PPN}.

Definition 18 *The **system throughput**, denoted by τ_{out}, is defined as the number of data tokens produced by the network per time unit.*

Process $P6$ needs 13 time units (1+10+1+1) to produce its first token. Then, it produces a new token each 10 cycles which is dictated by the slowest process $P2$. If we apply the process merging transformation to processes $P2$ and $P3$, then compound process $P23$ becomes the most computationally intensive process of the network. Processes $P2$ (10 time units) and $P3$ (1 time unit) are sequentialized and thus it will take 10+1=11 time units instead of 10 time units for process $P6$ to produce a new token, as shown in the time line denoted by τ_{out}^{P23}. We observe that the throughput of this network is lower than the throughput of the initial PPN. The fourth time line, denoted by τ_{out}^{P45}, shows the system throughput after merging processes $P4$ and $P5$. In this case, however, we see that the system throughput is not affected, i.e., it is the same as the throughput of the initial PPN, because the two merged and sequentialized processes do not dictate the system throughput. Thus, a designer can safely merge these processes and achieve the same system throughput as the initial PPN.

With the following example, we show that considering the process workload W_{P_i} only is not enough; a second factor that needs to be taken into account is the *rate of producing tokens*. Consider the PPN in Figure 4.2 which is topologically the same as in the previous example. The only difference is that both datapaths transfer a different

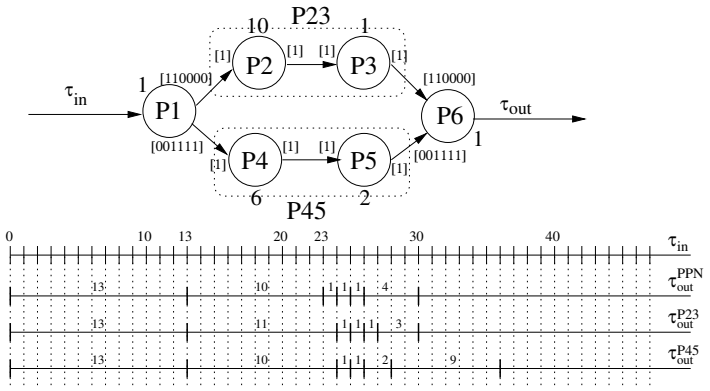

Figure 4.2: Production Rate Influencing the System Throughput

number of tokens. This is indicated with the patterns `[110000]` and `[001111]` at which process $P1$ writes to its outgoing FIFO channels. A "`1`" in these patterns indicates that data is read/written and a "`0`" that no data is read/written. So, the FIFO channel connecting $P1$ and $P2$, for example, is written the first two iterations of $P1$, but not in the remaining 4. As a consequence of these patterns, more tokens are communicated through the second datapath $DP2 = (P1, P4, P5, P6)$. Therefore, we observe that, despite process $P2$ largest workload of 10 time units, process $P4$ with a workload of 6 is more dominant. Therefore, merging processes $P4$ and $P5$ leads to a lower network throughput compared to merging $P2$ and $P3$, as can be seen in the time lines τ_{out}^{P45} and τ_{out}^{P23} in Figure 4.2. We observe a trend which is completely different from the previous example. According to Figure 4.2, a designer can safely merge processes $P2$ and $P3$ as opposed to $P4$ and $P5$ to achieve a system throughput that is equal to the throughput of the initial PPN.

In the last motivating example, we consider the PPN shown in Figure 4.3. The processes always read and/or write a single token when they are executed. Therefore, one could expect that this example is different from the example in Figure 4.2, but similar to the example in Figure 4.1. We show, however, that neither case applies and that a third factor needs to be taken into account. In this example, process $P1$ is the computationally most intensive process with a workload of 53 time units. If a designer wants to merge processes, a logical choice would be to merge $P2$ and $P3$ and not to consider the heavy process $P1$.

Processes $P2$ and $P3$ both have a workload of 25 time units and thus the compound process $P23$ has a summed workload of 50 time units, which is smaller than process $P1$ (53 time units). For this reason, we expect performance results that are equally good as the initial PPN. However, when we measure the performance results of both the initial PPN and the transformed PPN on the ESPAM platform [61], there is a

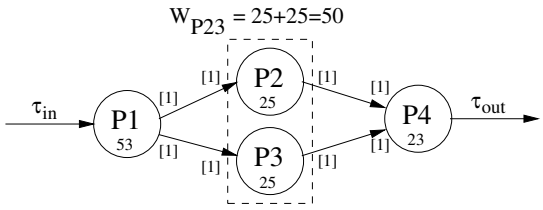

Figure 4.3: Sequentialized FIFO Accesses Influencing the System Throughput

20% degradation in the performance results. Although the workload of compound process $P23$ is lower than $P1$, the compound process reads sequentially from two input channels and writes sequentially to two output channels. This makes it the heaviest process in the network. So, besides sequential execution of the process workloads, we observe that *sequential FIFO reading/writing* is another aspect that should be taken into account.

The 3 examples above show that it is not trivial to merge processes and to achieve performance results as close as possible to the initial PPN. Therefore, we want to have a compile-time framework to evaluate the system throughput such that the best possible merging can be selected. Our compile-time framework is based on the throughput modeling techniques presented in Section 4.4.

4.3 Restrictions on the Throughput Modeling

A number of restrictions apply on the throughput model as presented in Section 4.4. First of all, we consider acyclic PPN graphs. Cycles in a PPN are responsible for sequential execution of some of the processes involved in the cycle. The sequential execution can vary from a single initial delay, to a delay at each iteration of some of the processes. For accurate throughput modeling, these cycles must be taken into account which we do not study in this work. The reason is that throughput modeling for acyclic networks is already a very difficult task, which is even more challenging for cyclic networks. There are recent works that started to investigate the performance analysis of cyclic dataflow graphs [86], but more research is required in that area in the future.

Secondly, it is important to state that our goal is not to compare different PPNs, but to compare transformed PPNs derived from a single PPN. Therefore, in the throughput modeling, we do not take into account the latency of a token, i.e., the time that elapses between injecting a token in the PPN and the time when that token leaves the PPN. Thus, we do not calculate the total execution time of PPNs, but only want to capture the throughput trend instead. The reason is that the framework should be fast,

and only as accurate as needed to correctly capture the throughput trend for different process mergings.

Thirdly, the process workload W_{P_i} and the costs for FIFO communication are parameters in our system throughput modeling. These are constant values that should be provided by the designer who can obtain them, for example, by executing the function and FIFO read/write primitives once on the target platform. The reader is referred to Section 3.6 for a discussion on the modeling of the process workload and FIFO read/write primitives with constant values. Although our approach is extensible to heterogeneous MPSoCs, we restrict ourself to MPSoCs with programmable homogeneous cores. The reason is that a process function implemented as software cannot be merged with a process function that is implemented as a hardware IP core. Similarly, one cannot merge two processes both implemented as IP cores. This means that once the process workload of a given process is determined, that this process workload value is the same for all programmable homogeneous cores in the target platform.

Finally, we do not study the effect of different buffer sizes. Although buffer sizes play an important role in the performance results, there are studies [17] showing that saturation points can be found where performance does not increase for larger buffer sizes. The `pn` compiler can find such points and we use buffer sizes that correspond to these points, i.e., the buffer sizes that give maximum performance.

4.4 Throughput Modeling

We introduce first the solution approach to model the throughput of polyhedral process networks with an example. Then, we define all concepts and steps of the throughput model in detail. Finally, we present the overall algorithm for the throughput modeling.

4.4.1 Process Throughput and Throughput Propagation

The solution approach for the overall Polyhedral Process Network (PPN) throughput modeling relies on calculating the throughput τ_{P_i} of a process P_i for all processes and propagation of the lowest process throughput to the sink processes. For a process P_i, the propagation consists of selecting either the aggregated incoming FIFO throughput $\tau_{F_{aggr}}$ or the isolated process throughput $\tau_{P_i}^{iso}$:

$$\tau_{P_i} = min(\tau_{F_{aggr}}, \tau_{P_i}^{iso}), \tag{4.1}$$

Before defining formally $\tau_{F_{aggr}}$ and $\tau_{P_i}^{iso}$ (in Sections 4.4.2 - 4.4.4), we first give an intuitive example of the solution approach applied on the PPN shown in Figure 4.3

and explain the meaning of Equation 4.1. First, the workload of each process is taken into account and let us assume that it takes 10, 20, 10, 10 time units for processes $P1, P2, P3, P4$, respectively, for executing its function. This means that, for example, $P1$ can read and produce a new token every 10 time units if there is input data. Thus, we define the isolated process throughput to be $\tau_{P1}^{iso} = \frac{1}{10}$ tokens per time units (excluding communication costs for the sake of simplicity). Similarly for the other processes, we define $\tau_{P2}^{iso} = \frac{1}{20}, \tau_{P3}^{iso} = \frac{1}{10}, \tau_{P4}^{iso} = \frac{1}{10}$. However, the required input data for a process can be delivered with a different throughput, i.e., the aggregated incoming FIFO throughput $\tau_{F_{aggr}}$. Consequently, the lowest throughput ($\tau_{F_{aggr}}$ or $\tau_{P_i}^{iso}$) determines the actual process throughput τ_{P_i}. Therefore, the minimum throughput value is selected as shown in Equation 4.1. This is repeated for all processes by iteratively applying Equation 4.1 on each process to select the lowest throughput and to propagate it to the sink processes. First, the PPN graph is topologically sorted to obtain a linear ordering of processes, e.g., $P1, P2, P3, P4$. In step I)

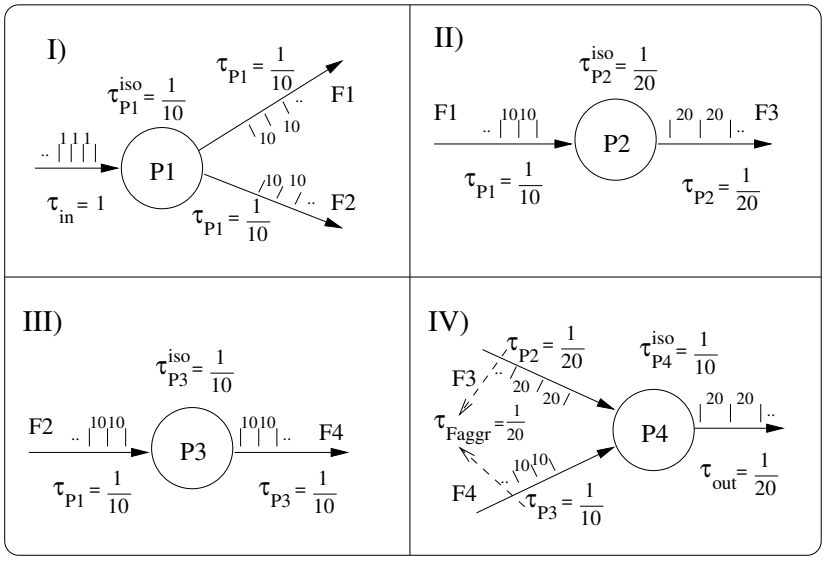

Figure 4.4: Throughput Propagation Example

of Figure 4.4, process $P1$ is the first process to be considered. While it receives tokens at each time unit ($\tau_{in} = 1$), it is ready to execute again after 10 time units due to the process workload ($\tau_{P1}^{iso} = \frac{1}{10}$). We see that the actual process throughput is determined by the process itself (it is the slowest) and Equation 4.1 is used to find this: $\tau_{P1} = min(1, \frac{1}{10}) = \frac{1}{10}$ with which it writes to both its outgoing FIFO channels $F1$ and $F2$.

If we continue with the second process in step II), we see that $P2$ receives tokens

from $P1$ with a throughput of $\tau_{P1} = \frac{1}{10}$. However, $P2$ is twice slower than $P1$ which is delivering the data: $\tau_{P2} = min(\frac{1}{10}, \frac{1}{20}) = \frac{1}{20}$. Thus, we know that $P2$ writes its results to FIFO channel $F3$ with a throughput of $\frac{1}{20}$.

In step III), we calculate the throughput for process $P3$. It receives data from $P1$ with a throughput of $\tau_{P1} = \frac{1}{10}$, and it can process data with a throughput of $\tau_{P3}^{iso} = \frac{1}{10}$. We compare what is slower by calculating $\tau_{P3} = min(\frac{1}{10}, \frac{1}{10}) = \frac{1}{10}$ and set this as the throughput at which $P3$ writes to FIFO channel $F4$.

Finally, we consider process $P4$ in step IV). Process $P4$ reads from two FIFO channels $F3$ and $F4$, which are written by $P2$ and $P3$ with different throughputs. Therefore, the FIFO throughput must be aggregated in order to have a single throughput value at which data arrives. If we assume that both channels are read per process iteration of $P4$, then the slowest FIFO throughput determines the aggregated FIFO throughput. For this example, $\frac{1}{20}$ is the slowest component and we set $\tau_{F_{aggr}} = \frac{1}{20}$. Applying Equation 4.1 shows that the data is delivered with a lower throughput than $P4$ can actually process: $\tau_{P4} = min(\frac{1}{20}, \frac{1}{10}) = \frac{1}{20}$ and set this to be the process throughput. In this way, we have propagated the slowest throughput from $P2$ to the sink process $P4$, which in the end determines the overall system throughput. In the next sections we exactly define how the (isolated) process throughput and (aggregated) FIFO throughput can be calculated.

4.4.2 Isolated Throughput of a (Compound) Process

Definition 19 *The **isolated process throughput** of a process P_i, denoted by $\tau_{P_i}^{iso}$, is the number of tokens produced by P_i per time unit when the input rate of its input data is ∞.*

We illustrate the isolated process throughput with the example shown in Figure 4.5.

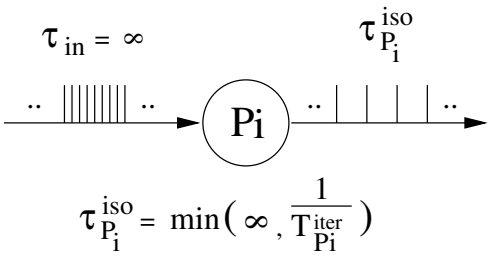

Figure 4.5: Isolated Process Throughput

We model the input data to arrive infinitely fast, i.e., $\tau_{in} = \infty$, such that the time $T_{P_i}^{iter}$ that is required for one process iteration, determines the throughput at which

tokens are produced by P_i. This means that the isolated process throughput is determined only by the workload W_{P_i} of a process and the number of FIFO reads/writes per process iteration provided that no blocking occurs:

$$\tau_{P_i}^{iso} = \frac{1}{T_{P_i}^{iter}}, \tag{4.2}$$

where $T_{P_i}^{iter}$ is the time to execute one process iteration as we have defined in Formula 3.9. It is important to note that two factors as identified in the motivating examples are taken into account in modeling the isolated process throughput: the time $T_{P_i}^{iter}$ for one process iteration includes the *process workload* W_{P_i} and also the number of *sequential FIFO accesses* (i.e., the data transfers).

In a similar way, we must also model the isolated throughput $\tau_{P_m}^{iso}$ of a compound process P_m in order to evaluate the system throughput for a PPN with merged processes. Assume that P_m is formed by merging processes P_i and P_j with iteration domains D_{P_i} and D_{P_j}, respectively. We define the isolated compound process throughput as $\tau_{P_m}^{iso} = \frac{1}{T_{P_m}^{iter}}$, where

$$T_{P_m}^{iter} = \frac{|D_{P_i}|}{|D_{P_j}|} \cdot (T_{P_i}^{iter} + T_{P_j}^{iter}) + \frac{|D_{P_j}| - |D_{P_i}|}{|D_{P_j}|} \cdot (T_{P_j}^{iter}) \tag{4.3}$$

with $|D_{P_i}| \leq |D_{P_j}|$. To model the time $T_{P_m}^{iter}$ for executing the compound process, we take into account the generated schedule of the compound process as produced by the `pn` and ESPAM tools [61, 95]. The execution of the process functions are interleaved as much as possible. This means that per iteration of the compound process, all functions are sequentially executed if this is allowed by the inter-process dependencies. In case of inter-process dependencies, an offset is calculated for the producer-consumer pair to ensure correct program behavior, and then the function execution is interleaved again. Therefore, we calculate fractions where the execution of the functions overlap and multiply it with the process iteration costs of these functions, i.e., the first term in Equation 4.3. And then we consider for the remaining iterations the cost of the process with the largest domain size only, i.e., the second term in Equation 4.3. Note that the coefficients in Equation 4.3 represent these fractions which should sum up to 1. Formula 4.4 below shows how $T_{P_m}^{iter}$ is calculated when n process are merged into a compound process P_m.

$$T_{P_m}^{iter} = \frac{|D_1|}{|D_n|} \cdot (\sum_{i=1}^{n} T_i^{iter}) + \sum_{j=2}^{n} \left(\frac{|D_j| - |D_{j-1}|}{|D_n|} \cdot (\sum_{k=j}^{n} T_k^{iter}) \right) \tag{4.4}$$

where the different process iteration domains have been sorted and renumbered according to their domain sizes, i.e., $D_1 \leq .. \leq D_{i-1} \leq D_i \leq D_{i+1} \leq .. \leq D_n$.

4.4.3 FIFO Channel Throughput

The throughput of a FIFO-channel is determined by the throughput of the processes accessing it. Let us consider the example shown in Figure 4.6. Assume that $P1$ executes 500 times, i.e., $|D_{P1}| = 500$, and each time it writes to $F1$ and $F2$.

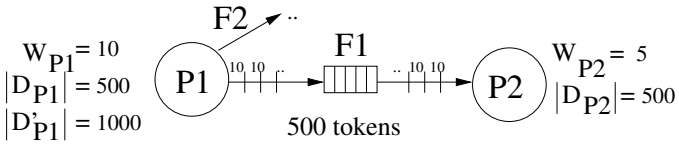

Figure 4.6: FIFO Channel Throughput

Process $P1$ needs 10 time units to produce a token. Consumer process $P2$ is twice as fast and needs only 5 time units to consume a token, but still it receives tokens only each 10 time units due to the slower producer. As a result, $P2$ blocks on reading and waits for data, which follows the operational semantics of the PPN model of computation: a process stalls if it tries to read from an empty FIFO channel and proceeds only if data is available again. This example shows that, to calculate the FIFO throughput τ_{f_i} of a FIFO channel f_i, the minimum is taken of the FIFO write throughput $\tau_{f_i}^{Wr}$ and the FIFO read throughput $\tau_{f_i}^{Rd}$:

$$\tau_{f_i} = min(\tau_{f_i}^{Wr}, \tau_{f_i}^{Rd}), \tag{4.5}$$

where $\tau_{f_i}^{Wr} = \tau_{P1}$ (see Equation 4.1) and $\tau_{f_i}^{Rd} = \tau_{P2}^{iso}$ (see Equation 4.2). Let us consider another example where $P1$ executes 1000 times, i.e., $|D'_{P1}| = 1000$ as also shown in Figure 4.6. Assume that in one iteration of $P1$ data is written to FIFO channel $F1$, and in the next iteration to $F2$. This is repeated such that in total 500 tokens are written to both FIFOs $F1$ and $F2$. To compensate for a producer that does not write data to a FIFO channel at each iteration, we define a coefficient that divides the total number of tokens transfered over a channel by the iteration domain size of a producer process P_i. This coefficient denotes an average production rate, expressed in a number of producer iteration points. Note that this takes into account the different *production rates* of processes as also identified in the motivating example in Figure 4.2. By multiplying this coefficient with the process throughput, we define FIFO write/read throughput $\tau_{f_i}^{Wr}$ and $\tau_{f_i}^{Rd}$ of a FIFO channel f_i as shown

in Equations 4.6 and 4.7. In this way, we model a lower throughput if necessary.

$$\tau_{f_i}^{Wr} = \frac{|OP_{P_i}^j|}{|D_{P_i}|} \cdot \tau_{P_i} \tag{4.6}$$

$$\tau_{f_i}^{Rd} = \frac{|IP_{P_i}^j|}{|D_{P_i}|} \cdot \tau_{P_i}^{iso}, \tag{4.7}$$

For the example, we see that $\tau_{f1}^{Wr} = \frac{500}{1000} \cdot \frac{1}{10} = \frac{1}{20}$ and the FIFO read throughput is $\tau_{f1}^{Rd} = \frac{500}{500} \cdot \frac{1}{5} = \frac{1}{5}$. Consequently, the FIFO throughput is $\tau_{f1} = min(\frac{1}{20}, \frac{1}{5}) = \frac{1}{20}$ tokens per time unit.

