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 automatically 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.

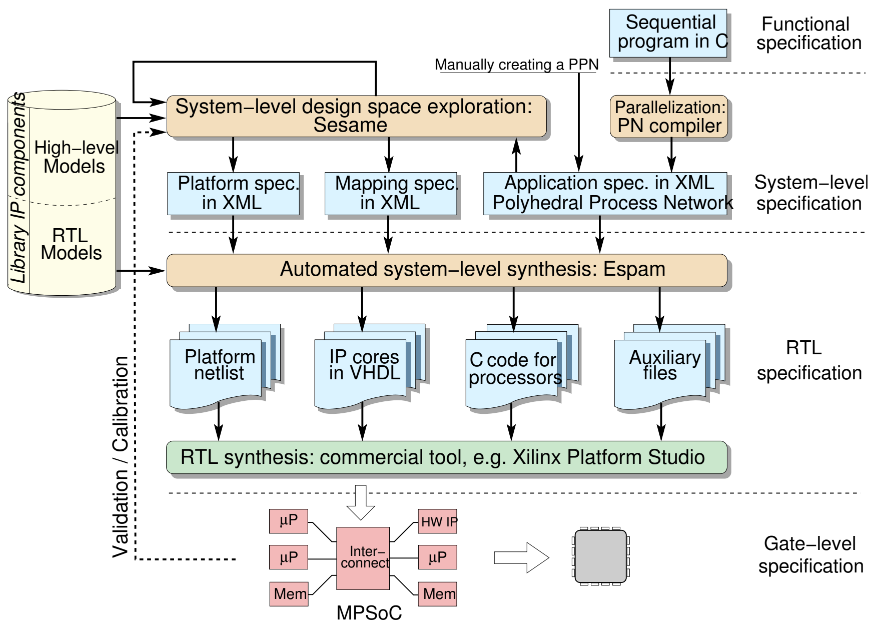
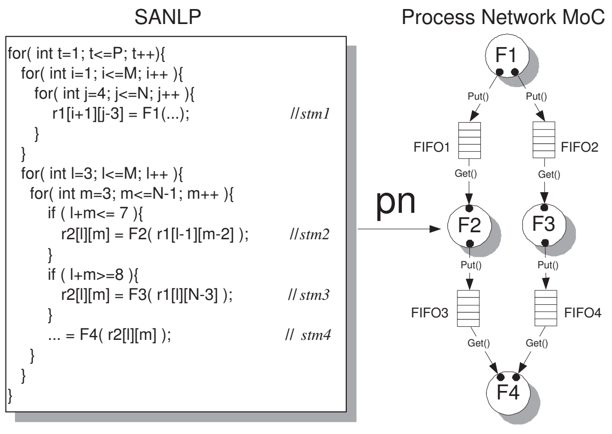


Figure 1.1: Daedalus tool-flow overview

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 processes 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).



```
for( int t=1; t<=P; t++){
  for( int i=1; i<=M; i++ ){
    for( int j=4; j<=N; j++ ){
      r1[i+1][j-3] = F1(...);                     //stm1
    }
  }
  for( int l=3; l<=M; l++ ){
   for( int m=3; m<=N-1; m++ ){
     if ( l+m<= 7 ){
       r2[l][m] = F2( r1[l-1][m-2] );             //stm2
     }
     if ( l+m>=8 ){
       r2[l][m] = F3( r1[l][N-3] );               //stm3
     }
     ... = F4( r2[l][m] );                        // stm4
   }
  }
}
```

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

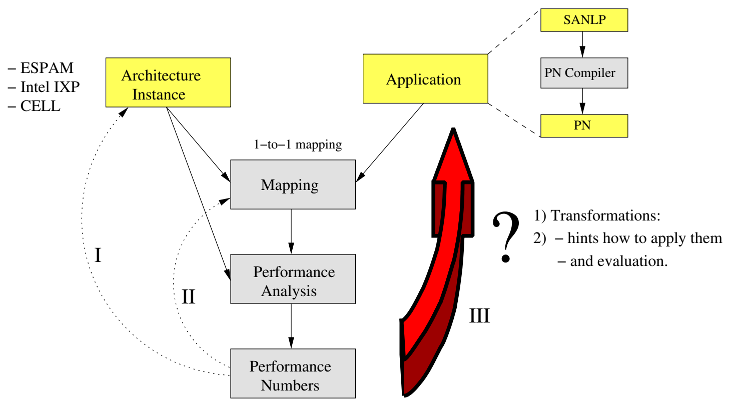


Figure 1.3: The Y-chart Approach

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 different 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.

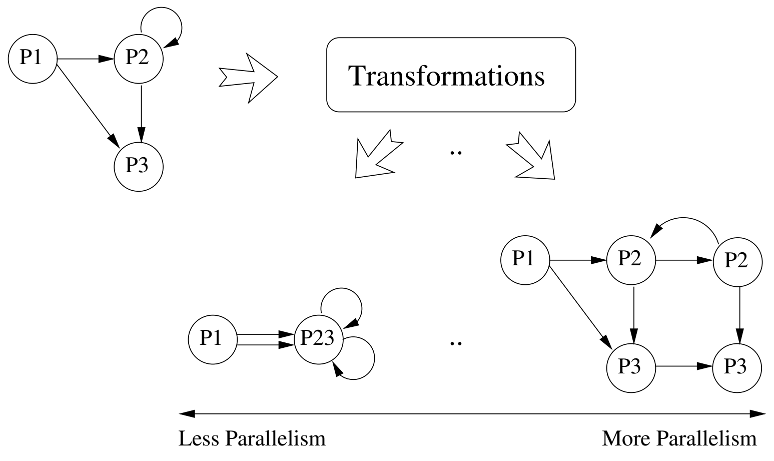


Figure 1.4: Deriving Different PPNs using Process Splitting and Merging Transformations

- **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 partition 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 considered. 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 extension 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.