System-Level Design Methodology for Streaming Multi-Processor Embedded Systems

Hristo N. Nikolov

System-Level Design Methodology for Streaming Multi-Processor Embedded Systems

PROEFSCHRIFT

ter verkrijging van de graad van
Doctor aan de Universiteit Leiden, op gezag
van de Rector Magnificus prof. mr. P.F. van der Heijden,
volgens besluit van het College voor Promoties
te verdedigen op donderdag, 16 April 2009
te klokke 16.15 uur

door

Hristo N. Nikolov
geboren te Gabrovo, Bulgaria
in 1974

Samenstelling promotiecommissie:

promotor	Prof.dr.ir. Ed F. Deprettere	
co-promotor	Dr.ir. Todor Stefanov	
overige leden:	Prof.dr. Daniel Gajski	(University of California, Irvine, USA)
	Prof.dr. Rainer Leupers	(Aachen University of Technology, Germany)
	Prof.dr.ir. Angel Popov	(Technical University of Sofia, Bulgaria)
	Prof.dr. Henk Corporaal	(Technical University Eindhoven)
	Prof.dr. Joost Kok	
	Prof.dr. Harry Wijshoff	
	Prof.dr. Frans Peters	

The work in this thesis was carried out in the Artemisia project supported by PROGRESS/STW.

System-Level Design Methodology for Streaming Multi-Processor Embedded Systems
Hristo Nikolov Nikolov. -
Thesis Universiteit Leiden. - With ref. - With summary in Dutch

ISBN 978-90-9024163-0

Copyright © 2009 by Hristo Nikolov Nikolov, Leiden, The Netherlands.
All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without permission from the author.

Printed in the Netherlands

To my daughters Michaela and Anetta;
To my wife Boyanka for all the support and understanding...

Contents

Acknowledgments **xi**

1 Introduction **1**

1.1 Problem statement . . . 3

1.2 Solution approach . . . 4

1.2.1 Platform-based design at system level . . . 8

1.2.2 Kahn Process Network model of computation . . . 11

1.3 Scope of Work . . . 12

1.4 Research Contributions . . . 14

1.5 Related Work . . . 17

1.6 Dissertation Outline . . . 22

2 Embedded System-level Platform synthesis and Application Mapping – ESPAM **25**

2.1 The Multiprocessor Platform . . . 26

2.1.1 Multiprocessor memory architecture . . . 26

2.1.2 Data communication and synchronization mechanism . . . 27

2.1.3 Platform interconnect protocol . . . 27

2.1.4 Implementation details . . . 28

2.1.5 System-level platform model . . . 32

2.2 Automated MPSoC Synthesis . . . 33

2.2.1 Platform specification . . . 34

2.2.2 Platform synthesis . . . 35
2.3 Automated Programming of MPSoCs . . . 43
2.3.1 Automated Derivation of Process Networks . . . 44
2.3.2 Automated programming – input specification . . . 46
2.3.3 Code generation: SW code for processors . . . 47
2.4 Dedicated IP core integration with ESPAM . . . 51
2.4.1 Uniform structure of a KPN process . . . 52
2.4.2 IP Module – basic idea and structure . . . 53
2.4.3 IP core types and interfaces . . . 56
2.5 Discussion . . . 56
2.5.1 Motivating example . . . 57
2.5.2 Process network instance . . . 58
2.5.3 Preserving the consistency of our PNs with dynamic parameters . . . 59
2.5.4 Respecting the conditions . . . 59
2.6 Conclusions . . . 61

3 Techniques for Narrowing the Design Space 63
3.1 System performance . . . 66
3.1.1 Process throughput and system performance . . . 69
3.1.2 Throughput in case of merged processes . . . 70
3.1.3 Buffer sizes and system performance . . . 71
3.1.4 Dataflow feed-back loops . . . 74
3.2 Rules for MANY-TO-ONE mapping generation . . . 75
3.3 Applying the mapping rules . . . 80
3.3.1 Polyhedral process networks (PPN) . . . 81
3.3.2 Isolated average throughput of a PPN process . . . 82
3.3.3 Process throughput in case of dataflow loops . . . 83
3.3.4 Data rate of the streams in a PPN . . . 85
3.3.5 Computing buffer sizes of the FIFO channels in PPNs . . . 86
3.4 Conclusion . . . 90

4 Case studies 91

4.1 Experimental setup . . . 92
4.2 Homogeneous MPSoCs design with DAEDALUS . . . 92
4.2.1 Design time . . . 93
4.2.2 Performance results and accuracy of the DSE numbers . . . 94
4.2.3 Synthesis results . . . 96
4.2.4 Conclusions . . . 96
4.3 Heterogeneous MPSoCs design with DAEDALUS . . . 97
4.3.1 Design time . . . 97
4.3.2 Performance results . . . 98
4.3.3 Synthesis results . . . 99
4.3.4 Conclusions . . . 99
4.4 Putting DAEDALUS to work . . . 100
4.4.1 Simulation-level DSE . . . 101
4.4.2 Implementation-level DSE . . . 104
4.4.3 Conclusions . . . 107

5 Summary and Conclusions 109

Bibliography 113

Samenvatting 119

Curriculum Vitae 121

Acknowledgments

It is my privilege and great pleasure to convey my gratitude to those who have, directly or indirectly, supported me and helped me during the PhD study.

First, I would like to thank all the people I have worked with in Bulgaria and who have played a role in building my knowledge, my experience and expertise. I am grateful to my teachers for all the things I have learned from them and, in particular, to my mentors during the master project I had at TU-Sofia for showing me the way to the scientific research, and especially, for encouraging me to continue and to do a PhD. Also, I am thankful to all my colleagues and friends at Innovative MicroSystems Ltd. and Fabless Ltd. who contributed to the successful start of my engineering career. Many thanks for the great, enthusiastic atmosphere. Working for these companies complemented the background of knowledge I have obtained at the University with industrial experience related to systems-on-chip design and digital design for FPGAs. Now when I write these words, I realize how much the scientific and engineering background I have built while studying and working in Bulgaria helped me during the PhD research.

The work presented in this dissertation has been supported by PROGRESS, the embedded systems and software research program of the Dutch Technology Foundation STW, under the project ARTEMISIA (Project number LES 6389). I would like to acknowledge PROGRESS and STW for financially supporting my research and the dissemination of the achieved results at various scientific forums worldwide. In particular, special acknowledments go to all people involved in the administration of the ARTEMISIA project.

This dissertation is the result of work conducted at the Leiden Institute of Advanced Computer Science (LIACS), Leiden University, in collaboration with researchers from University of Amsterdam and Delft University of Technology. For the successful collaboration, I express my gratitude to all the people with whom I worked in the context of the ARTEMISIA project and with whom I had very interesting discussions, both scientific and non-scientific. I would like also to acknowledge the people from TU/e with whom I had interesting discussions at several joint projects meetings organized by PROGRESS.

I am glad to say that I enjoyed the time of the PhD study in our group at LIACS. It was a research path that, although it was not easy, led to fruitful results. Moreover, the conducted research enriched my knowledge and expertise in multiprocessor systems-on-chip and design automation for embedded multi-processor systems. It was pleasure working together with my colleagues and my supervisor prof. Ed Deprettere. In addition, I want to thank our former secretary Gonnie for helping me to settle in The Netherlands.

Many thanks to all my friends who know how much I appreciate our friendship. I am pleased to note that for the past several years, it turned out that the two thousands kilometers between Bulgaria and The Netherlands are nothing for real friendship and “every time we meet as we have never separated”, as a friend of mine says. Also, I am lucky that some of my friends are close to me, here in The Netherlands. Hereby, I express my special gratitude to them for always giving me a helping hand when needed! In addition, I would like to thank my close relatives and my family, especially my mother and my brother, for believing in me and for their lifetime support.

Finally, with my deepest love I express my gratitude to Boyanka, my wife, for her love and trust; For sacrificing her professional career in Bulgaria and following me in this PhD adventure.

Hristo N. Nikolov
March, 2009
Leiden, The Netherlands

Chapter 1

Introduction

In a paper published in April 1965 [1], Gordon Moore discussed the future of electronics. Among his predictions for integrated circuits was that the number of circuit components fabricated on a single silicon chip would double every each year, reaching 65000 by 1975[1]. Moore's prediction fit the facts so well that people began referring to it as Moore's Law. It is still known as Moore's Law, even when Moore altered his projection to a doubling every two years in 1975. Since then, the spectacular rate of progress in semiconductor technology has made possible dramatic advances in computers and has led to the emerging of the embedded (electronic) Systems-on-Chip (SoC) concept[2], which in turn have significantly altered almost all areas of human endeavor. In particular, the embedded systems have become the electronic engines of modern consumer and industrial devices, from automobiles to satellites, from washing machines to high-definition TVs, from cellular phones to complete base stations.

Through the years, the increasingly demanding complexity of applications have significantly expanded the scope and the complexity of these SoCs, i.e., the more available resources provided by every new generation of technology have been used to implement more and more sophisticated and diverse system features. Currently, for modern embedded systems in the realm of high-throughput multimedia, imaging, and signal processing, the complexity of embedded applications has reached a point where the performance requirements of these applications can no longer be supported by embedded systems based on a single processing component. Thus, the emerging embedded SoC platforms are increasingly becoming multiprocessor platforms (MPSoCs) encompassing a variety of hardware (HW) and software (SW) components. The ever increasing requirements imply also that, for efficiency and performance, in an MPSoC different application tasks have to be executed by different types of processing components which are optimized for the execution of particular tasks. It is a common knowledge that higher performance is achieved by a dedicated (customized and optimized) HW IP core because it works more efficiently than programmable processors.

[1] At that time, no chips had been manufactured with more than 60 components.

[2] Embedded systems are application domain specific information processing systems that are tightly coupled to their environment.

Evidently, highest efficiency and performance is achieved by MPSoCs consisting of only dedicated IP cores. However, dedicated IPs lack flexibility in making design modifications, a feature playing an important role in the time-to-market competition. Therefore, most of today's MPSoCs are heterogeneous in nature, i.e., a constellation of programmable processors and dedicated IPs, delivering high flexibility and high performance at the same time.

The long design cycle and the ever increasing time-to-market pressure impose clear requirements for systematic and, moreover, automated design methodologies for building heterogeneous MPSoCs. In such methodologies, the intrinsic computational power is not only used effectively and efficiently, but also the time and effort to design a system containing both hardware (HW) and software (SW) remains acceptable. Although embedded systems have been designed for decades, the systematic design of such systems with well defined methodologies, automation tools and technologies has gained attention primarily in the last 10-15 years. For example, a well adopted approach to deal with the embedded SoC design complexity is the Top-Down methodology which allows the designers to manage design complexity at different (hierarchical) levels of implementation details. Currently, this approach is successfully used together with the hardware/software (HW/SW) co-design methodology where HW and SW are designed (almost) independently and concurrently. This allows hardware and software integration testing during the early stages of design resulting in reduced number of design cycles, and consequently, in reduced overall design time. Nowadays end, applying the Top-Down and the HW/SW co-design methodologies with the support of electronic design automation (EDA) tools, is the most efficient design philosophy offering benefits such as reduced design time, design reuse, flexibility in making design changes, faster exploration of alternative architectures, and increased productivity.

Unfortunately, most of the current methodologies for multiprocessor system design are still based on descriptions at register transfer level (RTL) of design abstraction created by hand using, for example, VHDL or C. Such methodologies were effective in the past when SoC platforms based only on a single processor or processor-coprocessor architectures were considered. However, applications and platforms used in many of today's new system designs are mainly based on heterogeneous multiprocessor platforms. As a consequence, the designs are so complex that traditional design practices are now inadequate, because creating RTL descriptions of complex MPSoCs is error-prone and time-consuming even by using the Top-Down methodology. In addition, the complexity of high-end, computationally intensive applications in the multimedia domain further exacerbates the difficulties associated with the traditional hand-coded RTL design and HW/SW co-design methodologies. To execute an application on a MPSoC, the system has to be programmed, which is performed in several steps. First, the application is partitioned into tasks. Second, tasks are assigned (mapped on) to processors (programmable and/or non programmable). Finally, based on the mapping, the MPSoC is programmed, which requires writing program code for each of the programmable processors using languages such as C/C++. The program code includes code implementing the tasks' behavior and code for synchronization the data movement between the tasks (processing components, respectively). In recent years, a lot of attention has been paid to the building of MPSoCs. However, insufficient attention has been paid to the development of concepts, methodologies, and tools for efficient programming of such systems, so that the programming still remains a major difficulty and challenge [2]. Today, system designers experience difficulties in programming MPSoCs because the way an application is specified by

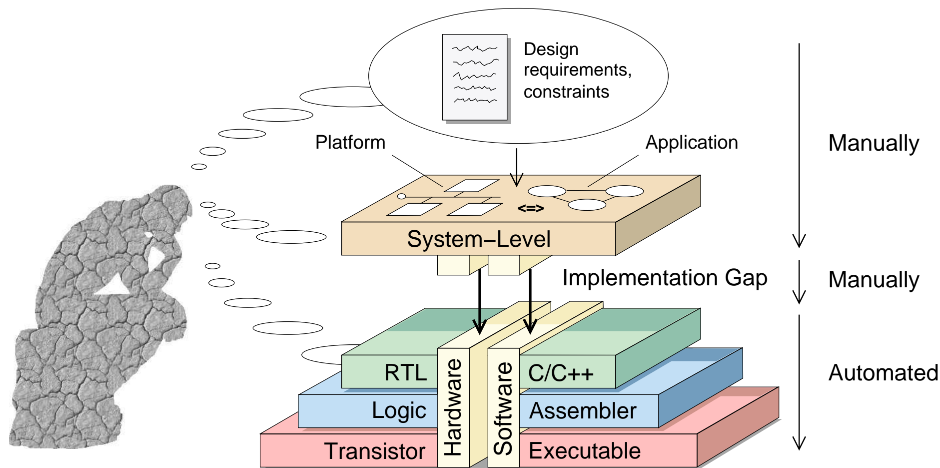


Figure 1.1: The Implementation Gap.

an application developer, typically as a sequential program, does not match the way multiprocessor systems operate, i.e., multiprocessor systems contain processing components that run in parallel.

1.1 Problem statement

For all the reasons stated above, we conclude that:

1) The use of an RTL specification as a starting point for multiprocessor system design methodologies is a bottleneck. Although the RTL specification has the advantage that the state of the art synthesis tools can use it as an input to automatically implement an MPSoC, we believe that a multiprocessor system should be specified at a higher level of abstraction. This is the only way to solve the problems caused by the low level (detailed) RTL specification. The concept of system-level design of embedded systems, which raises the abstraction level of the design process above RTL to cope with design complexity, has been around for several years already and has shown a lot of potential. Despite of this, system-level design of (heterogeneous) MPSoCs still involves a substantial number of challenging design tasks. For example, MPSoCs need to be modeled and simulated to study system behavior in order to evaluate a variety of different design options. Once a good candidate has been found, it needs to be implemented, which involves the synthesis of its architectural components. However, moving up from the detailed RTL specification to a more abstract system-level specification opens (typically a large) gap between the deployed system-level specifications and actual physical implementations. We call it `implementation gap` which is illustrated in Figure 1.1. Indeed, on the one hand, the RTL specification is very detailed and close to an implementation, thereby allowing an automated synthesis path from RTL specification to implementation. This is obvious if we consider the current commercial synthesis tools where the RTL-to-netlist synthesis is very well developed and efficient. On the other hand, the com-

plexity of today's embedded systems forces us to move to higher levels of abstraction when designing a system, but currently, there exists no mature methodologies, techniques, and tools to move down from the high-level system specification to an implementation. Therefore, the implementation gap has to be closed by devising a systematic and automated way to convert a system-level specification effectively and efficiently to an RTL specification.

2) Programming multiprocessor systems is a tedious, error-prone, and time consuming process. On the one hand, the applications are typically specified by application developers as sequential programs using imperative programming languages such as C/C++ or Matlab. Specifying an application as a sequential program is relatively easy and convenient for application developers. However, the sequential nature of such specification does not reveal the available concurrency in an application because only a single thread of control is considered. Also, memory is global and all data resides in the same memory source. On the other hand, system designers need parallel application specifications, because when an application is specified using a parallel model of computation (MoC)[3], the programming of multiprocessor systems could be done in a systematic and automated way. This is so because the multiprocessor platforms contain processing components that run in parallel, and a parallel MoC represents an application as a composition of concurrent tasks with a well defined mechanism for inter-task communication and synchronization.

The facts discussed above suggest that to program an MPSoC, system designers have to partition an application into concurrent tasks starting from a sequential program (delivered by application developers) as a reference specification. Then, they have to assign the application tasks to different processors[4] and to write specific program code for each programmable processor. Partitioning of an application into tasks consumes a lot of time and effort because the system designers have to study the application in order to identify possible task- and/or data-level parallelism that is available, and to reveal it. Moreover, an explicit synchronization for data communication between the application tasks is needed. This information is not available in the sequential program and has to be specified by the designers explicitly. Therefore, an approach and tool support are needed for application partitioning and code generation, i.e., (C/C++) code for each processor of an MPSoC, to allow systematic and automated programming of MPSoCs. Currently, for a wide range of processors, the path from C/C++ to final executable code is fully automated.

In this dissertation, we address the issues of design, program, and implementation of MPSoCs in a specific way which allows us to devise a particular solution of closing the implementation gap. A motivation and an overview of the solution is presented in the next section.

1.2 Solution approach

In this section, we give an overview of the solution approach we propose in order to close the implementation gap described in Section 1.1. The ideal approach would be a tool (or set of tools) that could automatically identify a set of application tasks and map them onto a multiprocessor platform guaranteeing the correct functionality and timing with optimal re-

[3] A model of computation is the definition of the set of allowable operations used in computation.
[4] This step may also involves SW/HW partitioning decisions.

source utilization. This tool should take a design description at the pure functional level together with performance and other constraints, and considering a target platform, it should produce optimized implementation. The ideal situation is not fulfilled (yet) for the general case, however, in this dissertation we present our methodology in which the issues of automated design, programming, and implementation of MPSoCs are addressed in a particular way, focusing on a particular application domain. Based on its characteristics, we make some assumptions (see section "Scope of work") which enabled the development of techniques to close the implementation gap.

As we mentioned already, the state of the art Top-Down and HW/SW co-design methodologies have been a topic of interest for years, but the proposed methodologies lack productivity and effectiveness when targeting MPSoCs design. In addition, these methodologies fail in raising the level of abstraction above RTL. Therefore, a new *design philosophy* is needed to address the aforementioned design challenges. At the same time, we believe that this new design philosophy must exploit the great potential and the advantages of the Top-Down and HW/SW co-design methodologies (see Section 1) that they offer for single-processor systems design.

In this dissertation we propose a methodology, implemented in a tool-flow called DAEDALUS [3,4], for automated design, programming, and implementation of MPSoCs starting at a high level of abstraction. The methodology is built on the concept of Platform-Based Design (PBD) [5] being a promising new approach to master the ever growing complexity of today's embedded systems. The main idea is starting from a functional specification of an application and a description of an MPSoC at system level, to refine and translate them to lower RTL descriptions in a systematic and automated way. The proposed methodology is illustrated in Figure 1.2. It starts with an application written as a sequential C program which represents the required system behavior at functional level. In DAEDALUS, there are specifications at three additional levels of abstraction, namely at SYSTEM-LEVEL, RTL-LEVEL, and GATE-LEVEL.

Definition 1.2.1 (System level)
System level is a level of abstraction above RTL including both hardware and software.

The SYSTEM-LEVEL specification in DAEDALUS consists of three parts written in XML format:

1. *Application Specification*, describing an application in a parallel form as a set of communicating application tasks.
2. *Platform Specification*, describing the topology of a multiprocessor platform. The type of platforms we consider is presented in Section 2.1.5.
3. *Mapping Specification*, describing the relation between all application tasks in *Application Specification* and all components in *Platform Specification*.

The application specification captures the initial application in a parallel form. For this purpose, we use the Kahn Process Network (KPN) [6] model of computation, i.e., a network of concurrent processes communicating via FIFO channels. For applications specified as

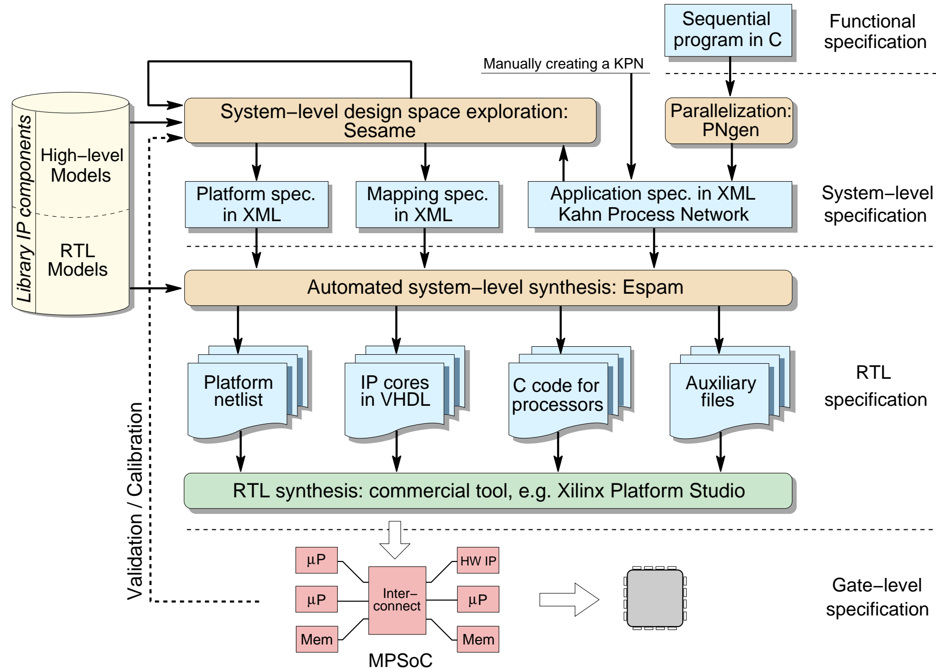


Figure 1.2: DAEDALUS System Design Flow.

parameterized static affine nested loop programs in C (a class of programs discussed in Section 2.3.1), KPN descriptions can be derived automatically by using the PNGEN tool [7], see the top right part in Figure 1.2. In case the application does not fit in this class of programs, the application specification needs to be derived by hand. The platform and the mapping specifications can be created manually or can be generated automatically. Specifying a multiprocessor platform by hand is a simple task that can be performed in a few minutes, because the high-level platform specification does not contain any details about the MPSoC components and, e.g., their physical interfaces. Describing a mapping in XML format is even simpler than writing a platform specification.

The components in the platform specification are taken from a library of IP components, see the left part of Figure 1.2. The library consists of predefined generic parameterized components which constitute the platform model in the DAEDALUS design flow. The platform model is a key component in the proposed solution approach because it allows alternative MPSoCs to be easily built by instantiating components, connecting them, and setting their parameters in an automated way. The components in the library are represented at two levels of abstraction: High-level models are used for constructing and modeling multiprocessor platforms at system level. Low-level models of the components are used in the translation of the multiprocessor platforms to RTL, ready for final implementation.

The platform and the mapping specifications can be generated automatically as a result of a design space exploration. For this purpose, we use the SESAME tool [8] (see the top of

Figure 1.2) developed at the University of Amsterdam. As input, SESAME uses the KPN application specification and the high-level models of the components from our library. The output is a set of pairs, i.e., a platform specification and a mapping specification, each pair representing an optimal mapping of the initial application onto a particular MPSoC in terms of performance and given certain constraints.

The SYSTEM-LEVEL specification of an MPSoC is systematically and automatically translated to RTL-LEVEL in several steps. In the beginning, the platform specification is used to construct a platform instance. The platform instance is an abstract model of an MPSoC because, at this stage, no information about the target physical platform is taken into account. The model defines only the key system components of the platform and their attributes. Then, the abstract platform model is refined to an elaborate (detailed) parameterized RTL model which is ready for an implementation on a target physical platform. The refined system components are instantiated by setting their parameters based on the target physical platform features. Finally, program code for each programmable processor in the multiprocessor platform is generated in accordance with the application and mapping specifications. The described SYSTEM-LEVEL to RTL-LEVEL translation is performed by the ESPAM tool [9], see Figure 1.2. Details about the platform model and ESPAM are given in Chapter 2.

As output, ESPAM delivers a hardware (synthesizable VHDL code) description of an MPSoC and software (C/C++) code to program each processor in the MPSoC. The hardware description, namely a RTL-LEVEL specification of a multiprocessor system, is a model that can adequately abstract and exploit the key features of a target physical platform at the register transfer level of abstraction. It consists of two parts: *1) Platform topology*, a netlist description defining in greater detail the MPSoC topology; *2) Hardware descriptions of IP cores*, containing predefined and custom IP cores (processors, memories, etc.) used in *Platform topology* selected from *Library IP Cores*. Also, it generates custom IP cores needed as a glue/interface logic between components in the MPSoC. ESPAM converts the XML application specification to efficient C/C++ code including code implementing the functional behavior together with code for synchronization of the communication between the processors. This synchronization code contains a memory map of the MPSoC, and read/write synchronization primitives. The generated program C/C++ code for each processor in the MPSoC is given to a standard GCC compiler to generate executable code.

A commercial synthesizer can convert the generated hardware RTL-LEVEL specification to a GATE-LEVEL specification, thereby generating the target platform gate-level netlist, see the bottom part of Figure 1.2. This GATE-LEVEL specification is actually the system implementation. The current version of ESPAM facilitates automated multiprocessor platform synthesis and programming targeting Xilinx FPGA technology, and thus, we use development tools (a GCC compiler and a VHDL synthesizer) provided by Xilinx [10] to generate the final bit-stream file that configures a specific FPGA. We use the FPGA platform technology for prototyping purpose, however, the generated FPGA MPSoC implementations may also be the final system implementation if, e.g., certain system requirements are met. In addition, the results we obtain from prototyping are used for validation/calibration of the high-level models in order to improve accuracy of the design space exploration process. The techniques in the ESPAM tool are flexible enough to target other physical platform technologies.

With DAEDALUS, we propose a model-driven design methodology and below we highlight its key characteristics:

- To address the challenges associated with the programming of MPSoCs presented in Section 1.1, in the proposed design methodology we use a parallel model of computation, namely the Kahn Process Network (KPN) MoC [6], to represent an application as a set of (concurrent) application tasks. These tasks are further mapped onto programmable (ISA) and non-programmable (dedicated IPs) processing components of an MPSoC. Exploiting the KPN MoC, we propose techniques for programming the ISA processors in an automated way.
- DAEDALUS facilitates design of heterogeneous systems where both programmable and non-programmable processors are used as processing components. In case of non-programmable processing components, we propose an approach for automated integration of predefined (third-party) dedicated IP cores. An IP core can be created by hand or it can be generated automatically from C descriptions using high-level synthesis tools like, e.g., the PICO tool from Synfora [11]. High-level (behavioral) synthesis is out of the scope of this dissertation and the DAEDALUS system design methodology.
- To facilitate automated implementation of MPSoCs, we have identified a platform model which captures very well the operational semantics of the KPN MoC. This allows system-level platform descriptions to be refined and translated to detailed RTL descriptions in an automated way. The good match between the KPN MoC and our platform model results in efficient implementations when KPNs are executed on such platforms;
- Our PBD methodology starts with application, platform, and mapping specifications at system level. By applying our techniques, the system-level models are translated to HW platform descriptions at RTL, and SW code executed on the processors of the platform. From RTL to final implementation, DAEDALUS utilizes state of the art (commercial) synthesis and compiler tools;
- By using the proposed application and platform models, a design space exploration at system level is enabled. It allows evaluating the performance of different application to platform mappings and alternative HW/SW partitionings. Such exploration result in a number of promising system design candidates, each defined by an application, a platform, and a mapping specification.

The PBD concept and the KPN MoC are motivated in the following sections. Our platform model is discussed in detail further in this dissertation.

1.2.1 Platform-based design at system level

The concept of a platform encapsulates the notion of reuse, facilitating the adaptation of a common design to a variety of different (domain specific) applications [5, 12]. The platform-based design at system level is a powerful approach that has the potential of addressing the

MPSoC design challenges, in both HW and SW design, in a unified way. We chose PBD because this approach:

- Includes both hardware and embedded-software design;
- Favors the use of high levels of abstraction for the initial design specification;
- Facilitates effective design exploration;
- Achieves detailed implementation by refinement.

The principles of PBD in our approach consist of starting at the highest level of abstraction, i.e., System-level in Figure 1.1, which includes application and platform specification, hiding unnecessary details of an implementation. In PBD, important parameters of the implementation are summarized in an abstract model(s) and design space exploration is limited to a set of available components, i.e., the IP library in Figure 1.2. Furthermore, the design is carried out as a sequence of refinement steps that go from the initial specification towards the final implementation using platforms at various levels of abstraction.

Below, we give definitions associated with the PBD approach of our design methodology presented in this dissertation.