4.4.4 Aggregated FIFO Throughput

The throughput of a process τ_{P_i} is either determined by the FIFO throughput from which it receives its data, i.e., $\tau_{F_{aggr}}$, or by the computational workload of the process itself, i.e., $\tau_{P_i}^{iso}$, as shown in Equation 4.1. $\tau_{P_i}^{iso}$ is computed with Equation 4.2. Here we show how to compute $\tau_{F_{aggr}}$, which deals with the problem how to model the throughput of data in case there are multiple incoming FIFO channels. This is illustrated with the example in Figure 4.7.

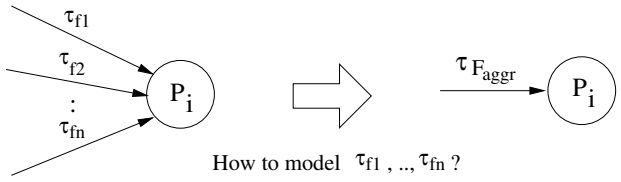

Figure 4.7: Modeling Multiple Incoming FIFO Channels

Process P_i has n incoming FIFO channels each with its own throughput. We need to model these different incoming FIFO channel throughputs as one throughput value, i.e., $\tau_{F_{aggr}}$, because we must determine what is slower: the arrival of the input data or the process itself. The throughput of the incoming FIFO channels are aggregated according to the way the process function input arguments are read.

To illustrate the calculation of the aggregated FIFO throughput, let us first consider Process P in Figure 4.8, which has **one** input argument value a that is read from two different input ports $IP1$ and $IP2$. Thus, two tokens are delivered, but only one is read for each iteration of the consumer process. The other token will be read in another iteration. To model the throughput at which data arrives, the sum is taken of the FIFO throughput $F1$ and $F2$, i.e., $\tau_{F_{aggr}} = \tau_{f_1} + \tau_{f_2}$. Effectively, this means that

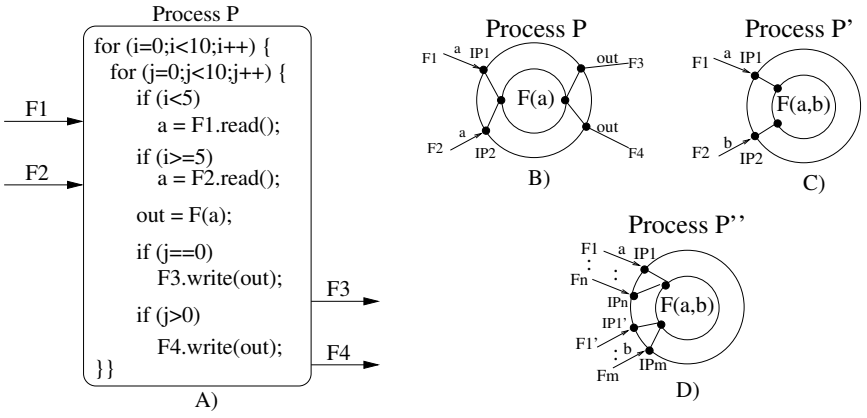

Figure 4.8: Process Structure (left) and FIFO Throughput Aggregation (right)

the aggregated incoming FIFO throughput becomes higher, which corresponds to the behavior that one token is needed but two are delivered. Note that any imbalance in the number of tokens transfered over each FIFO channel has already been taken into account in the FIFO read/write throughput as defined in Equation 4.6 and 4.7.

Process P' in Figure 4.8 is the second example, which reads its **two** input arguments values a and b from FIFOs $F1$ and $F2$. Thus, both FIFOs are read per process iteration of P'. If one FIFO throughput is fast and the other one is slower, then the slowest FIFO throughput determines the aggregated FIFO throughput. Therefore, we select the minimum throughput in this case, i.e., $\tau_{F_{aggr}} = min(\tau_{f_1}, \tau_{f_2})$.

Finally, the general case is illustrated with process P'' in Figure 4.8, i.e., it combines the previous two examples. Process P'' has multiple function input arguments and multiple incoming FIFO channels per input argument. To calculate the aggregated FIFO throughput, the throughput is summed of all the FIFO channels that are connected to one function input argument (the first example). Next, the minimum throughput, i.e., the slowest throughput, is taken of all the throughputs for the different function input arguments (the second example). Thus, the aggregated FIFO throughput $\tau_{F_{aggr}}$ for P'' is calculated as follows:
$\tau_{F_{aggr}} = min(\tau_{f_1} + .. + \tau_{f_n},\ \tau'_{f_1} + .. + \tau'_{f_m})$.

The general formula to calculate the aggregated FIFO throughput $\tau_{F_{aggr}}$ is given below:

$$\tau_{F_{aggr}} = min(\sum_{i=1}^{n} \tau_{f_i}, ..., \sum_{j=1}^{m} \tau_{f_j}) \tag{4.8}$$

where each sum corresponds to the sum of the throughputs of a number of FIFO channels connected to *one* process function input argument. Thus, the first term sums the throughput τ_{f_i} of n different FIFO channels connected to one process function input argument, and the last term sums the throughput τ_{f_j} of m different FIFO channels connected to another process function input argument. Finally, the minimum is taken to determine the slowest FIFO throughput.

4.4.5 System Throughput Calculation Algorithm

Up to now, we have formally defined all the components that allow the throughput calculation and propagation to be done in a systematic and automated way. The pseudo code of the throughput calculation and propagation algorithm is shown in Algorithm 1.

Algorithm 1 : PPN Throughput Estimation Pseudo-code

Require: PPN: a Polyhedral Process Network
Require: W_{P_i}: the computational workload of all processes.
Require: $C^{Rd,Wr}_{intra,inter}$: the costs for the FIFO read/write primitives.
$list \leftarrow$ Create topological ordering for PPN
for all process $P_i \in list$ **do**
1) Calculate $\tau^{iso}_{P_i} = set\_isolated\_throughput(P_i, W_{P_i}, C^{Rd,Wr}_{intra,inter})$
2) Set $\tau^{Rd}_{f_j}$ for all incoming FIFOs f_j of P_i.
3) Set τ_{f_j} for all incoming FIFOs f_j of P_i.
4) Calculate $\tau_{F_{aggr}} = calc\_fifo\_aggr(\tau_{f_j}, .., \tau_{f_n})$
5) Set $\tau_{P_i} = min(\tau^{iso}_{P_i}, \tau_{F_{aggr}})$
6) Set $\tau^{Wr}_{f_j}$ for all outgoing FIFO f_j of P_i.
end for
return $\tau^{PPN}_{out} = \tau_{P_{|list|}}$

This algorithm was introduced informally with the example in Section 4.4.1. Here we give the formal solution by applying Algorithm 1 on this example. All steps of Algorithm 1 are shown in Figure 4.9. The example PPN in Figure 4.3 consists of 4 processes and thus we obtain first a topologically ordered list of 4 processes, i.e., $list = \{P1, P2, P3, P4\}$. For each of these processes, we calculate the throughput at which the incoming data arrives, how fast a process can actually process this data, and the slowest value is propagated to the outgoing FIFO channels. The most interesting steps are $4.2.1 - 4.4$ in Figure 4.9, because the throughput of FIFO channels $F3$ and $F4$ are aggregated. Process $P4$ needs input tokens from both channels for each of its process iterations. Since the slowest FIFO throughput determines the aggregated FIFO throughput, the minimum FIFO throughput is selected in step 4.4.

$W_{P1} = W_{P3} = W_{P4} = 10,\ W_{P2} = 20$
$C^{Rd} = C^{Wr} = 0$

0 $list = \{P1, P2, P3, P4\}$
1.1 $\tau_{P1}^{iso} = \frac{1}{10}$
1.2 $\tau_{f_{in}}^{Rd} = \infty$
1.3 $\tau_{f_{in}} = \infty$
1.4 $\tau_{F_{aggr}} = \infty$
1.5 $\tau_{P1} = min(\frac{1}{10}, \infty) = \frac{1}{10}$
1.6.1 $\tau_{F1}^{Wr} = \frac{1}{10}$
1.6.2 $\tau_{F2}^{Wr} = \frac{1}{10}$

2.1 $\tau_{P2}^{iso} = \frac{1}{20}$
2.2 $\tau_{F1}^{Rd} = \frac{1}{20}$
2.3 $\tau_{F1} = min(\tau_{F1}^{Wr}, \tau_{F1}^{Rd}) = \frac{1}{20}$
2.4 $\tau_{F_{aggr}} = min(\frac{1}{20}) = \frac{1}{20}$
2.5 $\tau_{P2} = min(\frac{1}{20}, \frac{1}{20}) = \frac{1}{20}$
2.6 $\tau_{F3}^{Wr} = \frac{1}{20}$

3.1 $\tau_{P3}^{iso} = \frac{1}{10}$
3.2 $\tau_{F2}^{Rd} = \frac{1}{10}$
3.3 $\tau_{F2} = min(\tau_{F2}^{Wr}, \tau_{F2}^{Rd}) = \frac{1}{10}$
3.4 $\tau_{F_{aggr}} = min(\frac{1}{10}) = \frac{1}{10}$
3.5 $\tau_{P3} = min(\frac{1}{10}, \frac{1}{10}) = \frac{1}{10}$
3.6 $\tau_{F4}^{Wr} = \frac{1}{10}$

4.1 $\tau_{P4}^{iso} = \frac{1}{10}$
4.2.1 $\tau_{F3}^{Rd} = \frac{1}{10}$
4.2.2 $\tau_{F4}^{Rd} = \frac{1}{10}$
4.3.1 $\tau_{F3} = min(\tau_{F3}^{Wr}, \tau_{F3}^{Rd}) = \frac{1}{20}$
4.3.2 $\tau_{F4} = min(\tau_{F4}^{Wr}, \tau_{F4}^{Rd}) = \frac{1}{10}$
4.4 $\tau_{F_{aggr}} = min(\frac{1}{10}, \frac{1}{20}) = \frac{1}{20}$
4.5 $\tau_{P4} = min(\frac{1}{20}, \frac{1}{10}) = \frac{1}{20}$
4.6 $\tau_{out}^{PPN} = \tau_{P4} = \frac{1}{20}$

Figure 4.9: Throughput Calculation

In this way, we have propagated the slowest throughput of process $P2$ to the sink process, which determines in the end the overall system throughput.

4.5 Case-Studies

In this section we map two different nested loop kernels on the ESPAM platform prototyped on a Xilinx Virtex 2 Pro FPGA. Each process is mapped one-to-one on a MicroBlaze softcore processor and the processors are point-to-point connected. FIFO communication is implemented with FSL links and a FIFO access costs 10 clock cycles. We investigate if our throughput modeling captures the differences in performance results for different process merging configurations and process workloads.

4.5.1 Merging Light-Weight Producers

In the first experiment, we merge two light-weight producers (workload of 54 time units) into a single process, and we should observe that the new compound process does not become the process that determines the system throughput, i.e., the through-

put of the PPNs before and after the process merging are the same. Then, we increase the workload of the producers to 59 time units such that we intentionally introduce a new bottleneck in the PPN. The throughput of the PPN after the process merging should be less than the initial PPN, and we test whether this is captured by our throughput model.

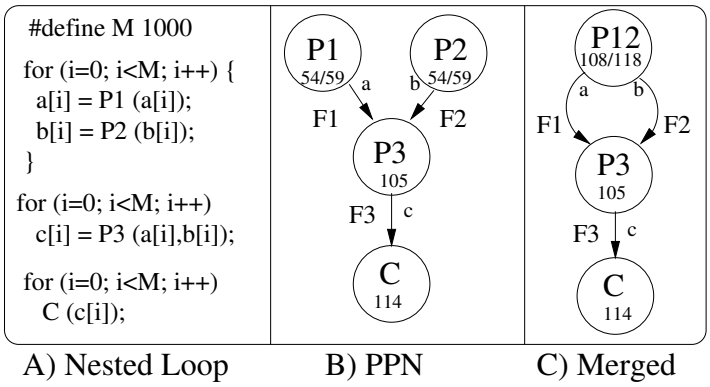

Figure 4.10: Example PPN

Figure 4.10 shows the nested loop program in A), the derived PPN in B), and the PPN with producers $P1$ and $P2$ merged in C). We calculate the throughput of the PPN before and after merging by applying Algorithm 1.

Figure 4.11 shows the analysis for process $P1, P2, P3$ and C. In process $P3$, two FIFO throughput values are aggregated as shown in step 3.4 of the throughput calculation in Figure 4.11. We find a process throughput of $\tau_{P3} = \frac{1}{135}$ for process $P3$, which is propagated to C such that the system throughput is $\tau_{out}^{PPN} = \tau_C = \frac{1}{135}$ as well.

Next, we calculate the system throughput for the PPN with processes $P1$ and $P2$ merged into one compound process. The throughput calculation is shown in Figure 4.12, and thus we find a system throughput of $\tau_{out}^{PPN'} = \frac{1}{135}$. Since we find a throughput of $\tau_{out} = \frac{1}{135}$ for both PPNs before and after merging, we predict that the initial PPN and transformed PPN$'$ perform equally well. This is confirmed by the actual measured performance results shown in Figure 4.13. That is, the first and second bar in Figure 4.13 denote the cycle numbers for the initial PPN and transformed PPN$'$, which are the same.

Then we increase the workload of the producer processes and intentionally create a compound process that is the most compute intensive process. We check if this is captured by our throughput model by analyzing the throughput of the PPNs before and after the merging. The throughput model gives a throughput of $\frac{1}{135}$ and $\frac{1}{138}$

0 $list = \{P1, P2, P3, C\}$

1.1 $\tau_{P1}^{iso} = \frac{1}{54+0+10} = \frac{1}{64}$
1.2 $\tau_{f_{in}}^{Rd} = \infty$
1.3 $\tau_{f_{in}} = \infty$
1.4 $\tau_{F_{aggr}} = \infty$
1.5 $\tau_{P1} = min(\frac{1}{64}, \infty) = \frac{1}{64}$
1.6 $\tau_{F1}^{Wr} = \frac{1000}{1000} \cdot \frac{1}{64} = \frac{1}{64}$

2.1 $\tau_{P2}^{iso} = \frac{1}{54+0+10} = \frac{1}{64}$
2.2 $\tau_{f_{in}}^{Rd} = \infty$
2.3 $\tau_{f_{in}} = \infty$
2.4 $\tau_{F_{aggr}} = \infty$
2.5 $\tau_{P2} = min(\frac{1}{64}, \infty) = \frac{1}{64}$
2.6 $\tau_{F2}^{Wr} = \frac{1000}{1000} \cdot \frac{1}{64} = \frac{1}{64}$

3.1 $\tau_{P3}^{iso} = \frac{1}{105+(2\cdot10)+10} = \frac{1}{135}$
3.2.1 $\tau_{F1}^{Rd} = \frac{1000}{1000} \cdot \frac{1}{135}$
3.2.2 $\tau_{F2}^{Rd} = \frac{1000}{1000} \cdot \frac{1}{135}$
3.3.1 $\tau_{F1} = min(\frac{1}{64}, \frac{1}{135}) = \frac{1}{135}$
3.3.2 $\tau_{F2} = min(\frac{1}{64}, \frac{1}{135}) = \frac{1}{135}$
3.4 $\tau_{F_{aggr}} = min(\frac{1}{135}, \frac{1}{135}) = \frac{1}{135}$
3.5 $\tau_{P3} = min(\frac{1}{135}, \frac{1}{135}) = \frac{1}{135}$
3.6 $\tau_{F3}^{Wr} = \frac{1000}{1000} \cdot \frac{1}{135} = \frac{1}{135}$

4.1 $\tau_{C}^{iso} = \frac{1}{114+10+0} = \frac{1}{124}$
4.2 $\tau_{F3}^{Rd} = \frac{1000}{1000} \cdot \frac{1}{124} = \frac{1}{124}$
4.3 $\tau_{F3} = min(\frac{1}{135}, \frac{1}{124}) = \frac{1}{135}$
4.4 $\tau_{F_{aggr}} = \frac{1}{135}$
4.5 $\tau_{C} = min(\frac{1}{135}, \frac{1}{124}) = \frac{1}{135}$
4.6 $\tau_{out}^{PPN} = \tau_{C} = \frac{1}{135}$

Figure 4.11: Throughput Estimation of Processes $P1, P2, P3, C$ in Figure 4.10 B)

for the initial and transformed PPN, respectively. Thus, the throughput calculation indicates that the throughput of the merged PPN is lower, which is confirmed by the third and fourth bar in the measured performance results in Figure 4.13.

0 $list = \{P12, P3, C\}$
1.1 $\tau^{iso}_{P12} = \frac{1}{54+54+0+2\cdot10} = \frac{1}{128}$
1.2 $\tau^{Rd}_{f_{in}} = \infty$
1.3 $\tau_{f_{in}} = \infty$
1.4 $\tau_{F_{aggr}} = \infty$
1.5 $\tau_{P12} = min(\frac{1}{128}, \infty) = \frac{1}{128}$
1.6.1 $\tau^{Wr}_{F1} = \frac{1000}{1000} \cdot \frac{1}{128} = \frac{1}{128}$
1.6.2 $\tau^{Wr}_{F2} = \frac{1000}{1000} \cdot \frac{1}{128} = \frac{1}{128}$
2.1 $\tau^{iso}_{P3} = \frac{1}{105+2\cdot10+1\cdot10} = \frac{1}{135}$
2.2.1 $\tau^{Rd}_{F1} = \frac{1000}{1000} \cdot \frac{1}{135} = \frac{1}{135}$
2.2.2 $\tau^{Rd}_{F2} = \frac{1000}{1000} \cdot \frac{1}{135} = \frac{1}{135}$
2.3.1 $\tau_{F1} = min(\frac{1}{128}, \frac{1}{135}) = \frac{1}{135}$
2.3.2 $\tau_{F2} = min(\frac{1}{128}, \frac{1}{135}) = \frac{1}{135}$
2.4 $\tau_{F_{aggr}} = min(\frac{1}{135}, \frac{1}{135}) = \frac{1}{135}$
2.5 $\tau_{P3} = min(\frac{1}{135}, \frac{1}{135}) = \frac{1}{135}$
2.6 $\tau^{Wr}_{F3} = \frac{1000}{1000} \cdot \frac{1}{135} = \frac{1}{135}$

3.1 $\tau^{iso}_{C} = \frac{1}{114+10+0} = \frac{1}{124}$
3.2 $\tau^{Rd}_{F3} = \frac{1000}{1000} \cdot \frac{1}{124} = \frac{1}{124}$
3.3 $\tau_{F3} = min(\frac{1}{135}, \frac{1}{124}) = \frac{1}{135}$
3.4 $\tau_{F_{aggr}} = \frac{1}{135}$
3.5 $\tau_{C} = min(\frac{1}{135}, \frac{1}{124}) = \frac{1}{135}$
3.6 $\tau^{PPN}_{out} = \tau_{C} = \frac{1}{135}$

Figure 4.12: Throughput Estimation after merging $P1$ and $P2$

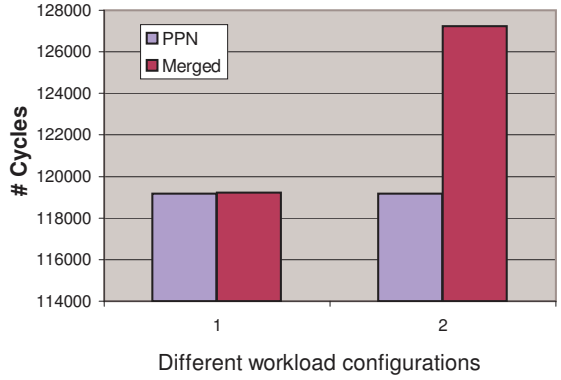

Figure 4.13: Measured Performance Results Before/After Merging $P1$ and $P2$

4.5.2 Merging Processes in Networks with Different Data Paths

In this experiment we consider the more complicated network shown in Figure 4.14 that combines different properties. First of all, it has processes with different domain sizes. Processes $P1$ and $P2$ execute 500 times, while the other processes execute 1000 times. As a result, coefficients will scale down the $F1$ and $F2$ FIFO read throughput. Second, two data paths come together in process $P3$ where one token

is needed per iteration of $P3$ similar to the example in Figure 4.8 B). Third, in process $P6$ two datapaths are joined as well where both tokens are needed for each iteration, similar to the example in Figure 4.8 C). We estimate the system through-

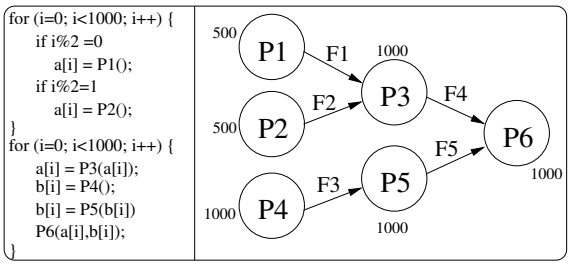

Figure 4.14: Nested-loop Program and its Derived PPN

put by applying Algorithm 1 again and test the throughput modeling with 3 different process workload configurations. Each configuration is a tuple where the first value corresponds to the workload of process P1, the 2nd value to workload of P2, etc. Figure 4.15 shows the measured performance results and for each configuration the initial PPN in Figure 4.14 is used as a reference (the first bar) and different mergings are shown in the 2nd, 3rd and 4th bars. For example, the second bar denotes the performance results after merging processes P1, P2 and P3. If we take the 2nd workload configuration as an example, our model finds the following throughputs: $\frac{1}{65}, \frac{1}{100}, \frac{1}{65}, \frac{1}{80}, \frac{1}{75}$. Thus, the estimation indicates that the first merging (i.e., $\frac{1}{100}$), leads to a lower throughput than the initial PPN (i.e., $\frac{1}{65}$). The second merging ($\frac{1}{65}$) gives the same performance results, and the third ($\frac{1}{80}$) and fourth ($\frac{1}{75}$) are worse than the initial PPN. From these estimations, we conclude that processes $P2$ and $P4$ can be merged and achieve the same system throughput. This estimation is correct as confirmed by the actual measured performance results shown in Figure 4.15.

4.6 Discussion and Summary

We have presented a solution approach for throuhgput modeling of Polyhedral Process Networks (PPNs) to evaluate process merging transformations. Our approach takes into account all major factors that influence the throughput. Therefore, we can accurately capture the throughput trend and select the best possible merging as illustrated with the experiments.

The throughput model defined in this chapter, requires the cost estimations of the process workloads and the FIFO communication primitives, similar to the process splitting transformation. Therefore, the same remark with respect to the modeling of the workload and FIFO communication with a constant value should be taken into

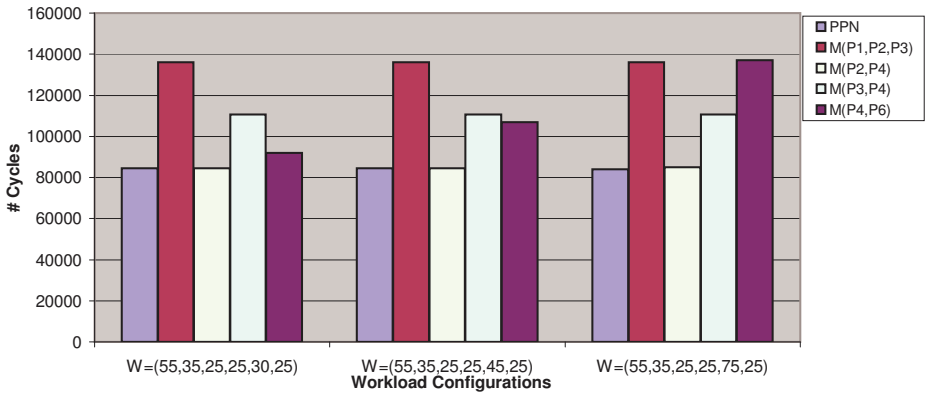

Figure 4.15: Measured Results on the ESPAM Platform

account. For an in-depth discussion, the reader is referred to Section 3.6.

Our throughput model calculates an average throughput for a given PPN, i.e., we do not take into account the dynamic behavior how output tokens are produced. This is best illustrated with the coefficient used in Formula 4.6 to determine the FIFO write throughput: the number of tokens written to a FIFO channel is divided by the total number of process iterations. However, the calculation of average throughput values allows efficient evaluation of the process merging transformations on the ESPAM platform, for two reasons. First, recall from Section 4.3 that the process workload is the same for all programmable cores in the target platform, i.e., we use a homogeneous MPSoC and assign the processes one-to-one to the cores. Second, also recall that we use buffer sizes that give maximum performance, which are calculated by the `pn` compiler. This is different in the work of [86], where the workload of a processor can vary as multiple processes can be assigned to that processor. To estimate buffer sizes and/or the system performance in this case, the dynamic behavior of the platform and application are important. In Section 1.3, we have indicated that this dynamic behavior is captured with maximum and minimum values of arrival/service curves. This throughput calculation is more complex than our approach, which we do not need for evaluating the process merging transformation on the ESPAM platform, because we assign the processes one-to-one and use buffer sizes that give maximum performance.

Chapter 5

Appling Transformations in Combination

In Chapter 3 we have discussed a compile-time approach for evaluating the *process splitting* transformation [51, 78, 79], and in Chapter 4 an approach for evaluating the *process merging* transformation [53]. These two parameterized transformations play a vital role in meeting the performance/resource constraints. The splitting transformation is parameterized in the sense that a given process can be split up in many different ways, and the designer must choose a specific splitting factor (i.e., the number of created copies). For the merging transformation, it is obvious that the designer must decide which processes to merge. The problem is that, for both transformations, the designer must select a particular process(es) to apply the transformations on in order to achieve good results. This is not a straightforward task as we explain in Section 5.2.2. In addition to this, both transformations can be applied one after the other and in a different order with different parameters which may, or may not, give better results than applying one transformation only. Therefore, in this chapter we

- investigate whether applying the two transformations in combination can give better performance results than applying only one,
- propose a solution approach that solves the very difficult problem of determining the best order of applying the transformations and the best transformation parameters,
- relieve the designer from the difficult task of selecting processes on which the applied transformations have the largest positive performance impact, and
- present a solution approach that exploits available data-level parallelism in cyclic PPNs and/or PPNs with stateful processes.

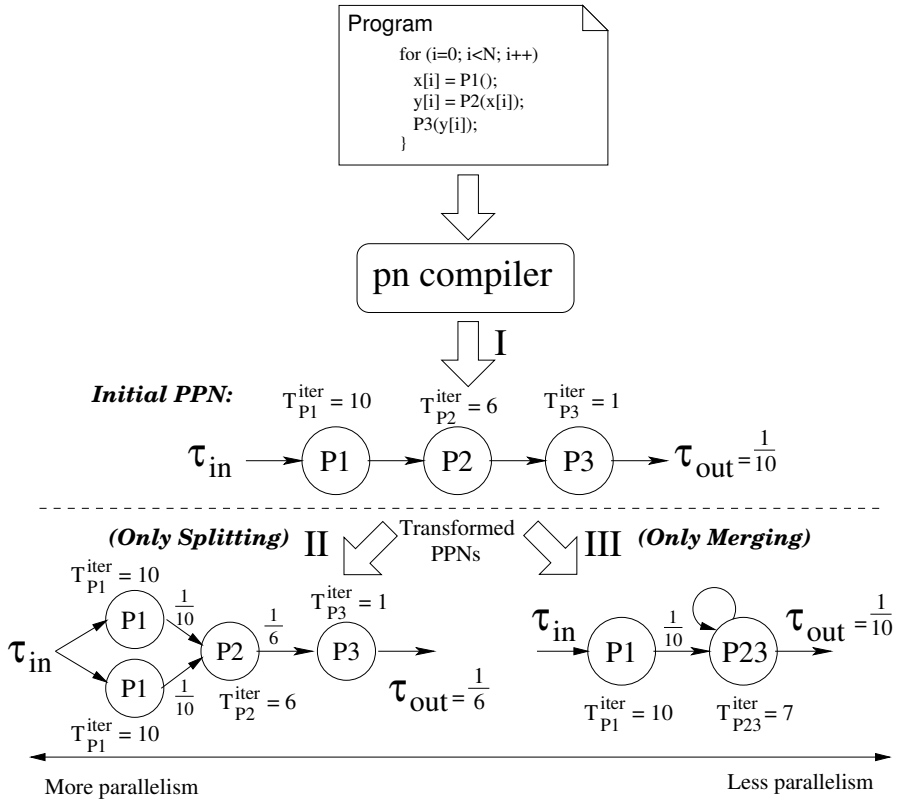

Figure 5.1: Deriving and Transforming Process Networks

In this Chapter, we apply the different transformations in combination on the initial PPN shown in Figure 5.1. Arrow II is an example of applying the process splitting transformation on process $P1$. The transformed network has two processes $P1$ executing the same function such that the data tokens are delivered twice faster to the consumer process $P2$. Recall from Chapter 3, that we refer to the two processes $P1$ as process partitions of $P1$. Arrow III is an example of transforming the initial PPN by applying the merging transformation on processes $P2$ and $P3$ to create compound process $P23$. The problem how to apply each transformation has been discussed in the previous chapters. However, still a remaining challenge is to devise a holistic approach to help the designer in transforming and mapping PPNs onto the available processing elements of the provided target platform to achieve even better performance results using the two transformations in combination. In the next section, we first investigate the effects on the performance results of applying both transformations in combination. Next, we propose a solution how to order them, and finally we present two case-studies.

5.1 Impact of the Transformation on Performance Results

We investigate whether applying both the process splitting and merging transformations in combination gives better performance results than applying only one transformation. Consider the initial and transformed PPNs in Figure 5.1. Each process P_i is annotated with the time $T_{P_i}^{iter}$ that is required to execute one process iteration, which includes the time for executing the process function and also the FIFO communication costs, see Definition 3.9. For example, a process iteration of $P1$ is completed in 10 time units, i.e., $T_{P1}^{iter} = 10$, while $P2$ is a computationally less intensive process as one process iterations is completed in only 6 time units, i.e., $T_{P2}^{iter} = 6$. Process $P1$ determines therefore the system throughput of the initial PPN. The throughput is denoted by τ_{out} and we define it as the number of tokens produced by the network per time unit (see Definition 18 in Section 4.2). Since $P1$ is the most computationally intensive process that fires each 10 time units, the throughput and number of produced tokens is $\frac{1}{10}$ tokens per time unit. Now we show and discuss many different examples in this section to illustrate how difficult it is for a designer to apply transformation, even for such a simple initial PPN as shown in Figure 5.1.

5.1.1 Transforming a PPN to Create More Processes

If we want to increase the performance results for a given PPN, the number of processes can be increased using the process splitting transformation to benefit from more parallelism. In this subsection we, therefore, show two different PPNs consisting of 4 processes that are derived from the same initial PPN consisting of 3 processes. The first transformed PPN is derived from the initial PPN in Figure 5.1 using only the process splitting transformation, and the second is derived from the initial PPN using both the process splitting and merging transformation.

Transformed $PPN1$ (only splitting)

We split up process $P1$ two times as shown in Figure 5.1. Then there are 2 processes that generate data in parallel for consumer process $P2$. As a result, process $P2$ receives its input data twice as fast. Therefore, we say that process $P2$ receives its data with an aggregated throughput of $\frac{1}{10}+\frac{1}{10} = \frac{1}{5}$. We know that the slowest process in a network determines the system throughput and to check this, we compare the incoming throughput of a process with the time it takes to fire that process function. While $P2$ receives its input data with a throughput of $\frac{1}{5}$ tokens per time unit, it can only produce tokens with a throughput of $\frac{1}{6}$ ($T_{P2}^{iter} = 6$). This means that the input tokens arrive faster than $P2$ can process them. To calculate the overall system throughput, we therefore propagate the throughput $\tau = \frac{1}{6}$ of $P2$ to sink process $P3$ and compare what is slower: the arrival of the input data, or the firing of process $P3$.

We see that $P3$ can process data much faster than it actually receives since $T_{P3}^{iter} = 1$, but still it produces tokens with a throughput of $\frac{1}{6}$ caused by the slowest process $P2$. The overall system throughput is therefore $\tau_{out} = \frac{1}{6}$ and is determined by $P2$. Thus, we have derived a PPN that gives a throughput $\tau_{out} = \frac{1}{6}$ that is much better than the initial throughput $\tau_{out} = \frac{1}{10}$.

Now we investigate whether we can derive another network with 4 processes, using both the process splitting and merging transformations in combination, that gives even better performance results than our previous example.

Transformed $PPN2$ (splitting+merging)

We apply first the process splitting transformation on processes $P1$, $P2$, and $P3$ from the initial PPN in Figure 5.1 to derive the transformed PPN shown in Figure 5.2 A). Two independent data paths are created each consisting of 3 processes.

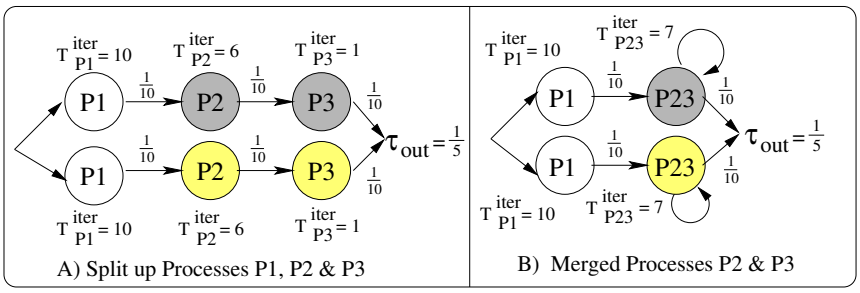

Figure 5.2: Transformed PPN2: Splitting and Merging to Create 4 Processes

In each data path, process $P1$ is the bottleneck process such that tokens are delivered with a throughput of $\frac{1}{10}$. Since there are two data paths, we say that the overall system throughput of the transformed PPN in Figure 5.2 A) is $\tau_{out} = \frac{1}{5}$. When we merge $P2$ with $P3$, process $P1$ remains the bottleneck and the throughput is unaffected as shown in Figure 5.2 B). Thus, we have derived a PPN with 4 processes that gives better performance results compared to the previous example *Transformed $PPN1$ (only splitting)* shown in Figure 5.1. That is, applying both transformations in combination achieves a throughput of $\tau_{out} = \frac{1}{5}$, while applying only the process splitting transformation gives a throughput $\tau_{out} = \frac{1}{6}$. In fact, to create a PPN with n processes from the initial PPN in Figure 5.1, the best performance results that can be achieved by using the process splitting transformation only, will never be better than the best performance results that can be achieved by applying both transformations in combination. Therefore, this example shows that both transformations must be used in combination to achieve better performance results.

5.1.2 Transforming a PPN to Reduce the Number of Processes

A designer sometimes needs to reduce the number of processes for a given PPN in order to meet resource constraints. Another reason to merge processes, is that in some cases the same performance can be achieved using less processes. In this subsection, our objective is to derive a PPN consisting of 2 processes when this is required for one of the two reasons mentioned above. We start with the initial PPN in Figure 5.1 that has 3 processes and investigate again whether the combination of applying the transformations is important when the number of processes in the network must be reduced.

Transformed $PPN3$ (only merging)

A transformed PPN with 2 processes is shown in the bottom right part of Figure 5.1, which is obtained by applying only the process merging transformation. The resulting network has the same throughput as the initial PPN, but uses one process less. By merging 2 light-weight processes $P2$ and $P3$, process $P1$ remains the most computationally intensive process. As a result, the system throughput remains the same as in the initial network, i.e., $\tau_{out} = \frac{1}{10}$.

Transformed $PPN4$ (splitting+merging)

An alternative using both the process splitting and merging transformations is shown in Figure 5.3.

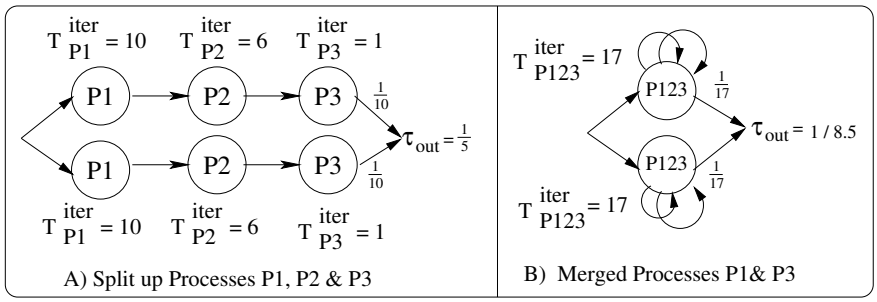

Figure 5.3: Transformed PPN4: Creating 2 Load-Balanced Tasks

All processes are first split up two times as shown in Figure 5.3 A). Then, two compound processes are created by merging a process partition of each process into a compound process $P123$ as shown in Figure 5.3 B). The time for one process iteration of the compound process is $T^{iter}_{P123} = T^{iter}_{P1} + T^{iter}_{P2} + T^{iter}_{P3} = 17$ time units, because all process functions are executed sequentially. This means that the compound

process delivers tokens with a throughput of $\tau_{P123} = \frac{1}{17}$. Since we have 2 compound processes, the resulting overall throughput is $\tau_{out} = \frac{1}{8.5}$, which is better than the throughput $\tau_{out} = \frac{1}{10}$ of our previous example *Transformed $PPN3$ (only merging)* shown in Figure 5.1. This is another example which shows that both transformations should be applied in combination to obtain better performance results, which cannot be obtained by only one transformation (i.e., the merging transformation in this case).

5.1.3 The Optimization Pitfall: Performance Degradation

We have shown that there is great potential in using both transformations in combination, but a designer should be very careful how the transformations are applied, otherwise performance degradation may be encountered. We illustrate this with two examples using both the process splitting and merging transformations. First we show an example for a PPN with 4 process and then for a PPN with 2 processes.

Transformed $PPN5$ (splitting+merging)

We start with the initial PPN in Figure 5.1, which consists of 3 processes, and split up both processes $P1$ and $P2$ to obtain the PPN shown in Figure 5.4 A).

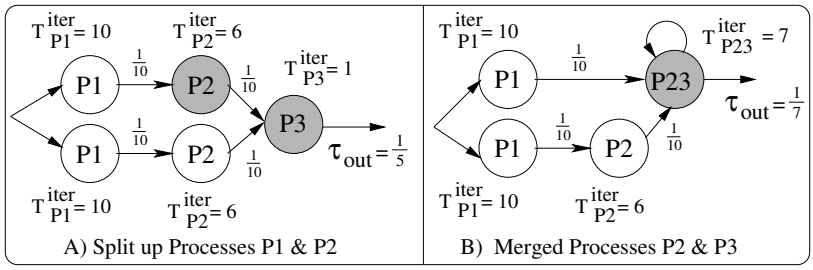

Figure 5.4: Transformed PPN5: Splitting and Merging to Create 4 Processes

The network has a throughput of $\frac{1}{5}$ using 5 processes, while our objective is to use 4 processes. Therefore, we merge two light-weight processes $P2$ and $P3$. The created compound process $P23$ has a process iteration time $T^{iter}_{P23} = 7$ time units and is the bottleneck process of the network. The overall system throughput is, therefore, determined by $P23$ and is $\tau_{out} = \frac{1}{7}$. In this way, we have derived another PPN with 4 processes that performs better than the initial process network ($\tau_{out} = \frac{1}{10}$). However, the throughput $\tau_{out} = \frac{1}{7}$ is worse than the throughput achieved by applying only the splitting transformation, i.e., *transformed $PPN1$ (only splitting)* in Figure 5.1 with a throughput of $\tau_{out} = \frac{1}{6}$ and subsequently also worse than *Transformed $PPN2$* shown in Figure 5.2 B) that has a throughput $\tau_{out} = \frac{1}{5}$.

Transformed $PPN6$ (splitting+merging)

We have shown two examples to transform the initial PPN in Figure 5.1 into a PPN with 2 processes in *Transformed $PPN3$* and *Transformed $PPN4$*. Both give good performance results, but now we give an example of a PPN that performs worse. Another possibility to create a PPN with 2 process is to first split up the computationally most intensive process $P1$ as shown in Figure 5.5 A). Then, two compound processes

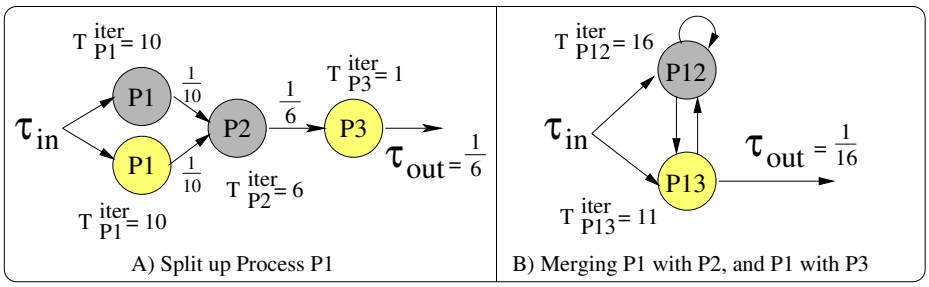

Figure 5.5: Transformed PPN6: Splitting and Merging to Create 2 Processes

are created, one by merging process $P1$ with $P3$, and the other one by merging process $P1$ and $P2$. We see that a topological cycle is introduced by merging processes in this way and we find that the system throughput is $\tau_{out} = \frac{1}{16}$ tokens per time unit. This result is worse than *Transformed $PPN3$* and *Transformed $PPN4$* that have a throughput of $\tau_{out} = \frac{1}{10}$ and $\tau_{out} = \frac{1}{8.5}$, respectively.