Definition 1.2.2 (Platform)
The platform is a *library of components* that can be assembled to generate a design. The library contains *processing* blocks that carry out the appropriate computation and also *communication* blocks and *memory* blocks that are used to interconnect the processing blocks.

Definition 1.2.3 (Platform model)
The platform model includes the library of components, and defines the way the components can be assembled assuming particular (inter-component) communication and synchronization mechanisms.

Definition 1.2.4 (Platform instance)
A platform instance is a set of components that is selected from the the platform and whose parameters are set. The components in a platform instance are connected in accordance with the platform model.

Definition 1.2.5 (Platform instance refinement)
Refinement is a process of adding (implementation) details to the original platform instance.

The refined platform instance does not necessarily represent a final implementation, however, it is closer than the original platform instance since it contains more details about the target implementation.

Definition 1.2.6 (Mapping)
In the proposed methodology, mapping is an assignment of application tasks to processing components of a platform instance.

The notion of a platform is associated with a set of potential solutions to a design problem where each platform instance implements a design point, i.e., a particular solution. Therefore, we need to capture the process of mapping functionality, i.e., what the system is supposed to do, to platform computational, communication, and memory components that will be used to build a platform instance. This process is an essential step for refinement, which provides a mechanism to proceed towards implementation by closing the implementation gap in a structured way. In addition, taking into account the MPSoC design challenges, we advocate that in order to allow systematic and automated system design where the fundamental steps of functional partitioning, allocation of computational resources, integration, and verification are supported,

1. Applications have to be specified in some parallel model of computation (MoC), at a high level of abstraction;
2. Platform instances have to be specified in a parameterized abstract form (a platform model);
3. Methods have to be provided to map the former onto the latter.

A well known principle in designing complex systems is the *separation of concerns*, initially introduced by Edsger Dijkstra in his essay from 1974 ”On the role of scientific thought” [13]. Separation of concerns is one of the key principles in software engineering and object oriented programming. However, it is an important principle in PBD as well [14]. The main goal is to design systems so that different kinds of concerns are identified and separated (optimized independently) in order to cope with complexity, and to achieve the required quality factors such as robustness, adaptability, maintainability, and reusability. The principle can be applied in various ways. For instance, in PBD, it is important to keep communication and computation components well separated as different methods are usually needed and used to represent and to refine these components. Communication plays a fundamental role in determining the properties of models of computation. Subsequently, special care is needed in defining the communication mechanism of a platform model since it may help or hinder design/components reuse and performance.

Based on the foregoing discussion, we state that the PBD at system level is an attractive candidate to form the basis for new design methodologies. Moreover, if linked to the Top-Down and HW/SW co-design methodologies at RTL, it results in a synergy that can be very productive. In our case, we create this link by closing the implementation gap. In addition, the main goals of reduced design time, design re-use, flexibility in making design changes, faster exploration of alternative platform instances and mappings, and increased productivity, can not be achieved without tools supporting this new design methodology. Therefore, in our approach we are equally interested in developing techniques for:

- Raising the design abstraction to system level by utilizing the platform-based design concept to deal with design complexity;
- Automated translation of the system-level models to RTL descriptions, therefore, closing the implementation gap in a systematic and automated way.

1.2.2 Kahn Process Network model of computation

As discussed in Section 1.1, programming multiprocessor systems is a tedious, error-prone, time consuming process and we argued that in order to facilitate an automated programming, a parallel MoC is required for application representation.

But what should this MoC be ?

Many parallel MoCs exist [15], and each of them has its own specific characteristics. Evidently, to make the right choice of a parallel MoC, we need to take into account the application domain we target. In this dissertation, we consider only data-flow dominated applications in the realm of multimedia, imaging, and signal processing that naturally contain tasks communicating via streams of data. Such applications are very well modeled by using the parallel data-flow MoC called Kahn Process Network (KPN) [6, 16].

Gilles Kahn defined a formal model for networks of concurrent processes that communicate through unbounded First-In First-Out (FIFO) channels carrying streams of data tokens [6, 16]. Processes produce tokens and send them along a communication channel where they are stored until the destination process consumes them. Communication channels are the only method processes may use to exchange information. For each channel there is a single process that produces tokens and a single process that consumes tokens. Multiple producers or multiple consumers connected to the same channel are not allowed. Kahn requires the execution of a process to be suspended when it attempts to get data from an empty input channel. At any given point, a process is either enabled or it is blocked waiting for data on only one of its input channels. When enabled, a process may access only one channel at a time and when blocked on a channel, a process may not access other channels.

Kahn showed that requiring processes to block when attempting to read from empty channels allows processes to be represented as continuous functions over a complete partial order (the set of streams of data elements with a prefix order). A program graph can be represented as a collection of equations that have a unique minimum solution that corresponds to the history of all tokens produced on all streams. Thus, systems that obey Kahn's model are determinate: the history of tokens produced on the communication channels is uniquely determined by the equations representing the program graph and does not depend on the execution order [6]. This implies that as long as blocking reads are enforced, the results of a computation are unique and correct whether the processes are executed sequentially, concurrently, or in parallel. The number of tokens produced, and their values, are determined by the definition of the system and not by the scheduling of operations. However, the number of data elements that must be buffered on the communication channels during execution does depend on the execution order and is not completely determined by the KPN definition.

Because process networks expose parallelism and make communication explicit, they are well suited for targeting MPSoC implementations of a variety of signal processing and scientific computation applications such as embedded signal and image processing. Many researchers [8, 17–21] have already indicated that KPNs are suitable for efficient mapping onto multiprocessor platforms. In addition, we motivate our choice of using the KPN MoC by observing that the following characteristics of a KPN can take advantage of the parallel resources available in multiprocessor platforms:

- *The KPN model is* determinate*:* Irrespective of the schedule chosen to evaluate the network, the same input/output relation always exists. This gives a lot of scheduling freedom that can be exploited when mapping process networks onto multi-processor architectures;

- *Distributed Control:* The control is completely distributed to the individual processes and there is no global scheduler present. As a consequence, distributing a KPN for execution on a number of processing components is a simple task;

- *Distributed Memory:* The exchange of data is distributed over FIFO channels. There is no notion of a global memory that has to be accessed by multiple processes (processors). Therefore, resource contention is greatly reduced if systems with distributed memory are considered;

- *Simple synchronization:* The synchronization between the processes in a KPN is done by a blocking read mechanism on FIFO channels. Such synchronization can be realized easily and efficiently in both hardware and software.

1.3 Scope of Work

In this section, we outline the assumptions and restrictions regarding the work presented in this dissertation. Most of them are discussed in further detail, where appropriate, throughout the dissertation.

Applications

One of the main assumptions is that we consider only data-flow dominated applications in the realm of multimedia, imaging, and signal processing, that naturally contain tasks communicating via streams of data. The streams can represent any type of information, such as audio samples, image blocks, or video frames. Typically, the streams have one source and one sink, and must be non-lossy. Usually, reordering of data items (tokens) in streams is not acceptable. The transformations that are performed on data streams can be quite complex and their granularity is design-dependent. These transformations may consume data from any number of streams and produce data to any number of streams. Such applications are very well modeled by using the KPN data-flow model of computation [6]. We consider KPNs that are input-output equivalent to static affine nested loop programs. The properties of such programs are discussed in Section 2.3.1. We are interested in this subset of KPNs because they are analyzable at design time, e.g., FIFO buffer sizes and execution schedules are decidable. Moreover, such KPNs can be derived automatically from the corresponding sequential programs [7, 22–24].

Application and platform models

The KPN choice as an application model is very important since it influences the platform model and the work/techniques presented in this dissertation. KPNs assume unbounded communication buffers. Writing is always possible and thus a process blocks only on reading from an empty FIFO. In the physical implementation, however, the communication buffers have bounded sizes, and therefore, a blocking write synchronization mechanism is used as well. The problem of deciding whether a general Kahn Process Network can be scheduled with bounded memory is undecidable [25, 26]. However, in our case this is possible because the process networks are derived by using the PNGEN tool from static affine nested loop programs (SANLPs), which programs require finite amount of memory to execute. In SANLPs, loop bounds, variable indexing functions, and condition expressions are all affine functions[5] of loop iterators and (static) parameters. This enables such programs to be modeled in terms of polyhedral domains, i.e., to represent a KPN, we use polyhedral descriptions. Therefore, the process networks we consider in this dissertation are actually *polyhedral process networks* (PPNs)[6], being a subset of the Kahn process networks. In addition, we compute buffer sizes of the FIFO channels (see Section 3.3.5) such that a deadlock-free execution of the considered KPNs on our platform instances is guaranteed. The scheduling of process networks using bounded memory has been discussed in [25, 27]. Also, a number of tools and libraries have been developed for executing KPNs [28, 29]. In contrast to these approaches, the platform model we propose and use to construct (multiprocessor) platform instances does not require scheduling and run-time deadlock detection and resolution. Instead, the processing components in our platform model are self-scheduled following the KPN operational semantics using a blocking read/write synchronization mechanism, i.e., the KPNs are self-scheduled when executed on the MPSoCs. The main objective in devising the platform model was to allow building of MPSoCs which execute KPNs efficiently. In the proposed approach, we do not target particular processing components design rather than integrating such (taken from an IP library) in MPSoCs. Therefore, the main goal in order to achieve efficient KPN execution, is to enable efficient data communication between the processing components, i.e., a communication with minimum communication overhead. We achieved this by taking the main characteristics of the KPN MoC (see Section 1.2.2) into account when devising the platform model.

Multiprocessor platform instances – MPSoCs topology and execution model

With respect to the proposed application and platform models, we consider MPSoCs in which the processing components, i.e, programmable processors and/or HW IP cores, communicate data only through distributed memory units. Each memory unit can be organized as one or several FIFOs. The data communication among the processing components is realized by blocking read and write implemented in software and hardware. Such MPSoCs match and support very well the KPN operational semantics, thereby achieving high performance when KPNs are executed. If the number of processing components in a platform instance is less

[5] Affine functions represent vector-valued functions of the form: $f(x_1, ..., x_n) = A_1x_1 + ... + A_nx_n + b$.

[6] For brevity, in this dissertation, we keep the notation 'KPN' because both, the PPNs and the KPNs, obey the same semantics. Some details about PPNs are given in Section 2.3.1 and Section 3.3.1.

than the number of processes of a KPN, then some of the programmable processors execute more than one process. These processes are scheduled at compile time and the generated program code for a given processor does not require/utilize an operating system. In our approach, we do not consider (high-level, behavioral) synthesis of HW IP cores. Instead, we propose an automated integration of predefined (third-party) HW IPs into (heterogeneous) MPSoCs. We do not impose restrictions on how the IP cores are created, i.e., by hand or by employing high-level design tools. In order an IP core to be added to the components library, however, an IP core has to implement the computation of only a single KPN process. We do not support sharing of an IP core between several KPN processes, i.e., more than one KPN processes to be implemented by a single dedicated IP. Additional requirements for the considered IP cores and their interfaces are discussed in Section 2.4.3. The programmable processors and the HW IP cores in our platforms can be connected in crossbar, point-to-point, or shared bus communication topologies. Details are given in Section 2.1.5.

Tool inputs

The input to the PNGEN tool is an application written as a static affine nested loop program (SANLP) in C. SANLP is a sequential program with some restrictions, discussed in Section 2.3.1. These restrictions allow for automated derivation of KPNs from SANLPs as described in Section 2.3.1. The PNGEN tool partitions a SANLP into processes only at function boundaries, i.e., the programmer divides the SANLP into functions, thus guiding the granularity of the automatically derived processes. Many applications in the considered domain (see above) can be represented as SANLPs. The ESPAM tool accepts as an input three specifications: an application specification, a platform specification, and a mapping specification. The application specification is a KPN either derived by PNGEN or a manually created. The platform specification is restricted in the sense that it must contain only components taken from the library of predefined parameterized components. The library allows and ensures that many alternative (multiprocessor) platform instances can be constructed and all of them fall into the class of MPSoCs we consider (see above). The mapping specification gives the relation between processes and processing components. Based on this, ESPAM determines automatically the most efficient mapping of FIFO channels onto distributed memory units. The platform and the mapping specifications can be created manually or automatically generated by the SESAME tool as a result of a design space exploration.

1.4 Research Contributions

The work presented in this dissertation focuses on the design, programming, and implementation of multiprocessor systems (MPSoCs) starting from high (system) level of abstraction. Below, we outline our main contributions:

Closing the implementation gap

In this dissertation, we present our methods and techniques [9] for systematic and automated multiprocessor system design, programming, and implementation. They bridge the gap between the *system-level* specification and the *RTL* specification in a particular way which we consider as the main contribution of the dissertation. These methods and techniques have been implemented in a tool called ESPAM (Embedded System-level Platform synthesis and Application Mapping). More specifically, with ESPAM a system designer can specify a multiprocessor platform instance at a high level of abstraction in a short amount of time, say a few minutes. Then, ESPAM refines this specification to a real implementation, i.e.,

1. Generates a synthesizable (RTL) HW description of the MPSoC and
2. Generates SW code for each processor,

in an automated way, thereby closing in a particular way the implementation gap mentioned earlier. This reduces the design and programming time from months to hours. As a consequence, an accurate exploration of the performance of alternative multiprocessor platform instances becomes feasible at implementation level in a few hours.

System-level platform model matching the KPN programming (application) model

Our methods and techniques to closing the implementation gap are based on the underlying programming model and system-level platform model we use. Recall that ESPAM targets data-flow dominated (streaming) applications for which we use the Kahn Process Network (KPN) [6] model of computation as a programming (application) model. By carefully exploiting and efficiently implementing the simple communication and synchronization features of a KPN (see Section 1.2.2), we have identified and developed a set of generic parameterized components which we call a platform [9]. The platform and the way its components can be connected and synchronized comprise our platform model. We consider the platform model an important contribution of this dissertation because the set of components allows system designers to specify (construct) fast and easily many alternative multiprocessor platform instances that are implemented and programmed by ESPAM. The approach we propose is general enough and allows for building heterogeneous MPSoCs, i.e., different types of programmable processors and dedicated (third-party) HW IP cores, connected together in different communication topologies. In addition, the good match between the KPN MoC and the platform model results in efficient implementations when KPNs are executed on the considered MPSoCs.

Computing minimum KPN FIFO sizes that guarantee maximum performance

The automated MPSoC design and programming is enabled by using the KPN MoC. However, deriving a KPN specification is a time consuming process and confirmation of this fact can be found in the many system-level design approaches that use the KPN model [28–36].

The KPN model has been widely studied in our group at Leiden Embedded Research Center (LERC)[7] for almost a decade. The work presented in [37] is the first approach, known in the literature, to derive a KPN specification from a static affine nested loop program (SANLP). Several years of research in this direction resulted in techniques implemented in the COMPAAN tool [22, 24] for automated translation of SANLPs written in Matlab to KPN specifications. Although these techniques are very advanced, they do not address the problem of what the buffer sizes of the communication FIFO channels should be. This is a very important problem because if the FIFO buffers are undersized, this leads to a deadlock in the KPN behavior.

Recently, we have developed techniques for *improved* derivation of KPNs [7] from applications specified as sequential C programs. These techniques, implemented in the PNGEN tool [7], allow for automated computation of efficient buffer sizes that guarantee deadlock-free execution of our KPNs. In addition, in this dissertation we present an approach to compute minimum buffer sizes that guarantee maximum performance when KPNs are executed onto the considered MPSoCs. This is another important contribution of this dissertation because we are interested in high-performance multiprocessor systems and with our approach, the highest (theoretical) performance is achievable with reduced memory requirements.

Systematic mapping of application tasks to processing cores

The decision of mapping application tasks to processing components is crucial in order to achieve high performance of the MPSoCs at reduced cost. Assuming that the data communication is efficient and does not introduce communication overhead, the maximum performance is achieved when every task is executed on a separate processor. However, this may introduce large resource overhead because due to task data dependences, most of the time processors may stay idle waiting for data. Therefore, the purpose of the mapping is to group tasks and assign them to processing components in a way that the number of processing components is minimized and the workload is balanced between the components without (or with reasonable) penalty in the overall performance.

Mapping application tasks to processors in an ad-hoc manner may lead to efficient implementations, however, it heavily depends on the expertise of the designer. In addition, for large design space, e.g., an application consisting of many application tasks and a platform that offer different types of processing components, the most efficient mapping can be easily overlooked. This motivated us to research techniques that aim at systematically mapping of application tasks to processing cores in an MPSoC. We devised an approach which exploits the properties of our application and platform models to narrow the design space in a systematic way. More precisely, we defined mapping rules used to create mappings that require fewer number of processing cores without compromising the achieved system performance. Moreover, the proposed approach can be effectively used to complement the techniques in the SESAME tool for reducing the design space that need to be traversed in the design space exploration process.

[7] Leiden University, The Netherlands

1.5 Related Work

Systematic and automated application-to-platform mapping has been widely studied in the research community. The closest to our work is the SystemC-based design methodology presented in [38]. The proposed methodology consists of an automated design space exploration, performance evaluation, and automatic platform based system generation. But unlike DAEDALUS, [38] does not allow for automated parallelization of applications (it requires applications to be specified by hand in SystemC), nor design space exploration at application level. Similarly to our approach, the input for the design flow in [38] contains an executable application specification (written in SystemC), a target architecture template (in both approaches built from components taken from a component library) and mapping constraints of the SystemC modules (in our methodology we have a mapping giving a relation between the application and the architecture). In order to automate the design process, the SystemC application has to be written in a synthesizable subset of SystemC, called SysteMoC [39], whereas our restriction of the initial C program is to be a SANLP (see Section 2.3.1). The synthesizable subset of SystemC is required because for the IP core generation the authors use high-level synthesis tools, e.g, Mentor CatapultC or Forte Cynthesizer which is a major difference with our concept for heterogeneous MPSoCs design. Instead, in this dissertation we propose an approach for dedicated IP core integration based on an HW Module generation consisting of a wrapper around a predefined IP core.

The Eclipse work [40] defines a scalable architecture template for designing stream-oriented multiprocessor SoCs using the KPN model of computation to specify and map data-dependent applications. The Eclipse template is slightly more general than the templates presented in this dissertation. However, the Eclipse work lacks an automated design and implementation flow. In contrast, our work provides such automation starting from a high-level system specification.

Recent work related to multi-processor system design for data-streaming applications is the MAMPS flow presented in [41]. Applications in MAMPS are described as SDF graphs in xml format. These graphs express topological features only without capturing any functional behavior. This is a major difference with DAEDALUS design flow in which applications are specified as fully-functional sequential C programs, automatically parallelized (as KPNs) by the PNGEN tool. The functional specification of an application enables fully-automated programming of the target multi-processor systems. That is, the ESPAM tool generates software code including $computation$ code implementing the functional behavior and $control$ code for synchronization of the communication between the processors of an MPSoC. In contrast, the automated software code generation in MAMPS includes only the $control$ code, i.e., the model of the SDF actor execution and arbitration. Another difference with the DAEDALUS design flow is that the work presented in [41] targets only homogeneous MPSoCs comprised of $MicroBlaze$ processors [42] connected point-to-point through dedicated FIFO links while DAEDALUS supports homogeneous and heterogeneous MPSoCs with processing components being $MicroBlaze$ processors, $PowerPC$ processors [43], and/or dedicated HW IP cores. Moreover, the connections between the processing components can be either point-to-point, crossbar, or shared bus. The work in [41] focuses on multiple (SDF) applications executed on the same platform. In addition, the authors take into account the fact that these applications

may not always run simultaneously by considering multiple use-cases. With DAEDALUS, multiple applications can be mapped on the same platform, however, DAEDALUS does not support "use-cases" as defined in [41].

In our automated design flow for MPSoC programming and implementation, we use a parallel model of computation to represent an application and to map it onto alternative MPSoC architectures. A similar approach is presented in [44]. Jerraya et al. propose a design flow concept that uses a high-level parallel programming model to abstract hardware/software interfaces in the case of heterogeneous MPSoC design. Details are presented in [45] and [46]. In [45] a design flow for the generation of application-specific multiprocessor architectures is presented. This work is similar to our approach in the sense that we also generate multiprocessor systems based on instantiation of generic parameterized architecture components as well as communication controllers to connect processors to communication networks. However, many steps of the design flow in [45] are performed manually. As a consequence, a full implementation of a system comprising 4 processors connected point-to-point takes around 33 hours. In contrast, our design flow is fully automated and a full implementation of a system comprising several processors connected point-to-point, or via a crossbar or a shared bus, takes around 2 hours.

The Polis environment [47] provides an automated design flow starting from high-level specifications and targeting optimized machine code for reconfigurable architectures. It uses a model of computation (MoC) called Extended Finite State Machines (EFSM). This is a major difference from our work since we use the KPN MoC. The EFSM MoC is well suited for control dominated applications whereas the KPN MoC is most suitable for stream oriented applications.

C-HEAP is a top-down design methodology presented in [18]. It generates instances of an architecture template containing multiple processing devices, local cache memories, global shared memory, and a communication network. This work is similar to our approach in the sense that we also generate platform instances based on our platform model. In their work however problems with the cache coherence are reported. In our approach we do not use global shared memory and local cache memories, thus memory contention is avoided.

System-level semantics for system design formalization is presented in [48]. It enables design automation for synthesis and verification to achieve a required design productivity gain. Using Specification, Multiprocessing, and Architecture models, a translation from behavior to structural descriptions is possible at system level of abstraction. Our approach is similar but in addition, it defines and uses application and platform models that allow an automated translation from the system level to the RTL level of abstraction.

In [46] Gauthier et al. present a method for the programming of MPSoCs by automatic generation of application-specific operating systems (OS), and automatic targeting of the application code to the generated OS. In the proposed method, the OS is generated from a OS library and includes only the OS services specific to the application. The input to the code generation flow consists of structural information about the MPSoC, allocation information (memory map of the MPSoC), and high-level task descriptions. By contrast, in our programming approach we do not use operating systems. For each processor of a MPSoC our tool generates sequential code that contains control (for communication, synchronization,

and task scheduling) and application specific code. Another major difference is that in our approach the allocation information (the memory map of a MPSoC) and the task descriptions are generated automatically.

The Multiflex system presented in [49] is an application-to-platform mapping tool. It targets multimedia and networking applications and integrates a system-level design exploration framework. Multiflex uses Symmetric Multi Processing (SMP) and Distributed System Object Component (DSOC) programming models. SMP supports concurrent threads accessing shared memory. DSOC model supports heterogeneous distributed computing using message passing. The MultiFlex tools map these models onto the StepNP MPSoC platform architecture. The relation to our work is that ESPAM also targets the mapping of multimedia and data streaming applications onto a particular MPSoC platform. A design space exploration is included in our design flow as well. However, in our design flow we use Kahn Process Networks as the parallel programming model instead of SMP and DSOC used in Multiflex. The benefit of using KPNs is related to the KPN model properties that allow us to derive KPNs in an automated way from applications specified as sequential programs. Multiflex does not support at all automatic derivation of SMP or DSOC. In [49] a design time of 2 man-months is reported for a MPEG4 multiprocessor system. The design time includes manual application partitioning, automated architecture exploration and optimization. In this paper, we show that by using our design flow a complete design including partitioning, exploration, implementation, and programming of a similar multiprocessor system (a JPEG encoder) is achieved within 2 hours.

There are several approaches for HW design based on the ANSI C standard such as Handel-C and SpecC. Handel-C is a C-based hardware description language commercialized by Celoxica [50]. In contrast to our approach for multiprocessor systems design, Handel-C targets dedicated HW implementations on FPGAs. To express parallelism and event sensitivity in Handel-C, a designer has to use annotations (construct *par*) in the programming code. In our approach, a designer specifies an application as a sequential program using a subset of the ANSI C standard without any special annotations. The parallelism is revealed by our PNGEN tool and determined by the granularity of the function calls used by the designer. Another difference is that Handel-C is based on Hoare's communicating sequential processes (CSP) model [51] while we use the KPN MoC. In both models, processes communicate through channels, yet the synchronization is different. In Handel-C data transfer can only complete when both the source and destination are ready for it. In the KPN model, a channel is organized as a FIFO buffer where write and read operations perform in parallel as long as the buffer is not full or empty, leading to more independent parallel execution of the processes.

The SpecC language, as introduced in [52], is a modeling language for the specification and design of embedded systems at system level. In [52] the authors propose a design methodology based on a library of reusable components that includes several steps such as partitioning, scheduling, communication refinement, code generation. This is similar to our methodology and design flow in the sense that we also use a library of predefined components and our methodology includes similar steps. The main difference, however, is that SpecC is an extension of the C programming language implying that the designer has to study it, although he/she might be familiar with the ANSI C standard. Also, with SpecC the designer has to specify the possible parallelism of an application in an explicit way. In contrast, the appli-

cation specification in our methodology is a C program written using a subset of the ANSI C standard, i.e., SANLP explained in Section 2.3.1. In addition, the parallelization and the communication refinement steps in our design methodology are automated by the PNGEN and ESPAM tools.

A method for automatic generation of embedded software is presented in [53]. The proposed design flow consists of several software refinement steps and intermediate models to generate efficient ANSI C code from system specification written in a SLDL language. This work is similar to our work in the sense that our ESPAM tool generates efficient C/C++ code for processors in a MPSoC. The difference is that some of the software refinement steps in [53] have to be performed manually whereas our software generation is fully automated. Moreover, we generate software for processors in a MPSoC starting from an application specified as sequential program in the widely accepted C language. In [53] a designer has to specify an application using the specific SLDL language.

The Task Transaction Level (TTL) interface presented in [54] is a design technology for programming of embedded multi-processor systems. A multi-tasking application programming interface (API) is provided for parallel execution of streaming applications in a shared memory space. The interaction between application tasks is performed by using communication primitives with different semantics, allowing blocking or non-blocking calls, in-order or out-of-order data access, and direct access of data in a channel to avoid unnecessary data movement. Our programming approach is similar to TTL in the sense that we also target streaming applications and we also use communication primitives. However, in our approach we consider only MPSoC architectures with distributed memory because such architectures give better timing performance compared to shared memory architectures. TTL is more flexible because it supports many communication primitives but programming a MPSoC by using TTL requires a lot of manual work which is hard (in some cases even impossible) to automate. In [6] Kahn proved that by using infinite FIFO queues, the blocking read in-order mechanism is sufficient to realize communication and synchronization in any streaming application modeled as a process network. Due to practical reasons, blocking write is needed as well because a FIFO implementation can not have an infinite size. However, using a blocking write mechanism and finite memory resources may lead to deadlock of a KPN when executed. Therefore, we developed techniques for computing FIFO sizes such that a deadlock-free execution of our KPNs on our platforms is guaranteed – see Section 3.3.5. In this sense, the blocking read and write, both in-order, form the minimum set of communication primitives realizing the communication mechanism of a process network when targeting real implementations. Other communication/synchronization mechanisms add more flexibility but at a certain price. In comparison with TTL, our platform model supports only the two basic primitives which allows us to fully automate the programming of MPSoCs as we will show in this dissertation.

A recent work describing an exploration framework for building efficient FPGA multiprocessor systems for data-flow and stream oriented applications is presented in [55, 56]. This framework explores architectures and allocates application tasks to maximize throughput. The architecture topologies are limited to a network of $MicroBlaze$ processors interconnected using buses (the slow On-chip Peripheral Bus – OPB) and direct FSL links. This work is related to our work in the sense that (for prototyping) we also target FPGA multiproces-

sor systems for data-flow and stream oriented applications using $MicroBlaze$ processors. However, we have developed a different concept of how to connect processors into a homogeneous or heterogeneous multiprocessor system. Our concept relies on communication controllers and memories for communication and synchronization between processors that allow to connect not only $MicroBlaze$ processors using buses and FSL links but also to connect $MicroBlaze$ processors, HW IP cores, and/or $PowerPC$ processors connected in a point-to-point network or, e.g., via a crossbar. In addition, our concept is fully implemented, i.e., our ESPAM tool generates automatically a synthesizable (RTL) specification of a multiprocessor system along with the program code executed on each processor. In [55, 56], the authors do not discuss if they generate automatically RTL-synthesizable multiprocessor systems and how the systems are programmed.

Companies such as Xilinx and Altera provide approaches and design tools attempting to facilitate efficient implementations of processor-based systems on FPGAs. These tools are the Embedded Development Kit (EDK) [10] for Xilinx chips, and the System On a Programmable Chip (SOPC) builder [57] for Altera chips. A recent survey of multiprocessor solutions [58] shows that these state-of-the-art tools support only processor-coprocessor systems and shared memory bus-based multiprocessor systems which can not always meet the performance requirements of todays (streaming) applications. In contrast, our work proposes a platform model that supports different communication topologies (not only a shared bus) and allows different types of processors to be connected in heterogeneous multiprocessor platforms. In addition, we use a parallel model of computation to represent an application and to map it onto multiprocessor platforms. Exploiting the properties of our platform and application models allows for automated MPSoC synthesis and implementation, application dependent self-scheduling of the platform resources, and fully automated MPSoC programming.