In this section, we have shown that it is necessary to apply both the process splitting and merging transformations in combination to achieve better performance results that cannot be achieved by applying only one transformation in isolation. On the other hand, performance degradation may be encountered if the transformations are not applied properly. So the question is how a designer should apply the transformations properly, i.e., choosing the best possible order of transformations and their parameters. In the next section, we show our solution approach that addresses these issues.

5.2 Compile-Time Solution for Transformation Ordering

Before introducing our solution in a more formal way, we show how our approach intuitively works for the examples discussed in Section 5.1. We have already shown 3 different PPNs consisting of 4 processes that were derived from the same initial PPN. The first transformed PPN is obtained by using only the splitting transformation as shown in Figure 5.1. In two other examples, shown in Figure 5.2 B) and

Figure 5.4 B), different networks were obtained by consecutively using the process splitting and merging transformations. Our solution approach, however, gives a different solution and also gives better performance results as we show with the examples in Figure 5.6.

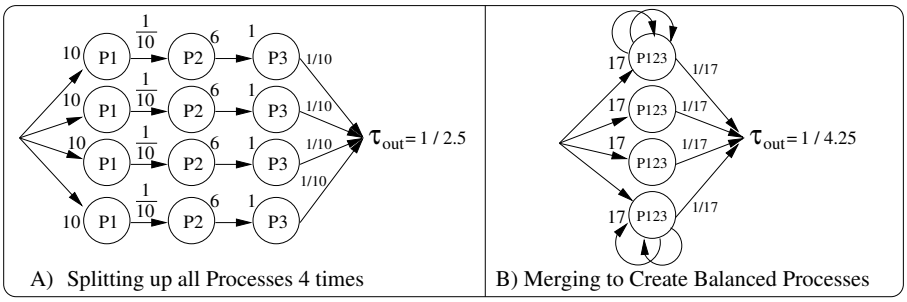

Figure 5.6: Creating 4 Load-Balanced Tasks

In our simple, elegant but yet very effective solution approach, we first split up all processes with a splitting factor that is specified by the designer. This splitting factor can, for example, be the number of available processing elements of the target platform, or simply the number of tasks the designer wants to create. Since in our examples the goal is to transform and create a PPN with 4 processes, we split up all processes 4 times as shown in Figure 5.6 A). In this way, we create a PPN consisting of 12 processes. Next, we merge back process partitions into compound processes such that they contain one process partition of each process. Figure 5.6 B) shows these compound processes $P123$. Note that the self-edges for two compound processes have been omitted for the sake of clarity. The time to execute one process iteration of the compound processes is 17 time units, which is obtained by summing the process iteration time of the individual processes. Thus, we know that each compound process produces $\frac{1}{17}$ tokens per time unit. Since there are 4 compound processes, the overall system throughput $\tau_{out} = \frac{4}{17} = \frac{1}{4.25}$, which is better than all other transformed PPNs with 4 processes shown in Figure 5.1, Figure 5.2 B), and Figure 5.4 B).

The initial PPN in Figure 5.1 is transformed in a similar way if the number of processes needs to be reduced. We have already shown 2 examples and our solution is already given in Figure 5.3; all processes are first split up 2 times, and then compound processes are created by merging different process partitions such that the resulting transformed network consists of 2 processes.

5.2.1 Creating Load-Balanced Tasks

While we illustrated our solution approach with examples in the previous section, a more formal description of our solution approach is given with the pseudo-code in Algorithm 2. We create a number of tasks from an initial PPN based on the combination of two transformations: *i)* the processes are split-up first, and *ii)* load-balanced tasks are created by using the process merging transformation.

Algorithm 2 : Task Creation Pseudo-code

Require: A Polyhedral Process Network PPN with n processes,
Require: A process splitting factor u.
  for all $P_i \in PPN$ **do**
    $\{P_{i1}, P_{i2}, .., P_{iu}\} = split(P_i, u)$
  end for
  for $i = 1$ to u **do**
    $P_{Ci} = merge(\{P_{1i}, P_{2i}, .., P_{ni}\})$
  end for
  return all compound processes P_{Ci}

Algorithm 2 uses two functions: `split` and `merge`. For the former, we refer to Chapter 3 in which it is shown that a process can be split up in many different ways and how to select the best splitting. We use the approach in Chapter 3 to select and perform the processes splitting. For the process merging transformation, we rely on the approach described in Chapter 4. We add to this approach a procedure to cluster producer-consumer pairs of processes. By clustering producer-consumer processes, communication between these processes stays within one compound process after merging. Thus, it tries to avoid communication and synchronization of different compound processes. An example of this is given in Figure 5.6. One process partition of $P1$ has only one channel to $P2$, which in turn has only one channel to $P3$. Merging processes in this sequence results in compound processes that do not have any communication channels among them. It is not always possible to obtain completely independent compound processes. If one producer process has multiple channels to consumer processes, as shown in Figure 5.7 A), one particular consumer has to be selected and merged with the producer.

If we start with the first partition of $P1$, i.e., grey process $P1$ in Figure 5.7 A), then we see that it has two outgoing channels to two process partitions of $P2$. Regardless which partition of $P2$ is chosen for merging, the resulting compound processes will have channels for data communication between them as shown in Figure 5.7 B). In our approach, we simply consider the first outgoing channel and corresponding consumer process, and merge it with the producer. We mark this consumer as being merged already, to avoid that it will be selected again.

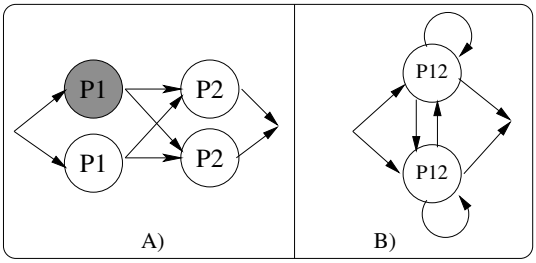

Figure 5.7: Different Merging Options

5.2.2 Selecting Processes for Transformations

Our solution approach in Section 5.2.1 solves another problem indicated in the introduction of this chapter, i.e., how to select processes to be transformed on which the transformations have the largest positive performance impact. For the process splitting, it is important to find the bottleneck process of the network, because splitting is the most beneficial when applied on the bottleneck process. For process merging, it is important to avoid merging the bottleneck process, i.e., not introducing an even larger bottleneck process. In general, however, it is not possible at all to determine a single bottleneck process. The reason is that, in PPNs, different data paths can transfer a different number of tokens. As a result, different processes can determine the overall system throughput at different stages during the execution of the network, which we illustrate with the example shown in Figure 5.8.

The network has two datapaths $DP1 = (P1, P2, P3, P6)$ and $DP2 = (P1, P4, P5, P6)$ that transfer a different number of tokens. This is the result of the communication patterns `[1100000]` and `[0011111]` at which process $P1$ writes to its outgoing FIFO channels. A "`1`" in these patterns indicates that data is read/written and a "`0`" that no data is read/written. So, the FIFO channel connecting $P1$ and $P2$, for example, is written the first two firings of $P1$, but not in the remaining 5 firings. As a consequence of these patterns, more tokens are communicated through the second datapath $DP2$. At the bottom of Figure 5.8, the different time lines of the processes are shown. Each block corresponds to a firing of that process producing data, and the arrow indicates the dependent consumer process. In this way, a full simulation of the process network is shown. We observe that, despite process $P2$'s largest process iteration time $T_{P2}^{iter} = 10$ time units, process $P4$ with $T_{P4}^{iter} = 6$ is determining the throughput most of the time. This illustrates that, in general, due to the varying and possibly complicated communication patterns, it is not possible to decide which process to split up for a more balanced network. Our solution approach in Section 5.2.1, solves this problem as the transformations are applied on all

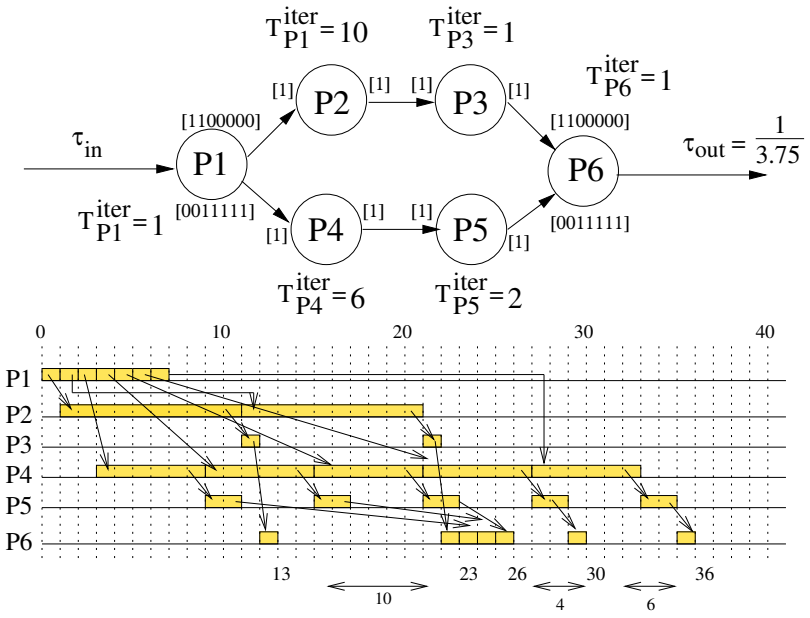

Figure 5.8: What is the Bottleneck Process: $P2$ or $P4$?

processes and, therefore, it is not necessary to select particular processes.

5.3 Exploiting Data-Level Parallelism

The idea of our approach presented in Section 5.2 is to create load-balanced tasks that exploit *data-level parallelism* as much as possible. In this section, we want to show that our simple solution always results in performance gains when there is data-level parallelism to be exploited. The degree of data-level parallelism that can be exploited is determined by:

1. Processes with *self-edges* in a PPN. Similar to the definition used in [31], we refer to data-level parallelism when processes do not dependent on previous firings of itself. Obviously, when there is no self-edge, the process is *stateless* and an arbitrary number of independent process partitions can be created that run in parallel. When a process has a self-edge, however, it produces data for itself and there exists a dependency between different firings of that process. Then, we refer to such a process as *stateful*.

2. *Cycles* in a PPN. A cycle can be responsible for sequential execution of the processes involved in the cycle. If this is the case, we call it a *true cycle*.

Despite stateful processes and topological cycles, PPNs may still reveal some data-level parallelism which is exploited by our solution approach. This means that our solution approach gives better performance results when there is data parallelism to be exploited, and the same performance as the initial PPN if there is nothing to be exploited. In addition to cycles and stateful process, the workload balancing of the initial PPN is another important factor that determines whether performance gains are possible. We therefore first discuss this workload balancing before we elaborate how to exploit more data-level parallelism for stateful processes and cyclic PPNs.

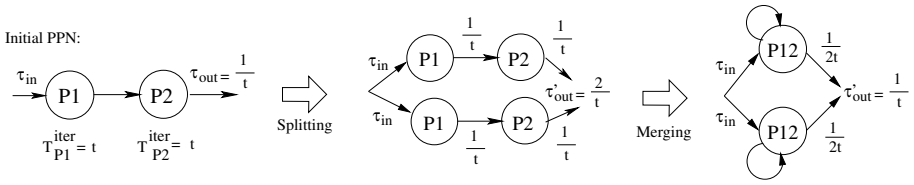

Figure 5.9: Simple Acyclic Producer/Consumer

Balanced PPNs

Let us consider the PPN shown in Figure 5.9 and its two processes $P1$ and $P2$.

- The PPN and its processes $P1$ and $P2$ shown in Figure 5.9 are balanced, because $T_{P1}^{iter} = T_{P2}^{iter} = t$ time units. The throughput of the PPN is therefore $\tau_{out} = \frac{1}{t}$. If we apply splitting and merging, as illustrated with the arrows in Figure 5.9, then a compound process has a throughput of $\tau = \frac{1}{2t}$. Since there are two compound processes the overall throughput is $\tau'_{out} = 2 \cdot \frac{1}{2t} = \frac{1}{t}$. Thus, we see that the new throughput τ'_{out} is the same as the throughput of the initial PPN, that is, $\tau'_{out} = \tau_{out}$.

Now let us consider the other case:

- Suppose that the PPN in Figure 5.9 and its processes $P1$ and $P2$ are imbalanced, then we have $T_{P1}^{iter} = t$ and $T_{P2}^{iter} = t + x$, where $x > 0$. The throughput of the initial PPN is $\tau_{out} = \frac{1}{t+x}$. Then, we apply our solution approach and create 2 independent streams. Each compound process has a throughput of $\tau = \frac{1}{T_{P1}^{iter} + T_{P2}^{iter}} = \frac{1}{2t+x}$. Since we have 2 parallel streams, the throughput is $\tau'_{out} = \frac{2}{2t+x}$. If we want to know when splitting and merging is worse compared to the initial PPN, then we have: $\frac{2}{2t+x} < \frac{1}{t+x}$. From this inequality it follows that $x < 0$, which contradicts with the assumption that the

network is imbalanced, i.e., $x > 0$. Thus, the new throughput is the same or better than the initial PPN, i.e., $\tau'_{out} \geq \tau_{out}$.

We have shown that $\tau'_{out} = \tau_{out}$ when the initial network is already balanced and $\tau'_{out} \geq \tau_{out}$ when this is not the case. In other words, applying our approach results in performance gains when there is something to be gained by load balancing. Next, we discuss how our approach exploits data-level parallelism for PPNs with cycles and/or stateful processes.

5.3.1 Stateful Processes

When a stateful process is split up, then the different process partitions must communicate data as a result of a dependency between different process iterations. The question whether partitions of a split up process have overlapping executions or not depends on the *distance*, in terms of a number of process firings, between data production and consumption. If data is produced by a process for the next firing of the same process (i.e., the distance is 1), then there is no data-level parallelism to be exploited and splitting such a process results in sequential execution of the process partitions. However, when the distance is larger than 1, then some copies of that process have some data parallelism that can be exploited by the process splitting transformation. If, for example, the distance between data production and consumption is 5, then 5 process firings can be done in parallel before communication and synchronization is required again. Applying our solution approach, splits up all processes first. As a result, the same functions are executed by several process partitions. The necessary FIFO communication channels are automatically derived in case the split up processes are stateful. In this way, the different process partitions overlap their firings when this is allowed by the self-dependences, i.e., the dependence distance is larger than 1, and synchronize their firings when necessary.

5.3.2 Cycles

For transforming processes that form a topological cycle, it is important to realize that the process splitting and merging transformations do not re-time any of the process firings. This means that the process firings are not re-scheduled, but only assigned to different process partitions. Therefore, a cycle present in the initial PPN, will *not* be removed by our approach and the transformed PPN will have a cycle as well. The behavior of the cycle is the most important factor that determines whether performance improvements are possible or not and we illustrate this with 3 different examples in Figure 5.10. There are 2 extremes: the first is a true cycle for which nothing can be gained, and the second is a doubling of the throughput by creating 2 independent streams. A third example shows a network that gives performance results between

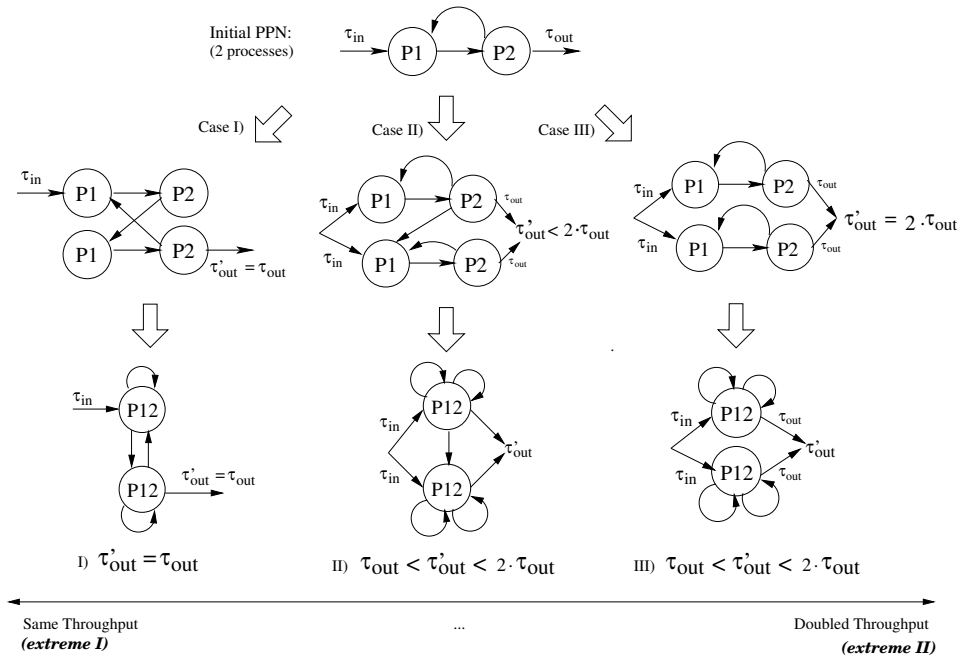

Figure 5.10: Throughput Possibilities after Splitting a Cycle 2 Times

the two extremes. For the three examples in Figure 5.10, we discuss how: *i)* the initial load balancing, and *ii)* the inter-process dependencies after splitting play a role on the performance results.

Extreme I (same throughput): We already mentioned that for true cycles all processes involved in such a cycle execute sequentially. That is, data is typically read once from outside the cycle and then data is produced/consumed for/from processes belonging to that cycle. For the initial PPN in Figure 5.10, this can mean that $P1$ reads from its input channel once, and then produces/consumes from the 2 channels to/from $P2$. If $P1$ injects a data token in the cycle in one firing and reads a token from the feedback channel in the next firing, then processes $P1$ and $P2$ execute in a pure sequential way. It is clear that for this type of cycles, performance gains are not possible. Applying our solution approach on a true cycle, as shown with `Case I` in Figure 5.10, gives the same performance results as the initial PPN. The reason is that after splitting, the cycle is present as a path connecting $P1$, $P2$, $P1$, $P2$, $P1$, and after merging this sequential firing sequence is not changed as the dependencies and sequential execution do not allow any overlapping executions.

Extreme II (doubled throughput): Another extreme is a transformed network with independent data paths. The initial PPN from which this transformed PPN is derived,

is topologically the same as the initial PPN in `Case I`, but the behavior is different, i.e., it is not a true cycle because *P1* injects first, for example, at least 2 tokens before reading data from the cycle. Thus, depending on the behavior of the cycle, splitting processes can result in different paths where the cycle connects only processes in the same path. In other words, independent streams can be created as illustrated with `Case III` in Figure 5.10. This can easily happen when we split processes, for example, 2 times such that the even executions of that process are assigned to one process partition, and the odd executions to another partition. If the cycle and thus the dependent producer and consumer executions are from even to even executions and from odd to odd executions, then the communication remains local to one data path as shown in `Case III` of Figure 5.10. This is an example of a cyclic PPN that has the potential to scale linearly with the number of created streams. Having a transformed PPN with independent data paths, however, does not automatically mean that performance gains are possible. Besides the dependencies as we have just discussed, the workload balancing of the initial PPN is another important factor. For our example with the 2 independent data paths, it can still happen that the same throughput as the initial network is achieved, i.e., $\tau'_{out} = \tau_{out}$, when the initial network is already perfectly balanced. That is, for a network that is already balanced, there is nothing to be gained with load-balancing. On the other hand, when the two processes are highly imbalanced, then a doubling of the throughput can be approached.

Between the 2 Extremes: The last case to be discussed from Figure 5.10, is `Case II` that gives performance results between the two extremes as discussed above. After splitting and merging, the compound processes are connected with one communication channel. Depending on how many times synchronization and data communication occurs between the compound processes, the performance results can be the same as for a true cycle (i.e., sequential execution), or the performance results can approach a doubling of the throughput if synchronization does not play a role as, for example, data is communicated only once.

5.4 Case-Studies

To illustrate that our approach works for PPNs with stateful processes and cycles, we consider 2 different algorithms and implement their initial PPN and transformed PPNs onto the ESPAM platform prototyped on a Xilinx FPGA [60], [61]. We measure the performance results to check that indeed the maximum performance gains are obtained allowed by inter-process dependencies. First, we focus on the QR algorithm, which is a matrix decomposition algorithm that is interesting as the compute processes have self-edges (stateful processes) and, in addition to this, the PPN is cyclic. Second, we consider a simple pipeline of processes and we show that our ap-

proach is as good as the initial network if the network is already perfectly balanced.

5.4.1 QR Decomposition: a PPN with Stateful Processes and Cycles

A QR decomposition of a square matrix A is a decomposition of A as $A = QR$, where Q is an orthogonal matrix and R is an upper triangular matrix. Our implementation and corresponding PPN is shown in Figure 5.11 A). It consists of 2 source processes, 1 sink process, and 2 compute processes denoted by V and R. This network is highly imbalanced as process R fires more times and is also computationally more intensive than V. Applying the process splitting transformations on processes V and R gives as a result the network shown in Figure 5.11 B). We apply our solution approach and merge process partitions of V with R (and not V with V) to create compound processes $VR1$ and $VR2$. We do this by considering first one partition of V in the network and see that it has outgoing FIFO channels to another partition of V and to one partition of R. These two process partitions are merged and in a similar way the second compound process is created. The final result and transformed PPN is shown in Figure 5.11 C). In all our experiments, we assume that source and sink processes cannot be transformed. The reason is that, for example, these processes read and write data from/to a memory location, which can only be done by one process sequentially and, thus, not by multiple processes in parallel.

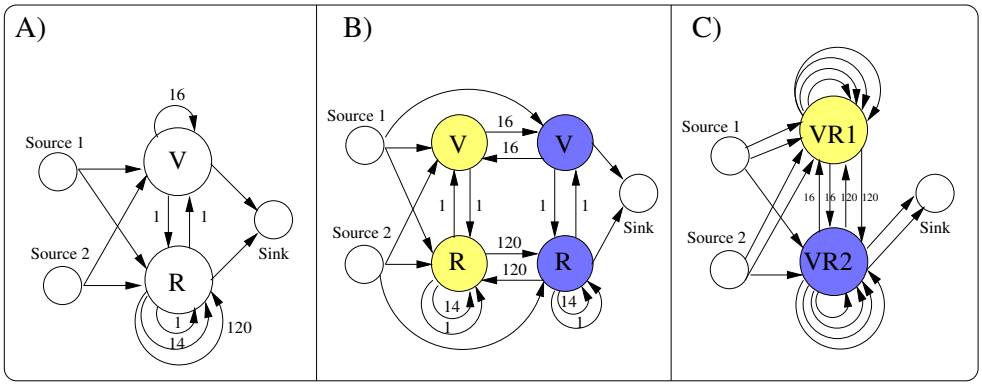

Figure 5.11: A) Intial PPN for QR Decomposition Algorithm, B) PPN with split up processes V and R, and C) load-balanced PPN with compound processes.

The resulting network is perfectly balanced. To implement the network, we apply a one-to-one mapping of processes to processors and thus 5 processors are used in total. To be more specific, the processes are executed as software routines on soft-core MicroBlaze processors, which are point-to-point connected. Figure 5.12 shows the corresponding measured performance results on the ESPAM platform [60], [61],

prototyped on a Xilinx FPGA. The source and sink processes both finish one process iteration in only 1 time unit, while the compute processes V and R are the computationally intensive processes which need respectively 100 and 450 time units for one process iteration.

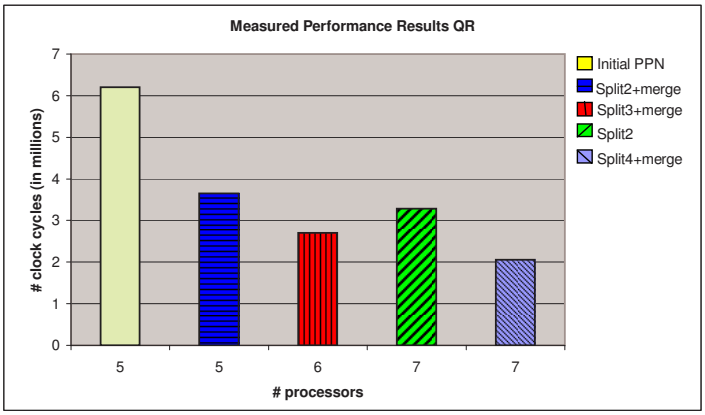

Figure 5.12: Measured Performance Results of QR on the ESPAM Platform

The first bar serves as our reference point and it corresponds to the performance results of the initial PPN shown in Figure 5.11 A). The QR network needs around 6 million cycles to finish its execution and uses 5 processors. For the same number of processors, our transformation approach gives much better performance results as shown by the second bar; the compute processes are split-up 2 times and different partitions are merged, which is denoted by *split2+merge* and shown in Figure 5.11 C). When we apply our approach and create 3 compound processes, denoted by *split3+merge*, then we even further improve performance results using 6 processors as shown by the third bar. Next, we compare the results of applying only the process splitting transformation, denoted by *split2* and shown in Figure 5.11 B), with our approach of splitting processes 4 times and merging different process partitions into compound processes, denoted by *split4+merge*. Both experiments use 7 processors and the 4th and 5th bars show the corresponding performance results. It can be seen that creating balanced partitions gives better performance results than applying only the splitting transformation. Note that the initial PPN with 5 processors executes mostly in a sequential way, i.e., no data-level parallelism is exploited. By applying our approach, i.e., splitting the compute processes 2, 3, and 4 times, we exploit data level parallelism and achieve speed ups of 1.7, 2.3, and 3, respectively.

The QR algorithm is an example of `Case II` in Figure 5.10. The self-edges in Figure 5.11 A) are annotated with their minimum buffer size capacity as computed by the `pn` compiler [95]. Process V, for example, has a self-channel that should

have a capacity of at least 16 tokens to avoid a deadlock. This means that 16 tokens are produced and buffered before they are finally consumed by the same process: 16 firings of that processes could be done in parallel before data communication and synchronization are required again. We showed results for splitting up the stateful processes 2, 3, and 4 times in the experiments. After applying our approach, we see in Figure 5.11 C) that the self-channels appear as the channels connecting the compound processes. These observations make clear that the cycles are not true cycles as we have discussed in the previous section and that there is data-level parallelism to be exploited by applying our solution approach. This is, indeed, confirmed by the measured performance results. Our approach almost scales linearly by increasing the number of compound processes (2nd, 3rd, and 5th bars in Figure 5.12) compared to the initial PPN, indicating that we exploit all available data-level parallelism.

5.4.2 Transforming Perfectly Balanced PPNs

We have shown that stateful processes and cycles in PPNs restrict data-level parallelism and thus influence performance results. In this section we show that the process workload, and thus the process iteration time $T_{P_i}^{iter}$, is another aspect that should be taken into account. To illustrate this, we consider a simple PPN consisting of a pipeline of 4 processes. The goal of this experiment is to verify that our approach, compared to applying only the process splitting transformation, does not give worse performance results for PPNs that are already balanced. To check this, we generate the following 4 PPNs as also shown in Figure 5.13: *i)* the initial PPN, *ii)* a PPN with process $P2$ split up 2 times, *iii)* a PPN with processes $P2$ and $P3$ split up 2 times and different partitions merged, and *iv)*, a PPN with processes $P2$ and $P3$ split up 3 times and different partitions merged.

For each process network, we vary the workload of process $P3$ and assign 4 different values. As a result, the process iteration time T_{P3}^{iter} is 1, 50, 75, and 100 time units. This means that process $P2$ is the bottleneck when T_{P3}^{iter} is 1, 50, and 75 time units. By increasing it to 100, both $P2$ and $P3$ are equally computationally intensive. Recall that we do not transform source and sink processes $P1$ and $P4$ in our experiments. We therefore say that the network is *imbalanced* when T_{P3}^{iter} is 1, 50, or 75 time units, and *balanced* when we choose T_{P3}^{iter} to be 100. We expect that:

- The more balanced the network becomes by increasing the workload of $P3$, the less is gained by splitting only process $P2$ two times (network II in Figure 5.13);
- Our transformation approach (network III in Figure 5.13) gives better performance results when the network is imbalanced;

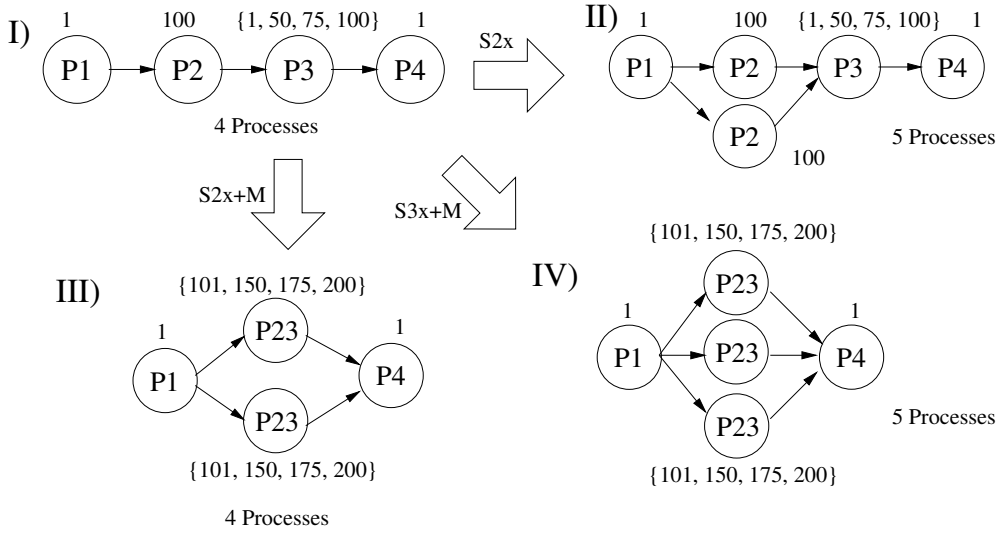

Figure 5.13: Splitting vs. "Splitting+Merging" with Different Workloads

- Our approach can even achieve better results by creating more than 2 compound processes (network IV in Figure 5.13), while this is not possible using the same number of processors and thereby applying only the process splitting transformation.

We make 2 comparisons and measure the performance results on the ESPAM platform of PPNs with an equal number of processes, i.e., PPNs with 4 processes and PPNs with 5 processes. First, we compare the initial PPN (i.e., network `I` in Figure 5.13) with the network on which process splitting and merging has been applied (i.e., network `III` in Figure 5.13). Second, we compare network `II` with network `IV` from Figure 5.13.

Figure 5.14 shows the measured performance results for the 2 different PPNs with 4 processes. The x-axis shows the different T_{P3}^{iter} configurations when the workload of process $P3$ is increased, and the y-axis the corresponding cycles counts. Because we map the processes one-to-one onto processors, there are 4 processors used in this experiment. For each workload configuration, the first bar corresponds to process network `I` in Figure 5.13 and the second bar to process network `III`. The initial PPN gives the same performance results for all different workload configurations as the overall throughput is $\tau_{out} = \frac{1}{100}$ determined by process $P2$. Our approach gives better results for unbalanced networks. However, as the workload of process $P3$ is increased, the network becomes more balanced and less can be gained by transformations targeting the same number of processors. Figure 5.14 shows that

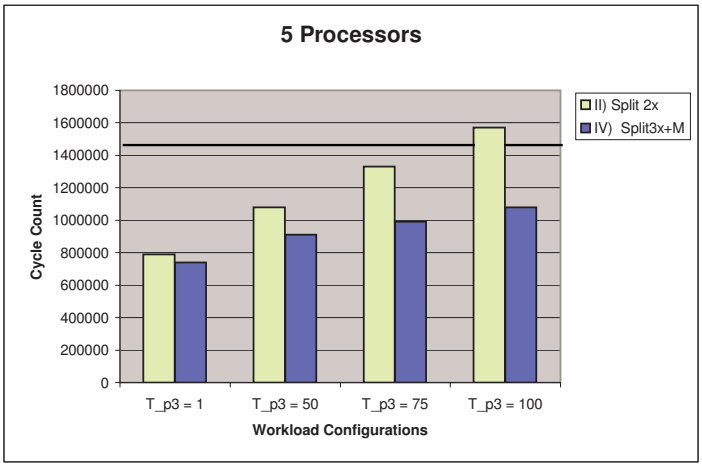

Figure 5.14: Initial PPN (PPN I) vs. Split2x + Merging (PPN III)

the difference between the initial PPN and the transformed PPN becomes smaller. The last 2 bars show the results for the PPNs where the initial network is already balanced, i.e., $T_{P3}^{iter} = 100$. It can be seen that our approach is slightly worse than the initial PPN, although the difference is not significant as it is only 2% off. The reason is that the transformations introduce a small overhead in the compound processes, which consist of additional control to execute the different functions. In the ideal case when there is no overhead, the throughput of one compound process is $\frac{1}{200}$ and thus the aggregated throughput of both compound processes is $\frac{1}{100}$, which is the same as the initial PPN. Due to the additional control, however, the process iteration time is not $T_{P23}^{iter} = 200$, but a little bit higher which finally results in the minor and not significant performance degradation. The ratio of the workload and the control overhead is important for the actual overhead and performance degradation. In our experiments, the workload of the compound processes is 200 assembly instructions. In most applications however, the process workload will be much larger such that the overhead subsequently will have less impact on the performance results and will be negligible (i.e., less than 2%).

Figure 5.15 shows the comparison between PPNs with 5 processes. That is, we compare our solution approach that splits up all processes 3 times and merges back different partitions, with applying only the process splitting transformation. For each workload configuration, the first bar corresponds to network `II` in Figure 5.13, and the second bar to network `IV`. The bold horizontal line in Figure 5.15 is the reference corresponding to the performance results of the initial PPN.

We see that applying only process splitting for process $P2$ is less beneficial as the

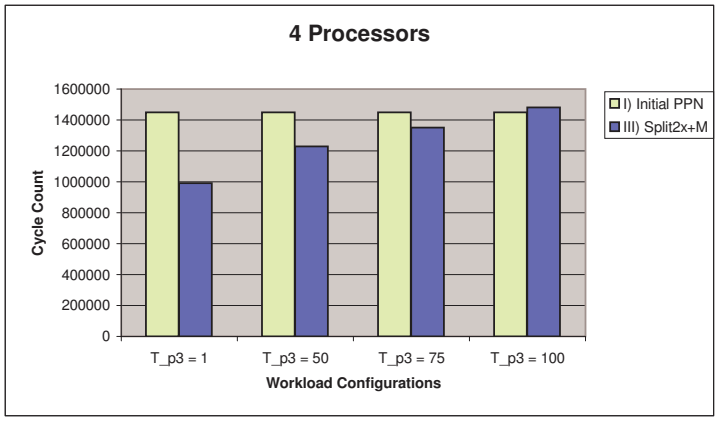

Figure 5.15: "Splitting 2x" (PPN II) vs. "Splitting 3x + Merging" (PPN IV)

network becomes more balanced as illustrated with the $1^{st}, 3^{th}, 5^{th}$, and 7^{th} bars. When the network is balanced, i.e., the 7^{th} bar, the performance results are a bit worse than the initial PPN due to some additional control introduced by the transformations as discussed before. For splitting and merging the processes 3 times, however, we see that better performance results are obtained as illustrated with the $2^{nd}, 4^{th}, 6^{th}$, and 8^{th} bars in Figure 5.15. The reason is that 3 balanced compound processes execute as 3 independent streams in parallel. Each compound process delivers tokens with a throughput of $\frac{1}{200}$ (when the time for one process iteration of processes $P2$ and $P3$ is 100 time units). The overall system throughput is therefore $\tau_{out} = \frac{3}{200} \approx \frac{1}{67}$. If only $P2$ is split up, then the overall system throughput will be determined by $P3$ and remains $\tau_{out} = \frac{1}{100}$. We see that our approach gives better performance results for all workload configurations. By increasing the workload and thus also T_{P3}^{iter}, the cycle count goes up, but not as steep compared to applying only the process splitting. In addition, our approach would also scale for more than 5 processors, as an arbitrary number of independent streams can be created.

5.5 Discussion and Summary

We have shown that better performance results are obtained when both the process splitting and merging transformations are applied in combination, as opposed to applying only one of these transformations. Furthermore, we have shown that it is very difficult to identify a single bottleneck process in a PPN, since there can be many different bottleneck processes during the execution of a PPN. Our approach solves the problem of selecting a process on which the transformations have the largest impact,

as first all processes are split up and then perfectly load-balanced compound processes are created using the process merging transformation. Furthermore, we have shown that our approach also works for process networks with cycles and stateful processes. If in the initial PPN there is data-level parallelism to be exploited, then our approach gives better performance results compared to the initial PPN by exploiting this parallelism to the maximum. The same performance results are obtained when no data-level parallelism is available in the initial PPN.

After applying our solution approach a designer may end up with a transformed PPN which performance is the same as the initial PPN. As already explained, the reason can be that the initial PPN is already perfectly balanced, or cycles can be present in the PPN that restrict the data-level parallelism. If we focus on cyclic PPNs, then we know that performance gains are not possible when processes involved in a true cycle are split up. This makes it clear that it is desired to indicate to the designer when a PPN contains a true cycle. Therefore, we sketch an approach how true cycles can be detected, i.e.,

- we investigate if the number of input tokens that the processes read from outside the cycle can serve as a metric to detect true cycles.

We consider the two example PPNs shown in Figure 5.16, which are different in the number of tokens read from outside the cycle.

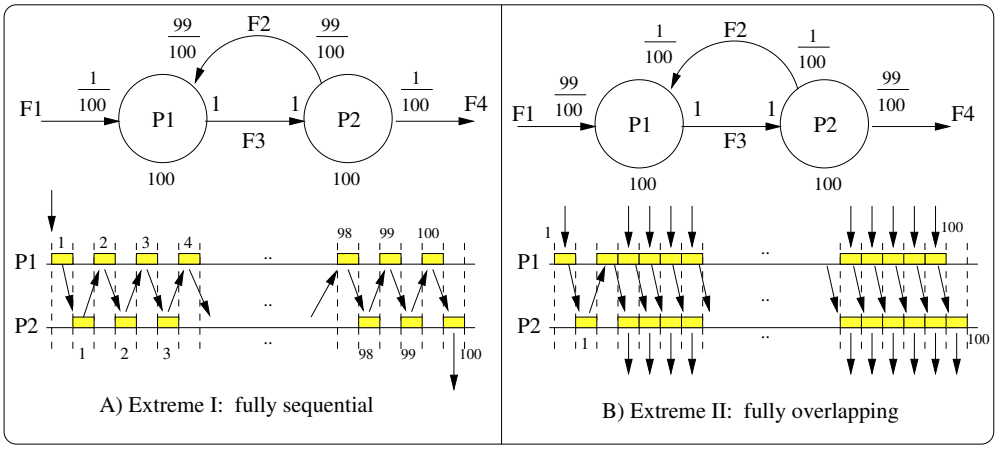

Figure 5.16: Different Behavior of a Cycle

The cyclic PPNs are topologically the same, but the behavior of the cycles are different. That is, processes $P1$ and $P2$ both have 100 process iterations, but the cyclic PPNs are different in the total number of input tokens read from processes that are involved in the cycle. In Figure 5.16 A), process $P1$ reads data only once from a

process that is not part of the cycle (i.e., the process writing to FIFO channel $F1$), and 99 times from FIFO channel $F2$ that is written by $P2$, i.e., a process involved in the cycle. These channels are therefore, respectively, annotated with the fractions $\frac{1}{100}$ and $\frac{99}{100}$. The behavior of process $P1$ is the following: it injects one token in the cycle in one iteration, and in a next iteration it needs to read a token from the cycle first, before it can inject one token again. This leads to sequential execution of both processes, as illustrated with the time lines of $P1$ and $P2$ in Figure 5.16 A). From this example, we learn that a cycle is a true cycle when the processes read the input data only *once* from outside the cycle and then *always* read/write from/to the cycle.

The other extreme is shown in Figure 5.16 B), where all the input data is read from outside the cycle, except only one input token. Thus, topologically the PPN in Figure 5.16 B) is the same as in A), but the behavior of the cycle is different. That is, process $P1$ reads 99 tokens from FIFO $F1$ that is *not* part of the cycle, and only once from FIFO $F2$ that is part of the cycle. This makes both processes $P1$ and $P2$ from that point of view independent, i.e, the cycle does not sequentialize the executions of $P1$ and $P2$, which results in overlapping execution of both processes as illustrated with the time lines in Figure 5.16 B). This example shows that the cycle is not a true cycle, because all the input data (except one token) is read from outside the cycle.

From the two extreme cases presented in Figure 5.16, we learn that the number of input tokens that the processes read from outside the cycle, can serve as a metric to detect the behavior of a cycle, i.e., whether it is a true cycle. If only one token is read from outside and all the others are read/written from/to the cycle, then the cycle is a true cycle. A true cycle should be easy to detect at compile-time with similar techniques presented in the previous chapters, i.e., polyhedral analysis and counting integer points in polyhedral descriptions of input/output port domains. Thus, a designer can be informed when a cyclic PPN contains a true cycle for which performance gains are not possible. Besides the information on the behavior of the cycles, a designer may also be interested in *how much parallelism* there is, in case there is something to be gained. Therefore, we investigate whether the number of input tokens that are read from outside a cycle, is also an indication how much the processes inside the cycle can overlap their executions.