SPIRIT [59] is a consortium which aims at ”Enabling Innovative IP Re-use and Design Automation”. It defines several standards, e.g., IP-XACT, and one of the main purposes of this consortium is to provide a well-defined XML Schema for meta-data that documents the characteristics of IPs. While the consortium is focused mainly on the general IP-XACT standard, we target automated IP core integration in our multiprocessor systems. IP-XACT allows to use general interfaces for connection between the IPs, where for each interface a reference bus definition is required. Depending on the complexity of an interface, a bus definition may require a lot of error-prone and time consuming manual work. In order to simplify and automate the IP core integration in our MPSoCs, we define and support only two interfaces, i.e., one data interface and one control interface, that an IP core has to provide. We do not consider this as a limitation of our approach because 1) these interfaces allow an efficient IP core integration in the multiprocessor platforms we consider and 2) the two interfaces are sufficient for integration of IP cores performing computations in the domain we are interested in, i.e., multimedia, image, and signal processing.

There are several initiatives such as VISA [60] and OCP-IP [61] aiming at specifying ”open” interface standards, which will ease the integration effort required to incorporate IP cores into a system-on-chip (SoC). The Open Core Protocol (OCP) defines a bus-independent interface between IP cores that reduces design time, design risk, and manufacturing costs. The OCP is equivalent to VSIA’s proposed Virtual Component Interface (VCI). While the VCI

addresses only data flow aspects of core communications, the OCP is a superset of VCI that also supports configurable sideband control signaling and test harness signals. Although OCP and VCI could remove some of our IP interfacing issues, we do not use these interfaces in our DAEDALUS framework because in many cases they do not comply with our main goal, which is, to integrate IPs in our multiprocessor systems in such a way that the highest possible overall system performance is achieved for a given application. Indeed, the main focus of OCP or VCI is to guarantee interoperability and re-usability of a wide variety of IP cores in a "plug-and-play" fashion but this is achieved at the expense of more general, application-independent, and relatively slow interfaces and protocols. In our approach, we provide a mechanism to integrate third-party dedicated HW IP cores into heterogeneous systems by means of HW modules generated by ESPAM. Each HW module contains a wrapper around a third-party IP core. Our IP wrappers developed in ESPAM are not meant to be as general as OCP and VCI, i.e., our wrappers support efficient integration of the specific type of IPs defined in Section 2.4.3. This fact and the fact that our wrappers are customized for every application, i.e., they are automatically generated according to the KPN specification of an application, guarantee that the highest possible overall system performance is achieved.

1.6 Dissertation Outline

The remaining part of this dissertation is organized as follows. Chapter 2 presents the approach we propose to close the implementation gap between the System and the RTL abstraction levels of description introduced in Section 1.1. The chapter describes in great details the models, methods, and techniques we have developed and implemented in the ESPAM tool for systematic and automated multiprocessor system design, programming, and implementation. First, we motivate the choice of the target multiprocessor systems with a discussion about the mechanism for efficient data communication and synchronization between the processing components allowing efficient execution of KPNs. Then, we introduce the system-level platform model used in ESPAM to construct (abstract) MPSoC instances at system level and present how these instance are translated MPSoC descriptions at RTL. This is followed by a discussion about the automated programming of the MPSoCs and we give details on how ESPAM converts processes to software code for every processor in a homogeneous MPSoC. In this chapter, we also present the approach for building heterogeneous MPSoCs with ESPAM where both programmable processors and dedicated IP cores are used as processing components.

Exploiting the fact that we target MPSoCs executing applications modeled as Kahn process networks, in Chapter 3, we propose techniques for mapping processes to processing components, i.e., mapping rules, which aim at utilizing as less MPSoC components as possible without compromising the performance of the system when executed. By applying the mapping rules, the design space is effectively pruned to a set, consisting of the most promising design points from which, based on certain criteria, the designer can choose the best one for final implementation. First, we explain what system performance means when we consider MPSoCs that execute KPNs. Next, we comment on the factors that affect system performance. Then, after presenting the mapping rules, we discuss how the rules can be applied in practice considering the KPN application model we use.

In Chapter 4, we present three case studies that we conducted in order to demonstrate, validate, and evaluate the methods and techniques for automated MPSoC design presented in Chapter 2 in terms of overall design time, achieved performance, and HW resource utilization. In addition, we comment on the accuracy of the results obtained by performing high-level system simulations (during the DSE process) compared to real implementation numbers. The first case study illustrates a complete design flow with DAEDALUS for a JPEG encoder application, starting from a sequential program, performing system-level DSE with SESAME, synthesizing design instances with ESPAM, and prototyping them by using commercial synthesis and compiler tools. In the second case study, we address heterogeneous MPSoCs where both programmable processors and dedicated IP cores are used as processing components in MPSoCs executing a JPEG encoder, a Sobel edge detection, and a Discrete Wavelet Transform. We illustrate the approach, discussed in Section 2.4, for integrating of predefined IP cores into heterogeneous systems by using automatically generated IP Modules. The purpose of the last case study is to push DAEDALUS "to the limit" in order to check how large and complex systems can be designed using the proposed methodology and considering the constraints imposed by the FPGA technology we currently use for prototyping.

Finally, we conclude this dissertation in Chapter 5 with a summary of the presented research work along with some concluding remarks.

Chapter 2

Embedded System-level Platform synthesis and Application Mapping – ESPAM

In this chapter, we motivate and present in detail the platform model and the target MPSoCs we consider, together with methods and techniques for systematic and automated multiprocessor system design, programming, and implementation. These methods and techniques bridge in a particular way the gap between the system level and the register-transfer level (RTL) of design abstraction introduced in Section 1.1. The approach is implemented in the ESPAM tool (Embedded System-level Platform synthesis and Application Mapping) which is the core tool in the DAEDALUS design flow presented in Section 1.2.

This chapter is organized as follows. First, we motivate the choice of the target MPSoCs. This is done by a discussion about the mechanism for efficient data communication and synchronization between the processing components allowing efficient execution of KPNs. Then, in Section 2.1.5, we introduce the system-level platform model used in ESPAM to construct (abstract) MPSoC instances at system level. In Section 2.2, we present how the abstract high-level platform instance is refined and translated systematically and automatically to an MPSoC instance at RTL. This is followed by a discussion in Section 2.3 about the automated programming of the MPSoCs generated by ESPAM. It includes a brief introduction of the static affine nested loop programs (SANLP) and the automated generation of KPNs using the PNGEN tool. Furthermore, we present details on how ESPAM converts processes in this KPN model to software code for every processor in a homogeneous MPSoC. In Section 2.4, we present our approach for building heterogeneous MPSoCs where both programmable processors and dedicated IP cores are used as processing components. We conclude the chapter in Section 2.6.

2.1 The Multiprocessor Platform

The advancement from single core to multi-core processors is expected to continue resulting in many-core chips with up to hundreds or thousands of cores per chip. This progress raises new important questions concerning new designing and programming paradigms. Connecting the cores, determining the right memory subsystem, ensuring coherence and consistency of data, all require a deep understanding of issues and innovative application of ideas. In this section, we address these issues in a particular way by motivating and defining the type of multiprocessor platform we consider. We target data-flow dominated (streaming) applications which we model in an explicit parallel form using the Kahn Process Network (KPN) model of computation. At the same time, we target high-performance multiprocessor systems to execute these applications. Therefore, the main objective is to devise a multiprocessor platform for efficient execution of applications specified as Kahn process networks, and subsequently, to enable automated multiprocessor systems design and implementation. We achieve this by exploiting the properties and the operational semantics of a KPN, and translating it to the topology and the execution model, i.e., the communication and synchronization mechanism, of the target multiprocessor platform. More precisely, in this section we discuss the MPSoC memory architecture, the mechanism for data communication and inter-processor synchronization, and the MPSoC interconnect topology and protocol.

2.1.1 Multiprocessor memory architecture

Increasing the number of the processing components in an MPSoC, increases the possibility of more computation to be carried out in parallel. However, by increasing the number of the processing components, the access to data becomes the system bottleneck that limits the available parallelism and the achieved overall performance, respectively. At the heart of the trouble is the so-called memory wall, i.e., the disparity between how fast a processor can operate on data and how fast it can get the data it needs. This is the major factor limiting the performance of systems when they rely on a single memory shared between the processing cores and used to load and store data. Increasing the number of processing components in such shared memory MPSoCs only exacerbates this problem. Therefore, boost in memory bandwidth is needed which can be achieved only by considering different memory units, distributed between different processing components. Consequently, in this dissertation we propose MPSoC platform with distributed memory architecture.

A multiprocessor system with distributed memory can deliver higher performance compared to a shared memory system. However, a major drawback of distributed memory systems arises when it comes to program them due to difficulties associated with multiprocessor synchronization and validity of data located in different parts of the distributed memory. We address this important issue, and in Section 2.3, we present an approach for automated programming of the target MPSoCs. Automated programming of MPSoC with distributed memory is facilitated by the fact the KPN MoC we use as an application model also assumes distributed memory, i.e., data is stored either in private memory of processes or in communication FIFO channels.

2.1.2 Data communication and synchronization mechanism

The applications in the multimedia domain we target, are often characterized by having a complex array index manipulation scheme and a large number of data accesses [62] where a processing component needs to generate the addresses of the memory locations in order to retrieve and store data. Address calculations often involve linear and polynomial arithmetic expressions which have to be calculated during program execution. Memory address computation can significantly degrade the performance and increase power consumption [63,64]. Therefore, it is very important to carry out the memory accesses and the related address computations in an effective way.

With respect to this, accessing the memory in a FIFO-like manner is the most efficient way because, in this case, address computation is limited only to increment operation, and consequently, the address generators are comprised by simple counters. Moreover, FIFO communication has proven to be very efficient and it has been widely used in digital signal/image processing and multimedia systems for decades. Therefore, in the MPSoC platform we propose, data between the processing components is communicated through FIFO buffers. This matches very well the KPN operational semantics, thus, leading to minimal communication overhead when KPNs are executed on the target MPSoCs. That is, the synchronization between the processing components is realized by simple blocking read and blocking write operations on empty and full FIFO buffers, respectively. In addition, this synchronization mechanism enables an important feature of the considered MPSoC platform, i.e., being self-synchronizing in a local-synchronous, global-asynchronous fashion.

Another benefit of using FIFOs for communication between processing components is that the simple FIFO interface, i.e., a data bus and two control signals only (empty and read signals, or full and write signals, respectively) facilitates efficient heterogeneous MPSoC design in which different types of processing components communicate data through FIFO channels. We exploit this property in the approach we present in Section 2.4 for dedicated IP core integration in heterogeneous MPSoCs.

2.1.3 Platform interconnect protocol

Recall that we target multiprocessor systems which allow efficient execution of KPNs. Since a KPN is a set of concurrently executing processes communicating data among them, providing efficient data communication (with low overhead) is crucial for multiprocessor system executing KPNs. A data stream in a KPN has a source: The process that writes data to it, and a sink: The process that reads data from it. The inter-processor communication protocol realizing the streaming data between processing components in an MPSoC is simplest, i.e., it introduces the lowest communication overhead, when components are connected point-to-point. However, point-to-point inter-processor communication links may not be always feasible in general, and sharing memories and links for data communication may be necessary. Therefore, we need to devise an alternative approach to connect processing components (by sharing communication memories and links) with the objective to reduce the interconnection complexity in a way that keeps the communication overhead low. In the considered MPSoCs, data communication between the processing components is materialized

in a communication structure which allows for point-to-point, shared bus, crossbar, or even network-on-chip communication topologies. We do not advocate any of these communication topology types, though each and every type has its own efficiency merits. Consequently, the MPSoC (communication) performance depends on the properties of the used communication topology type and the corresponding communication and synchronization mechanism. Considering the distributed memory and the FIFO communication in the target MPSoCs as well as the KPN model, below we motivate the communication mechanism we propose which leads to low overhead and reduced complexity of the communication structure.

Data between the processing components is communicated through dedicated (distributed) memories, i.e., communication memories (CM), in which the communication FIFOs are located. All CMs in an MPSoC are connected through a communication structure. In order to achieve data communication with low overhead in the target MPSoCs, we devised an approach that keeps the communication and synchronization as close as possible to the KPN operational semantics because it guarantees the highest possible communication performance. More precisely, we propose a request-based mechanism for accessing communication memories through the communication structure. In this approach, a processing component is connected to a CM, being its *local* CM, and all other CMs are seen as *remote* CMs of that processing component because the remote CMs are accessible only through the communication structure. Hence, a processing component has a direct access only to its local CM. Consequently, we consider that each processing component can write only to its local communication memory and has to rely on the communication structure to read data from all other communication memories. Thus, a processing component can always write if there is room in its local CM. If this CM is large enough, the processing component may never block on writing, which mimics the infinite FIFOs of the KPN model.

In addition, the FIFO communication between the processing components in the target MPSoCs and the restriction that the processing components use the communication structure only for reading data from remote CMs, allow the communication structure to be very simple. That is, it implements connections in one direction only between communication memories and processing components. A data interface of a processing component consists of address, data, and control buses. However, in the proposed FIFO communication mechanism where the CMs are organized as FIFOs buffers, the addresses to these buffers are generated locally as a result of the memory accesses, see Section 2.1.4. Therefore, there is no need to propagate the address bus through the communication structure, which results in reduced number of signals that needs to be switched. Moreover, the uni-directional communication (for reading data) reduces the load on the communication structure because write accesses are performed locally, and at the same time, reduces the complexity of the switch matrix since data buses only in one direction needs to be switched.

2.1.4 Implementation details

A simplified example of an instance of the multiprocessor platform we propose, is depicted in Figure 2.1. For brevity, this example of three processing components (uP) and memories (CM) connected through a communication structure (Interconnect) shows only details that matter to illustrate the main features of the MPSoC platform discussed above, and leading to

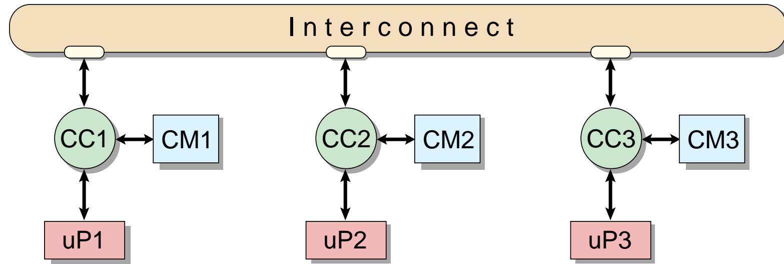


Figure 2.1: Example of a target MPSoC platform instance.

efficient executions of KPNs. Communication and synchronization are the essentials in any MoC. In the KPN MoC, inter-process communication and synchronization is by means of blocking FIFO channels. In the proposed platform the processing components communicate data through (distributed) communication memories (CM), see Figure 2.1. Each CM is organized as one or more FIFO buffers. The inter-processor synchronization in the platform is implemented in a simple and efficient way by blocking read/write operations on empty/full FIFO buffers located in the communication memory. The efficiency is achieved by employing the separation of concern principle discussed in Section 1.2.1, i.e., a processing component (active component) is used to implement the KPN process behaviors, and the communication and synchronization is managed by a *communication controller* (CC). As illustrated in Figure 2.1, a CC connects a communication memory to the data bus of the processor (uP) it belongs to, and to an interconnect component. In addition, each CC implements the multi-FIFO organization of a CM as well as the processors local bus-based protocol for accessing the CM and the interconnect component. The latter is used to access FIFO buffers located in remote CMs. The usage of a CC is convenient for efficient implementation of the communication and synchronization mechanism, independent of the type of processing components.

Data communication and synchronization

We propose an efficient inter-processor synchronization mechanism for data communication in our platform exploiting the fact that the target MPSoC execution platform instances execute KPNs in which the data streams are modeled in FIFO channels. The proposed synchronization policy is a simple FIFO blocking write and read protocol. For better efficiency, we propose dual port communication memories, i.e., the CMs in Figure 2.1. In systems with a point-to-point interconnect topology, dual port FIFO buffer memories avoid arbitration of memory access because a memory connects only two processing components. Hence, both components in this case can perform a FIFO operation simultaneously (via a communication controller CC in Figure 2.1), the synchronization being blocking write/read operations on the FIFO buffers.

In case the system communication topology is not point-to-point, the memories and the communication links are shared between different processing components. We have devised a general approach to connect and to synchronize processing components that communicate data through distributed communication memories (CM). Assuming dual port memories, one

communication memory port is dedicated to a processing component (therefore, the memory becomes its local CM) and the other memory port is connected (through a CC) to the communication component. In this case, arbitration of memory access is significantly reduced because a processing component always can access its own (local) communication memory. Arbitration is required when other processing components need to access the memory using the communication component. We have developed a request-based synchronization mechanism for accessing remote communication memories. A processing component accesses a remote CM in three steps:

1. Request a connection to a remote communication memory (CM);
2. Transfer data from the remote CM;
3. Release the connection after the data transfer is completed.

A request to read from a FIFO located in a remote CM is generated to the communication component. A connection to the CM is granted only if the communication line is currently available and there is data in the corresponding FIFO. When a connection is not granted, the processing component blocks (its execution is suspended) until the connection is granted. Once the request is granted, the processing component has a direct connection to the remote CM. After transferring the data, the connection has to be released in order for other processing components to be able to read data from different FIFOs located in the same CM. A CM is not used by the processing components as a local memory for processing data. Instead, a CM and a communication link are used only to copy data locally for further processing which reduces the load on the communication component of the system. The request-based mechanism for synchronization and data communication between processing components in the MPSoC is implemented by the proposed communication controller (CC). In case a processing component is a programmable processor, we propose a SW/HW implementation mechanism. It consists of SW synchronization primitives that interact with HW communication controllers.

Communication controller

The structure of the communication controller (CC) we propose is shown in Figure 2.2. It consists of two main parts: INTERFACE Unit and FIFOs Unit. The INTERFACE Unit contains a Control Module, i.e., an address decoder, fifos' control logic, and logic to generate read requests to the communication component, and a processor interface (PI) module that implements the data bus protocol of a particular processing component. When a processing component has to write data to its local communication memory (CM), it first checks if there is room in the corresponding FIFO by reading its status. If the FIFO is full, the processing component blocks. Otherwise, it sends the data to the CC. The Control Module decodes the FIFO address sent by the processing component along with the data and generates control signals (select FIFO, write data, or read status) to the Write Module of the FIFOs Unit. The latter implements the multi-FIFO behavior. For each FIFO buffer, the FIFOs Unit contains read and write counters that indicate the read and write positions into the buffer. These counters are used as read and write address generators and their values are used to determine the status (empty or full) of a FIFO. The FIFOs Unit also includes a memory interface (MI)

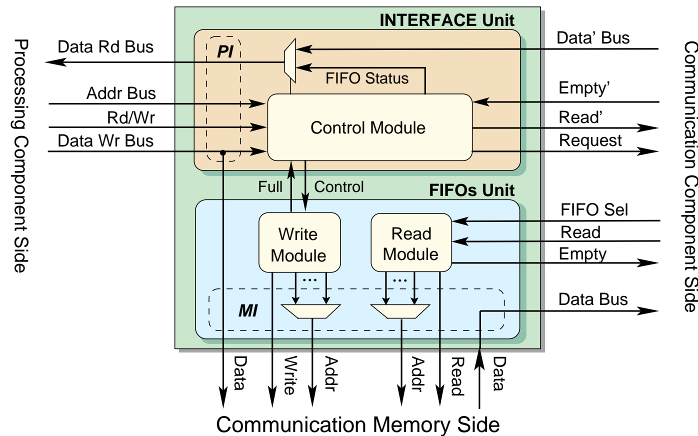


Figure 2.2: Proposed communication controller structure.

module that implements the access protocol to the communication memory connected to the CC, see the bottom part of Figure 2.2. Notice that since we consider dual-port memories and separate read and write logic, a FIFO in a CM can be accessed for read and write operations simultaneously by different processing components or two FIFOs in a CM can be accessed at the same time – one for read operation and one for write operation.

The structure of the CC is devised in a way that if a new type of a processing component is to be added to the platform, then that requires changes only to the PI module of the INTERFACE Unit of the CC. Modification is needed in order to implement the data bus protocol of the new processing component and to translate it to the interface of the Control Module of the CC. Similarly, if a CM is implemented by another type of memory component, e.g., a single-port (static or dynamic) memory, then – again – only the memory interface (MI) part of the FIFOs Unit has to be modified in order to match the timing characteristics and the physical interface of the new memory component. For a single-port memory, the MI module would also contain a simple arbiter for accessing the memory. A priority on writing is an appropriate policy to resolve access contention.

Recall that a processing component can access FIFOs located in other CMs via a communication component only for read operations. First, the processing component checks if there is any data in the FIFO the processor wants to read from. When a processor checks for data, the INTERFACE Unit sends a request to the communication component for granting a connection to the CM in which the FIFO is located. A connection is granted only if a communication line is available and there is data in the FIFO. When a connection is not granted, the processing component blocks until a connection is granted. When a connection is granted, the CC connects the data bus of the communication component to the data bus of the processing component and the latter reads the data from the CM where the FIFO is located. After the data is read the connection has to be released in order to allow other processing components to access the same CM. When data is read from a FIFO in a CM, the signals to the Read Module of the FIFOs Unit (*FIFO Sel* and *Read*) signals are generated by the communication component via the bottom part of the *communication component side* as a response to a request from another CC (processing component respectively), see Figure 2.2.

2.1.5 System-level platform model

The multiprocessor platform model we propose consists of a library of generic parameterized components and defines the way the components can be assembled assuming the (inter-component) communication and synchronization mechanisms discussed in Section 2.1.4. The platform model is used in the ESPAM tool for automated multiprocessor systems design and implementation. To enable efficient execution of KPNs with low overhead, the platform model allows for building MPSoCs that strictly follow the KPN semantics. Moreover, the platform model allows easily to construct platform instances at a high level of abstraction and to refine and translate them systematically and automatically to MPSoCs instances at RTL. This is achieved by applying techniques which are presented in Section 2.2. The generated MPSoCs are compliant with the multiprocessor platform we consider and discussed above. In addition, the platform model allows, together with the the KPN MoC we use as an application (programming) model, the generated platform instances to be programmed automatically by ESPAM.

Platform components

To support systematic and automated synthesis of MPSoCs, we have carefully identified a set of components which comprise the multiprocessor platform we consider. It contains the following components.

- **Processing Components**. The processing components implement the functional behavior of an MPSoC. The platform supports two types of processing components, namely programmable (ISA) processors and non-programmable, dedicated IP cores. The processing components have several parameters such as *type*, *number of I/O ports*, program and data *memory size*, etc.
- **Memory Components**. Memory components are used to specify the local program and data memories of the the programmable processors and to specify data communication storages (buffers) between the processing components (*Communication Memories*). In addition, the platform supports dedicated FIFO components used as communication memories in MPSoCs with a point-to-point topology. Important memory component parameters are *type*, *size*, and *number of I/O ports*.
- **Communication Components**. A communication component determines the interconnection topology of a multiprocessor platform instance. Some of the parameters of a communication component are *type* and *number of I/O ports*.
- **Communication Controller**. Compliant with our approach to build MPSoCs executing KPNs, communication controllers are used as glue logic realizing the synchronization of the data communication between the processors at hardware level. A communication controller implements an interface between processing, memory, and communication components. There are two types of CCs in our library. In case of a point-to-point topology, a CC implements only an interface to the dedicated FIFO components used as communication memories. If an MPSoC utilizes a communication component,

then the communication controller realizes a multi-FIFO organization of the communication memories. The structure of this type of communication controller was already discussed in Section 2.1.4. Important CC parameters are *number of FIFOs* and the *size* of each FIFO.

- **Memory Controllers**. Memory controllers are used to connect the local program and data memories to the ISA processors. Every memory controller has a parameter *size* which determines the amount of memory that can be accessed by a processor through the memory controller.

- **Peripheral Components** and **Controllers**. They allow data to be transferred in and out of the MPSoC platform, e.g., a Universal Asynchronous Receive-Transmit (UART). We have also developed a multi-port interface controller allowing for efficient (DMA-like) data communication between the processing cores by sharing an off-chip memory organized as multiple FIFO channels [65]. General off-chip memory controller is also part of this group of library components. In addition, *Timers* can be used for profiling and debugging purposes, e.g., for measuring execution delays of the processing components.

- **Links**. Links are used to connect the components in our system-level platform model. A link is transparent, i.e., does not have any type, and connects ports of two or more components together.

In our approach, we do not consider the design of processing components. Instead, we use IP cores (programmable processors and dedicated IPs) developed by third parties and propose a communication mechanism that allows efficient data communication (low latency) between these processing components. The devised communication mechanism is independent of the types of processing and communication components used in the platform instance. This results in a platform model that easily can be extended with additional (processing, communication, etc.) components.

2.2 Automated MPSoC Synthesis

Using the proposed platform model, a system designer can construct many alternative platform instances at a high (system) level of abstraction by connecting processing, memory, and communication components together using permissive interconnection rules. These platform instances are then automatically synthesized (i.e., translated to RTL descriptions) and programmed by ESPAM. Recall that for system design, ESPAM requires three input specifications: the *Application, Platform, and Mapping Specifications*. The application specification is in terms of a Kahn Process Network (KPN), the platform specification provides the topology of a multiprocessor platform, and the mapping associates components in the application specification and the platform specification together. In the mapping specification, mapping of KPN channels to communication memories is not specified explicitely. This mapping is implicit in the way processes are assigned to processors, see Section 2.3.3. Below, we give details about the platform specification which is followed by description of the system-level

```
1  <platform name = "myPlatform">
     <processor name = "uP1" type = "MB"
                dm = "18000" pm = "48000" >
        <port name = "IO1"/>
5    </processor>
     <processor name = "uP2" type = "MB"
                dm = "20000" pm = "4000" >
        <port name = "IO1"/>
10   </processor>
     <processor name = "uP3" type = "MB"
                dm = "6000" pm = "5000" >
        <port name = "IO1"/>
15      <port name = "IO2" type = "OPB" />
     </processor>
     <peripheral name="OMC" type="ZBTCTRL" size="1M" >
        <port name = "IO1" type = "OPB" />
20   </peripheral>
     <peripheral name="UART" type="UART" size="256" >
        <port name = "IO1" type = "OPB" />
     </peripheral>
25   <network name = "CB" type = "Crossbar">
        <port name = "IO1"/>
        <port name = "IO2"/>
        <port name = "IO3"/>
     </network>
30   <link name = "BUS1">
        <resource name = "CB" port = "IO1" />
        <resource name = "uP1" port = "IO1" />
     </link>
35   <link name = "BUS2">
        <resource name = "CB" port = "IO2" />
        <resource name = "uP2" port = "IO1" />
     </link>
40   <link name = "BUS3">
        <resource name = "CB" port = "IO3" />
        <resource name = "uP3" port = "IO1" />
     </link>
     <link name = "BUS4" />
45      <resource name = "uP3" port = "IO2" />
        <resource name = "OMC" port = "IO1" />
        <resource name = "UART" port = "IO1" />
     </link>
   </platform>
```

Figure 2.3: Platform specification.

to RTL MPSoC synthesis steps performed by ESPAM. The XML format of the application and the mapping specifications are discussed further, where appropriate, in this chapter.

2.2.1 Platform specification

For the discussions in this section, we use an example of a multiprocessor platform instance containing 3 processing components. The system-level specification of this MPSoC instance is depicted in Figure 2.3. The specification is written in XML format and consists of four parts which define processing components (three processors, lines 2-16), peripheral components (lines 18-24), communication (network) component (Crossbar, lines 25-29), and links (lines 30-48). The links specify the connections of the processors to the communication component. Every component has an instance name and different parameters characterizing the component. For example, all the processing components are $MicroBlaze$ programmable processors ($type =$ “MB“) and every core has the program (pm) and data (dm) memory size specified. The memory size affects the way the memory system is synthesized and optimized which is explained further in this chapter. Every processor has one port ($IO1$) which represents the local memory bus (LMB) of a processor. This port type is the default type in ESPAM, therefore, in the platform specification it can be omitted. Processor $uP3$ has another port specified ($IO2$) which is of type on-chip peripheral bus (OPB). This type represents the processor’s peripheral bus and it is used to connect peripheral components to the processor. In our example there are two peripherals, i.e., an off-chip memory controller OMC and a universal asynchronous receive transmit $UART$. The right part of Figure 2.3 specifies the connections between the components, i.e., the processors are connected to the communication component and the peripherals are connected to processor $uP3$.

Note that in the specification, a designer does not have to take care of memory structures, interface controllers, and communication and synchronization protocols. Our ESPAM tool takes care of this in the platform synthesis process by implementing our general approach to connect and synchronize processing cores of arbitrary types via a communication component and communication controllers as discussed in Section 2.1. In this way, unnecessary details are hidden at the beginning of the design, keeping the abstraction of the input platform specification very high.

2.2.2 Platform synthesis

The automated translation of the high-level specifications to RTL descriptions of an MPSoC goes in several steps. They are illustrated in Figure 2.4 and are grouped in:

- **Models initialization.** Using the platform specification, an MPSoC instance is created by initializing an abstract platform model in ESPAM. Based on the application and the mapping specification, three additional abstract models are initialized: application, schedule, and mapping models.
- **Platform synthesis.** ESPAM elaborates and refines the abstract platform model to a detailed parameterized platform model, compliant with the approach discussed in Section 2.1. Based on the application and the mapping models, a parameterized process network model is created as well.
- **Platform generation.** Parameters are set and ESPAM generates a platform instance implementation using the RTL version of the components in our library. In addition, ESPAM generates program code for each programmable processor.