Let us consider an example where a process reads *half* of its input tokens from outside the cycle, and the other half from inside the cycle. A cyclic PPN with this behavior is shown in Figure 5.17 A). Similar to the example in Figure 5.16, the processes have 100 iterations, but now process $P1$ reads in total 50 tokens from input port $IP1$, i.e., from outside the cycle, and it reads in total 50 tokens from input port $IP2$, i.e., from a process that is part of the cycle. The FIFO channels are therefore annotated with the fractions $\frac{50}{100}$. With this example, we want to indicate that *i)* the communication pattern's influence on the behavior of cycles, and *ii)* that these communication patterns are important for all processes involved in the cycle.

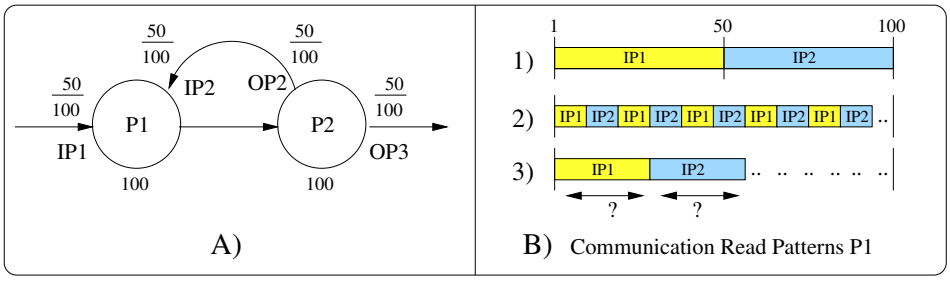

Figure 5.17: What is the behavior when half of the data is read from outside?

Three different communication patterns are shown in Figure 5.17 B), which classifies how process *P1* can read its input data from two input ports *IP1* and *IP2*. Example `1)` shows the time line of process *P1* that has 100 process iterations. Process *P1* needs one input token from either one of its two input ports per process iteration. It reads consecutively 50 tokens from input port *IP1* in the first 50 iterations. Then, 50 tokens from *IP2* are consecutively read in the remaining 50 process iterations. A different pattern is shown with example `2)`. In one process iteration, one token is read from input port *IP1*, and in the next iteration one token is read from *IP2*, which is repeated 50 times. Thus, the tokens are read one by one from different input ports. Example `3)` does not read all tokens consecutively from one port as in example `1)`, it also does not read only 1 token as in example `2)`, but a number of tokens between these extremes.

Figure 5.18 shows the overlap of the two processes involved in the cycle that read/write data with the different communication patterns as we have identified above, i.e., it shows the time lines of processes *P1* and *P2*. We experiment with different communication patterns selected from Figure 5.17 B) and want to show that there is overlap to some extent in all the examples. Each block in the time lines corresponds to one process iteration, i.e., the yellow, blue, white, and red boxes. The executions of *P1* are annotated with the input ports from where *P1* reads its input data (i.e., *IP1* or *IP2*). And the executions of *P2* are annotated with the output ports where *P2* writes its output data to (i.e., output port *OP2* or *OP3*). The arrows denote dependencies, i.e., how data is communicated, and thus a simulation of the cyclic PPN is shown.

There are many combinations of different communication patterns possible for the processes involved in the cycle, because a process does not only have 3 options to read/write in a particular pattern, but these patterns can also be ordered differently inside each process (see process *P2* in Figure 5.18 A and B). Figure 5.18 shows some representative examples of a cycle with different communication patterns and it also illustrates that the overlap in process executions for some examples is minimal (e.g., in Figure 5.18 C), while the overlap for some other examples is substantial (e.g.,

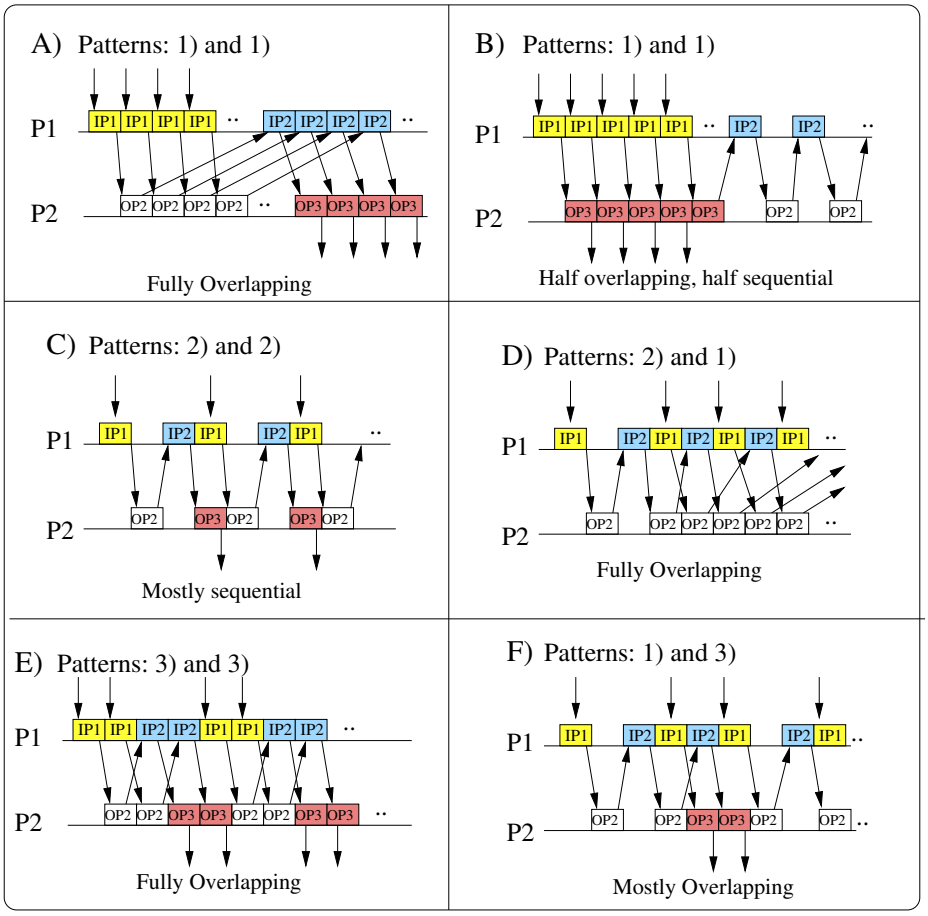

Figure 5.18: Behavior of a Cycle with Different Read/Write Patterns

in Figure 5.18 A), D) and E)). Note that the number of input data that is read from outside the cycle is the same for all examples, i.e., 50 tokens, while the behavior of the cycles are very different. Therefore, we conclude that the number of input tokens read from outside the cycle cannot serve as a metric how much the processes overlap and thus how much can be gained with applying our solution approach. More sophisticated analysis is required, which is therefore left for future research.

Recall that our approach first splits up all processes in a PPN before process instances are merged back (see Algorithm 2). Our final remark is about the order in which the process splitting transformation is applied consecutively on all processes. That is, we did not investigate whether applying the splitting transformation in a different order has an effect on, for example, the number of FIFO channels and/or the final performance results of the transformed PPN. This is left for future research.

Chapter 6

Executing PPNs on Fixed Programmable MPSoC Platforms

In Chapter 1, we have indicated that the Daedalus tool-flow instantiates a specific hardware platform, called ESPAM [61] , prototyped on a FPGA to execute PPNs as efficiently as possible. Recall that we also argued in Chapter 1 that such a specific hardware platform may not always be available to a designer. Therefore, we want to investigate how to execute PPNs onto commercial-off-the-shelf (COTS), programmable MPSoC platforms. In this chapter, we address the issue how to execute polyhedral process networks onto COTS programmable MPSoC platforms and experiment with 2 different platforms: the Intel IXP network processor [1] and the CELL BE processor [39]. The Intel IXP is interesting as it has hardware support for FIFO communication to some extent, i.e., the IXP is highly optimized for streaming data, albeit in the form of internet packets. This makes the Intel IXP a dedicated streaming processing platform. As a second platform, we chose to experiment with a more general purpose MPSoC computing platform, i.e., the Cell platform, which lacks any hardware support for FIFO communication. For both platforms, there is a mismatch with the PPN model of computation. The mismatch is related to the FIFO read/write primitives used in the PPN model of computation and the way FIFO communication is supported by the hardware platform. This mismatch is the most evident in the Cell processor because it lacks any hardware support for FIFO communication, while the IXP has FIFO support to a certain extent. Taking this mismatch into account, we want to investigate in this chapter, how FIFO communication can be realized in the most efficient way using the provided communication infrastructure of these two COTS programmable MPSoC platforms.

6.1 The Programmable Platforms

In this section, we briefly discuss the Cell processor and the Intel IXP processor architectures, i.e., we discuss the interesting components of both platforms and explain the mismatch between the processor architectures and the PPN model of computation.

The Cell

The Cell BE platform [39] is a very good representative example of a state-of-the-art heterogeneous programmable MPSoC platform. A high-level schematic of the Cell architecture is shown in Figure 6.1. It has a PowerPC host processor (PPE) and a set of eight computation-specific processors, known as synergistic processing elements (SPEs). The memory subsystem offers private memories for each SPE processing elements and a global memory space, to which only the PPE has direct access. Each SPE has a Memory Flow Controller (MFC) for handling all data transfers. All processors and I/O interfaces are connected by the coherent interconnect bus which is a synchronous communication bus.

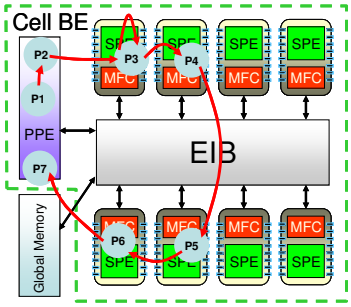

Figure 6.1: A 7-process PPN mapped onto the Cell BE platform.

The mismatch mentioned earlier is illustrated with the example in Figure 6.1. A PPN consisting of 7 processes and 7 FIFO channels is mapped onto the Cell BE platform. Processes $P1$, $P2$ and $P7$ are mapped onto the PPE, and processes $P3$ to $P6$ are mapped onto different SPEs. The FIFO communication channels must be mapped onto the Cell BE communication, synchronization and storage infrastructure. On the one hand, the semantics of the FIFO communication is very simple: Producer and Consumer processes in a producer/consumer pair interact asynchronously with the communication channel to which they are connected. The synchronization between the Producer and the Consumer is by means of blocking read/write on empty/full FIFO channels. On the other hand, in the Cell BE platform the processors are connected to a synchronous communication bus and there is no specific HW support for

blocking FIFO communication. Therefore, the PPN communication model and the Cell BE communication infrastructure do not match. The FIFO channels have to be realized by using the private memory of a SPE, and/or the global memory, and the Cell BE specific synchronization methods which may be costly in terms of communication latency. The challenge is how to do this in the most efficient way, i.e., to minimize the communication latency.

The Intel IXP

The IXP Network processor [1] is built to operate in real-time on internet traffic while being completely programmable. The architecture uses microengines that have hardware multi-threading support and various communication structures to move streams of data around as efficiently and quickly as possible. We focus on the IXP2400 of which a schematic is shown in Figure 6.2.

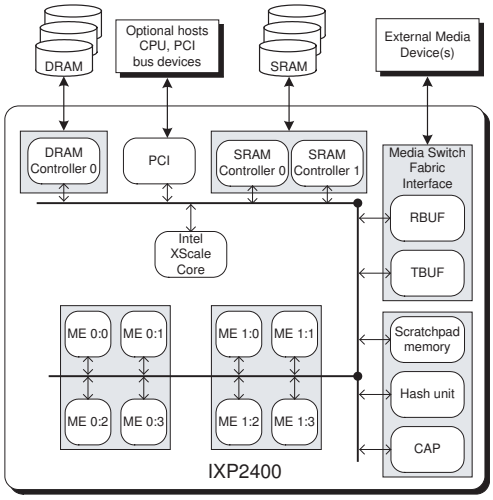

Figure 6.2: IXP2400

The IXP2400 has an Intel XScale Core and eight microengines (ME0.0 - ME1.3) clustered in two blocks of 4. Other relevant parts are the specialized controllers to communicate data with off-chip DRAM and SRAM, the scratch path memory, and the Media and Switch Fabric (MSF) Interface. The MSF interface governs the communication with the Ethernet connection. The IXP2400 can receive and transmit data on this interface at a speed of 2.5Gbps. The XScale core is a RISC general-purpose processor similar to the processing units found in other hardware, including other embedded computers, handhelds and cell phones. The intended use of XScale on the

network processors is for controlling and supporting processes on the microengines where needed.

A microengine is a simple RISC-processor that is optimized for fast-path packet processing tasks. Its instruction set is specifically tuned for processing network data. It consists of over 50 different instructions including arithmetic and logical operations that operate at bit, byte, and long-word levels, and can be combined with shift and rotate operations in a single instruction. Integer multiplication is supported; floating point operations are not. The microengine has special registers to communicate data quickly and efficiently with DRAM and SRAM, its neighbors and local memory. For example, to communicate with neighboring microengines within a cluster, this microengine can uses special hardware support. Via special registers, it can send data to a neighbor and receive data from a neighbor. This could be used to implement FIFO communication, but only for one channel with a very limited buffer capacity. Furthermore, the microengines have access to hardware rings for accessing circular buffers located in the scratchpad and SRAM memories. In Figure 6.2, it can be seen that these buffers are accessed via the bus. Hence, we remark that the IXP has hardware support for FIFO communication (i.e., the available rings), but it is not as dedicated as in the ESPAM platform [60, 61] where each processor has a dedicated communication memory which can be organized as one or more FIFOs.

6.2 Realizing FIFO Communication

In Section 6.1, we have introduced the IXP and Cell processors. Now we show how we map PPNs onto these platforms. This means that each components of the PPN must be expressed in terms of the C language, i.e., a source-to-source translation. These sources can be compiled with the C compiler for the given platform to generate an executable. We focus on the realization of the FIFO communication, because it is the most platform dependent implementation that must use the provided communication infrastructure of the target platform as efficiently as possible. Thus, we indicate the possible mismatch in the PPN model of computation and the target platform. The processes of a PPN are mapped one-to-one onto processing elements. We do not further elaborate on this. Instead, the reader is referred to [50, 58] for more details.

FIFO Communication on The Cell

In mapping PPN processes onto processing elements of the Cell BE platform, different assignments are possible, i.e., processes can be mapped onto the PPE or onto one of the SPEs. This results in different types of FIFO communication channels. For example, in Figure 6.1 processes P_1 (producer) and P_2 (consumer) are mapped onto

the PPE. Therefore, we say that the FIFO channel connecting them is of PPE-to-PPE type. If the producer and the consumer is the same process that is mapped onto a SPE (like process P_3 in Figure 6.1), then we refer to that FIFO channel as a SPE-to-self FIFO channel. Similarly, we identify PPE-to-self, SPE$_i$-to-SPE$_j$, PPE-to-SPE, and SPE-to-PPE types of FIFO communication channels. It is important to define these channel types as all of them require different implementations since different components of the Cell BE platform are involved. To summarize, we identify the following classes of FIFO channels, classified by connection type: *a*) *class self* (PPE-to-self and SPE-to-self), *b*) *class intra* (PPE-to-PPE), and *c*) *class inter* (SPE$_i$-to-SPE$_j$, PPE-to-SPE and SPE-to-PPE).

The first two classes of FIFO channels are easy to implement efficiently, as FIFOs from these classes are realized using just local (for producer and consumer processes) memories and local synchronization is utilized. In the remainder of this section we therefore focus on the *class inter* FIFO channels, which connect the producer and consumer processes mapped onto two different processing elements. The first issue to be addressed is where the memory buffer of a FIFO has to reside. The Cell BE platform provides two memory storages, thus, the buffer can *i)* reside in global memory or *ii)* can be distributed over the local memories of the producer and consumer processes. The advantage of the former approach is the shared memory that is easily accessible (in a mutually exclusive way). The disadvantage, however, is a substantial synchronization overhead. For example, a SPE process with a FIFO channel of type *inter*, should not only compete for the memory resource, but also move the data from the global storage to the local memory prior to computation. The implication of this is an enormous synchronization overhead and we therefore do not consider this as an option to implement the FIFO buffers. For the second approach, i.e., when the memory buffer of a FIFO channel is distributed between local memories, the issue is how to efficiently implement the FIFO semantics. The issue is that the processors need to be synchronized to ensure mutually exclusive access to the FIFO buffer. This processor synchronization is costly and is necessary as the CELL does not provide hardware support for FIFO communication, i.e., the mismatch between the PPN model of computation and the target platform as we mentioned earlier.

In our approach to realize FIFO communication on the Cell and to minimize the number of processor synchronizations, a number of tokens are grouped and send at once, i.e., token packetizing is used. Packetizing decreases the number of DMA data transfers and subsequently it also decreases the number of synchronizations. Determining the packet size is a very important issue, i.e., depending on the process that initiates the data transfers, deadlocks may occur if the packet size is not chosen correctly. We have therefore chosen to use a run-time solution that simply transfers all available generated data. We refer to this solution as the *FIFO pull strategy* which we discuss next. The reader is referred to [58] for a discussion on the FIFO push

strategy and a comparison between the pull and push strategies.

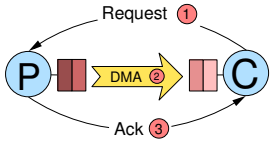

Figure 6.3: Pull strategy for *class inter* FIFO channels.

The FIFO communication between a producer/consumer pair of processes using the pull strategy, consists of 3 steps as shown in Figure 6.3:

1. *Read request* (1). The consumer first tries to read from its local buffer. If it is empty, then it sends a request message to the producer and is blocked on reading awaiting an acknowledgement message from the producer. The request message contains the maximum number of tokens the consumer can accept.

2. *Data transfer* (2). The producer which receives the read request can either be blocked on writing to its local storage or be busy executing a function. If it is blocked, it serves other requests immediately. If it is executing then it immediately serves the request after execution. In either case, the producer handles the request and transfers all tokens it has available for the consumer as one packet by means of a DMA transfer.

3. *Acknowledgement* (3). The producer notifies the consumer after completion of the data transfer issuing a message containing the total number of tokens which have been transferred as one packet in the previous step.

Thus, the *pull* strategy requires two synchronization messages for each DMA data transfer (step (1) and (3)) and the packetizing of tokens is realized in step (2). For every read request of a single data token, the producer sends all its available data to the consumer. Therefore, we refer to this mechanism as dynamic packetizing. The only way to control the dynamic packetizing is by setting the size of the memory buffer, i.e., the larger the size, the larger the packet's size that can be assembled.

FIFO Communication on The Intel IXP

Since the FIFO is such a central element in the IXP, different implementations exist. We have found that six different FIFO types can be realized on the IXP as shown in Figure 6.4. The various realizations make a different trade-off between speed, claimed resources, and size. A short description of the different realizations is given below:

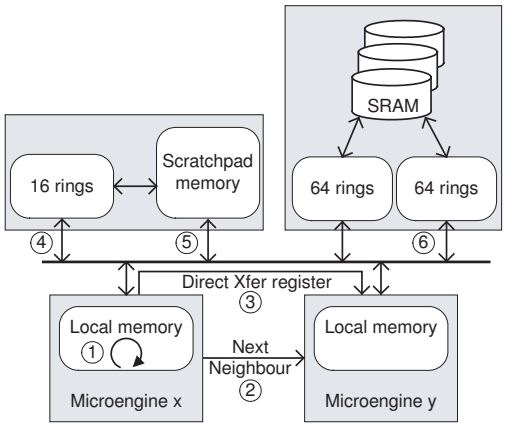

Figure 6.4: FIFO options on the IXP2400

1. *Local memory*. Each microengine has a fast accessible local memory of 640 longwords (2Kb) that is shared among all threads running on that microengine. It can be used to implement a very fast FIFO channel between processes mapped onto the same microengine.

2. *Next-neighbor*. Each microengine has 128 next-neighbor registers. They can be used to implement a very fast, dedicated, FIFO channel between processes mapped onto a limited set of other microengine that are neighbors. The registers can be used in three modes: an extra set of general purpose registers, one FIFO channel of 128 longwords, or as 128 separate registers accessible by the neighboring microengine.

3. *Direct Transfer registers*. Each microengine has 128 SRAM and 128 DRAM registers. They can be used to exchange date with any other process on a microengine. The direct transfer registers use the standard bus and are slower than the previously explained communication mechanism.

4. *Scratchpad rings*. There are 16 sets of special ring registers available on the scratchpad unit. These ring registers provide hardware support to implement the head and tail pointers of a FIFO channel located on the scratchpad memory.

5. *Scratchpad memory*. The ring registers can also be implemented in software directly. These software ring registers implement the head and tail pointers of a FIFO channel located on the scratchpad memory. This is much slower mechanism than the hardware ring register support.

6. *SRAM*. SRAM rings are hardware supported FIFO channel implementations. Each SRAM memory channel has a queue descriptor table which can hold 64 values. Since the IXP2400 has two SRAM memory channels, a total of 128 rings are available.

When very fast FIFO channels are needed, the local memory or next-neighbor registers should be employed. If this is not possible, the hardware support ring registers and scratchpad memory should be used. If this is not possible, the software supported rings should be used. Finally, the SRAM supported FIFO channels should be used. They are the slowest, but can hold the largest amount of data. We implemented a very simple assignment strategy for the processes and FIFO channels of a PPN. FIFO channels are assigned in a greedy way to the fastest possible location. If the FIFO buffer is too large for that location, it is assigned to the next fastest FIFO location.

6.3 Performance Results

We use the techniques presented in Section 6.2 to execute PPNs on the IXP and Cell. We measure the performance results and compare them with results on the dedicated ESPAM platform that is designed to execute PPNs as efficiently as possible.

The Cell

In this section we present several experiments of PPNs mapped onto the Cell platform. The goal is to show the impact of tokens packetizing on synchronization overhead induced in the *class inter* FIFO channels using the *pull* strategy. In addition, we compare the results of two PPNs mapped onto the Cell with the results for the same PPNs mapped onto the ESPAM platform. The Cell experiments are carried out on the `Playstation 3` platform, where the program code has been compiled with IBM's XLC compiler and the `libspe2` library.

In the first experiment we map a JPEG encoder application onto the Cell BE platform. The encoder takes a stream of frames with sizes of 512×512 pixels and applies the JPEG algorithm on these frames. The corresponding PPN consists of 7 processes and 15 FIFO channels. We map the computationally intensive processes DCT, Q and VLE on different SPEs, whereas the other processes are mapped onto the PPE. For this application, buffer sizes of 1 will give a deadlock free network, which means that we can observe token packetizing only when the buffer sizes are increased. Therefore, we run the PPN with four different configurations: we use FIFO buffer sizes of 1, 16, 32 and 48 tokens.

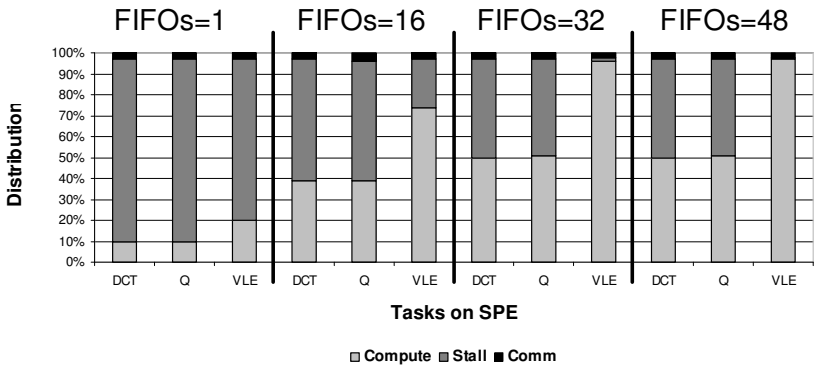

Figure 6.5: Distribution of times the DCT, Q and VLE processes of JPEG encoder spent in computation, stalling and communication for non-packetized and packetized versions.

All bars in Figure 6.5 depict the distribution of the time the DCT, Q and VLE tasks spend in computing, stalling and communicating. Each bar shows how much time processes spend on real computations and thus also how much time is spent in the communication overhead. While stalling, a process is awaiting the synchronization messages from other processes, i.e., showing the synchronization overhead. In the communicating phase, a process is transferring the actual data. The first 3 bars in Figure 6.5 correspond to the configuration with all buffer sizes set to one token. The remaining bars show results of configurations with larger buffer sizes illustrating the effect of token packetizing. We observe a redistribution between computation and stalling fractions in all tasks: the stalling parts have been decreased, while the computation parts were increased. Thus, the packetizing decreases the synchronization overhead. In Figure 6.6, the overall performance of the PPN with different buffer sizes is shown. We observe that the performance increases when the processors spend less time in synchronization.

In four more experiments, we want to investigate the benefits of packetizing in applications with different computation-to-communication ratios. For this purpose, we mapped JPEG2000, MJPEG, Sobel, and Demosaic applications onto the Cell BE. The first two application have coarse-grained computation tasks, while the latter two are communication dominant. For each application, we compare the throughput of the sequential version running on the PPE and two parallel versions: the first one is with minimum buffer sizes that guarantee deadlock free network, i.e., without packetizing possible, and the second, with buffer sizes which are larger than the previous version to allow packetizing. The experiments are depicted in Figure 6.7. Note that the y-axis is a log scale of the throughput in Mbs (mega bits per second).

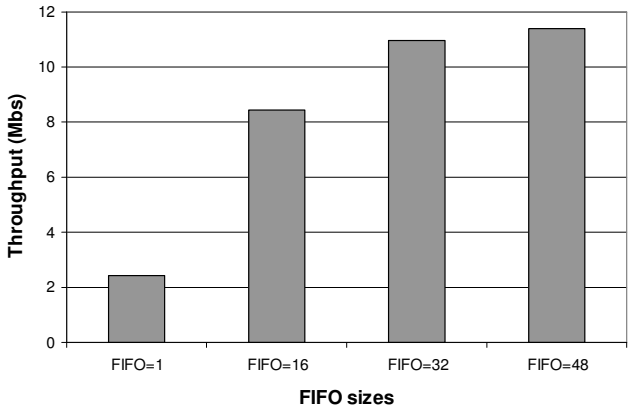

Figure 6.6: Throughput of JPEG encoder with different FIFO sizes

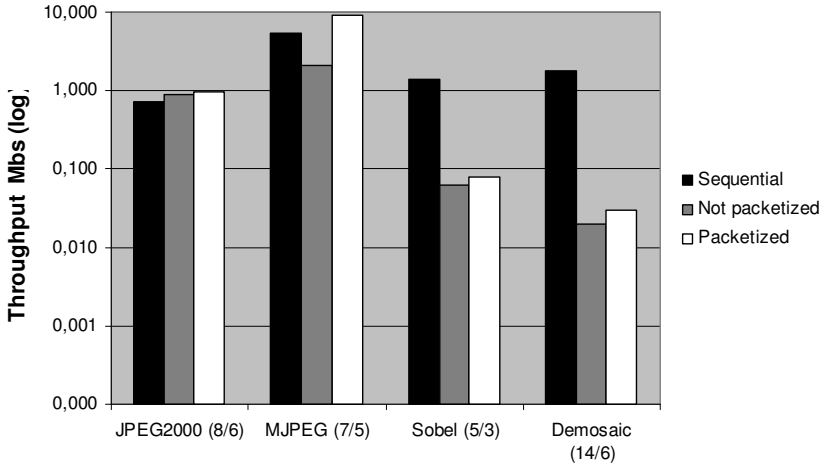

Figure 6.7: Throughput comparison of sequential, non-packetized and packetized versions of JPEG2000, MJPEG, Sobel, and Demosaic applications.

For all algorithms, the packetized versions work better than the non-packetized version. As the JPEG2000 and MJPEG are characterized by their coarse grain tasks, the communication overhead is insignificant and we see that the parallel versions are faster than the sequential version for all, but non-packetized MJPEG algorithms. The Sobel and the Demosaic kernels have very lightweight tasks. Thus, the introduced inter-processor communication and overhead are more costly than the computations themselves. This is the reason the bars in the third and fourth experiments in Figure 6.7 show a significant slow-down compared to the sequential application. The

conclusion is not to consider parallelization for communication dominant applications on the Cell BE platform. We, therefore, ignore the Sobel and Demosaic applications in the comparison of the performance results for applications mapped onto both the Cell and the ESPAM, i.e., we focus on the JPEG2000 and MJPEG applications. The measured performance results for the JPEG2000 application on the Cell and ESPAM are shown in Table 6.1.

Arch.	# clock cycles	CPU freq.	time (sec)
Cell	$288 \cdot 10^6$	3.2 Ghz	0.09
ESPAM	$60 \cdot 10^6$	100 Mhz	0.6

Table 6.1: Measured Performance Results JPEG2000

We observe that the execution time (i.e., the fourth column) is much smaller on the Cell than on the ESPAM platform. This result is mainly due to the clock frequency of the Cell that is a factor of 30 higher than the ESPAM platform. Despite this factor of 30 in the clock frequency, the execution time is *not* 30 times better on the Cell, instead, it is only 7 times better. We observe the same trend for the execution times of the MJPEG application shown in Table 6.2.

Arch.	# clock cycles	CPU freq.	time (sec)
Cell	$1200 \cdot 10^6$	3.2 Ghz	0.375
ESPAM	$300 \cdot 10^6$	100 Mhz	3

Table 6.2: Measured Performance Results MJPEG

The execution time of the MJPEG application is roughly 10 times better on the Cell compared to the ESPAM. Hence, we conclude that the Cell platform is a good platform to obtain low absolute execution time numbers, but it is not necessarily the most efficient platform. The reason is that its clock frequency is 30 times higher than the ESPAM platform (i.e., the Cell is power hungry), but the execution times are not 30 times better. Instead, they are only 10 times better. In other words, the Cell is faster in terms of execution time, but it is not proportional to its much higher clock frequency. The reason is that the FIFO primitives are more costly on the Cell than on the ESPAM platform, i.e., there is more overhead because the Cell communication infrastructure does not support any FIFO communication with hardware components. The reason that the ESPAM platform runs at 100 Mhz, is that it is prototyped on FPGAs. If the ESPAM platform is implemented in ASIC technology as the Cell and the IXP, then the frequency can be higher. As a result, the ESPAM platform would become better in terms of performance than the Cell and the IXP running at a lower frequency, which means that it is also more power efficient.

The Intel IXP

In the experiments for the IXP processes, we consider the QR matrix decomposition algorithm. The corresponding PPN consists of 5 nodes and 12 FIFO channels, see also Figure 5.11. Each process is mapped on a microengine and all FIFO channels are mapped on hardware assisted scratchpad memory rings, i.e., option 4 in Figure 6.4. The reason to map all FIFOs using option 4 is that all FIFOs fit in that memory and that we can actually test the provided hardware support for FIFO communication of the IXP. Note that despite this hardware support for FIFO communication, there is a small software interface implementation that takes care of the FIFO read/write function calls in the read/write phases of the processes. We do not consider the next neighbor link, i.e., option 1 in Figure 6.4, because only 1 FIFO can be mapped and the storage space is limited. Furthermore, we chose not to implement the process functions such that we measure only the FIFO communication in the PPN. Processing a 5x6 version of QR took 40247 cycles on the IXP as shown in the first row of Table 6.3. Note that this measurement only says something about the FIFO communication (read and write phase of a process) as no real function is executed.

Arch.	# clock cycles	CPU freq.	time (μs)
IXP	40247	600 Mhz	67
ESPAM: 5 MB	3865	100 Mhz	39
ESPAM: full HW	213	108 Mhz	2

Table 6.3: Measured Performance Results QR

To assess the efficiency of our FIFO implementation on the Intel IXP processor, we create two ESPAM hardware solutions prototyped on a Xilinx FPGA for the same QR application and compare the performance numbers. We create a platform with 5 Microblaze microprocessors for each process of the PPN, and connect the processors with a dedicated crossbar. The other hardware platform that we create does not use any microprocessors, but all functionality is implemented in hardware, i.e., a full hardware solution. The measured performance results for these two hardware platforms are respectively shown in row 2 and 3 in Table 6.3. It can be seen that the 5 MicroBlaze microprocessor platform executes the QR application in 3865 cycles, while the full hardware implementation executes in 213 cycles. If we take into account the frequencies of the different platforms, i.e., the 3rd column in Table 6.3, then we can compute the execution times that are shown in the 4rd column. These execution times allow a comparison of the QR PPN implementation on 3 different platforms. We observe that the IXP implementation is the slowest, despite the fact that it is running at the highest frequency. We conclude that the more dedicated the communication gets, the higher the performance, i.e., there is roughly a factor of 30 between the ex-

tremes: the software solution on the IXP and the ESPAM full hardware solution. In the IXP we need to share a bus, in the FPGA with MicroBlazes we share a crossbar to communicate between MicroBlazes, and in the full hardware implementation, only dedicated FIFO channels are used to communicate between processes. Moreover, in the IXP there is still some synchronization and control required to handle all FIFO accesses, while in the hardware platforms the producer/consumer pairs are truly decoupled. If we compare the IXP with the 5 MicroBlaze processor ESPAM platform, then the execution time is almost 2 times worse, while the frequency of the IXP is 6 times higher. Thus, Table 6.3 illustrates the penalty that must be paid for mapping PPNs onto a platform that does not support the execution of PPN as efficiently as ESPAM does.

6.4 Discussion and Summary

In this chapter, we showed approaches to execute PPNs on the Cell and IXP platforms, i.e., two commercial-off-the-shelf (COTS), programmable MPSoC platforms. We compared the measured performance results of PPNs executed on these 2 platforms with the performance results obtained on the ESPAM platform. The Cell, IXP, and ESPAM platforms can be characterized as follows: the ESPAM is the most dedicated regarding the execution of PPNs and is prototyped on a FPGA, the IXP is dedicated to streaming data, but is not as dedicated in executing PPNs as ESPAM, and the Cell is the most general purpose compute platform. The platforms run at different frequencies: the ESPAM platform is prototyped on a FPGA and thus runs at 100 Mhz, the IXP runs at 600 Mhz, and the Cell at 3.2 Ghz.

In Section 6.3, we have shown experiments of PPNs executed on these 3 different platforms. Thus, we were able to compare the execution time of the PPNs. From the experiments in Section 6.3, it becomes clear that the IXP processor is not the best platform a designer can select if he/she is free to choose any of these 3 platform as the target platform. Despite the FIFO support in the IXP, the measured execution times of the PPNs are higher than on the dedicated ESPAM platform, while the IXP's clock frequency is 6 times higher than the ESPAM platform. The Cell platform on the other hand, can be a very good platform candidate. Its very high clock frequency compensates the lack of the hardware FIFO support and the overhead caused by the software implemented FIFO communication. However, this overhead makes the Cell not the most efficient platform. While its frequency is 30 times higher than the ESPAM platform, the execution time is only 7 times better. Therefore, we conclude that the Cell is the best platform to obtain the lowest absolute performance numbers. However, the Cell is also the most power hungry solution since it runs at 3.2 Ghz. The frequency of the ESPAM platform can be increased if implemented on the ASIC

technology like the Cell. Then, the ESPAM platform would not only give the best absolute performance results, but it would also be more power efficient. In addition, we remark that PPNs with very light-weight tasks can result in execution times on the Cell that are worse than the sequential version of the application. Again, the reason is the expensive software implemented FIFO communication on the Cell platform. This fact indicates that the designer must carefully take into account the properties of the PPN and the platform in his decision to choose a particular platform. In any case, the ESPAM platform is the most efficient one because it is dedicated to execute PPNs as efficiently as possible.

Chapter 7

Conclusions

In this dissertation, we addressed the problem of how to transform a Polyhedral Process Network (PPN) in order to meet performance/resource constraints. Transformations are crucial because deriving PPNs from a sequential program specifications without performing any transformations does not guarantee that the resource and/or performance constraints are met. The reason is that the `pn` compiler creates one process in the parallel application specification (PPN) for each program statement in the sequential program. As a result, the derived PPN and its processes can be highly imbalanced as some program statements can be much more computationally intensive than others. Therefore, compile-time analysis of PPNs and transformations should assist the designer in transforming the PPN when some design constraints are not met.

The research work presented in this dissertation mainly focused on how the process splitting and merging transformations should be applied to achieve the best possible performance results. The process splitting transformation creates more processes in a given PPN to exploit more data-level parallelism in the application. The process merging transformation is used to reduce the number of processes in a PPN. Before our work presented in this dissertation, i.e., our compile-time approaches to evaluate the process splitting and merging transformations, the problem was that the transformations were defined but it was the designer's responsibility to apply them. We have shown in this dissertation that it is not trivial for a designer to apply these transformations. The reason is that there are many possibilities to apply a particular transformation and many factors influence the final performance results. As a consequence, there can be great differences in the achieved performance results, and they also can easily get worse than the results of the initial PPN if the transformations are not applied carefully. To assist the designer in transforming a PPN, we have defined metrics that are important for the final performance results. Furthermore, we pre-

sented compile-time approaches to evaluate these metrics, such that the designer can select the best possible alternative. For the process splitting transformation discussed in Chapter 3, the analysis is performed locally on the process, while a throughput model for PPNs has been introduced for evaluating the process merging transformation in Chapter 4. Based on the results of the work presented in Chapters 3 and 4, we draw the following conclusion.

- **Conclusion I:** by defining all major factors that are important for the process splitting/merging transformation, and by taking into account the target platform characteristics, we can, at compile-time, evaluate and correctly predict how the process splitting/merging transformation should be applied to obtain the best performance results.

Compile-time hints to transform PPNs in a particular way were missing in the Daedalus tool-flow, as it could only explore different platform and mapping specifications. Thus, the research work presented in this dissertation addresses one very important aspect of the Y-chart approach, i.e., to evaluate and change the application specification after performance analysis. With our compile-time approaches, we can evaluate the process splitting and merging transformations, such that the best option to apply a transformation can be selected. Changing the application specification was identified in Chapter 1 as an important step in order to obtain a desired design point.

Besides approaches to help the designer in evaluating and applying the process splitting and merging transformations in isolation, we have also devised a holistic approach in Chapter 5 that combines both transformations. This solved the problem of ordering the process splitting and merging transformations, which is a difficult problem as there are many alternatives to apply the transformations one after the other and with different parameters. Furthermore, we solved the problem of selecting the processes on which a transformation should be applied.

- **Conclusion II:** by first splitting up all processes and by subsequently merging the different process instances into load-balanced compound processes, we solved the problem of ordering the different transformations and also on which process a particular transformation should be applied to obtain the largest positive performance impact.

There are two perspectives to look at our approach to combine the process splitting and merging transformations. The first one is presented in Chapter 5, i.e., to consider the combination of transformations as an optimization after the initial PPN has been derived. The second perspective is to look at this as an approach to derive PPNs in a different way than currently implemented in the `pn` compiler. That is, instead of creating one process for each program statement in the sequential application, a number of compound processes are created that contain a number of executions of

all program statements. Then, the designer will not be confronted with the initial PPN, but only with the transformed and load-balanced PPN. However, we did not emphasize on this perspective in this dissertation as this requires more research on the number of compound processes to be generated. Choosing the number of processes could be the responsibility of either the designer or the compiler, but the latter is clearly the preferred option as it may not be straightforward for the designer to decide when saturation of the performance occurs. For example, cycles in a PPN may, or may not lead, to sequential execution of the processes involved in the cycle. When the processes in a cycle execute sequentially, then we refer to it as a true cycle. Splitting the processes involved in true cycles would only introduce more processes and not improve the performance, because the processes already execute sequentially as we have also explained in Chapter 5. On the other hand, when the process executions in a cycle overlap, then the splitting transformation can result in performance gains. However, how much can be gained depends on the behavior of that cycle.

- **Conclusion III:** with our holistic approach that combines the splitting and merging transformations, we exploit all available data-level parallelism to the maximum such that our approach gives the best performance results using the two considered transformations when there is something to be gained, and the same performance results as the initial PPN when there is nothing to be gained.

In order to know how much can be gained by splitting processes, the behavior of the (self)-cycles that restrict the data-level parallelism in a certain way must be investigated. We sketched an approach how to detect true cycles, but left the question how many times a process should be split-up for future research.

In Chapter 6, we have presented two approaches to execute PPNs on commercial off-the-shelf (COTS), programmable MPSoC platforms, i.e., the Intel IXP network processor and the Cell platform. While the IXP has hardware support for FIFO communication to some extent, this is completely absent in the Cell platform. Thus, both the Cell and the IXP platform do not support the operational semantics of the PPN model of computation as efficiently as the ESPAM platform, which is especially tailored to execute PPNs as efficiently as possible. To make the FIFO communication more efficient on the Cell, we deployed an approach to transfer multiple data tokens when only one is requested by a consumer process. Thus, by grouping multiple tokens into one package, less FIFO read/write accesses need to be performed during the execution of a PPN. The execution of PPNs on the Cell and IXP processors enabled us to compare the execution times with the ESPAM platform. In Chapter 6, we showed that the ESPAM platform always gives the lowest cycle count, while the Cell is better in terms of execution times as a result of its very high clock frequency.

- **Conclusion IV:** The cycle count for PPNs executed on the ESPAM platform is always lower compared to the IXP and Cell platforms. It does not provide

the fastest execution times since its clock frequency is restricted to 100 Mhz, only because it is prototyped on an FPGA. With higher clock frequencies (e.g., an ASIC implementation, or advances in FPGA technology), the ESPAM platform would not only be the most efficient, but also the best platform to obtain the lowest execution times.