Models initialization consists of two steps, i.e., initializing the internal models in ESPAM that capture platform, application, and mapping information, and running a consistency check on the models in order to detect errors and to facilitate correct-by-construction MPSoC designs. Platform synthesis is comprised of several steps as well. These are platform and mapping models elaboration, process network synthesis (PN), and platform instance refinement steps. As a result of the platform elaboration, ESPAM creates a detailed parameterized model of a platform instance. After the elaboration, a refinement (optimization) step is applied by ESPAM in order to improve resource utilization and efficiency. In our approach, the mapping specification gives the relation between KPN processes and processing components only. Then, ESPAM determines automatically the most efficient mapping of FIFO channels to communication memories. This is done in the mapping elaboration step, in which the mapping model is analyzed and augmented with the mapping of FIFO channels to communication memories. The PN synthesis is a translation of the *Approximated Dependence Graph* (ADG) model and the *Schedule Tree* (STree) model into a (parameterized) process network model. The platform generation consists of setting parameters step which completely determines a platform instance, and code generation step which generates hardware and software descriptions of an MPSoC. Each of these steps is described below in detail. Automated programming and SW code generation are discussed in Section 2.3.

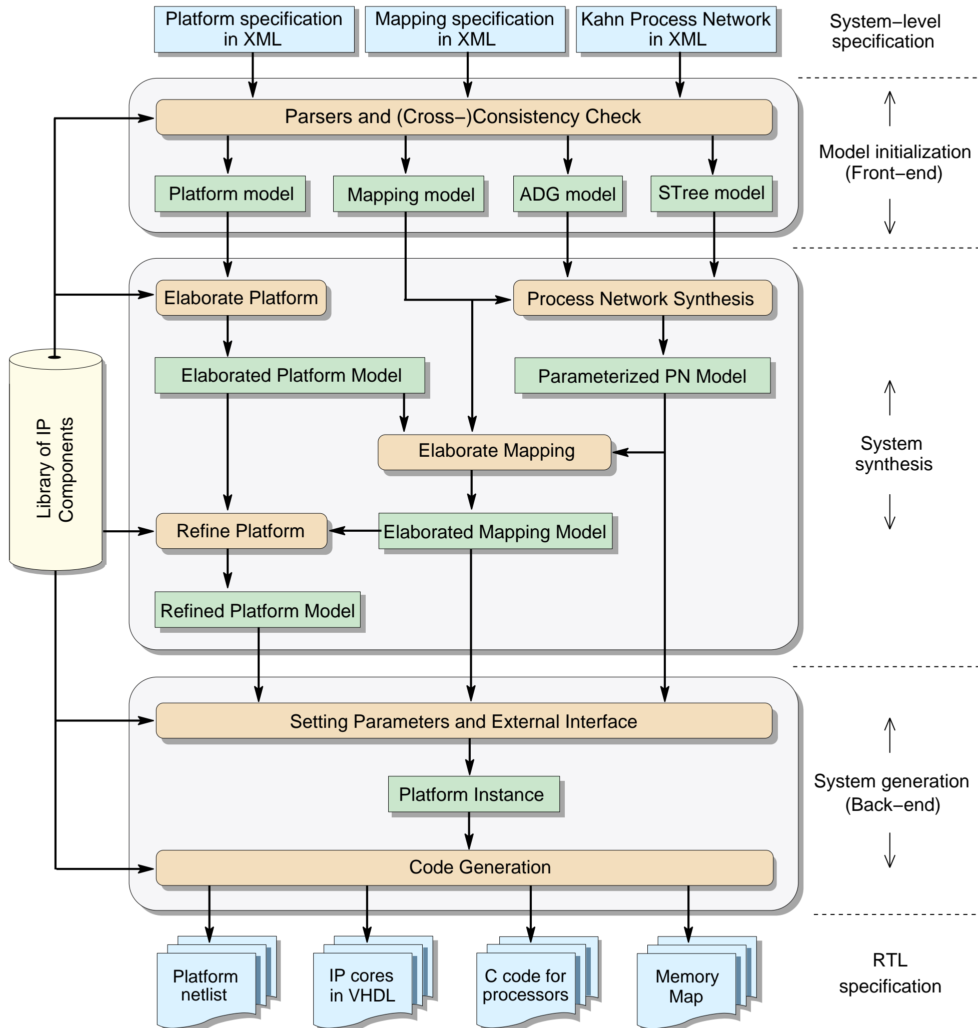


Figure 2.4: System-level to RTL MPSoC synthesis steps performed by ESPAM.

Models initialization

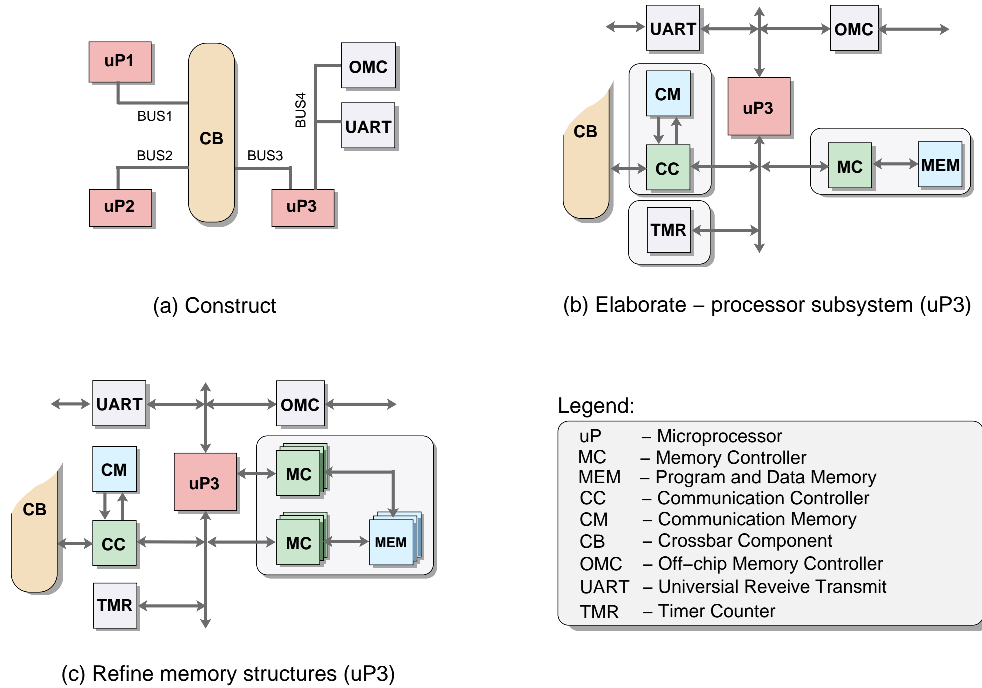
Models initialization is the first step in ESPAM for implementing an MPSoC. In this step, ESPAM constructs a platform instance from the input platform specification by initializing an abstract platform model. This is done by instantiating and connecting the components in the specification using abstract components from the library. The abstract model represents an MPSoC instance without taking target execution platform details into account. The model includes key system components and their attributes as defined in the platform specification. A graphical representation of an abstract MPSoC instance is depicted in Figure 2.5(a). It consists of 3 processing components ($uP1$, $uP2$, and $uP3$) connected to a crossbar (CB) communication component, and two peripheral components (OMC and $UART$) connected to $uP3$.

There are three additional abstract models in ESPAM which are also created and initialized, i.e., an application model, a schedule model, and a mapping model, see the top of Figure 2.4. The application specification consist of two annotated graphs, i.e., a KPN represented by an *Approximated Dependence Graph* (ADG) and a *Schedule Tree* (STree) representing one valid global schedule of the KPN. Consequently, the ADG and the STree models in ESPAM are initialized, capturing in a formal way all the information that is present in the application specification. The mapping model is constructed and initialized from the mapping specification. The objective of the mapping model in ESPAM is to capture the relation between the KPN processes in an application and the processing components in an MPSoC instance on the one hand, and the relation between FIFO channels and communication memories on the other. The mapping model in ESPAM contains important information which enables the generation of the memory map of the system in an automated way. This is a crucial part of the automated MPSoC programming that ESPAM provides. Recall that after initialization, the mapping model contains information about the mapping of the KPN processes to processing components only. Mapping of communication channels is related to the way processes are mapped to processing components, and therefore, the mapping of channels can not be arbitrary. This is performed by ESPAM automatically, i.e., during the mapping model elaboration step, ESPAM analyses the mapping model and determines the mapping of FIFO channels to communication memories.

After initializing the internal models, ESPAM runs a consistency check on the constructed platform model (instance) and cross-consistency check between the platform, the application and the mapping models. The consistency check on the platform is to find incorrect connections as well as to check whether all the used components are part of the library. In addition, all the ports connected to the same link have to be of the same type, e.g., local memory bus type or peripheral bus type. The cross-check guarantees that to every processing component in the mapping model corresponds a processing component in the platform model, and the processes in the mapping model correspond to the processes in the application model, respectively. In addition, every process has to be assigned to a processing component. The consistency check is an important step towards correct-by-construction designs.

Platform model elaboration

In this step, ESPAM elaborates the abstract platform model to a detailed parameterized model. The details in this model come from additional components added by ESPAM in order to construct a complete system. The elaborate step is illustrated in Figure 2.5(b). For brevity, only one processor ($uP3$) is shown. First, ESPAM generates the processors' memory subsystem, i.e., it automatically attaches memories (MEM) and memory controllers (MCs) to each processor. In addition, based on the type of the processors instantiated in the first step, the tool automatically synthesizes, instantiates, and connects all necessary communication controllers (CCs) and communication memories (CMs) compliant with our approach discussed in Section 2.1. Also, ESPAM provides an infrastructure for observability and control of the generated platform instances. For example, when execution delays are to be measured, a timer (TMR) peripheral component is connected to every processor. Using timers or not in our MPSoCs is controlled by a flag in the ESPAM tool when synthesizing a platform.



(a) Construct

(b) Elaborate – processor subsystem (uP3)

(c) Refine memory structures (uP3)

Figure 2.5: An MPSoC instance at different levels of details in ESPAM.

Since in the elaborate step ESPAM uses abstract models of the components, the created elaborate model is still abstract in the sense that no target specific issues are considered.

Process network synthesis

This step is a translation of the *Approximated Dependence Graph* (ADG) model and the *Schedule Tree* (STree) model into a (parameterized) process network (PN) model. The starting point for this PN synthesis step is the information captured in the ADG and the STree. This information is enough to generate a set of process networks with different topologies and degree of exploited parallelism which is determined by the mapping model. The process network model is created gradually by creating the PN topology, followed by creating the PN behavior. The topology of the process network is created by grouping nodes and edges of the ADG into processes and channels in the PN. The grouping is based on the information delivered by the mapping model. The behavior of the PN is created using procedures [66–68] that operate on the PN model and use the information from the STree. The sequential behavior of a process [6] implies that the function calls that have to be executed inside a process are executed in a sequential order. In this PN synthesis step, such order is derived from the STree in a way that the PN execution is deadlock free. For details about the process network synthesis, we refer to [66].

Mapping model elaboration

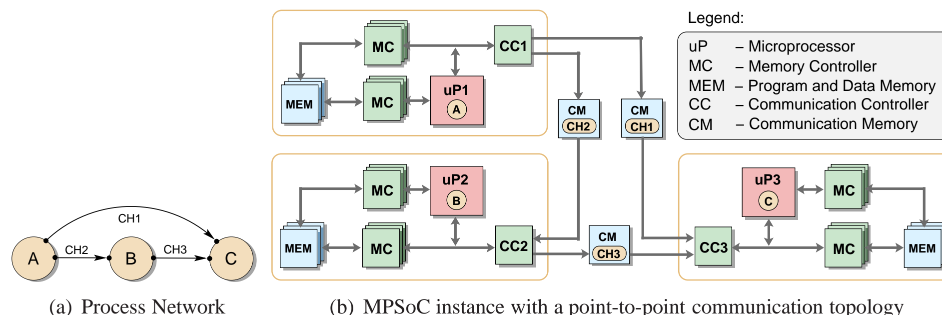
Elaboration of the mapping model is required in order to generate an assignment of FIFO channels to communication memories. Since the initialized mapping model gives only the relation between the processes and the processing components, mapping of FIFO channels to memories is implicit. The latter is performed automatically by ESPAM following the principle that a processor can write only to its local communication memory. As a result, the mapping model is augmented with channels which give the relation between the FIFO channels from the application model and the communication memories in the platform model. Recall that a communication controller (CC) organizes a communication memory as multiple FIFO channels. Therefore, the created FIFO mapping is used for determing important CC parameters, i.e., the number of channels of each CC and the size of each channel. This information is further used in the generation of the memory map of the system, i.e., for generation of the physical read and write addresses of the FIFO channels (described in Section 2.3.3).

Platform model refinement

After elaboration, the constructed platform model contains enough information to proceed with the generating of the system implementation at RTL. However, based on the type of the components used to build a platform instance, a refinement (optimization) of the detailed parameterized platform model is applied by ESPAM in order to improve resource utilization and efficiency. The refinement step includes program and data memory refinement and compaction in case of processing components with RISC architecture, memory partitioning, and building the communication topology in case of point-to-point MPSoCs.

Memory refinement and compaction. To keep the abstract model and the elaborate procedure general, we assume that a processor has a continuous address space used for program and data. Therefore, ESPAM instantiates initially a single memory and a memory controller connected to the data bus of the processor as shown in Figure 2.5(b). However, processors with a RISC architecture have separate program and data address spaces and use dedicated instruction and data busses for independent access to program and data. This is the case with the $MicroBlaze$ processor we use in the example. Therefore, ESPAM refines the processor memory system by instantiating and connecting to the processor an additional memory controller and a memory. This is followed by the memory compaction step which exploits the dual-port feature of the memory component we consider. ESPAM combines the program and the data memories of a programmable processor into a single memory unit, i.e., one memory port is connected to the instruction bus and the other to the data bus of a processor. Because of the dual port feature, the memory can still be accessed simultaneously allowing the processor to fetch opcode and data without compromising performance.

Using the proposed refinement, a single memory gives the opportunity for better utilization of the available memory resources. Consider $uP1$ from the example. It requires 18 KB of memory for data and 48 KB for program (see Figure 2.3). However, a single memory component, may have a size which is only a power of 2. Therefore, ESPAM would instantiate 32 KB for data and 64 KB for the program memory of $uP1$ if separate memories would be used. This means that 30 KB more than the required amount of memory is dedicated to $uP3$. By combining program and data into a single memory, only one memory of 64 KB is used which is exactly the required amount of memory.



(a) Process Network (b) MPSoC instance with a point-to-point communication topology

Figure 2.6: Point-to-Point Platform Synthesis Example.

Memory partitioning. The memory compaction alone does not lead to optimal memory utilization if we consider, e.g., processor $uP2$. It requires 20 KB of data and 4 KB of program memories. Even if both memories are combined (requiring 24 KB), ESPAM would instantiate 32 KB which is 8 KB more that the required amount. To cope with this issue, ESPAM applies another refinement on the memory system, i.e., it automatically partitions the memory using multiple controllers and memories. For processor $uP2$, ESPAM partitions the memory system into two memories (16 KB and 8 KB) each having a size which is a power of 2, using two program and two data memory controllers. Note that this refinement of the memory system does not depend on the type of the processor.

As a result of the memory refinement steps described above, there may be several program and data memory controllers connected to a processor and several physical memories comprising its program and data address space. These refinement steps, as illustrated in Figure 2.5(c), lead to better utilization of the memory resources which is very important in multiprocessor embedded system design. In [69], we made the important observation that the available on-chip memory (of an FPGA) is the only major factor limiting the size of an MPSoC that can fit on a single chip. Therefore, without paying special attention to the memory distribution between the processors in an MPSoC, it is not a matter of how efficient the system is implemented but whether it can be implemented at all. By applying the proposed refinement steps, memory savings can be substantial (and they increase with the number of processors) allowing to build larger MPSoCs.

Synthesis of a point-to-point communication topology. In case of a point-to-point communication topology, the number of direct connections between the processing components in an MPSoC is the same as in the Kahn process network it executes. Since there is no communication component such as a crossbar or a bus, there is no sharing of communication links, and consequently, there are no requests for granting connections. Therefore, no additional communication delay is introduced in the platform instance. Because of this, the highest possible communication performance can be achieved in such multiprocessor platforms.

Under the conditions that each communication memory (CM) contains only one FIFO channel and each processing component writes data only to its local CM (in compliance with the approach discussed in Section 2.1.4), the ESPAM tool synthesizes a communication topology

with point-to-point connections in the following automated way. First, for each process in the KPN, ESPAM instantiates a processor together with the processor's memory system and a communication controller (CC). This is done in the platform model elaboration step of the MPSoC synthesis as previously described. The memory refinement step is applied as well. Then, ESPAM finds all the channels which a process writes to. For every such channel, the tool instantiates a CM and assigns the channel to this CM. Finally, ESPAM connects the CM to the already instantiated CC of the corresponding processor.

In Figure 2.6(b), we give an example of a point-to-point multiprocessor platform instance generated by ESPAM. The MPSoC implements the KPN depicted in Figure 2.6(a) where each process is executed on a separate processor. There are three channels that have to be assigned to three CMs. Following the procedure above, ESPAM finds that $CH1$ and $CH2$ are written to by process A – see Figure 2.6(a). Assume that process A is assigned to be executed onto processor $uP1$ (process B onto processor $uP2$, and C onto $uP3$ respectively). Therefore, CMs corresponding to $CH1$ and $CH2$ are instantiated and connected to communication controller $CC1$ of processor $uP1$. Similarly, a CM corresponding to $CH3$ is instantiated and connected to $CC2$ of $uP2$. Process C is assigned to $uP3$ and since process C only reads data from $CH1$ and $CH3$ no more CMs are instantiated and connected. The communication controller of $uP3$ ($CC3$) is simply connected to the already instantiated CMs corresponding to $CH1$ and $CH3$. Similarly, $CC2$ is connected to the CM corresponding to $CH2$. Notice from Figure 2.6(b) that a CC is connected to more than one CM. When a CM contains only one FIFO, it is implemented as a dedicated FIFO component from the library which is more efficient than using the pair CC – CM. In order to connect one or more dedicated FIFO components to a processor, as in the case of a point-to-point communication topology, we use a simplified version of the communication controller (CC) described in Section 2.1.4. The simplified controller only translates the processor data bus signals to FIFO input/output signals.

Setting parameters and external interface

Setting the parameters of the components in the platform model completes a platform instance which allows an MPSoC description to be generated, ready for final implementation. Important parameters that are set at this step are:

- **Memory size.** Sets the size of the program and data memory components. Based on the mapping information, the size of the FIFO buffers in every communication controller are set as well;
- **Address space of the programmable processors.** Based on the size of the program and data memories, ESPAM sets proper addresses of all memory controllers of a processor as well as the address of the communication controller and all peripheral components;
- **Memory map of the system.** Based on the MPSoC topology and the addresses of the communication controllers, ESPAM sets the memory map of the system, i.e., the values of the physical read and write addresses of each FIFO chanel of the system are defined.

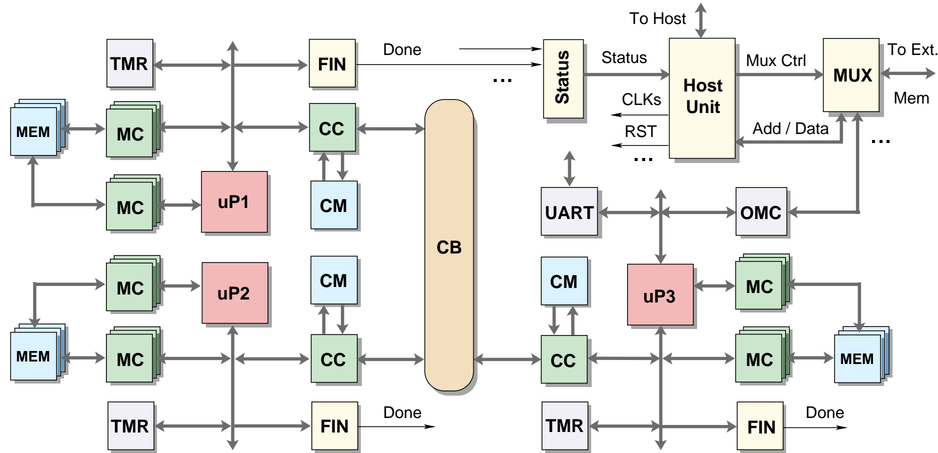


Figure 2.7: The target MPSoC platform instance generated by ESPAM.

In addition, based on the target physical platform features, ESPAM instantiates and connects to the platform instance external interface components realizing a connection between the platform instance and its environment, e.g., a host PC. This allows for sending of reset, start, or stop commands to the MPSoC, as well as reading the status of the MPSoC, reading and writing to the MPSoC memories, etc. If we consider again the platform specification example in Figure 2.3, a graphical representation of the final platform instance generated by ESPAM is shown in Figure 2.7. It consists of three processor subsystems synthesized by ESPAM as previously discussed in this section together with some external interface components. There are additional (FIN) controllers connected to the data bus of each processor indicating that a processor has finished an execution. This information is collected by a special $Status$ entity which generates the status of the MPSoC instance (see the top part in Figure 2.7). The other part of the interface (MUX) realizes an access to an external memory attached to the multiprocessor system.

Code generation: HW description of a platform instance generated by ESPAM

The final step in the platform synthesis process performed by ESPAM to convert the system-level specifications to RTL descriptions is the actual platform instance generation, i.e., hardware and software descriptions of the system as illustrated in Figure 2.4. The HW description generated by ESPAM consists of two parts:

- **Platform topology.** This is a netlist description defining the multiprocessor platform topology that corresponds to the platform instance synthesized by ESPAM, e.g., the instance in Figure 2.7. This description contains the components of the platform instance with the appropriate values of their parameters, and the connections between the components in the form compliant with the input requirements of the commercial tool used for low-level synthesis.

- **Hardware descriptions of the MPSoC components.** To every component in the platform instance corresponds a detailed description at RTL. Some of the descriptions are predefined (e.g., processors, memories, etc.), and ESPAM selects them from the library of components and sets their parameters in the platform netlist. However, some descriptions are generated by ESPAM, e.g., an IP Module used for integrating a third party IP core as a processing component in an MPSoC (discussed in Section 2.4). Based on the application specification, the IP Module description is generated by ESPAM every time an MPSoC instance is synthesized.

In ESPAM, a software engineering technique called *Visitor* [70] is used to visit the PN and platform model structures and to generate code. This code can be expressed in any programming language, i.e., ESPAM generates VHDL for the HW part and C/C++ for the SW part of the MPSoC description. The C/C++ software generated by ESPAM for each processor in the system consists of code implementing the functional behavior together with code for synchronization of the communication between the processors. The program code generated by ESPAM is given to a standard GCC compiler to generate executable code for each processor. The automated MPSoC programming and software code generation is discussed in the next section.

2.3 Automated Programming of MPSoCs

In this section, we present in detail our approach for systematic and automated programming of MPSoCs synthesized with ESPAM. Recall that we use the KPN MoC as programming model in ESPAM. Such model is created automatically by the PNGEN tool from sequential, static affine nested loop programs (SANLP). In order to program an MPSoC, ESPAM converts this KPN model to software (C/C++) code including *computation* code implementing the functional behavior and *control* code for synchronization of the communication between the processors. The synchronization code contains a memory map of the MPSoC, i.e., physical addresses of the FIFO channels, and read/write synchronization primitives. These primitives interact with the communication controllers, and together, they implement the blocking read/write synchronization mechanism. The primitives are inserted automatically by ESPAM in the places of the processors' code where read/write access to a FIFO is performed.

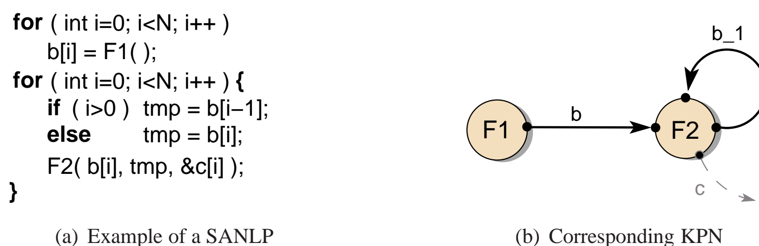
This section is organized as follows. First, we discuss the class of static affine nested loop programs we consider as well as how we derive KPNs from such SANLPs. Then, for the sake of clarity, we explain the main steps in the ESPAM programming approach by going through an illustrative example. First, we give an example of input application and mapping specifications for ESPAM. Next, from these example specifications we show how the SW code for each processor in an MPSoC is generated, and present our SW synchronization and communication primitives inserted in the code. Finally, we explain how the memory map of the MPSoC is generated.

2.3.1 Automated Derivation of Process Networks

The techniques we have recently developed for derivation of KPNs were implemented in the PNGEN tool [7]. The input to PNGEN is a SANLP written in C and the output is a KPN specification in XML format – see Figure 1.2. Below, we introduce the SANLPs with their restrictions and explain how a KPN is derived based on a modified data-flow analysis. We have modified the standard data-flow analysis in order to derive KPNs that have less inter-process communication FIFO channels compared to the KPNs derived by using previous work [22, 24].

SANLPs and modified data-flow analysis

SANLPs are programs that can be represented in the well-known polytope model [71]. That is, a SANLP consists of a set of statements and function calls, each possibly enclosed in loops and/or guarded by conditions. The loops need not be perfectly nested. All lower and upper bounds of the loops as well as all expressions in conditions and array accesses have to be affine functions of enclosing loop iterators and static parameters. The parameters are symbolic constants, i.e., their values may not change during the execution of the program. The above restrictions allow a compact mathematical representation of a SANLP through sets and relations of integral vectors defined by linear (in)equalities, existential quantification and the union operation. In particular, the set of iterator vectors for which a function call is executed is an integer set called the *iteration domain*. The linear inequalities of this set correspond to the lower and upper bounds of the loops enclosing the function call. For example, the iteration domain of function `F1` in Figure 2.8(a) is $\{i \mid 0 \leq i \leq N-1\}$. Iteration domains



(a) Example of a SANLP (b) Corresponding KPN

Figure 2.8: SANLP fragment and its corresponding KPN.

form the basis of the description of the processes in our KPN model, as each process corresponds to a particular function call. For example, there are two function calls in the program fragment in Figure 2.8(a) representing two application tasks, namely `F1` and `F2`. Therefore, there are two processes in the corresponding process network as shown in Figure 2.8(b). The granularity of `F1` and `F2` determines the granularity of the corresponding processes. The FIFO channels are determined by the array (or scalar) accesses in the corresponding function call. All accesses that appear on the left hand side or in an address-of (`&`) expression for an argument of a function call are considered to be *write accesses*. All other accesses are considered to be *read accesses*.

To determine the FIFO channels between the processes, we may perform standard array data-flow analysis [72]. That is, for each execution of a read operation of a given data element in a function call, we need to find the source of the data, i.e., the corresponding write operation that wrote the data element. However, to reduce communication FIFO channels between different processes, in contrast to the standard data-flow analysis and in contrast to [22, 24], we also consider all previous read operations from the same function call as possible sources of the data. That is why we call our approach a modified array data-flow analysis. The problem to be solved is then: given a read from an array element, what was the last write to or read from that array element? The last iteration of a function call satisfying some constraints can be obtained using Parametric Integer Programming (PIP) [73], where we compute the lexicographical *maximum* of the write (or read) source operations in terms of the iterators of the "sink" read operation. Since there may be multiple function calls that are potential sources of the data, and since we also need to express that the source operation is executed before the read (which is not a linear constraint, but rather a disjunction of n linear constraints, where n is the shared nesting level), we actually need to perform a number of PIP invocations.

For example, the first read access in function call `F2` of the program fragment in Figure 2.8(a) reads data written by function call `F1`, which results in a FIFO channel from process `F1` to process `F2`, i.e., channel `b` in Figure 2.8(b). In particular, data flows from iteration i_{w} of function `F1` to iteration $i_{\mathrm{r}} = i_{\mathrm{w}}$ of function `F2`. This information is captured by the integer relation

$$D_{\texttt{F1}\rightarrow\texttt{F2}} = \{(i_{\mathrm{w}}, i_{\mathrm{r}}) \mid i_{\mathrm{r}} = i_{\mathrm{w}} \wedge 0 \leq i_{\mathrm{r}} \leq N-1\}$$

For the second read access in function call `F2`, after elimination of the temporary variable `tmp`, the data has already been read by the same function call after it was written. This results in a self-loop channel `b_1` from `F2` to itself described as

$$D_{\texttt{F2}\rightarrow\texttt{F2}} = \{(i_{\mathrm{w}}, i_{\mathrm{r}}) \mid i_{\mathrm{w}} = i_{\mathrm{r}} - 1 \wedge 1 \leq i_{\mathrm{r}} \leq N-1\} \cup \{(i_{\mathrm{w}}, i_{\mathrm{r}}) \mid i_{\mathrm{w}} = i_{\mathrm{r}} = 0\}$$

In general, we obtain pairs of write/read and read operations such that some data flows from the write/read operation to the (other) read operation. These pairs correspond to the channels in our process network. For each of these pairs, we further obtain a union of integer relations

$$\bigcup_{j=1}^{m} D_j(\mathbf{i}_{\mathrm{w}}, \mathbf{i}_{\mathrm{r}}) \subset \mathbb{Z}^{n_1} \times \mathbb{Z}^{n_2},$$

with n_1 and n_2 the number of loops enclosing the write and read operation, respectively, that connect the specific iterations of the write/read and read operations such that the first is the source of the second. As such, each iteration of a given read operation is uniquely paired off to some write or read operation iteration.

2.3.2 Automated programming – input specification

Recall that the input to ESPAM consists of platform instance, application, and mapping specifications. The platform specification was already discussed in Section 2.2 and an example was given in Figure 2.3. In this section, we give an example of application and mapping specifications. We will use this example also in our discussion about the program code generated by ESPAM in the next section. Consider an application specified as a KPN consisting of five processes communicating through seven FIFO channels. A graphical representation of the application is shown in Figure 2.11(a). Part of the corresponding XML application specification for this KPN is shown in Figure 2.9(a). Recall that this KPN in XML format is generated automatically by our PNGEN tool using the techniques presented in Section 2.3.1. For the sake of clarity, we show only the description of process $P1$ and channel $FIFO1$ in the XML code. The other processes and channels of the KPN are specified in an identical way. $P1$ has one input port and one output port defined in lines 3-8. $P1$ executes a function called $compute$ (line 9). The function has one input argument (line 10) and one output argument (line 11). There is a strong relation between the function arguments and the ports of a process given at lines 4 and 7. The information how many times function $compute$ has to be fired during the execution of the application is determined by a parameterized *iteration domain* (see Section 2.3.1) which is captured in a compact (matrix) form at lines 12-15. There are two matrices representing the function domain which corresponds to a nested *for*-loop structure. It originates from the structure of the initial (static and affine) nested loop program. In this particular example, there is only one *for*-loop with index k and parameter N. The parameter is used in determining the upper bound of the loop. The range of the loop index k is determined at line 13. This matrix represents the following two inequalities:

$$\begin{aligned} k-2 &\geq 0 \\ -k+2N-1 &\geq 0 \end{aligned}$$

and therefore, $2 \leq k \leq 2N-1$. In the same way, the matrix at line 14 determines the range of parameter N, i.e., $3 \leq N \leq 384$. Similar information for each port is used to determine at which iterations an input port has to be read and consequently, at which iterations, an output port has to be written. However for brevity, this information is omitted in Figure 2.9(a). Lines 19-24 show an example of how the topology of a KPN is specified: $FIFO1$ connects processes $P1$ and $P3$ through ports $p1$ and $p4$.

An example of a mapping specification is shown in Figure 2.9(b). It assumes an MPSoC with four processing components, namely, $uP1$, $uP2$, $uP3$, and $uP4$, and five KPN processes: $P1$, $P2$, $P3$, $P4$, and $P5$. The XML format of the mapping specification is very simple. Process $P4$ is mapped onto processor $uP1$ (see lines 3-5), processes $P2$ and $P5$ are mapped onto processor $uP2$ (lines 7-10), process $P3$ is mapped for execution on processor $uP3$, and finally, process $P1$ is mapped on processor $uP4$. In the mapping specification, the mapping of channels to communication memories is not specified. Recall that this mapping is related to the way processes are mapped to processors, and therefore, the mapping of channels to communication memories can not be arbitrary. The mapping of channels is performed by ESPAM automatically which is discussed in the next section.

```
<application  name ="myKPN">
  <process   name ="P1">
    <port  name = "p2"  direction = "in"  />
       <var  name = "in_0"  type = "myType"  />
    </port
    <port  name = "p1"  direction = "out" />
       <var  name = "out_0" type = "myType" />
    </port
    <process_code     name = "compute" >
       <arg  name = "in_0"  type = "input" />
       <arg  name = "out_0" type = "output" />
       <loop   index = "k"  parameter = "N" >
          <loop_bounds  matrix = "[1,  1,0,–2;  1,–1,2,–1]"/>
          <par_bounds  matrix = "[1,0,–1,384;  1,0,  1, –3]"  />
       </loop
    </process_code
  </process>
       <!–– other processes omitted ––>
  <channel  name = FIFO1    size = "1">
    <fromPort name = "p1"/>
    <fromProcess  name = "P1"  />
    <toPort  name = "p4" />
    <toProcess name = "P3"  />
  </channel>
       <!–– other channels omitted ––>
</application>
```

(a) Application specification

```
<mapping name = "myMapping"  >

   <processor name = "uP1"   >
       <process  name = "P4"/>
   </processor>

   <processor name = "uP2"   >
       <process  name = "P2" />
       <process  name = "P5" />
   </processor>

   <processor name = "uP3"   >
       <proces    name = "P3" />
   </processor>

   <processor name = "uP4"   >
       <process  name = "P1" />
   </processor>

</mapping>
```

(b) Mapping specification

Figure 2.9: Example of Application and Mapping Specifications.

2.3.3 Code generation: SW code for processors

ESPAM uses the initial sequential application program, the corresponding KPN application specification, and the mapping specification to generate automatically software (C/C++) code for each processor. The code for a processor contains $control$ code and $computation$ code. The $computation$ code transforms the data that has to be processed by a processor and it is grouped into function calls in the initial sequential program. ESPAM extracts this code directly from the sequential program. The $control$ code (for-loops, if-statements, etc.) determines the control flow, i.e., when and how many times data reading and data writing have to be performed by a processor as well as when and how many times the $computation$ code has to be executed in a processor. The $control$ code of a processor is generated by ESPAM according to the KPN application specification and the mapping specification as we explain below.

According to the mapping specification in Figure 2.9(b), process $P1$ is mapped onto processor $uP4$ (see lines 16-18). Therefore, ESPAM uses the XML specification of process $P1$ shown in Figure 2.9(a) to generate the $control$ C code for processor $uP4$. The code is depicted in Figure 2.10(a). At lines 4-7, the type of the data transferred through the FIFO channels is declared. The data type can be a scalar or more complex type. In this example, it is a structure of 1 boolean variable and a 64-element array of integers, a data type found in the initial sequential program. There is one parameter (N) that has to be declared as well. This is done at line 8 in Figure 2.10(a). Then, at lines 10-19 in the same figure, the behavior of processor $uP4$ is described. In accordance with the XML specification of

```
1  #include "primitives.h"
   #include "memoryMap.h"

   struct myType {
5      bool flag;
       int  data[64];
   };
   int N = 384;

10 void main( ) {
     myType in_0;
     myType out_0;

     for ( int k=2; k<=2*N–1; k++ ) {
15       read( p2, &in_0, sizeof(myType) );
         compute( in_0, &out_0 );
         write( p1, &out_0, sizeof(myType) );
     }
19 }
```

(a) Control code for processor $uP4$

```
1  void read( int* port, void* data, int length ) {
     int *req_&_rd = 0xE0000000; // Address in a CC
     int *isEmpty = req_&_rd + 1;
     *req_&_rd = 0x80000000 | (port); // Write a request
5    for ( int i=o; i<length; i++ ) {
       // reading is blocked if a FIFO is empty
       while ( *isEmpty ) { }
       (byte* data)[i] = *req_&_rd; // read from a FIFO
     }
10   *req_&_rd = 0x7FFFFFFF&(inPort);
   }

   void write( int* port, void* data, int length ) {
     int *isFull = port + 1;
15   for ( int i=o; i<length; i++ ) {
       // writing is blocked if a FIFO is full
       while ( *isFull ) { }
       *port = (byte* data)[i]; // write to a FIFO
     }
20 }
```

(b) Read and write communication primitives

Figure 2.10: Source code generated by ESPAM.

process $P1$ in Figure 2.9(a), the function $compute$ is executed $2 * N - 2$ times. Therefore, a *for*-loop is generated in the $main$ routine for processor $uP4$ in lines 14-18 in Figure 2.10(a). The $computation$ code in function $compute$ is extracted from the initial sequential program. This code is not important for our example, hence, it is not given here for the sake of brevity. The function $compute$ uses local variables in_0 and out_0 declared in lines 11 and 12 in Figure 2.10(a). The input data comes from $FIFO2$ through port $p2$ and the results are written to $FIFO1$ through port $p1$ – see Figure 2.11(a). Therefore, before the function call, ESPAM inserts a *read* primitive to read from $FIFO2$ initializing variable in_0 and after the function call, ESPAM inserts a *write* primitive to send the results (the value of variable out_0) to $FIFO1$ as shown in Figure 2.10(a) at lines 15 and 17. When several processes are mapped onto one processor, a schedule is required in order to guarantee a proper execution order of these processes onto one processor. The ESPAM tool automatically finds a local static schedule from the STree model (see Section 2.2.2) based on the grouping technique for processes presented in [23].

SW communication and synchronization primitives

Recall that the FIFO channels are mapped onto the communication memories of our MPSoC platform instances and the multi-FIFO organization of a communication memory is realized by the corresponding communication controller (CC) as described in Section 2.1.4. A FIFO channel is seen by a processor as two memory locations in its communication memory address space. A processor uses the first location to read the status of the FIFO. The status indicates whether a FIFO is full (data cannot be written) or empty (data is not available). This information is used for the inter-processor synchronization. The second location is used to read/write data from/to the FIFO buffer, thereby, realizing inter-processor data transfer.

The described behavior is realized by the SW communication and synchronization primitives interacting with the HW communication controllers. The code implementing the *read* and *write* primitives used in lines 15 and 17 in Figure 2.10(a), is shown in Figure 2.10(b). Both read and write primitives have 3 parameters: $port$, $data$, and $length$. Parameter $port$ is the address of the memory location through which a processor can access a given FIFO channel for reading/writing. Parameter $data$ is a pointer to a local variable and $length$ specifies the amount of data (in bytes) to be moved from/to the local variable to/from the FIFO channel.

The primitives implement the blocking synchronization mechanism between the processors in the following way. First, the status of a channel that has to be read/written is checked. A channel status is accessed using the locations defined in lines 3 and 14. The blocking is implemented by while loops with empty bodies in lines 7 and 17. A loop iterates (does nothing) while a channel is full or empty. Then, in lines 8 and 18 the actual data transfer is performed.

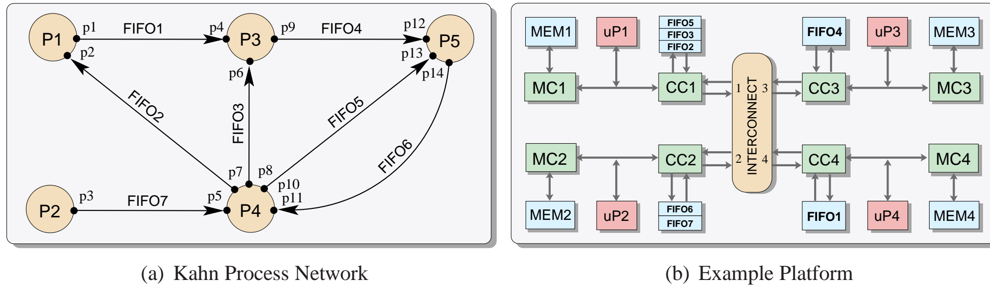
The *read* primitive in Figure 2.10(b) implements also the read request mechanism for accessing remote CMs using a communication component as discussed in Section 2.1.4. This is done in several steps. Variable *req_&_rd*, initialized at line 2, is a pointer to the communication controller. This pointer is used to write *read requests* to the CC and to read *data* from a remote CM. The request data has a special format which can be seen at line 4. It consists of the address (*port*) of the requested FIFO and a read request flag located at the most significant bit of the request word. The physical addresses of the FIFOs in our MPSoCs are discussed in the next section. Setting the request flag at line 4 triggers the generation of a read request by the CC to the communication component. At the same time, the CC asserts the status of the requested FIFO to *empty* until the connection to the remote memory is granted[1]. This blocks the processor in the *while*-loop at line 7. When the connection is granted, the CC propagates the actual status of the FIFO, and the processor reads the data performing the synchronization in the same way as described in the previous paragraph. Once the data has been transferred, the connection is released which is done at line 10 by clearing the request flag in the CC.

Note that the request mechanism for reading data is used only in MPSoCs utilizing a communication component (e.g. a crossbar or a bus). Therefore, in MPSoCs with a point-to-point communication topology, there are no read requests generated. In this case, the *read* synchronization primitive used by ESPAM has a simplified structure which is identical to the *write* primitive in Figure 2.10(b).

Memory map generation

Each FIFO channel in our MPSoCs has separate read and write ports. A processor accesses a FIFO for read operations using the *read* synchronization primitive. The parameter $port$ specifies the address of the read port of the FIFO channel to be accessed. In the same way, the processor writes to a FIFO using the write synchronization primitive where the parameter $port$ specifies the address of the write port of this FIFO. The FIFO channels are implemented in the communication memories (see Section 2.1.5), therefore, the addresses of the FIFO ports are located in the processors' address space where the communication memory segment is

[1] Recall that a connection is granted if the communication line is available and there is data in the requested FIFO.



(a) Kahn Process Network (b) Example Platform

Figure 2.11: Mapping Example.

defined. The memory map of a MPSoC generated by ESPAM contains the values defining the read and the write addresses of each FIFO channel in the system.

The first step in the memory map generation is the mapping of the FIFO channels in the KPN application specification onto the communication memories (CMs) in the multiprocessor platform. This mapping can not be arbitrary. ESPAM maps FIFO channels onto CMs of processors in the following automated way. First, for each process in the application specification ESPAM finds all the channels this process writes to. Then, from the mapping specification ESPAM finds which processor corresponds to the current process and maps the found channels in the processor's local CM. For example, consider the mapping specification shown in Figure 2.9(b) which defines only the mapping of the processes of the KPN in Figure 2.11(a) to the processors in the platform shown in Figure 2.11(b). Based on this mapping specification, ESPAM maps automatically $FIFO2$, $FIFO3$, and $FIFO5$ onto the CM of processor $uP1$ because process $P4$ is mapped onto processor $uP1$, and process $P4$ writes to channels $FIFO2$, $FIFO3$, and $FIFO5$. Similarly, $FIFO4$ is mapped onto the CM of processor $uP3$, and $FIFO1$ is mapped onto the CM of $uP4$. Since both processes $P2$ and $P5$ are mapped onto processor $uP2$, ESPAM maps $FIFO6$ and $FIFO7$ onto the CM of this processor.

After the mapping of the channels onto the CMs, ESPAM generates the memory map of the MPSoC, i.e., generates values for the FIFOs' read and write addresses. For the mapping example illustrated in Figure 2.11(b), the generated memory map is shown in Figure 2.12. Notice that $FIFO1$, $FIFO2$, $FIFO4$, and $FIFO6$ have equal write addresses (see lines 4, 6, 10, and 14). This is not a problem because writing to these FIFOs is done by different processors and these FIFOs are located in the local CMs of these different processors, i.e., these addresses are local processor write addresses. The same applies for the write addresses of $FIFO3$ and $FIFO7$. However, as explained in Section 2.1.5, all processors can read from all FIFOs via a communication component. Therefore, the read addresses have to be unique in the MPSoC memory map and the read addresses have to specify precisely the CM in which a FIFO is located. To accomplish this, a read address of a FIFO has 2 fields: a communication memory (CM) number and a FIFO number within a CM.

Consider for example $FIFO3$ in Figure 2.11(b). It is the second FIFO in the CM of processor $uP1$, thus this FIFO is numbered with 0002 in this CM. Also, the CM of $uP1$ can be accessed

```
1  #ifndef  _MEMORYMAP_H_
   #define  _MEMORYMAP_H_

   #define  p1   0xe0000008 //write addr. FIFO1
5  #define  p4   0x00040001 //read addr. FIFO1
   #define  p7   0xe0000008 //write addr. FIFO2
   #define  p2   0x00010001 //read addr. FIFO2
   #define  p8   0xe0000010 //write addr. FIFO3
   #define  p6   0x00010002 //read addr. FIFO3
10 #define  p9   0xe0000008 //write addr. FIFO4
   #define  p12  0x00030001 //read addr. FIFO4
   #define  p10  0xe0000018 //write addr. FIFO5
   #define  p13  0x00010003 //read addr. FIFO5
   #define  p14  0xe0000008 //write addr. FIFO6
15 #define  p11  0x00020001 //read addr. FIFO6
   #define  p3   0xe0000010 //write addr. FIFO7
   #define  p5   0x00020002 //read addr. FIFO7

19 #endif
```

Figure 2.12: The memory map of the MPSoC platform instance generated by ESPAM.

for reading through port 1 of the communication component *INTERCONNECT* as shown in Figure 2.11(b), thus this CM is uniquely numbered with 0001. As a consequence, the unique read address of $FIFO3$ is determined to be 0x00010002 – see line 9 in Figure 2.12, where the first field 0001 is the CM number and the second field 0002 is the FIFO number in this CM. In the same way, ESPAM determines automatically the unique read addresses of the rest of the FIFOs that are listed in Figure 2.12.

2.4 Dedicated IP core integration with ESPAM

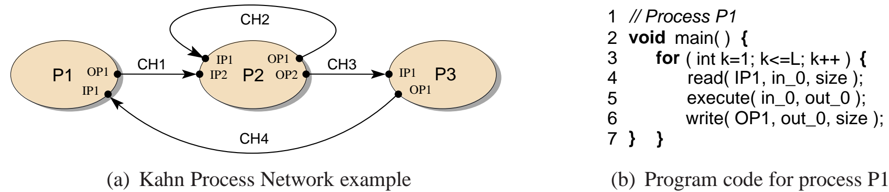
With the foregoing discussions in this chapter, we presented our methodology for multiprocessor system design implemented in ESPAM. It considers automated generation of homogeneous multiprocessor platforms, i.e., the processing components are only programmable (ISA) processors. However, in many cases, a homogeneous system may not meet the performance requirements of an application. For better performance and efficiency, in a multiprocessor system different tasks may have to be executed by different types of processing components which are optimized for execution of particular tasks. It is common knowledge that higher performance may be achieved by relying on dedicated (customized and optimized) IP cores. Moreover, many companies already provide dedicated customizable IP cores optimized for a particular functionality that aim at saving design time and increasing overall system performance and efficiency. Therefore, our platform model supports also dedicated IP cores as processing components. The idea of using dedicated IP cores in heterogeneous systems is very appealing because these systems deliver high flexibility and high performance at the same time. However, two major problems emerge, namely how to design and how to program heterogeneous MPSoCs. The lack of standard interfaces that an IP core has to provide in order to allow seamless integration, and the lack of automated programming approaches for heterogeneous multiprocessor systems only exacerbates these problems.

With ESPAM, we solve the problems mentioned above, and we provide an automated design and programming of heterogeneous multiprocessor systems where both programmable processors and dedicated IP cores are used as processing components. In our approach, we developed techniques for automated generation of an IP Module which consists of a wrapper around a dedicated and predefined IP core. This approach originates from the general idea implemented in Laura [74], i.e., generating of IP Modules based on the properties of the KPN model we use. Although using the same concept in ESPAM, we developed different techniques in order to enable systematic and automated IP core integration and connection to the other components of the system, i.e., programmable processors and different communication components. The structured, highly modularized, and parameterized IP Module we propose has been devised by carefully exploiting and efficiently implementing the simple communication and synchronization mechanisms of the KPN model. We have identified and developed an IP Module library which is a set of generic parameterized components used by ESPAM to compose an IP Module. This is done in the same way as ESPAM constructs an MPSoC instance, i.e., by instantiating components from the IP Module library, connecting them, and setting their parameters in correspondence with the KPN application specification. In addition, we defined clear interfaces of the components in an IP Module. This helped us to devise an efficient mechanism for connecting and synchronizing the components within an IP Module keeping high performance of the integrated IP cores. By making the IP Module structured and modularized, its components become more independent and loosely coupled. Therefore, we are able to design and optimize each component of the IP Module separately.

We have already presented our approach to design and programming of homogeneous MPSoCs. Based on that, in this section we present our approach to augment these MPSoCs with dedicated IP cores in a systematic and automated way. The basic idea in our approach is presented in Section 2.4.2. It is followed by a discussion on the type of the IPs supported by ESPAM, and the interfaces these IPs have to provide in order to allow automated integration. For details about the internal structure and the implementation of the IP Module we refer to [75].

2.4.1 Uniform structure of a KPN process

Our methodology and tool-flow for multiprocessor system design allow automated synthesis, programming, and implementation of multiprocessor platforms. As we have shown in Section 2.3, to automatically program an MPSoC the ESPAM tool generates program code for each processor in the system, generates the memory map of the system, and generates code that implements the synchronization and communication between the processors. In our methodology and design flow, the first step is partitioning of an application into concurrent tasks in the form of a Kahn process network (KPN) where the inter-task communication and synchronization is *explicitly* specified in each task. Such partitioning, discussed in Section 2.3.1 and performed automatically by the PNGEN tool [7], allows each task (Kahn process) or group of tasks to be compiled separately by a standard C compiler in order to generate an executable code for each processor in the platform. Regardless of the functional behavior specified by processes in a KPN generated by PNGEN, always ESPAM takes each process specification and generates a specific code for each process where the structure of the code is the same for all processes. This uniform structure is the basis of the proposed IP Module, and below, it is explained by an example.



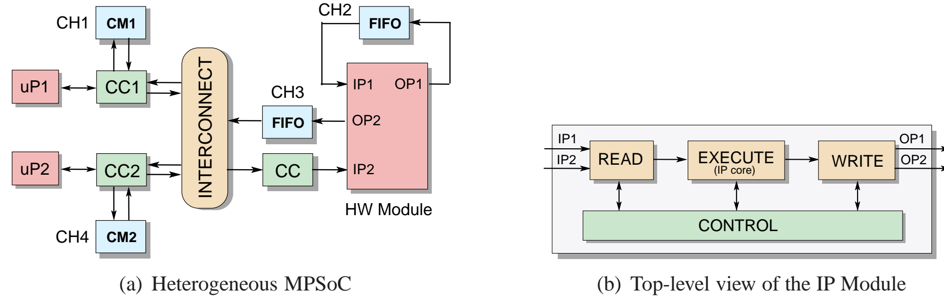
(a) Kahn Process Network example (b) Program code for process P1

Figure 2.13: Example of a KPN and program code.

Consider the KPN shown in Figure 2.13(a). Three processes ($P1$, $P2$, and $P3$) are connected through four FIFO channels ($CH1$, $CH2$, $CH3$, and $CH4$). Program code representing process $P1$ is shown in Figure 2.13(b). The structure of the code generated by ESPAM for each process is the same and consists of a ***CONTROL*** part, a ***READ*** part, an ***EXECUTE*** part, and a ***WRITE*** part. The same structure can be seen also for process $P2$ in Figure 2.15(a). The difference between $P1$ and $P2$, however, is in the specific code in each part. For example, the ***CONTROL*** part of $P1$ has only one for-loop whereas the ***CONTROL*** part of $P2$ has two for-loops. The blocking synchronization mechanism of our KPNs is implemented by read/write synchronization primitives. They are the same for each process and were discussed in detail in Section 2.3. The primitives are automatically generated and inserted in the program code by ESPAM in the places where a process has to read/write data from/to a FIFO channel. The ***READ*** part of $P1$ has one read primitive executed unconditionally whereas the ***READ*** part of $P2$ has two read primitives and if-conditions specifying when to execute these primitives.

2.4.2 IP Module – basic idea and structure

As we explained earlier, in the multiprocessor platforms we consider, the processors execute code implementing KPN processes, and communicate data between each other through FIFO channels mapped onto communication memories. Using communication controllers, the processors can be connected either point-to-point or via a communication component. We follow a similar approach to connect an IP Module to other IP Modules or programmable processors in our MPSoCs. We illustrate our approach with the example depicted in Figure 2.14. We map the KPN in Figure 2.13(a) onto the heterogeneous platform shown in Figure 2.14(a). Assume that process $P1$ is executed by processor $uP1$, $P3$ is executed by $uP2$, and the functionality of process $P2$ is implemented as a dedicated (predefined) IP core embedded in an IP Module. Based on this mapping and the KPN topology, ESPAM automatically maps FIFO channels to communication memories (CMs) following the rule that a processing component only writes to its local CM. For example, process $P1$ is mapped onto processing component $uP1$ and $P1$ writes to FIFO channel $CH1$. Therefore, $CH1$ is mapped onto the local CM of $uP1$ – see Figure 2.14(a). In order to connect a dedicated IP core to other processing components, ESPAM generates an IP Module (IPM) that contains the IP core and a wrapper around it. Such an IPM is then connected to the system using communication controllers (CCs) and communication memories (CMs), i.e., an IPM writes directly to its own local FIFOs and uses CCs (one CC for every input of an IP core) to read data from FIFOs located in CMs of other processors. The IPM that realizes process $P2$ is shown in Figure 2.14(b).



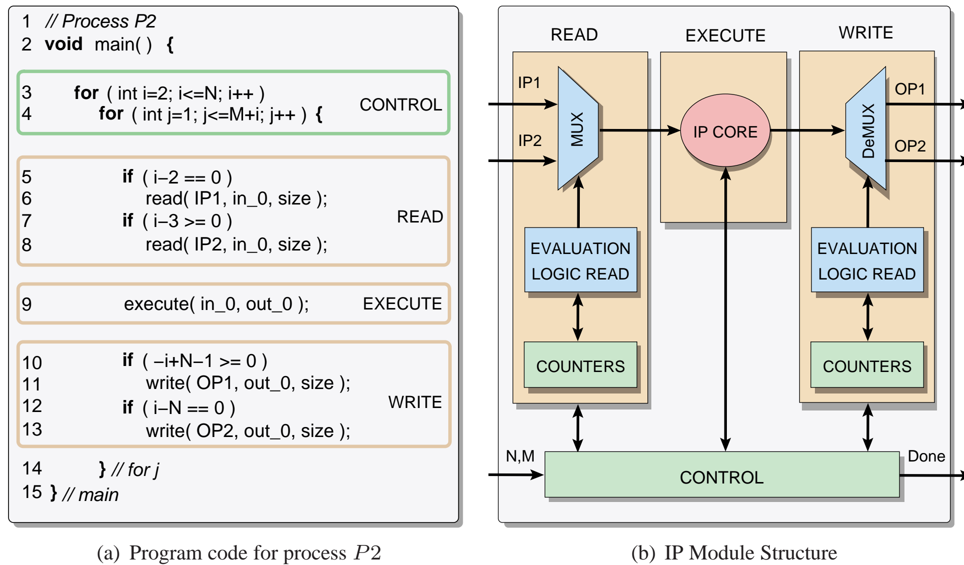
(a) Heterogeneous MPSoC (b) Top-level view of the IP Module

Figure 2.14: Example of heterogeneous MPSoC generated by ESPAM.

As explained in Section 2.3.1, our KPNs are derived automatically and the processes in our KPNs have always the same structure. It reflects the KPN operational semantics, i.e, read-execute-write using blocking read/write synchronization mechanism. Therefore, an IP Module realizing a process of a KPN has the same structure, shown in Figure 2.14(b), consisting of *READ*, *EXECUTE*, and *WRITE* components. A *CONTROL* component is added to capture the process behavior, e.g., the number of process firings, and to synchronize the operation of components *READ*, *EXECUTE*, and *WRITE*. The *EXECUTE* component of an IP Module (IPM) is actually a dedicated IP core to be integrated. It is not generated by ESPAM but it is taken from a library. The other components *READ*, *WRITE*, and *CONTROL* constitute the wrapper around the IP core. The wrapper is generated fully automatically by ESPAM based on the specification of a process to be implemented by the given IPM. Each of the components in a IPM have a particular structure which we illustrate with the example in Figure 2.15. Figure 2.15(a) shows the specification of process $P2$ in Figure 2.13(a) generated by ESPAM if $P2$ would be executed on a programmable processor. We use this code to show the relation with the structure of each component in the IP Modules generated by ESPAM, shown in Figure 2.15(b), when $P2$ is realized by an IP Module.

In Figure 2.15(a), the read part of the code is responsible for getting data from proper FIFO channels at each firing of process $P2$. This is done by the code lines 5–8 which behave like a multiplexer, i.e., the internal variable in_0 is initialized with data taken either from port $IP1$ or $IP2$. Therefore, the read part of $P2$ corresponds to the multiplexer MUX in the *READ* component of the IP Module in Figure 2.15(b). Selecting the proper channel at each firing is determined by the *if*-conditions at lines 5 and 7. These conditions are realized by the EVALUATION LOGIC READ sub-component in component *READ*. The output of this sub-component controls the MUX sub-component. To evaluate the *if*-conditions at each firing, the iterators of the *for*-loops at lines 3 and 4 are used. Therefore, these *for*-loops are implemented by counters in the IP Module – see the COUNTERS sub-component in Figure 2.15(b).