Thus, the most benefit from executing PPNs onto MPSoC platforms is obtained when the operational semantics of the PPN model of computation are supported by the target platform. The IXP processor, for example, runs at 600 Mhz which is 6 times higher than the ESPAM platform. Despite this higher clock frequency, however, the execution times are worse than for the ESPAM platform. The reason is that FIFO communication is supported to some extent, but not as efficiently as on the ESPAM platform. In the Cell platform on the other hand, FIFO communication is completely implemented in software. This makes the ESPAM platform the most efficient platform because it is especially tailored to execute PPNs and supports FIFO communication with hardware components. The Cell's clock frequency is 30 times higher than the ESPAM platform, but its performance results are only 10 times better. The only reason that the ESPAM is restricted to 100 Mhz, is because it is prototyped on FPGA technology and not in ASIC such as the Cell. If the ESPAM platform is implemented using ASIC technology, then it would not only be the most efficient, but also the fastest.

Finally, we remark that it is not beneficial for all applications to be executed as PPNs on MPSoC platforms. With the experiments in Chapter 6, we showed that performance results can also get worse compared to the sequential versions of the applications.

- **Conclusion V:** for applications with very fine-grain computations, and/or target platforms with high synchronization and communication costs, the gain of parallelization can be canceled by the costs for synchronization/communication.

In Chapter 5, we presented an approach to create compound processes by using the process splitting and merging transformations in combination. In that work, we assumed that the designer selects the number of compound processes to be created, which can, for example, be the number of available processors in the target platform. In our future work, we want to investigate if we can decide at compile-time how many compound processes to create before saturation of the system performance occurs. This optimization could, for example, result in a number of compound processes for a given PPN that is less than the available processors, which means that the other processors are available for other applications. Thus, we want to investigate how the maximum parallelism available in an application can be determined, and how it

can be exploited using the minimum number of resources by applying the process splitting and merging transformations.

Bibliography

[1] M. Adiletta, M. Rosenbluth, and D. Bernstein. The next generation of Intel IXP network processors. *Intel Technology Journal*, 06(03), 15 aug 2002.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] T. W. Ainsworth and T. M. Pinkston. On characterizing performance of the Cell Broadband Engine Element Interconnect Bus. In *NOCS '07: Proceedings of the First International Symposium on Networks-on-Chip*, pages 18–29, 2007.

[4] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[5] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.

[6] U. K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[7] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, 2004.

[8] S. Bhattacharrya, R. Leupers, J. Takala, and E. Deprettere, editors. *Handbook on signal processing systems*, chapter by Verdoolaege, S., Polyhedral process networks. Springer, 2010.

[9] S. S. Bhattacharyya and E. A. Lee. Scheduling synchronous dataflow graphs for efficient looping. *J. VLSI Signal Process. Syst.*, 6(3):271–288, 1993.

[10] A. Bik, M. Girkar, P. Grey, and X. Tian. Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems. *Intel Technology Journal Q1 (March) (2001) 1-9*, 2001.

[11] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on signal processing*, 44(2):397–408, 1996.

[12] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. A. Padua, P. Petersen, W. M. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In *LCPC '94: Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 141–154, 1995.

[13] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, 2008.

[14] P. M. Carpenter, A. Ramirez, and E. Ayguade. Mapping stream programs onto heterogeneous multiprocessor systems. In *CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 57–66, 2009.

[15] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. MAPS: an integrated framework for MP-SoC application parallelization. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 754–759, 2008.

[16] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10190, Washington, DC, USA, 2003. IEEE Computer Society.

[17] E. Cheung, H. Hsieh, and F. Balarin. Automatic buffer sizing for rate-constrained KPN applications on multiprocessor system-on-chip. In *Proc. of HLDVT*, pages 37–44, 2007.

[18] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: applications to analyze and transform scientific programs. In *ICS '96: Proceedings of the 10th international conference on Supercomputing*, pages 278–285, 1996.

[19] P. Clauss. Handling memory cache policy with integer points counting. In *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing*, pages 285–293, 1997.

[20] P. Clauss, V. Loechner, and D. Wilde. Deriving formulae to count solutions to parameterized linear systems using Ehrhart polynomials: Applications to the analysis of nested-loop programs, 1997.

[21] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. D. Gajski. System-on-chip environment: a SpecC-based framework for heterogeneous MPSoC design. *EURASIP J. Embedded Syst.*, 2008:1–13, 2008.

[22] B. K. Dwivedi, A. Kumar, and M. Balakrishnan. Automatic synthesis of system on chip multiprocessor architectures for process networks. In *Proc. of CODES+ISSS*, pages 60–65, 2004.

[23] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya. A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications. In *Proc. of EMSOFT*, pages 189–198, 2008.

[24] P. Feautrier. Parametric integer programming. *RAIRO Recherche Operationnelle*, 22, 1988.

[25] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.

[26] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 140–150, 1983.

[27] M. P. Forum. MPI: A Message-Passing Interface standard. Technical report, 1994.

[28] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and design of embedded systems*. Prentice-Hall, Inc., 1994.

[29] L. George and M. Blume. Taming the IXP network processor. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 26–37, 2003.

[30] A. H. Ghamarian, M. C. W. Geilen, T. Basten, and S. Stuijk. Parametric throughput analysis of synchronous data flow graphs. In *Proc. of DATE*, pages 116–121, 2008.

[31] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, 2006.

[32] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, 2002.

[33] K. Grüttner and W. Nebel. Modelling program-state machines in SystemC. In *FDL*, pages 7–12, 2008.

[34] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[35] S. Ha, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo. Hardware-software codesign of multimedia embedded systems: the PeaCE. In *RTCSA '06: Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 207–214, 2006.

[36] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, 1996.

[37] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[38] J. Y. Hur, S. Wong, and T. Stefanov. Design trade-offs in customized on-chip crossbar schedulers. *J. Signal Process. Syst.*, 58(1):69–85, 2010.

[39] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[40] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[41] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki, and K. Kuusilinna. UML-based multiprocessor SoC design framework. *ACM Trans. Embed. Comput. Syst.*, 5(2):281–320, 2006.

[42] J. Keinert, M. Streubuehr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith. SystemCoDesigner—an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):1–23, 2009.

[43] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[44] B. Kienhuis, E. F. Deprettere, P. v. d. Wolf, and K. A. Vissers. A methodology to design programmable embedded systems - the y-chart approach. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*, pages 18–37, 2002.

[45] A. Kumar, S. Fernando, Y. Ha, B. Mesman, and H. Corporaal. Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA. *ACM Trans. Des. Autom. Electron. Syst.*, 13(3):1–27, 2008.

[46] J.-Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[47] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, volume 75, pages 1235–1245, September 1987.

[48] C. Lemke. The dual method of solving the linear programming problem. *Naval Research Logistics Quarterly*, 1:36 – 47, 1954.

[49] L. Li, B. Huang, J. Dai, and L. Harrison. Automatic multithreading and multiprocessing of C programs for IXP. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 132–141, 2005.

[50] S. Meijer, B. Kienhuis, J. Walters, and D. Snuijf. Automatic partitioning and mapping of stream-based applications onto the Intel IXP network processor. In *SCOPES '07: Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, pages 23–30, 2007.

[51] S. Meijer, H. Nikolov, and T. Stefanov. On compile-time evaluation of process partitioning transformations for Kahn process networks. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 31–40, 2009.

[52] S. Meijer, H. Nikolov, and T. Stefanov. Combining process splitting and merging transformations for polyhedral process networks. In *Proc. of the 8th Int. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 97–106, 2010.

[53] S. Meijer, H. Nikolov, and T. Stefanov. Throughput modeling to evaluate process merging transformations in polyhedral process networks. In *Proceedings of the conference on Design, automation and test in Europe (DATE'10)*, pages 747–752, 2010.

[54] B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, and R. Lethin. Productivity via automatic code generation for PGAS platforms with the R-Stream compiler. In *APGAS'09 Workshop on Asynchrony in the PGAS Programming Model*, June 2009.

[55] 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*, pages 238–245, 2007.

[56] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[57] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[58] D. Nadezhkin, S. Meijer, T. Stefanov, and E. Deprettere. Realizing FIFO communication when mapping Kahn process networks onto Cell. In *SAMOS IX: International Symposium on Systems, Architectures, MOdeling and Simulation*, 2009.

[59] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[60] H. Nikolov, T. Stefanov, and E. Deprettere. Multi-processor system design with ESPAM. In *Proc. of CODES+ISSS*, pages 211–216, 2006.

[61] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(3):542–555, 2008.

[62] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia MP-SoC

design. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 574–579, 2008.

[63] L. noel Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *International Symposium on Code Generation and Optimization (CGO)*, pages 144 – 156, 2007.

[64] "OpenCL". "the open standard for parallel programming of heterogeneous systems http://www.khronos.org/opencl/", 2009.

[65] S. Pakin. Receiver-initiated message passing over RDMA networks. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, April 2008.

[66] K. K. Parhi and D. G. Messerschmitt. Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding. *IEEE Transaction on Computers*, 40(2):178–195, Feb. 1991.

[67] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput.*, 55(2):99–112, 2006.

[68] J. Pino, S. Bhattacharyya, and E. A. Lee. A hierarchical multiprocessor scheduling framework for Synchronous Dataflow Graphs. Technical Report UCB/ERL M95/36, EECS Department, University of California, Berkeley, 1995.

[69] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. A. Silber, and N. Vasilache. Graphite: Loop optimizations based on the polyhedral model for gcc. In *Proc. of the 4th GCC Developper's Summit*, pages 179–198, June 2006.

[70] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: part ii, multidimensional time. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 90–100, 2008.

[71] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 546–566, 1994.

[72] K. H. Rosen. *Discrete mathematics and its applications (2nd ed.)*. McGraw-Hill, Inc., New York, NY, USA, 1991.

[73] M. S. Schlansker and B. R. Rau. EPIC: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, 2000.

[74] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[75] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer. Np-click: A productive software development approach for network processors. *IEEE Micro*, 24(5):45–54, 2004.

[76] J. Sjodin, S. Pop, H. Jagasia, T. Grosser, and A. Pop. Design of graphite and the polyhedral compilation package. 2009.

[77] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

[78] T. Stefanov. Converting weakly dynamic programs to equivalent process network specifications, 2004. PhD thesis, Leiden University.

[79] T. Stefanov, B. Kienhuis, and E. Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *Proc. of CODES*, pages 7–12, 2002.

[80] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 899–904, 2006.

[81] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for Cyclo-Static and Synchronous Dataflow Graphs. *IEEE Trans. Comput.*, 57(10):1331–1345, 2008.

[82] N. N. S. Technologies. http://www.network-speed.com.

[83] J. Teich and L. Thiele. Exact Partitioning of Affine Dependence Algorithms. *Lecture Notes in Computer Science (LNCS), Springer*, 2268:133–151, 2002.

[84] L. Thiele, I. Bacivarov, W. Haid, and K. Huang. Mapping applications to tiled multiprocessor embedded systems. In *ACSD '07: Proceedings of the Seventh International Conference on Application of Concurrency to System Design*, pages 29–40, 2007.

[85] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *ISCAS*, pages 101–104, 2000.

[86] L. Thiele and N. Stoimenov. Modular performance analysis of cyclic dataflow graphs. In *EMSOFT 09: Proceedings of the 9th ACM international conference on Embedded software*, pages 127–136, Grenoble, France, 2009.

[87] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, 2002.

[88] M. Thompson and A. D. Pimentel. Towards multi-application workload modeling in Sesame for system-level design space exploration. In *SAMOS*, pages 222–232, 2007.

[89] A. Turjan. Compiling nested loop programs to process networks, 2007. PhD thesis, Leiden University, The Netherlands.

[90] A. Turjan, B. Kienhuis, and E. Deprettere. Translating affine nested-loop programs to process networks. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 220–229, 2004.

[91] S. van Haastregt and B. Kienhuis. Automated synthesis of streaming C applications to process networks in hardware. In *DATE*, pages 890–893, 2009.

[92] S. Verdoolaege. *Incremental Loop Transformations and Enumeration of Parametric Sets*. PhD thesis, Katholieke Universiteit Leuven, 2005.

[93] S. Verdoolaege. An integer set library for program analysis. *ACES symposium, Edegem, 7-8 september*, 2009.

[94] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor. Multi-dimensional incremental loop fusion for data locality. In *In Proceedings of the IEEE International Conference on Application Specific Systems, Architectures, and Processors*, pages 17–27, 2003.

[95] S. Verdoolaege, H. Nikolov, and T. Stefanov. pn: a tool for improved derivation of process networks. *EURASIP J. Embedded Syst.*, 2007(1):19–19, 2007.

[96] S. Verdoolaege and K. Woods. Counting with rational generating functions. *J. Symb. Comput.*, 43(2):75–91, 2008.

[97] S. Verdoolaege, K. M. Woods, M. Bruynooghe, and R. Cools. Computation and manipulation of enumerators of integer projections of parametric polytopes. CW Reports CW392, K.U.Leuven, Department of Computer Science, Mar. 2005.

[98] D. K. Wilde. A library for doing polyhedral operations. Technical Report RR-2157.

[99] X. D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe. A lightweight streaming layer for multicore execution, http://cag.lcs.mit.edu/commit/papers/07/zhang-dascmp07.pdf.

Index

Acknowledgments

This dissertation would not have been written without the help, assistance, and advise of many people. First of all, I would like to thank Alexandru Turjan for introducing me to the topics of compilation techniques and program analysis. Inviting me to write my master's thesis in Philips Research was really the kickstart for my research work later as a PhD-student. Alex, I learned a lot from your research mentality, interest in reading literature, and problem solving skills. It was therefore my pleasure to work briefly together again when you invited me for a PhD internship at NXP Semiconductors.

From the LERC group in LIACS, I am most thankful to Todor Stefanov and Hristo Nikolov. In the "second half" of my PhD time, when results needed to be produced, you pushed me and I pushed you. We had many interesting and challenging discussions that led to the fine results that we produced in such a short amount of time. While I was sometimes rushing, you were always checking and double-checking things and I really enjoyed working together.

From ACE Associated Compiler Experts B.V., I would like to thank Marcel Beemster, Marius Schoorel, Joseph van Vlijmen, and Martijn de Lange for giving me the right advise during my PhD, which really contributed to the successful second half my PhD.

The work presented in this dissertation has been supported by the MEDEA+ NEVA project 2A703. I would like to thank the NEVA project for financially supporting my research, and I am thankful to Sven Verdoolaege for proof reading this dissertation.

Finally, I would like to thank all my other friends, family, parents for their support. Wouter Meuleman in particular, since we finished the same bachelor and master studies, both continued as Phd-students, and thus shared many experiences. And last but not least, I would like to thank Senny for her understanding and support during my PhD time, and for her love!

Samenvatting

Deze dissertatie beschrijft methoden en technieken voor het analyseren en programmeren van multiprocessor systemen die zijn geïntegreerd in een enkele chip. We richten ons voornamelijk op applicaties voor de verwerking van signalen en beelden in ingebedde multimedia toepassingen. Deze toepassingen kunnen het best worden gekarakteriseerd als een verzameling van rekentaken die data uitwisselen in de vorm van datastromen. In de meeste van deze toepassingen zijn doorstroomsnelheden van cruciaal belang, waardoor rekentaken snel en, indien mogelijk, gelijktijdig moeten worden uitgevoerd. Deze eisen leiden vanzelf tot implementatiestructuren die bestaan uit meerdere, vaak ongelijke, processoren die autonoom rekenen en zijn aangesloten op een communicatie-, synchronisatie-, en geheugeninfrastructuur voor de uitwisseling van data. De complexiteit van zulke ingebedde multi-processor systemen heeft een niveau bereikt waardoor het noodzakelijk is geworden om het programmeren van deze systemen op systematische en automatische wijze uit te voeren.

Voor het efficiënt programmeren van multi-processor systemen heeft het Leiden Embedded Research Center (LERC) een ontwerpmethodologie ontwikkeld die uitgaat van twee principes. Het eerste is gebaseerd op het feit dat toepassingen gespecificieerd worden in termen van datastroom procesnetwerken, in het bijzonder *Polyhedral Proces Netwerken* (PPN), die goed passen bij de beoogde datastroom applicaties. Hierdoor is een ontwerper veel beter in staat, in tegenstelling tot monolitische en sequentiële applicatiebeschrijvingen, om autonome taken toe te kennen aan verschillende processoren van het multi-processor systeem. Het tweede principe heeft als doel multi-processor systemen te creëeren die naadloos aansluiten op de eigenschappen van de stroomgebaseerde toepassingen, waardoor de applicaties zo efficiënt mogelijk uitgevoerd kunnen worden. Deze ontwerptechnieken worden volledig ondersteund door het vertaalprogramma Daedalus. Dit is een vertaler die drie hoogniveau beschrijvingen (de applicatie, het multi-processor systeem, en de toekenning van applicatietaken aan rekeneenheden van het multi-processor systeem) automatisch

omzet naar een laagniveau beschrijving van het systeem. Dit stelt een ontwerper in staat om op volledig automatische wijze een applicatie te implementeren op een multi-processor systeem.

Deze dissertatie richt zich op de beschrijving van applicaties in de vorm van een Polyhedral Proces Netwerk (PPN), en dan met name op het omvormen van PPNs. Het probleem is dat PPNs automatisch afgeleid kunnen worden, maar niet noodzakelijk tot de gewenste doorstroomsnelheden leiden. Het omvormen van het PPN is dan noodzakelijk om het gewenste resultaat te bereiken. Het omvormen van een PPN kan op twee manieren gebeuren: een proces uit het PPN kan opgesplitst worden in meerdere parallele processen, of meerdere processen kunnen samengevoegd worden in één samengesteld proces. In het eerste geval, spreken we van de *process splitting* transformatie die toegepast wordt om de applicatie te versnellen, en in het tweede geval spreken we van de *process merging* transformatie dat toegepast wordt om het aantal processen in het PPN te verminderen indien nodig. Het probleem bestond eruit dat beide transformaties wel gedefinieerd waren, maar de ontwerper wist niet precies hoe deze het best toegepast konden worden. Er zijn namelijk vele verschillende mogelijkheden waarop een bepaalde transformatie toegepast kan worden, en vele verschillende factoren spelen een rol in de uiteindelijke doorstroomsnelheden van applicaties. Om de ontwerper te helpen met het zo efficient mogelijk toepassen van transformaties, benoemen we in hoofdstuk 3 de factoren die belangrijk zijn voor de process splitting transformatie, hoe deze geëvalueerd kunnen worden, en een aanpak voor het kiezen van de beste transformatie. In hoofdstuk 4 doen we hetzelfde, maar dan voor de process merging transformatie. Deze analyse is wezenlijk anders dan de process splitting transformatie, omdat het niet lokaal uitgevoerd wordt zoals bij de process splitting, maar globaal voor het hele PPN. Dat wil zeggen dat we voor het samenvoegen van processen een model voor de doorstroomsnelheid definiëren. Dit stelt de ontwerper in staat om een transformatie op een bepaalde manier uit te voeren, de doorstroomsnelheid te evaluëren, en het beste alternatief te kiezen. Daarnaast presenteren we in hoofdstuk 5 een aanpak die beide transformaties combineert. Hierdoor lossen we het probleem op dat de transformaties op vele verschillende mogelijkheden achter elkaar toegepast kunnen worden (in verschillende volgordes). Tenslotte presenteren we in hoofdstuk 6 technieken om PPNs op multi-processor systemen uit te voeren. We beschrijven technieken voor het afbeelden van de verschillende elementen van PPNs op de Intel IXP network processor en de Cell platform.

Curriculum Vitae

Sjoerd Meijer was born on the 1^{st} of December, 1979, in Leiderdorp, the Netherlands. In 1998, he received his VWO, or pre-university, high-school diploma at the Louise de Coligny Scholengemeenschap, in Leiden, the Netherlands. Sjoerd Meijer started his studies in computer science at the The Hague University of Applied Sciences and received his bachelor degree in 2002. He continued his studies in computer science at the Leiden University and wrote his master's thesis in Philips Research, Eindhoven. Sjoerd Meijer received his master degree in 2005 and continued the research work as a PhD-student in the Leiden Embedded Research Centre (LERC), which is part of the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University. He was involved in the NEVA project 2A703, Networks on Chips Design Driven by Video and Distribution Applications, and conducted research in the area of automatic parallelization of program code, analysis and transformations for parallel program specifications. The research work culminated in the writing of this Ph.D. dissertation in 2010. Since June 2010, Sjoerd Meijer is working as a compiler engineer in ACE Associated Compiler Experts B.V.