The write part in Figure 2.15(a) is similar to the read part. The only difference is that the write part is responsible for writing the result to proper channels at each firing of $P2$. This is done in code lines 10–13. This behavior is implemented by the de-multiplexer DeMUX sub-component in the *WRITE* component in Figure 2.15(b). DeMUX is controlled by the EVALUATION LOGIC WRITE sub-component which implements the *if*-conditions at lines 10 and 12. Again, to implement the *for*-loops, ESPAM uses a COUNTERS sub-component.



(a) Program code for process $P2$ (b) IP Module Structure

Figure 2.15: Example of a IP Module and its components' structure.

Although, the counters correspond to the control part of process $P2$, ESPAM implements them in both the *READ* and *WRITE* blocks, i.e., it duplicates the *for*-loops implementation in the IP Module. This allows the operation of components *READ*, *EXECUTE*, and *WRITE* to overlap, i.e., they can operate in pipeline which leads to better performance of the IP Module.

The execute part in Figure 2.15(a) represents the main computation in $P2$ encapsulated in the function call at code line 9. The behavior inside the function call is realized by the dedicated IP core depicted in Figure 2.15(b). As explained above, this IP core is not generated by ESPAM but it is taken from a library of predefined IP cores provided by a designer. An IP core can be created by hand or it can be generated automatically from C descriptions using high-level synthesis tools like, e.g., the PICO tool from Synfora [11]. In the IP Module, the output of sub-component MUX is connected to the input of the IP core and the output of the IP core is connected to the input of sub-component DeMUX. In the example, the IP core has one input and one output. In general, the number of inputs/outputs can be arbitrary. Therefore, every IP core input is connected to one MUX and every IP core output is connected to one DeMUX.

Notice that the loop bounds at lines 3–4 in Figure 2.15(a) are parameterized. The *CONTROL* component in Figure 2.15(b) allows the parameter values to be set/modified from outside the IP Module at run time or to be fixed at design time. Another function of component *CONTROL* is to synchronize the operation of the IP Module components and to make them to work in pipeline. Also, *CONTROL* implements the blocking read/write synchronization mechanism. Finally, it generates the status of the IP Module, i.e., signal $Done$ indicates that the IP Module has finished an execution.

2.4.3 IP core types and interfaces

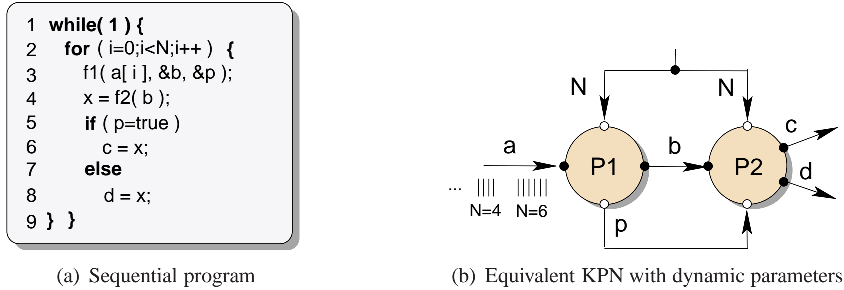
In this section we describe the type of the IP cores that fit in our IP Module idea and structure discussed above. Also, we define the minimum data and control interfaces these cores have to provide in order to allow automated integration in MPSoC platforms designed with ESPAM.

1. In the IP Module, an IP core implements the main computation of a KPN process which in the initial specification is represented by a function call. Therefore, an IP core has to behave like a function call as well. This means that for each input data, read by the IP Module, the IP core is *executed* and produces output data after an arbitrary delay.
2. In order to guarantee seamless integration within the data-flow of our heterogeneous systems, an IP core must have unidirectional data interfaces at the input and the output that do not require random access to read and write data from/to memory. Good examples of such IP cores are the *multimedia cores* at http://www.cast-inc.com/cores/. In addition, the PICO tool [11] can generate IPs that fall into the class of the considered IP cores from specifications written in C, i.e., PICO allows for synthesis of IP cores providing the required unidirectional data interfaces.
3. To synchronize an IP core with the other components in the IP Module, the IP core has to provide `Enable/Valid` control interface signals. The `Enable` signal is a control input to the IP core and is driven by the *CONTROL* component in the IP Module to enable the operation of the IP core when input data is read from input FIFO channels. If input data is not available, or there is no room to store the output of the IP core to output FIFO channels, then `Enable` is used to suspend the operation of the IP core. The `Valid` signal is a control output signal from the IP and is monitored by component *CONTROL* in order to ensure that only valid data is written to output FIFO channels connected to the IP Module.

2.5 Discussion

Recall that the initial applications in the DAEDALUS system design methodology are restricted to parameterized static affine nested loops programs (SANLP). This restriction is imposed by the PNGEN tool in order to allow automated KPN derivation and computation of buffer sizes that guarantee deadlock-free KPN behavior. However, ESPAM and the techniques for MPSoC synthesis presented in this chapter are not restricted to KPNs equivalent to this class of sequential programs. The presented techniques can be applied on more general KPNs as well. For example, we have developed techniques for converting weakly dynamic nested loop programs (WDNLP) to equivalent KPN specifications [23]. Similar to an SANLP, in a WDNLP loop boundaries and variable indexing functions are affine functions of loop iterators and static parameters, while the expressions in condition statements are arbitrary functions of loop iterators, parameters, and *data variables*. The inclusion of data variables makes the programs dynamic. The generation of such process networks is not automated yet.

Recall that the SANLPs may contain parameters, but their values may not change during the execution of the program, therefore, they are static parameters. The same is also true



(a) Sequential program

(b) Equivalent KPN with dynamic parameters

Figure 2.16: Motivating example.

for WDNLPs. Similarly, most deterministic data-flow models, whether static or dynamic and whether actor-based or process-based [76] do not support dynamic configuration of parameters. However, many realistic streaming-data applications that are naturally specified in terms of data-flow models require parameterization. For example, a function $p = f(token)$ in an active entity (thread or process) may return a value for the parameter p which is the upper bound of a loop $for(i = 1, i \leq p, i++)$ in another active entity. The restriction for the parameters to be static does not allow modeling of such behavior with our KPN model which significantly limits its expressiveness, and consequently, the applicability of the DAEDALUS approach to real-life applications. This motivated us to extend our approach to programs/KPNs that can deal with dynamic parameters. This is the topic of the next subsection. Automated derivation of such KPNs is out of the scope of this discussion.

2.5.1 Motivating example

Consider the program shown in Figure 2.16(a). This program processes an input data stream a and produces output data streams c and d. The *while(1)* construct at line 1 indicates that the streams may be infinite. The data is processed in blocks by functions $f_1()$ at line 3 and $f_2()$ at line 4. The size of the blocks is determined by the for-loop at line 2. In this program, there are five data variables (a, b, c, d, and x) and two control variables, i.e., N, used as an upper bound for the for-loop at line 2 and p, used in the evaluation of the if-condition at line 5. Values for p are produced by function $f_1()$. A functionally equivalent process network is depicted in Figure 2.16(b). It consists of processes $P1$ and $P2$ that execute functions $f_1()$ and $f_2()$, respectively. There is a direct relation between the control variables used in the sequential program and the dynamic parameters in the KPN with dynamic parameters. In such KPNs, if data is involved in control statements, we consider it as a dynamic parameter. We distinguish two types of parameters: global and local. A global parameter is an external parameter, i.e., not produced by any process in the network. A local parameter is an internal parameter, i.e., it is produced and consumed by processes in the network. According to these definitions, in the example in Figure 2.16(b), N is a *global* parameter, and p is a *local* parameter, both being dynamic (control) parameters. To transfer the parameter values between the processes in the network, we use control (FIFO) channels. In this example, there are three control channels: two channels for parameter N and one channel for parameter p.

```
1  // Process P1
2
3  while(1) {
4    N = read_N(); // global parameter
5    for(i=1;i<=N;i++) {
6      a(i) = read_a();
7      f1(a[i], &b, &p);
8      write_p(); // local parameter
9      write_b();
10   }
11 }
```

```
12 // Process P2
13 while(1) {
14   N = read_N(); // global parameter
15   for(i=1;i<=N;i++) {
16     b = read_b();
17     x = f2(b);
18     p = read_p(); // local parameter
19     if( p=true )
20       write_x_to_c();
21     else
22       write_x_to_d();
23 } }
```

Figure 2.17: Proposed structure of the processes in Figure 2.19(b).

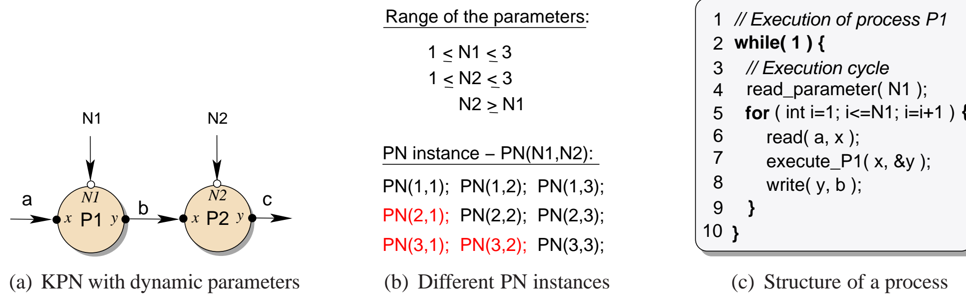
The functionality and the structure of processes $P1$ and $P2$ is shown in Figure 2.17. $P1$ reads and transforms a block of data (lines 6 and 7), which size is determined by the value of the global parameter N. Values for N are generated outside the network at run time. $P1$ and $P2$ read N (lines 4 and 14) which is to become the upper bound in a loop in both processes. Function $f_1()$ in process $P1$ outputs the local parameter p (line 8) which is of type *Boolean*, and data b (line 9). Process $P2$ reads data b (line 16), the parameter p (line 18), and sends the output of its function $f_2()$ to one of the process outputs (c, d) depending on the value of the local parameter p (lines 19-22).

As is the case with all data-flow models, the main question here is whether the PNs with dynamic parameters are *consistent*, and can execute in *bounded memory*. Consistency has to do with a balancing of the production and consumption of tokens in the network. When this balancing is dependent on dynamic parameters, consistency conditions may be violated. Execution in bounded memory is a necessary condition for the processing of infinite streams (non-terminating execution). Our KPNs with dynamic parameters execute in bounded memory and below, we address the consistency problem only.

2.5.2 Process network instance

Consider the KPN representing a producer-consumer pair, shown in Figure 2.18(a). N_1 and N_2 are FIFO channels of the global parameters N_1 for process $P1$ and N_2 for process $P2$, respectively. Each parameter can take values within a fixed range. PN(N_1, N_2) denotes an instance of the KPN[2]. There is generally a relation between the parameters, in this example N_1 and N_2. Therefore, some instances PN(N_1, N_2) are invalid instances. For the PN network in Figure 2.18(a), all different instances are shown in Figure 2.18(b). Instances $PN(2, 1)$, $PN(3, 1)$, and $PN(3, 2)$ are invalid because they violate the condition $N_2 \geq N_1$. Similarly, an instance $PN(2, 4)$ is invalid because N_2 is out of its range. Figure 2.18(c) shows the structure of a process we propose to deal with dynamic parameters. Network instances are selected by reading parameter values at run time. For this purpose, we add a *read parameters* phase, see line 4, prior to the actual processing at lines 5-9. Because reading parameters and data processing are repeated (possibly infinite number of times), we call it a process *execution cycle* (lines 3-9). When all actors in a PN have performed an execution cycle, a network instance has performed an execution.

[2] From now on, we consider the terms PN and KPN to be equivalent.



(a) KPN with dynamic parameters (b) Different PN instances (c) Structure of a process

Figure 2.18: A producer-consumer example of a PN with dynamic parameters.

Definition 2.5.1 (Consistency of a PN instance)
A PN instance is consistent if after an execution, the number of tokens written to any channel is equal to the number of tokens read from it.

2.5.3 Preserving the consistency of our PNs with dynamic parameters

The validity of the PN instances is a necessary but not a sufficient condition to preserve the PN consistency when changing parameter values at run time. A valid set of parameters corresponds to a valid (and consistent) PN instance. However, the transition from a valid instance to another valid instance at an arbitrary point may violate the consistency of the instances. Therefore, we defined the following three conditions which are sufficient to preserve consistency when changing parameter values dynamically at run time.

C1: *Parameter sets have to correspond to valid network instances.*

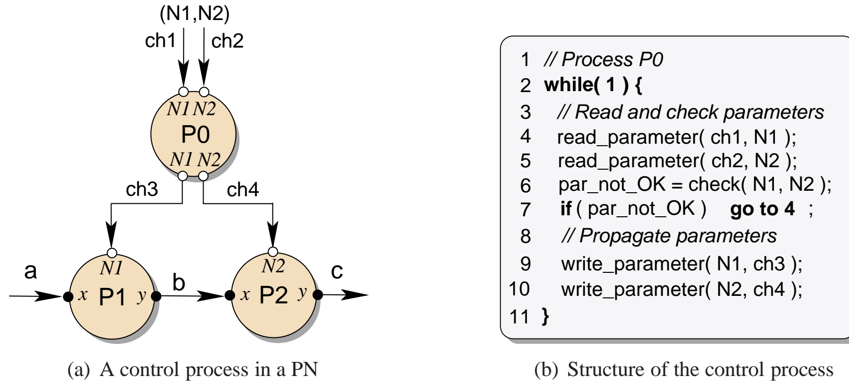
C2: *A valid parameter set has to initiate a network instance execution.*

C3: *Processes may read new parameters from a valid set (corresponding to the selection of a new valid network instance) after they have completed a process execution cycle.*

In other words, parameter values may be changed either before or after an execution cycle of the processes. This is taken into account by the proposed execution cycle of a process illustrated in Figure 2.18(c).

2.5.4 Respecting the conditions

Because a PN may have many dynamic parameters distributed over different processes, respecting condition $C1$ may not be feasible. For example, in $PN(2,1)$ discussed in Section 2.5.2, the values of the parameters are within the specified range ($N_1, N_2 \in [1..3]$). However, because of the condition $N_2 \geq N_1$, instance $PN(2,1)$ is not valid. Since N_1 is a parameter only for $P1$ and N_2 is a parameter only for $P2$, it is not possible to check in each process whether $(N_1, N_2) = (2,1)$ is a valid parameter set. Therefore, to respect the



(a) A control process in a PN

(b) Structure of the control process

Figure 2.19: Introducing a control process in our PNs with dynamic parameters.

three conditions above, we introduce a *control process* $P0$ as depicted in Figure 2.19(a). This process reads global parameter values (N_1, N_2) and propagates them to the other processes only when the set is valid. Process $P0$ reads and writes from and to Kahn FIFO channels, called *control channels* ($ch1$, $ch2$, $ch3$, and $ch4$). The behavior of the control process in Figure 2.19(a) is given in Figure 2.19(b). It consists of two parts, namely *read and check parameters* (lines 3-7), and *propagate parameters* (lines 8-10). The process body respects conditions $C1$ and $C2$ in the following way. First, $P0$ reads values for parameters N_1 and N_2 from the control channels $ch1$ and $ch2$, respectively, using a blocking read synchronization mechanism. Then, at line 6, it is checked whether the parameter values define a valid PN instance. If not, then an error is indicated (not shown in the example) and parameters are read again (see line 7). This behavior respects condition $C1$. Notice that a parameter set may be not valid because of just one parameter value. Nevertheless, all the parameters have to be read again: The current PN instance is invalidated (discarded) and a new set of parameter values is read again. After reading and successfully passing the validity check, the values of parameters N_1 and N_2 are written (lines 9 and 10) to the control channels ch_3 and ch_4 which will take them to the destination processes $P1$ and $P2$. Thus, the combined writing of parameter values by the control process $P0$, and the reading of these parameters by the destination processes respects condition $C2$, because only a valid parameter set will cause a process to initiate an execution cycle and, consequently, an execution of a PN instance.

The FIFO organization of the control channels and the blocking synchronization mechanism (the KPN semantics) keep the right order of selecting new network instances, i.e., the order in which the parameter sets are generated outside the network and written to the control channels. Since new parameter values are read by the processes after performing an execution cycle, parameter values selecting alternative PN instances may be written to the control channels while a PN instance is being executed. In addition, the proposed mechanism allows the processes to read the parameter values independently of each other without violating the conditions we defined for preserving the consistency. However, these conditions are valid only for consistent PN instances. Therefore, a consistency check is required, either at design time or at run time. In our approach, a consistency check is performed at design time and only checking for selection of valid instances is performed by the control actors at run time.

For more details about the consistency check and our approach to deal with dynamic parameters at run time, we refer to [77] where the presented approach is generalized for the SBF MoC [78]. Although the presented PNs have to be created currently by hand, the proposed structure and execution cycle of a process can be used as guidance in describing process networks with dynamic parameters and employing such PNs in the DAEDALUS design flow.

2.6 Conclusions

In this chapter, we presented our system design methods and techniques implemented in the ESPAM tool for automated multiprocessor system design, implementation, and programming. Using a platform model and specifications at system level of abstraction, ESPAM can automatically synthesize and program heterogeneous MPSoCs in which both programmable processors and dedicated IP cores are used as processing components. This automation significantly reduces the design time and with DAEDALUS, i.e., starting from a sequential application and going down to complete implementation, e.g., to an MPSoC prototyped on an FPGA, is only a matter of hours. In addition, the high level of the input specifications allows a system designer easily to construct many alternative platforms instances which are automatically implemented by ESPAM. As we will show in Chapter 4, this enables fast exploration of design points at implementation level with 100% accuracy during the early stages of design. At the same time, using high-level input specifications is less error-prone compared to lower levels of abstraction, e.g., RTL, at which MPSoC designs are captured, analyzed, and synthesized from.

Chapter 3

Techniques for Narrowing the Design Space

In Chapter 2, we presented methods and techniques for systematic and automated multi-processor system design, programming, and implementation for closing the implementation gap introduced in Section 1.1. The application and the platform models considered in the presented approach, were also discussed in this chapter. The proposed system-level design methodology for systematic and automated MPSoC design is implemented in the DAEDALUS design flow presented in Section 1.2. Designing an MPSoC with DAEDALUS includes essentially an MPSoC instance generation (see Chapter 2) and mapping (assignment) of application tasks to processing components of that instance. In the MPSoC design process, different numbers and types of processing components (from the platform model) can be used to construct an MPSoC instance as well as different mappings can be considered. This leads (usually) to a large number of potential alternative designs. That is, given an application specified as a KPN, there are many different MPSoC implementation possibilities. The set of all different possibilities comprises the so called design space. Evidently, some of the points in this design space will correspond to MPSoC instances that satisfy the initial requirements and some will correspond to MPSoCs which do not. The key issue here is to reduce the number of different implementation possibilities to a subset, consisting of the most promising design points from which, based on certain criteria, the designer can choose the best one. Then, the question is how to find this subset of design points. Naïvely, all the points of the design space need to be evaluated and some of them have to be selected. However, traversing the whole design space may not be always feasible. Therefore, an alternative way to evaluate the design space is required. This can be achieved by a design space exploration (DSE) approach which provides mechanisms for:

- Guided selection of a design point, i.e., a mechanism to ’selectively walk’ through the design space without visiting all design points;
- Evaluation of a design point in order to accept or discard the point based on some performance/cost constraints.

Ideally, one would employ an analytic procedure for computing an optimal design point for a given problem. However, in most practical situations this is not possible. An alternative approach to find a good solution is to construct a parameterized system model, where the set of parameters represents a design point, and to simulate it. However, simulating many alternative design points is costly, both in terms of the effort it takes to create these design points in the first place, and also in terms of the time it takes to simulate a large number of design points. In particular, when the design space is very large, constructing all possible design points quickly becomes infeasible, therefore, better search techniques are needed to reduce the number of alternatives to be explored. Apart from exhaustive simulation (being a non selective exploration), there are many ways of realizing design space exploration which employ different kind of search strategies, e.g., from simple hill climbing to more complex methods such as genetic algorithms, simulated annealing, etc.

In DAEDALUS, DSE can be performed by the SESAME tool [8], illustrated at the top of Figure 1.2. The DSE is performed at system level by selecting design points and simulating high-level models of these points. Recall that with DAEDALUS, the design time for implementing an MPSoC instance is significantly reduced, which enables a DSE at implementation level as well. Therefore, in DAEDALUS a design space exploration can be performed at two different levels of abstraction, i.e.,

- At system level through high-level simulations by using the SESAME tool;
- At implementation level by prototyping design points and measuring actual numbers.

On the one hand, evaluation of design points at system level is (relatively) fast, however, the accuracy of the results is compromised. On the other hand, the results from the implementation-level DSE are accurate, yet, the exploration process may become slow when many design points have to be implemented and evaluated. Therefore, in DAEDALUS we apply the following strategy when designing an MPSoC:

1. Perform a DSE at system level to narrow down the design space to a few points, given particular performance/cost constraints;
2. Perform 100% accurate exploration in the narrowed design space by real MPSoCs implementations and measurements of actual numbers;
3. Select the design point which leads to the best MPSoC implementation.

The design space in the DAEDALUS design flow is defined by a 3-tuple (T, C, M), where T is the set of the application tasks, i.e., the Kahn processes, C is the set of the platform components, and M is the set of possible mappings (the elements of T to elements of C). Although with the platform-based design approach in DAEDALUS, design space exploration is limited to these three sets only, the design space is potentially huge and very complex, i.e., T and C may be large, hence M may be extremely large.

An open issue in DAEDALUS is how to 'walk' efficiently through this large design space. At implementation level, guidance comes from SESAME, i.e., we implement all the points

selected by the system-level DSE performed by using the SESAME tool. At system level, currently SESAME applies searching methods and techniques employing genetic algorithms to select points from the design space for evaluation in order to select the most promising ones. However, these methods are not tailored to the specific KPN model we use, and consequently, for large T and C sets, the whole design exploration process may take unreasonable amount of time.

In this chapter, we propose techniques to prune the design space by reducing the size of set M. More precisely, by exploiting the fact that we target MPSoCs executing applications modeled as Kahn process networks, we devised techniques for mapping processes to processing components based on mapping rules[1]. The main goal when mapping an application to an MPSoC is to minimize cost and to maximize performance, i.e., to utilize as less MPSoC components as possible without compromising the performance of the system when executed. By applying the mapping rules we propose in this chapter, the design space is effectively pruned by reducing the number of the possible mappings, guaranteeing that only the design points delivering highest performance are considered for further exploration (by using the SESAME tool).

Mapping of KPNs to MPSoC instances

Mapping is a process of binding the application (KPN) and the platform models together[2], i.e., the mapping gives the relation between the processes and the channels in a KPN, and the components in an MPSoC instance. We consider two types of mapping, namely ONE-TO-ONE and MANY-TO-ONE.

Definition 3.0.1 (ONE-TO-ONE mapping)
In a ONE-TO-ONE mapping, each process is mapped onto only one processing component, and each processing component has only one process mapped onto it. A KPN channel is mapped onto a communication memory in the MPSoC and each communication memory has only one channel mapped onto it, so that all the connections are point-to-point connections.

Definition 3.0.2 (MANY-TO-ONE mapping)
In a MANY-TO-ONE mapping, two or more processes are mapped onto one processing component and/or two or more channels are mapped onto one communication memory.

Notice that in a MANY-TO-ONE mapping, assignment of several processes to a single processing component is possible for programmable (ISA) processors only. This is because, according to our assumptions in Section 1.3, with DAEDALUS we do not support sharing of a dedicated IP core between several KPN processes.

In a ONE-TO-ONE mapping, each process is executed on a separate processing component, implying that all the parallelism expressed by a KPN is directly translated to the multiprocessor platform instance. With this respect, ONE-TO-ONE mapping (and a given set of platform components) leads to maximum performance when the MPSoC is implemented and

[1] The proposed mapping rules do not consider mapping of FIFO channels to communication memories because this is done automatically by ESPAM (see Section 2.3.3).

[2] The application and the platform models we consider were presented in Chapter 2.

executed. In a MANY-TO-ONE mapping, platform computation and communication components are shared between multiple Kahn processes and FIFO channels, respectively. This allows for implementation of an MPSoC with less processing and communication components, i.e., with reduced implementation cost, however, with additional execution overhead. Consequently, the performance (in terms of execution time) of a MANY-TO-ONE mapping, given the same type of platform components, can not be higher than the performance of the ONE-TO-ONE mapping. With the techniques we propose for pruning the set M, we allow MANY-TO-ONE mappings of a KPN to MPSoC instances that do not compromise performance. More precisely, the mapping rules guarantee that if respected, the resulting MPSoC instances deliver performance equal to the performance of the MPSoC corresponding to a ONE-TO-ONE mapping for the same KPN. Hence, the MPSoC instances defined by ONE-TO-ONE mapping and MANY-TO-ONE mappings which comply with the mapping rules, form the set of design points representing the pruned design space.

The remaining part of the chapter is organized as follows. First, we explain what system performance means in the context of MPSoCs that execute KPNs. With respect to this, we introduce some terminology that we use throughout the chapter. Also, in order to motivate and clarify the devised mapping rules, we comment on the factors that affect system performance. Then, in Section 3.2, the mapping rules are presented. This is followed by a discussion in Section 3.3 about how the rules can be applied in practice considering the specific (polyhedral) KPN application model we use. We conclude the chapter in Section 3.4.

3.1 System performance

Recall that we consider data-flow dominated applications in the realm of multimedia, imaging, and signal processing. These applications naturally consists of computational tasks transforming partial streams to partial streams that are passed from tasks to tasks.

Definition 3.1.1 (Data token)
A data token is a packet of data that can represent any type of information.

Definition 3.1.2 (Data stream)
A data stream is a sequence of data tokens.

A stream is characterized by its data (token) rate.

Definition 3.1.3 (Data rate of a stream, ρ_{str})
The data rate (ρ) of a stream (str) is determined by the time-distance between two consecutive data tokens in the stream.

That is, $\rho_{str} = \dfrac{1}{T}$, where T is the time-distance. Usually, the time-distance is given as an average value over some period of time. A stream has a source that puts data tokens to the stream, and a destination (sink) that consumes data tokens from the stream. Consider the example in Figure 3.1. It depicts a system with one input stream (in) and one output stream (out). For clarity of the discussion, we introduce systems with multiple input and

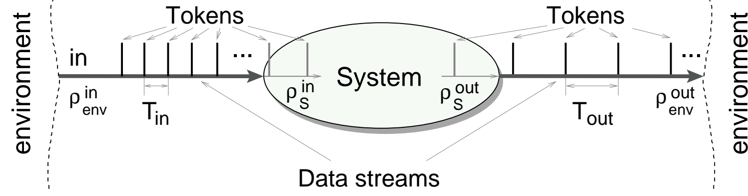


Figure 3.1: Example of a system.

output streams further in this chapter. The environment of the system produces data tokens to stream in at a rate ρ_{env}^{in} and the system consumes the tokens at rate ρ_{S}^{in}. Similarly, the system produces data tokens to stream out at a rate ρ_{S}^{out} and the environment consumes the tokens at rate ρ_{env}^{out}. Producing/consuming data to/from a stream can be performed at different rates, therefore, the actual rate of the stream is

$$\rho_{str} = min(\rho_{src}^{str},\ \rho_{snk}^{str}), \tag{3.1}$$

where, ρ_{src}^{str} is the rate at which the source (src) of the stream (str) produces data tokens to the stream, and ρ_{snk}^{str} is the rate at which the sink (snk) of the stream (str) consumes data tokens from the stream.

Recall that we model streaming applications by using the Kahn process network (KPN) dataflow model of computation [6] where the application tasks are processes and passing of partial streams is over FIFO buffered channels. Therefore, when we use the term *system*, we assume an MPSoC executing a KPN. Also, we consider systems that consume tokens from (possibly infinite) input streams and produce tokens to (possibly infinite) output streams. If input data is not available when a system attempts to consume it, then the process connected to the corresponding input stream is suspended, waiting for the data. In addition, we associate the term system performance with the throughput of the system.

Definition 3.1.4 (System throughput, τ_S)
The throughput (τ_S) is the sustained rate of data tokens at the output stream(s).

The data rate is given as an average value over some period of time. In case a system has multiple outputs, then the system throughput represents the sum of the data rates of all output streams. If input data is not available when a system attempts to consume it, then the rate at which the system generates results may be reduced (due to the delay introduced by waiting for the input data) which consequently may limit the performance of the system. Therefore, in order to capture the maximum (achievable) system performance, we introduce the term *isolated* throughput τ_S'.

Definition 3.1.5 (Isolated system throughput, τ_S')
The isolated throughput (τ_S') is the system throughput when isolated from its environment.

The isolated throughput indicates the (theoretical) maximum performance that can be achieved by the system since it depends only on the system itself and does not depend on its environment (like in case always input data is available). In addition, we express the rate of consuming data tokens from an input stream and the rate of producing data tokens to an output stream as a function of the isolated system throughput τ'_S:

$$\rho_S^{in} = k_{in}.\tau'_S,$$
$$\rho_S^{out} = k_{out}.\tau'_S,$$

where k_{in} and k_{out} are coefficients. For example, if the system in Figure 3.1 performs downsampling of a signal, then $k_{in} > 1$. Since we defined the isolated throughput of a system with multiple output streams as the sum of the data rates of all output streams, then the value of the coefficients of the output streams in such systems will be less than 1. Note that the system in Figure 3.1 has only one output stream, therefore, $k_{out} = 1$. The coefficients that determine the streams data rate are further discussed in Section 3.3.4.

Based on the foregoing discussion, we can summarize that system performance depends on the rates at which both the environment and the system produce and consume data tokens to/from data streams. According to Definition 3.1.4, we associate system performance τ_S with the data rate of the output stream ρ_{out} which can be expressed in the following way:

$$\begin{aligned} \tau_S = \rho_{out} &= min(\rho_{env}^{in},\ \rho_S^{in},\ \rho_S^{out},\ \rho_{env}^{out}) \\ &= min(\rho_{env}^{in},\ k_{in}.\tau'_S,\ k_{out}.\tau'_S,\ \rho_{env}^{out}). \end{aligned} \tag{3.2}$$

Recall that we are interested in techniques for narrowing down the design space in a way that preserves the design points corresponding to the MPSoC instances delivering maximum performance. We target MPSoCs executing applications modeled as Kahn process networks and, to narrow down the design space, we propose rules for mapping processes to processing components in a way that performance is not compromised. In order to achieve this, the mapping rules are devised by taking into account the factors that affect the performance of such systems. For motivation and better understanding of the mapping rules which we discuss in Section 3.2, we first present these factors and comment on their role in affecting system performance. The performance of a KPN executed on an MPSoC, is affected by the:

- Throughput of individual processes when executed on processing components;
- Throughput of processes when merged (grouped) for execution on a single processing component;
- Buffer sizes of the FIFO channels;
- Cycles in the KPN topology.

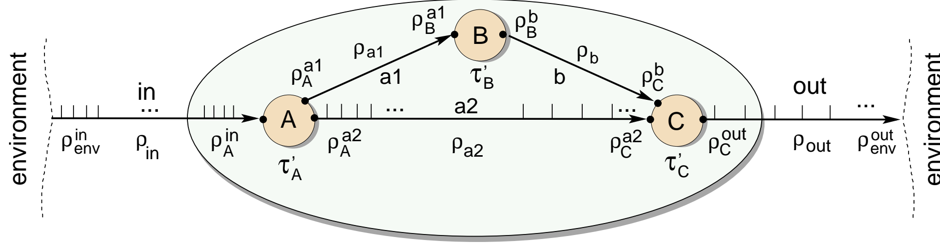


Figure 3.2: System as a process network.

3.1.1 Process throughput and system performance

Below, we show how the performance of a system, i.e., a process network executed on an MP-SoC, depends on the performance of the individual processes when executed on processing components. Consider the example in Figure 3.2 which represents the system in Figure 3.1 as a process network consisting of three processes (A, B, and C). We may consider every process in this system as a subsystem that consumes data from input stream(s) and produces data to output stream(s). Therefore, Equation 3.2 which was defined for a system holds for the processes as well and every process is characterized by its isolated throughput (performance).

Definition 3.1.6 (Isolated process throughput, τ'_P)
The isolated throughput (τ'_P) of a process P is its (maximum) throughput when isolated from the process network.

Note that the isolated throughput is not a metric of a process itself but it represents process performance when executed on a particular processing component.

Definition 3.1.7 (Data path)
A data path is a sequence of connected processes such that data from an input stream is transformed and propagated (through intermediate streams) to an output stream.

Hence, there are two data paths in Figure 3.2, i.e., ($A - B - C$) and ($A - C$).

In order to express the performance of this system (process network), we first associate the system performance τ_S with the data rate of its output stream out (see Definition 3.1.4), which actually is the output stream of process C. Therefore, by considering process C as a (sub)system, we apply Equation 3.2:

$$\tau_S = \tau_C = \rho_{out} = min(\ \rho_{a2},\ \rho^{a2}_C,\ \rho_b,\ \rho^b_C,\ \rho^{out}_C,\ \rho^{out}_{env}\ ). \tag{3.3}$$

In the same way, by following the data paths of the system, we can express the rate of all data streams. More precisely, we can apply the same approach by considering processes B and A as subsystems. By considering process B as a subsystem, we associate the performance τ_B with the data rate of stream b. Tokens are consumed by the the environment of subsystem B at rate ρ^b_C, see Figure 3.2. Consequently, we express ρ_b as:

$$\rho_b = min(\ \rho_{a1},\ \rho_B^{a1},\ \rho_B^{b},\ \rho_C^{b}\ ). \tag{3.4}$$

Similarly, by considering process A as a subsystem, we can express the rate of streams $a1$ and $a2$ as:

$$\rho_{a1} = min(\ \rho_{env}^{in},\ \rho_A^{in},\ \rho_A^{a1},\ \rho_B^{a1}\ ), \tag{3.5}$$

$$\rho_{a2} = min(\ \rho_{env}^{in},\ \rho_A^{in},\ \rho_A^{a2},\ \rho_C^{a2}\ ). \tag{3.6}$$

Finally, we substitute Equations 3.4, 3.5, and 3.6 in Equation 3.3 and obtain:

$$\tau_S \quad = \quad min(\ \rho_{env}^{in},\ \rho_A^{in},\ \rho_A^{a1},\ \rho_B^{a1},\ \rho_A^{a2},\ \rho_C^{a2},\ \rho_B^{b},\ \rho_C^{b},\ \rho_C^{out},\ \rho_{env}^{out}\ ). \tag{3.7}$$

That is, the throughput of the system is determined by the stream with the lowest data rate. This stream is referred to as the bottleneck stream. Consequently,

Definition 3.1.8 (Bottleneck process)
The bottleneck process is the process that causes the bottleneck stream in a system.

Therefore, system performance is limited by the isolated throughput of the bottleneck process. Note that if the bottleneck is caused by the environment (through ρ_{env}^{in} or ρ_{env}^{out}), then the system does not have a (real) bottleneck process.

3.1.2 Throughput in case of merged processes

Recall that the isolated throughput of a process P is determined by the time-distance T_P between consecutive tokens generated by the process. T_P is called also an *execution time* of process P to produce one data token. In case of a ONE-TO-ONE mapping in which every process is executed on a separate processing component, T_P is determined by the complexity of the process and the computational power of the processing component. However, in the MANY-TO-ONE mappings we target, several processes are merged together for concurrent execution on a single processing component. In this case, the throughput of the merged processes (compared to the throughput before merging) is affected as follows. Concurrent execution is achieved by interleaving the execution of the processes on the processing component over time. This produces the appearance of simultaneous (parallel) execution of the processes. In contrast with the real parallel execution however, the concurrent execution causes the time-distance between the generated tokens by the individual processes after merging to increase, and therefore, the isolated throughput of the processes to drop.

Assume that processes A and B in the example in Figure 3.2 are merged together. If some fair schedule (e.g. ROUND-ROBIN) is used for the execution of the merged processes, then the execution time can be represented by $T_{(AB)} = T_A + T_B$. Consequently,

$$\tau'_{(AB)} = \frac{1}{T_{(AB)}}, where \tag{3.8}$$

$$\tau'_{(AB)} < \tau'_A \text{ and } \tau'_{(AB)} < \tau'_B$$

Consequently, for the example in Figure 3.2 when process A is merged with process B, the system throughput is expressed by

$$\tau_S = min(\rho_{env}^{in}, k_{in}.\tau'_{(AB)}, k_1.\tau'_{(AB)}, k_2.\tau'_{(AB)}, k_3.\tau'_{(AB)}, k_4.\tau'_C, k_5.\tau'_{(AB)}, k_6.\tau'_C, k_{out}.\tau'_C,\ \rho_{env}^{out})$$

Merging of processes does not affect system performance as long as the new (compound) process does not become the bottleneck process of the system.

3.1.3 Buffer sizes and system performance

In a KPN, data is communicated through unbounded FIFO channels. For synchronization, the processes use a blocking read communication mechanism, i.e., if a process attempts to consume data that is not available, the process blocks (it is suspended) until data arrives. Blocking (on read) of the process execution means increasing the time-distance between consecutive tokens generated by the process. The increased time-distance reduces the process throughput. As a consequence, the process is no longer able to maintain its isolated throughput which is in line with the discussion presented in Section 3.1.1.

Although communication FIFO channels are unbounded in the formal definition of a KPN, they must be bounded in actual implementations. Therefore, to guarantee correct execution of a KPN, a blocking write synchronization mechanism is required as well. Blocking on read bounds the system performance to a performance determined by the isolated throughput of the bottleneck process, and blocking on write (due to finite buffer sizes) may additionally reduce system performance. However, in this section we show that there is a (lower) bound on the buffer sizes that guarantees the performance determined by Equation 3.7. Consider the example in Figure 3.3. It depicts a KPN consisting of four processes and four channels as shown in Figure 3.3(a). In this discussion, we consider that the environment is not the bottleneck, i.e., the data rate of the input stream (omitted in the figure) is higher than the rate of the data streams in the system. We also consider that the processes are executed on separate processing components, i.e., a ONE-TO-ONE mapping, and have equal isolated throughput. Figure 3.3(b) represents the performance curve as a function of the buffer sizes. In this figure, every point on the memory axis represents the sum of all FIFO buffer sizes of the KPN. The points on the performance axis (τ) illustrate the achieved throughput (in number of tokens per unit of time) given particular buffer sizes.

A KPN can execute in bounded memory if a deadlock-free execution is obtained with particular buffer sizes. Therefore, for such process networks, there is a point M_{min} representing the amount of memory distributed between the buffers of the FIFO channels in a way that deadlock-free execution is achieved. It has been shown in [79] that the performance of a KPN is a monotonic function of the FIFO buffer sizes. That is, for any point M, if

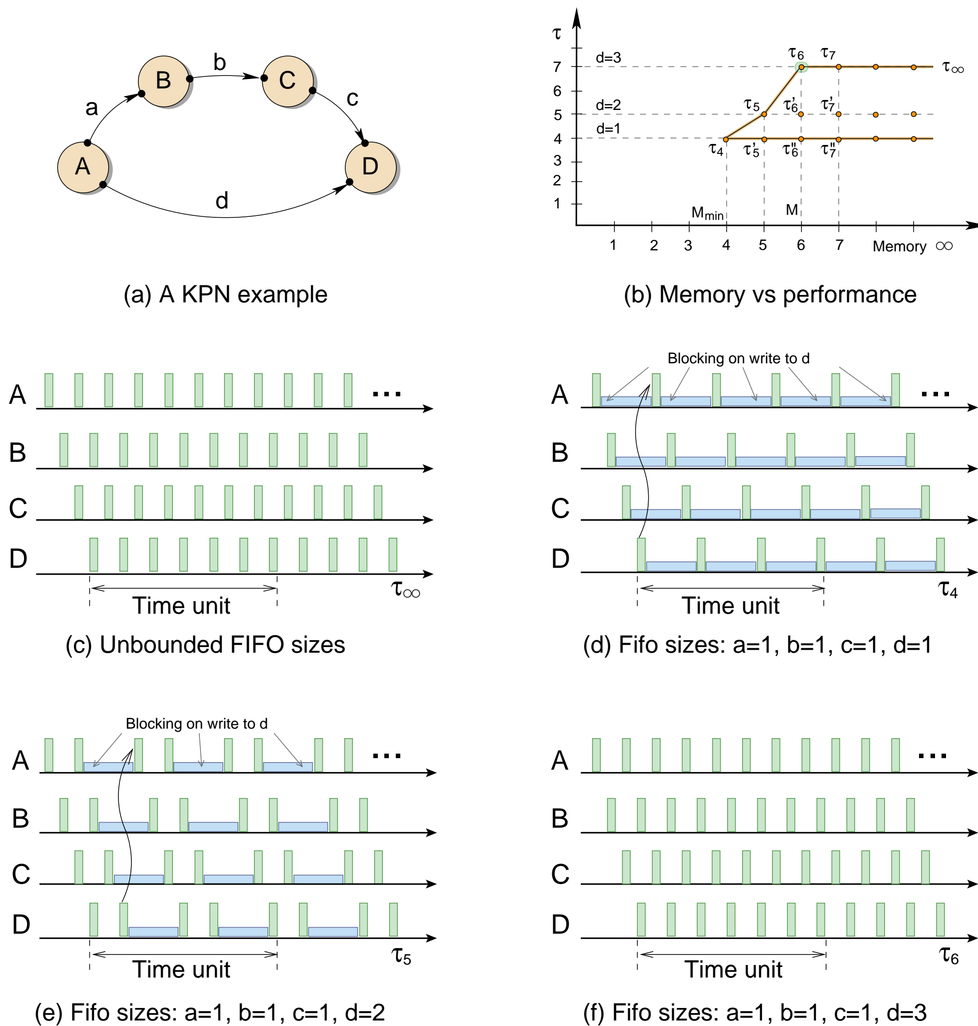


Figure 3.3: Memory vs performance.

$$\textit{Memory: } M_{min} \leq M < \infty,$$

then for the performance of the KPN, the following relation holds:

$$\textit{Throughput: } \tau_{M_{min}} \leq \tau_M \leq \tau_\infty.$$

Below we show that a point $M < \infty$ exists such that $\tau_M = \tau_\infty$. In addition, we illustrate how different buffer sizes affect system performance. Figure 3.3(c) shows the throughput of the processes in case of unbounded FIFO buffers. The ticks on the graph represent the production of data tokens by the corresponding processes (process names are given on the left of the figure) when executed on processing components. The output generated by process

D represents the time-distance between the tokens leaving the system, and consequently, the throughput τ_∞ of the system. For the chosen unit of time in this example, the KPN generates 7 tokens. Note that since the FIFO channels are unbounded, blocking on write does not occur and the performance is determined by the bottleneck process according to Equation 3.7.

Figure 3.3(d) illustrates the system performance corresponding to buffer sizes equal to 4 (point τ_4), i.e., $a = 1$, $b = 1$, $c = 1$, and $d = 1$. These are the minimum buffer sizes in this example that guarantee deadlock-free execution of the process network, $M_{min} = 4$. Assume that processes A always produce data to channels a and d, and process D always consumes data from channels c and d [3]. In this way, setting the buffer sizes of all FIFOs to 1, leads to temporally blocking of the processes during execution as we explain below. When process A produces a token, it is written to FIFO buffers a and d. This enables process B which reads data from a and writes data to b. Consequently, process C reads data from b and writes to c. Until then, process D is blocked on reading, and consequently, FIFO buffer d is full. Therefore, next time process A attempts to write to d, it will block on writing until D reads the data from it, see Figure 3.3(d). The blocking increases the time-distance between tokens generation and overall, the system performance is reduced to 4 tokens for the chosen time unit as the figure shows.

Figure 3.3(e) illustrates the system performance (τ_5) when the size of FIFO buffer d is increased to 2. The way processes block in this case leads to improved performance compared to point τ_4, however, since blocking on write still occurs, it is less than the performance when using unbounded FIFOs (τ_∞). Note that the blocking on write is avoided when the size of FIFO buffer d is increased to 3 (τ_6) which is illustrated in Figure 3.3(f). As a result, the achieved performance is equal to the performance when using unbounded FIFOs, i.e., $\tau_6 = \tau_\infty$ and further increasing the size of any FIFO buffers does not lead to better performance. Note also that the achieved performance depends on the memory distribution between the FIFO buffers. For example, τ_5 corresponds to a memory distribution $a = 1$, $b = 1$, $c = 1$, and $d = 2$. If the buffer size of channel c is increased to 2, then the overall memory becomes 6, however, the achieved performance is τ_6', see Figure 3.3(b). Note that $\tau_6' = \tau_5$ which means that the performance has not been improved with increasing the memory from 5 to 6. The performance of τ_6 is achieved only if the size of channel $d \geq 3$.

In our example, at point τ_6 no processes block on write. Therefore, τ_6 and point M represent the minimum buffer sizes that guarantee maximum performance determined by the bottleneck process. However, we assume that the processes have equal isolated throughput which in real systems is not very likely to be the case. When the processes have different throughput, some of them may still temporally block on write. Therefore, it might be assumed that this will lead to further drop in performance. It might be assumed also that increasing the buffer sizes would compensate for differences in the isolated throughput of the processes, and consequently, blocking on write would be avoided. However, this is not the case and increasing the buffer sizes above point M where the size of channel d is 3, is not needed, see point τ_6 in Figure 3.3(b). What will happen during execution is that data tokens will fill buffers ahead of the bottleneck process and the buffers after the bottleneck process will become almost empty. It means that always the bottleneck process will have data to consume

[3] In general, at different iterations processes of the polyhedral process networks we consider, may produce/consume data tokens to/from different FIFO channels.

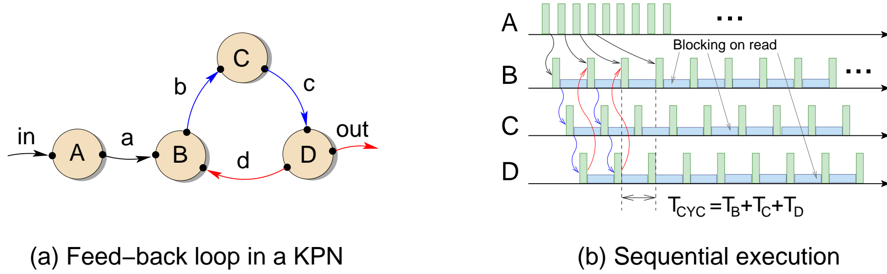


Figure 3.4: Dataflow cycle in a KPN topology.

and space where to write data. Therefore, the buffer sizes corresponding to τ_6 and point M guarantee that the bottleneck process will sustain its isolated throughput and the system performance is determined by Equation 3.7 (notwithstanding that other processes may block on writing). In Section 3.3.5, we present how these buffer sizes are computed for KPNs derived from SANLPs.

3.1.4 Dataflow feed-back loops

In the previous sections, we showed that the performance of a KPN executed on an MPSoC is determined by the throughput of the bottleneck process and the size of the FIFO channels used to communicate data between the processes. For brevity of the discussion, we assumed KPNs having a single input stream and a single output stream, and that the processes are connected in series. However, the data-flow in a KPN may form data-flow feed-back loops, which influence the performance of the KPN in a particular way when executed. Consider the example in Figure 3.4(a). It depicts a KPN containing four processes (A, B, C, and D) and four FIFO channels (a, b, c, and d). The execution time of the processes to transform a single data token is T_A, T_B, T_C, and T_D, respectively. The network has one input (in), one output (out), and one data path ($A - B - C - D$). Note that $B - C - D$ forms a data-flow loop, i.e., a *cycle* in the KPN topology. Below, we show how such cycles affect the performance. Assume that in the beginning, in order to execute, process B needs a token from channel a only, and after that, it continues reading tokens from both channels a and d. Note that process B will block on reading from either channel a or the feed-back channel d if data is not available when the process attempts to consume it. In this example, data produced by process B enables process C, and data produced by process C enables process D. Until then, process D is blocked on reading from channel c. This introduces a delay of process D (compared to process B) to start producing tokens on channel d. The delay causes process B to block on reading from channel d after producing a token on channel b[4]. The blocking on the feed-back channel d leads to a sequential execution of the processes involved in the cycle as illustrated in Figure 3.4(b), i.e., in this case the execution of processes B, C, and D is performed one after another. From now on, we refer to such cycles as *true* cycles. The

[4] Assume that process A produces tokens at a rate such that blocking of process B on reading from channel a is avoided.

sequential execution of these processes increases the time-distance between the output tokens generated out of the cycle (in our example, by process D) to a value we associate with the cycle: $T_{CYC} = T_B + T_C + T_D$. Consequently, the isolated throughput of the cycle (τ'_{CYC}) in Figure 3.4(a) is:

$$\tau'_{CYC} = \frac{1}{T_B + T_C + T_D},$$

meaning that the isolated throughput of a true cycle is lower than the isolated throughput of any of the processes involved in the cycle. It implies also that the isolated throughput of a true cycle can be lower than the isolated throughput of the bottleneck process. This is an important observation because, in such a case, the throughput of the true cycle will determine the KPN performance. Formally, according to Equation 3.7, for the example in Figure 3.4 we can write:

$$\tau_S = min(\ \rho_{env}^{in},\ \rho_A^{in},\ \rho_A^a,\ \rho_B^a,\ \rho_B^b,\ \rho_C^b,\ \rho_C^c,\ \rho_D^c,\ \rho_D^d,\ \rho_B^d,\ \rho_D^{out},\ \rho_{env}^{out}\ ).$$

However, due to the sequential execution of the processes involved in the true cycle,

$$\begin{aligned} \tau'_{CYC} &< \tau'_B, \\ \tau'_{CYC} &< \tau'_C, \\ \tau'_{CYC} &< \tau'_D, \end{aligned}$$

and we may consider the execution of the true cycle as a single entity, e.g., a process CYC with a throughput τ'_{CYC}. Consequently, for the KPN in Figure 3.4, we can apply Equation 3.7 in the following way:

$$\begin{aligned} \tau_S &= min(\rho_{env}^{in}, k_{in}.\tau'_A, k_1.\tau'_A, k_2.\tau'_B, k_3.\tau'_B, k_4.\tau'_C, k_5.\tau'_C, k_6.\tau'_D, k_7.\tau'_D, k_8.\tau'_B, k_{out}.\tau'_D,\ \rho_{env}^{out}) \\ &= min(\rho_{env}^{in}, k_{in}.\tau'_A,\ k_1.\tau'_A, k_2.\tau'_{CYC},\ k_{out}.\tau'_{CYC},\ \rho_{env}^{out}) \qquad (3.9) \end{aligned}$$

3.2 Rules for MANY-TO-ONE mapping generation

Based on the discussion in Section 3.1, we devised mapping rules for creating of MANY-TO-ONE mappings that guarantee maximum performance, i.e., performance equal to the performance of ONE-TO-ONE mapping. The main idea of generating MANY-TO-ONE mapping is to exploit the difference in the isolated throughput of the processes and to merge processes for execution on a single processing component (reducing the implementation cost) such that all the new (compound) processes have balanced throughput compared to each other. With respect to this, and in order to guarantee that a MANY-TO-ONE mapping results in an MP-SoC instance that delivers performance which matches the performance of the system with ONE-TO-ONE mapping, we propose the following mapping rules:

1. **Merge (group) processes in a way that the resulting (compound) process does not become a bottleneck.** According to Equation 3.7 and Equation 3.9, the overall system performance is determined by the bottleneck of the system, i.e., the process or the true cycle that produces or consumes data tokens from the stream having the lowest data rate. Therefore, the performance of the MPSoC will be negatively affected if a compound process (created as a result of the merging) becomes the process causing the lowest rate of a stream in the system, i.e., becomes the bottleneck.

2. **Do not merge the bottleneck process with other processes.** Recall that the execution time of the processes after merging is greater than the execution time of the processes before merging (see Section 3.1.2). Evidently, if the bottleneck process is merged (grouped) with other processes of the KPN, the resulting execution time of the new (compound) process will be higher than the execution time of the bottleneck process, and consequently, the isolated throughput of the new process will be lower than the isolated throughput of the bottleneck process before merging. This will further reduce the rate of the bottleneck stream, and thus, the overall system performance.

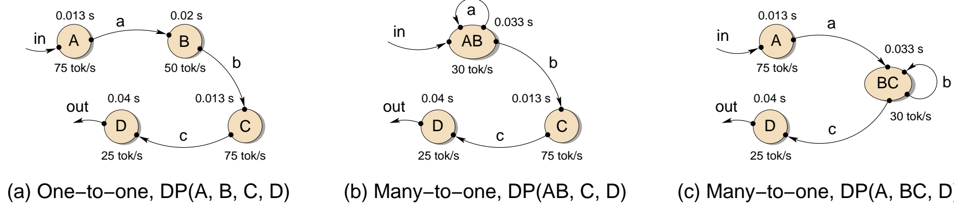
3. **Merge (group) the processes of a true cycle on a single processing component.** We have shown that the KPN processes involved in a true cycle (see Section 3.1.4) are executed sequentially one after another. Therefore, there is no benefit in mapping the processes of a true cycle on separate processing components since they will execute in sequence. Moreover, an additional delay to the execution time of the true cycle will be introduced if processes of a true cycle are grouped with other processes of the network (see Section 3.1.2). Therefore, the most appropriate approach is to merge the processes of a true cycle together, keeping the sequential execution of the cycle on a single processing component.

4. **Merge (group) processes along a data path with *neighboring* processes.** This means to merge only processes that have direct connections between each other. Merging processes of a data path that do not have direct connections introduces cycles in the KPN topology after merging. As we already showed, when these cycles are true cycles, system performance is compromised.

To check whether Rule 1 is respected after merging of processes, Equation 3.1 has to be re-evaluated for all the streams connected to the new process. Note that the rate of a stream (str) accessed by a process (mrg) created after merging is

$$\rho^{str}_{mrgd} = k.\tau'_{mrgd},$$

where τ'_{mrg} is the isolated throughput define by Equation 3.8. The data rate of the streams have to be compared with the data rate of the bottleneck stream in order to check whether the merging has created a new bottleneck stream. If this is the case, then the merging is not valid. Note that mapping Rule 2 does not consider the data rate of the input stream(s). In this way, the created mappings correspond to MPSoCs delivering the (theoretical) maximum performance. If however, the data rate of the input stream is known, and it is lower than the data rate of the bottleneck stream, then Rule 2 can be relaxed because the bottleneck is actually



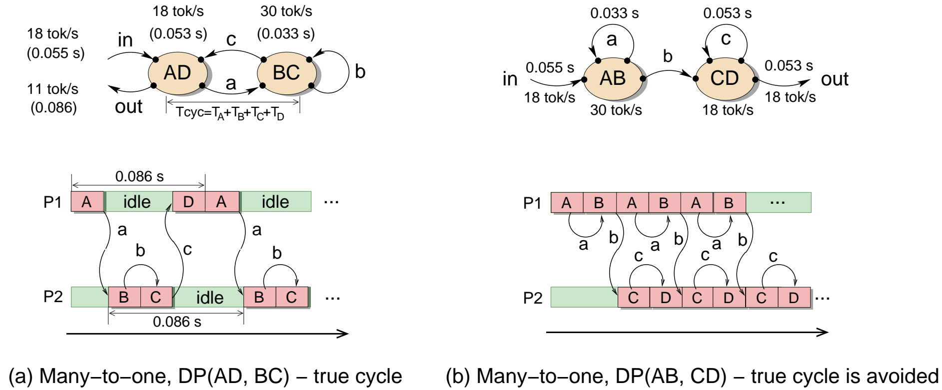
(a) One–to–one, DP(A, B, C, D) (b) Many–to–one, DP(AB, C, D) (c) Many–to–one, DP(A, BC, D)

Figure 3.5: Three design points (case 1) compliant with the mapping rules.

the source of the input data (the environment) which is not part of the system. In this case, still Rules 1, 3, and 4 have to be respected. Relaxing Rule 2 may result in a reduced number of processing components of the MPSoC instances, and therefore, in more cost-efficient implementations. An implication of Rule 4 is that the direct connections (FIFO channels) between the merged neighboring processes along a data path become self-loops after merging which, as we will show in Section 3.3.3, are not true cycles. In addition, respecting Rule 4 improves data locality, i.e., less data is communicated between the processing components in an MPSoC which means less communication overhead.

Example illustrating the mapping rules

Below, we use an example to illustrate how the mapping rules allow the design space to be reduced. Consider the process network in Figure 3.5(a). It consists of four processes (A, B, C, and D) connected in series, three FIFO channels (a, b, and c), and one input (in) and one output stream (out). Consequently, there is one data path in this KPN, i.e., $A-B-C-D$. In order to have a small design space for the purpose of the example, assume that this KPN can be mapped onto MPSoCs with only one type of processing components and the connections are point-to-point. Every process is annotated with two numbers, i.e., the process execution time and the isolated throughput when executed on this processing component. For brevity, the isolated process throughput is equal to the rate of consuming and producing data tokens from/to the corresponding channels, i.e., all the coefficients that determine the data rate of the streams are equal to 1. Consequently, the bottleneck process in this example is process D with an isolated throughput 25 tokens/sec. Note that since there is only one type of processing component, there is no need to specify to which instance of a processing component in the MPSoC a process is mapped on, i.e., a design point (DP) is defined only by the number of processing components and the mapping. To represent a design point, we use the following notation: The design point corresponding to a ONE-TO-ONE mapping is specified as $DP(A, B, C, D)$, and a design point corresponding to a MANY-TO-ONE mapping is specified as $DP(AB, C, D)$ when processes A and B are merged together. Defined in this way, the design space consists of 14 design points, i.e., there are 14 different implementation possibilities. The possibilities range from all processes mapped on a single processing component, i.e., $DP(ABCD)$, to every process mapped on a separate processing component: $DP(A, B, C, D)$. In this example, we consider one case to illustrate the pruning of the design space by applying all mapping rules and one case in which mapping Rule 2 can be relaxed, i.e.,



(a) Many–to–one, DP(AD, BC) – true cycle (b) Many–to–one, DP(AB, CD) – true cycle is avoided

Figure 3.6: Case 2: Two design points.

1. Input data rate is higher than the rate of the bottleneck stream, therefore, process D is the bottleneck process.

2. Input data rate is lower than the rate of the bottleneck stream, therefore, process D is not a (real) bottleneck process.

In case 1, there are only three design points that comply with the defined mapping rules, i.e., do not merge the bottleneck process and do not introduce new bottleneck processes. These points, $DP(A, B, C, D)$, $DP(AB, C, D)$, and $DP(A, BC, D)$, are illustrated in Figure 3.5. Note that the latter two points correspond to MPSoCs with three processing components which implies a reduced implementation cost. For case 2, assume that the data rate of the input stream is 18 tokens/sec (execution time=0.055 sec). In this case, the bottleneck is not process D but the input stream which allows for relaxing mapping Rule 2, i.e., to merge process D with other processes as well in order to achieve more cost efficient implementations. This results in the following additional design points: $DP(A, B, CD)$, $DP(AD, B, C)$, $DP(ABC, D)$, $DP(AD, BC)$, and $DP(AB, CD)$. The latter two points correspond to MPSoCs with only two processing components. These points are shown in Figure 3.6. Note that point $DP(AD, BC)$ violates Rule 4. We use this design point as an example to illustrate the importance of mapping Rule 4. By merging processes A with D and B with C, we create a true cycle (AD-BC) in the KPN topology which is shown at the top of Figure 3.6(a). The implication of this is that although the execution time of the merged processes results in isolated throughput which is higher than the input data rate, the system performance is limited to 11 tokens/sec. The reason is that processes A, B, C, and D are executed sequentially, one after another, as shown at the bottom part of the figure. In contrast, if we respect Rule 4 and merge processes A with B and C with D, see Figure 3.6(b), a true cycle is avoided. The isolated throughput of the merged processes (C and D) matches the input data rate, hence, this mapping is optimal, i.e., with respect to the rate of the input data stream, maximum performance is achieved with minimum number of processing components.

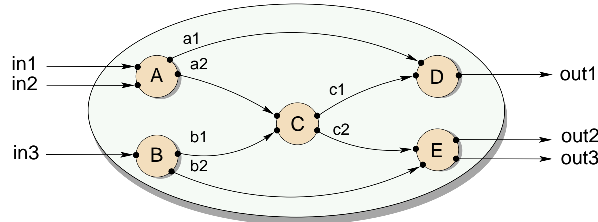


Figure 3.7: System with multiple input and output streams.

The mapping rules for systems with multiple input and output streams

In Section 3.1, we showed that the system performance is determined by the bottleneck process. For brevity of the discussion, we assumed that the system, i.e., the process network, had a single input and a single output. The static and affine nested loop programs (SANLP) that we use to derive process networks however, may result in KPNs with multiple input and output streams, and multiple data paths which have to be taken into account when applying the mapping rules. Consider the example in Figure 3.7. It represents a KPN with three input streams and three output streams. Also, some of the processes consume data from two inputs and some produce data to two outputs. Consequently, there are multiple data paths (see Definition 3.1.7) in this KPN topology as well. Since in such systems a single input stream may contribute to the generation of tokens on several output streams and several input streams may contribute to the generation of tokens to a single output stream, in order to apply the mapping rules defined is Section 3.2, we need to:

1. Identify for every output stream, a set of data paths contributing to the generation of tokens to that stream;
2. Identify the bottleneck, a process or a true cycle, for every set of data paths.

That is, for KPNs with multiple output streams, we need to identify a set of bottleneck processes (and/or true cycles) and to apply the mapping rules accordingly. The set of data paths that has to be considered for a single output stream is comprised by all data paths that end at the process generating data to that stream. Consider the example in Figure 3.7. The set of data paths for output stream $out1$ consists of the following paths:

$$(A - D),$$
$$(A - C - D),$$
$$(B - C - D).$$

Data to output streams $out2$ and $out3$ is generated by process E, and consequently, both streams lead to the same set of data paths:

$$(A - C - E),$$
$$(B - C - E),$$
$$(B - E).$$

Consequently, to apply the mapping rules, we need to find the bottleneck (a process or a true cycle) for every set of data paths. More precisely, we need to apply Equation 3.7 for every set of data paths in order to find the stream with the lowest data rate in the system and the process which causes the lowest data rate of the stream, respectively. Therefore, we need the values of the isolated throughput of the processes, and the rate of the data streams, i.e., values of the coefficients in the equation.

3.3 Applying the mapping rules

The mapping rules proposed for pruning the design space are general for process networks. However, pruning the design space in principal depends on the decidability of the considered model of computation. In the previous section, we only showed that there is a room for pruning of the design space, i.e., by respecting the proposed mapping rules, the number of different implementation possibilities is reduced. In order to apply the mapping rules, we need to find the bottleneck process(es) and to identify true cycles in the KPN topology. In order to do so, we need a mechanism to estimate the throughput of the individual processes and the processes when merged, the throughput of the true cycles, and the rate of the data streams. In addition, the presented discussion in the previous section assumes buffer sizes that do not affect performance (ideally, unbounded). Consequently, when bounded in an implementation, we need buffer sizes that guarantee maximum performance. Unfortunately, the (general) KPN MoC is not decidable at design time, therefore, the required information can not be obtained (at design time). In this section, we present how the mapping rules can be applied in practice considering specific properties of the application and platform models we use in the DAEDALUS design flow.

To represent KPNs, we use polyhedral descriptions, therefore, we call our KPNs polyhedral process networks (PPN). The PPNs are specific case of KPNs, i.e., PPNs are static and everything about the execution of the process networks is known at compile time. Moreover, the PPNs execute in finite memory and the amount of data communicated through the FIFO channels is also known. This enables techniques to estimate throughput of the processes when executed on processing components, to identify true cycles and their throughput, and to calculate buffer sizes, therefore, to apply the mapping rules. The approach is explained in the remaining part of this section. In Section 3.3.1, we present details about the representation of the PPNs we consider. In Section 3.3.2, we present an approach to calculate the isolated throughput of processes when executed on particular processing component, and the throughput when processes are merged. This is followed by a discussion in Section 3.3.3 how to identify true cycles in the considered PPNs and how to estimate the throughput of the

processes involved in the cycles. Computing the data rate of the streams in a PPN is discussed in Section 3.3.4. Finally, in Section 3.3.5, we present how to compute the minimum buffer sizes that guarantee maximum performance for PPNs derived from SANLPs.

3.3.1 Polyhedral process networks (PPN)

Recall that we consider KPNs that are input-output equivalent to static affine nested loop programs (SANLPs). Such process networks can be derived from SANLPs using the PNGEN tool. The PNGEN tool partitions a SANLP into processes only at function boundaries, i.e., the programmer divides the SANLP into functions (application tasks), thus guiding/determing the granularity of the automatically derived processes. Therefore, the parallelism in our KPN is expressed at the level of the application tasks as a process implements a single application task only. A process of a PPN consists of a *function*, *input ports*, *output ports*, and *control*. The function specifies how data tokens from input streams are transformed to data tokens to output streams. The function also has input and/or output arguments. The input and output ports are used to connect a process to FIFO channels in order to read data tokens initializing the function input arguments and to write data generated as a result of the function execution. The control specifies how many times the function is executed, which input ports to read and which output ports to write every time the function is executed. As a result of the restrictions imposed by the SANLPs discussed in Section 2.3.1, the control of a process can be compactly represented mathematically (using the polytope model [71]) in terms of linearly bounded sets of iterator vectors. A process has a *Process Domain* (DM_P) which is the set of all iterator vectors. Each iterator vector corresponds to one and only one integral point in a polytope[5]. The integral points are called also *iterations* because they correspond to the loop iterations in the initial SANLP. A function has a *Function Domain* (DM_F) which is a subset of DM_P. Similarly, input and output ports to which function arguments are bound, have *Input and Output Port Domains* (DM_{IP} and DM_{OP}, respectively) that are subsets of DM_P. The integral points in DM_{IP} (and DM_{OP}, respectively) specify the iterations in which a port is read (written respectively). Formally,

$$DM = \{P(p) \cap \mathbb{Z}^n\}, \tag{3.10}$$

where $P(p)$ is a parametric polytope,

$$P(p) = \{i \in \mathbb{Q}^n, p \in \mathbb{Z}^m \mid Ai \geq Bp + C\}, \tag{3.11}$$

where i is an iteration vector, A, B and C are integral matrices of appropriate dimensions, and p is a static parameter vector with an affine range,

$$R(p) = \{p \in \mathbb{Z}^m \mid Dp \geq E\}, \tag{3.12}$$

where D and E are integral matrices of appropriate dimensions. Also, $DM_F \subseteq DM_P$ subject to $A' \geq B'i + C'$. In the same way, $DM_{IP} \subseteq DM_P$ and $DM_{OP} \subseteq DM_P$. The

[5] Actually a linearly bounded lattice. Without loss of generality, we assume lattice matrices to be the identity.

number of integral points in a domain DM_X is denoted by I_X. For example, for a function domain DM_F, I_F represents the number of times function F is executed, and for an input port IP, I_{IP} represents the number of tokens consumed from the FIFO channel (the stream respectively) connected to this port.

To every function argument corresponds a set of input (or output) ports bound to the argument. In every execution of function F, its input arguments have to be initialized. At different iterations, an input argument may be initialized reading data from different ports bound to the function argument. However, at any iteration only one port may be used to initialize an input argument. Also, an input/output port may be bound to only one input/output argument. At any iteration, the value of an output argument may be written to more than one output ports. Formally,

1. An input argument of a function may be bound to more than one input port IPx with the following relations:

$$\begin{aligned} DM_{IP1} &\cup DM_{IP2} \cup ... \cup DM_{IPn} \equiv DM_F, \\ DM_{IP1} &\cap DM_{IP2} \cap ... \cap DM_{IPn} \equiv \emptyset \end{aligned}$$

2. An output argument of a function may be bound to more than one output port OP_x with the following relations:

$$\begin{aligned} DM_{OP1} &\cup DM_{OP2} \cup ... \cup DM_{OPm} \equiv DM_F, \\ DM_{OP1} &\cap DM_{OP2} \cap ... \cap DM_{OPm} \supseteq \emptyset \end{aligned}$$

3.3.2 Isolated average throughput of a PPN process

Recall that system performance is determined by the isolated throughput of the bottleneck process (τ'_P). Below, we present how to determine the value of τ'_P for PPN processes when targeting MPSoC execution. The isolated process throughput τ'_P is determined by the execution time T_P which is the time-distance between generation of output data tokens:

$$\tau'_P = \frac{1}{T_P} \tag{3.13}$$

The value of τ'_P represents the isolated throughput of a process when executed on a particular processing component in our platform model. Therefore, the throughput is determined by both, the function that a process realizes and the type of the processing component that executes the process. Moreover, the execution time T_P can vary in different iterations of the process due to data dependent execution time of the function transforming the input data tokens. In [80], it has been shown that data streaming architectures with varying processing delay (in which PPN implementations result) take advantage of average performance rather than worst case performance. Therefore, in PPNs we use the isolated average throughput of a process which is defined by the average execution time $Tavrg$ over the function domain DM_F:

$$Tavrg = \frac{1}{I_F}\sum_{i=1}^{I_F} T_{Fi} \tag{3.14}$$

where I_F is the number of integral points in the function domain DM_F, i.e., the number of iterations in which function F is executed, and T_{Fi} is the function execution time at iteration i. In order to determine its value, a particular processing component from the platform model has to be considered. The value of $Tavrg$ can be obtained by executing function F on the target processing component using some representative data set for a given function domain and measuring execution times. In addition, due to the uniform process structure presented in Section 2.4.1, always a process reads some ports (initializing input function arguments) prior executing the function and writes to some ports after the function is executed. Therefore, the process execution time T_P used in Equation 3.13, includes the delay T_{RD} to read a data token from an input FIFO, the average execution time of the function execution $Tavrg$, and the delay T_{WR} to write a data token to an output FIFO in the following way:

$$T_P = IN.T_{RD} + Tavrg + OUT.T_{WR}, \tag{3.15}$$

where IN is the number of input function arguments and OUT is the number of output ports to which the results are written.

Definition 3.3.1 (Isolated average throughput of a PPN process, τ'_P)
The isolated average throughput of a PPN process is the process throughput defined by Equation 3.13, Equation 3.14, and Equation 3.15.

Defined in this way, the isolated process throughput τ'_P represents the maximum rate at which data tokens are produced (to any output stream), i.e., τ'_P represents how often a process may execute its function and generate output data. We use τ'_P to compute the rate of the individual output streams of a process. Details are given in Section 3.3.4.

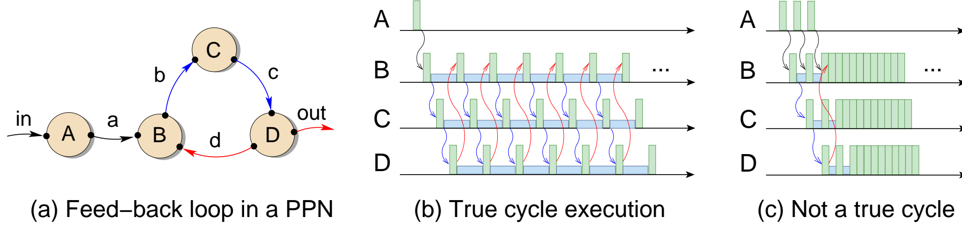
When processes are merged (grouped together) in order to execute on a single processing components, we assume a fair ROUND-ROBIN schedule. Therefore, according to the discussion in Section 3.1.2, the isolated throughput of the processes after merging is

$$\tau'_{merged} = \frac{1}{T_{merged}} = \frac{1}{\sum_{i=1}^{n} T_{Pi}},$$

where T_{Pi} is the execution time of a process i defined by Equation 3.15, and n is the number of the processes that are merged.

3.3.3 Process throughput in case of dataflow loops

As we showed in Section 3.1.4, if the data dependences between processes in a process network form a cycle, this may lead to a sequential execution of the processes involved in the



(a) Feed–back loop in a PPN (b) True cycle execution (c) Not a true cycle

Figure 3.8: Execution of a PPN containing cycles.

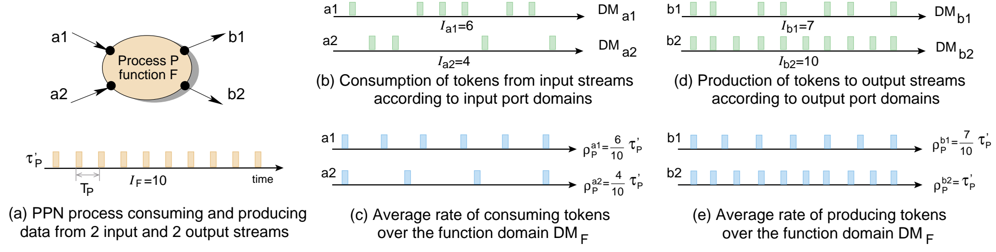
cycle. We call these cycles *true* cycles. Consider the example in Figure 3.8(a). The process network consists of four processes (A, B, C, and D). There is a cycle including processes B, C, and D. Figure 3.8(b) shows a true cycle execution in which process B reads in the beginning data from process A and then continues to read data from process D. This leads to the sequential execution of processes B, C, and D. Therefore, according to the discussion in Section 3.1.4,

$$\tau'_{cyc} = \frac{1}{T_B + T_C + T_D} \tag{3.16}$$

where T_B, T_C, and T_D are the execution time (defined by Equation 3.15) to process one data token of processes B, C, and D, respectively. Consequently, the processes in a true cycle have equal throughput, i.e, $\tau'_B = \tau'_C = \tau'_D = \tau'_{cyc}$.

Notice that in PPNs, not always a cycle leads to a sequential execution of processes, i.e., not all cycles are true cycles. If in the beginning of an execution, a sufficient number of data tokens are injected into the cycle, then it is not a true cycle. The number of tokens required to avoid an execution as a true cycle is equal to (or larger than) the number of processes involved into the cycle. For the example in Figure 3.8, we need to check for the third execution of the function realized by process B how many tokens have been consumed from stream a. Using the function domain DM_F and the corresponding input port domain DM_a, this can be done in a formal way. More technically, this can be done by computing Ehrhart polynomials (available in the polylib library) which allows for counting the number of integer points contained in a parameterized polyhedron [81]. If process B in Figure 3.8(a) reads sufficient data in the beginning of the execution from outside the cycle (from channel a) such that it provides enough data in order all processes to work in parallel, then the cycle actually does not limit the performance as indicated by Equation 3.16, i.e., the processes of the cycle will continue to execute in parallel even after process B starts reading data from the feed-back channel d. Therefore, the cycle is not a true cycle. This case is illustrated in Figure 3.8(c). In the beginning, process B reads 3 data tokens from channel a and feeds 3 data tokens to the cycle, respectively. This enables the parallel execution of the 3 processes involved in the cycle as the figure shows.

A self-loop is a special case of a dataflow feed-back loop (cycle). A self-loop in a PPN occurs when data produced by a process executing its function in one iteration is used by the same function in another iteration. According to the discussion above, a cycle is not a true cycle if



(a) PPN process consuming and producing data from 2 input and 2 output streams

(b) Consumption of tokens from input streams according to input port domains

(c) Average rate of consuming tokens over the function domain DM_F

(d) Production of tokens to output streams according to output port domains

(e) Average rate of producing tokens over the function domain DM_F

Figure 3.9: Average data rate of consuming and producing tokens from/to data streams.

in the beginning, the process first "injects" a number of data tokens equal to the number of processes involved in the cycle, and then starts consuming data from the feed-back channel. This actually is the case of self-loops, i.e., in a self-loop, there is only one process involved into the cycle and data is first written to the cycle and then consumed (otherwise it will lead to deadlock). Therefore, a self-loop is not a true cycle and the existence of a self-loop does not influence the isolated throughput of the corresponding process.

3.3.4 Data rate of the streams in a PPN

In Section 3.1.1, it was said that if a process network has multiple input/output streams and multiple data paths, then in order to apply the mapping rules we need to find a set of data paths for every output stream, and to identify the bottleneck process for every set of data paths. Recall that the bottleneck process is the process which consumes/produces tokens from/to the stream with the lowest data rate. In Section 3.1, we expressed the data rate of a stream as a function of the isolated throughput of the process that produces data to the stream and the process that consumes data from the stream. That is,

$$\rho_{str} = min(\rho_{P1}^{str}, \rho_{P2}^{str}) = min(k_1.\tau'_{P1},\ k_2.\tau'_{P2}),$$

where, $\rho_{P1}^{str} = k_1.\tau'_{P1}$ is the rate at which process $P1$ produces data to stream str, and $\rho_{P2}^{str} = k_2.\tau'_{P2}$ is the rate at which process $P2$ consumes data from the same stream, respectively. If a process of a PPN has multiple input/output ports, then in different iterations, data tokens are consumed/produced from/to different ports (streams) defined by the port domains. We illustrate this with an example in Figure 3.9(a) of a PPN process (P) with two input ports ($a1$, $a2$) and two output ports ($b1$ and $b2$). The process executes function F as the function domain DM_F has 10 integral points ($I_F = 10$). Consuming data tokens from the input ports is shown in Figure 3.9(b). During the function executions defined by the domain DM_F, the process reads 6 tokens from port $a1$ and 4 tokens from port $a2$. Producing data tokens to the output ports is shown in Figure 3.9(d): Process P generates 7 tokens to output port $b1$ and 10 tokens to port $b2$. Because consuming and producing tokens is application dependent, the term ρ_P^{str} actually represents an average rate of consuming/producing tokens over the function domain DM_F, see Figure. 3.9(c) and Figure 3.9(e). Recall that

$$\rho_P^{str} = k.\tau'_P,$$

where k is a coefficient and τ'_P is the isolated throughput of process P. In Section 3.3.2, we have already shown how to compute the isolated throughput τ'_P of a PPN process. To find the value of coefficient k, we use the corresponding port domain (DM_{port}) and function domain (DM_F) in the following way. Recall that the number of integral points in the port domain (I_{port}) represents the number of data tokens produced/consumed to/from a stream, and the number of integral points in the function domain (I_F) represents the number of times (iterations) the function F is executed. Therefore,

$$k = \frac{I_{port}}{I_F},$$

and consequently,

$$\rho_P^{str} = \frac{I_{port}}{I_F}.\tau'_P, \tag{3.17}$$

where τ'_P is the isolated process throughput. Always

$$DM_{port} \subseteq DM_F,$$

and consequently, always $I_{port} \leq I_F$. Therefore, the value of k is between 0 and 1, and

$$\rho_P^{str} \leq \tau'_P.$$

Note that if $I_{port} = I_F$, then $\rho_P^{str} = \tau'_P$ as illustrated in Figure 3.9(d). This is also the case when a process has a single input port and/or single output port.

3.3.5 Computing buffer sizes of the FIFO channels in PPNs

The formal definition of the KPN MoC assumes unbounded communication FIFO channels. However, they must be bounded in actual implementations. This implies a major problem when implementing a KPN because for the general KPN model, buffer sizes are not decidable at design time. Fortunately, for the considered PPNs, we devised an approach to compute a minimum buffer sizes that guarantee deadlock-free execution [7]. Note that minimum buffer sizes does not mean maximum performance because during execution, the processes may temporally block on write which additionally increases the execution delay of the processes (see Section 3.1.3). As a result, it is difficult to reason which process is the real bottleneck of the system. Therefore, in order to apply the mapping rules, we need buffer sizes that do not limit performance. Recall that the PPNs we consider, have finite process, function, and port domains. In particular, we can exploit the fact that the number of integral points of an

input/output port domain[6] (I_{port}) is finite. It means that if the corresponding FIFO channel has size equal to I_{port}, then blocking on write is avoided. Consequently, if all buffers are set to have sizes equal to the integral points of the corresponding port domains, then the PPN executes as it would have unbounded FIFO channels. That is:

$$\textit{Memory: } M_{min} \leq M_{max} < \infty,$$

where M_{min} corresponds to the minimum deadlock-free buffer sizes computed by PN-GEN [7], and M_{max} is the memory requirements defined by the port domains. Then, the relation of the performance determined by different buffer sizes of the PPN is:

$$\textit{Throughput: } \tau_{M_{min}} \leq \tau_{M_{max}} = \tau_{\infty}.$$

Note that M_{max} corresponds to buffer sizes that do not limit performance, therefore, we can use these buffer sizes when applying the mapping rules. However, depending on the application, the value of $M_{max} = I_{port}$ can be very large and impractical to use. Fortunately, for PPNs, we can compute buffer sizes, if they exists, corresponding to memory M such that

$$M_{min} \leq M < M_{max},$$
$$\tau_{M_{min}} \leq \tau_M = \tau_{M_{max}}.$$

The approach is presented below. First, we present the basic idea of computing minimum deadlock-free buffer sizes. Then, we present the way we compute buffer sizes that lead to maximum performance.

Computing minimal deadlock-free buffer sizes

Computing minimal deadlock-free buffer sizes is a non-trivial global optimization problem. This problem becomes easier if we first compute a deadlock-free schedule and then compute the buffer sizes for each channel individually. Note that this schedule is only computed for the purpose of computing the buffer sizes and is discarded afterwards because the processes in our PPNs are self-scheduled due to the blocking read/write synchronization mechanism. Although the schedule we compute may not be optimal, our computations do ensure that a valid schedule exists for the computed buffer sizes. The schedule is computed using a greedy approach. This approach may not work for process networks in general, but it does work for PPNs derived from the static affine nested loop programs we consider.

The basic idea is to place all iteration domains in a common iteration space at an offset such that the dependences in the initial program are respected. The offset is computed by the scheduling algorithm described in [82]. By fixing the offsets of the iteration domain in the common space, we have therefore fixed the relative order between any pair of iterations from any pair of iteration domains. The algorithm starts by computing for any pair of connected processes, the minimal dependence distance vector, being the difference between a read operation and the corresponding write operation. Then, the processes are greedily combined, ensuring that all minimal distance vectors are (lexicographically) positive. The end result

[6] In PPNs, for any input port (IP) and output port (OP) connected to a FIFO channel, $I_{IP} = I_{OP}$, see also Definition 2.5.1.

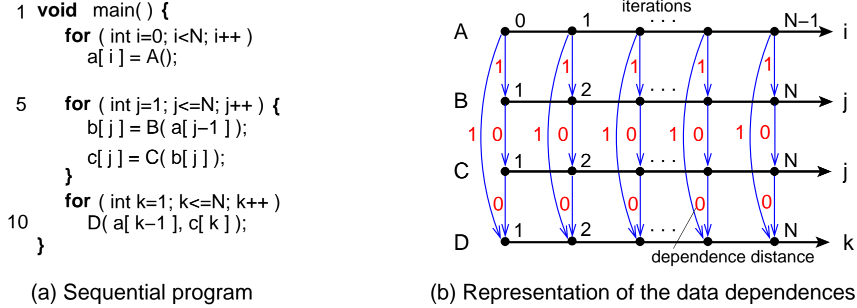


Figure 3.10: Example of a sequential program for the PPN in Figure 3.3(a).

is a schedule that ensures that every data element is written before it is read. For more information on this algorithm, we refer to [82], where it is applied to perform loop fusion on SANLPs.

Consider the sequential program in Figure 3.10(a). It results in the process network in Figure 3.3(a). Recall that we used this process network as an example to illustrate how buffer sizes affect performance. For illustrative purposes, we use the same process network to show how the minimum deadlock-free buffers as well as the buffer sizes that guarantee maximum performance are computed. The data dependences are depicted in Figure 3.10(b). The horizontal axes illustrates the single dimension of the iteration domains of the processes (function calls) A, B, C, and D, and the arrows show the data dependences. The value of the dependence distances are shown next to each arrow. As a next step, a valid global schedule is computed by combining processes together in a way that keeps the distance between write operations and the corresponding read operations minimal[7]. The result is shown in Figure 3.11(a). In this figure, next to each arrow, we also show the FIFO channels used to propagate the corresponding data at each iteration, e.g., FIFO a is used to propagate data between processes A and B. In the common iteration space, the horizontal axis represents the single dimension of the problem and the vertical axis represents the additional dimension that orders the statements inside the inner loop.

To compute the buffer sizes, we compute the number of read iterations $R(i)$ that are executed before a given read operation i and subtract the resulting expression from the number of write iterations $W(i)$ that are executed before the given read operation:

$$\#elements\ in\ FIFO\ at\ operation\ i:\ W(i) - R(i)$$

This computation can be performed entirely symbolically using the `barvinok` library [83] that efficiently computes the number of integer points in a parametric polytope. The result is a piecewise (quasi-)polynomial in the read iterators and the parameters. Then, the required buffer size is the maximum of this expression over all read iterations:

$$FIFO\ size\ =\ max(\ W(i) - R(i)\ )$$

[7] For the scheduling of processes having domains with different dimensions, all iteration domains are embedded in spaces of the same dimension (i.e. the dimensions are equalized), with a fixed coordinate value for the extra “dummy” dimensions. This is equivalent to (virtually) adding extra loops containing only one iteration.

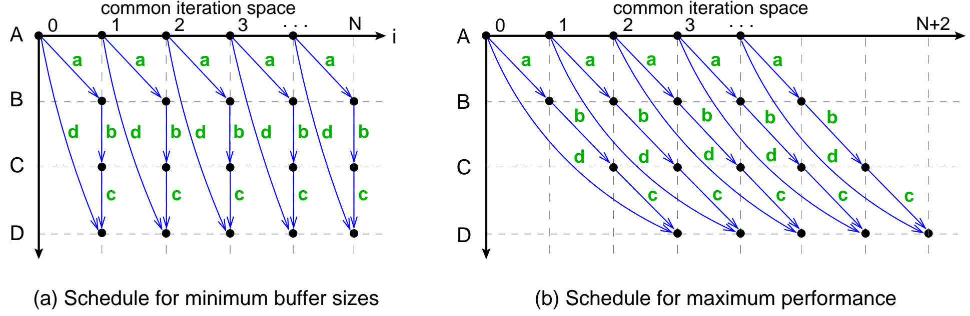


Figure 3.11: Two schedules for the PPN in Figure 3.3(a) used to compute minimum deadlock-free buffer sizes (a), and buffer sizes guaranteeing maximum performance (b).

To compute the maximum symbolically, we apply Bernstein expansion [84] to obtain a parametric *upper bound* on the expression.

Below, we show how the buffer sizes are computed based on the schedule in Figure 3.11(a). Consider FIFO a. Let the number of elements written to the FIFO by process A before iteration i is denoted as $W_A^a(i)$ and the number of elements read from the same FIFO by process B before iteration i is denoted as $R_B^a(i)$. Then, for every iteration $i, i \in [1, N]$, we compute the difference $W_A^a(i) - R_B^a(i)$ and assign the maximum difference as the buffer size of FIFO channel a. For example, consider the fourth iteration of the common iteration spaces $(i = 3)$. Then,

$$\begin{aligned} W_A^a(3) &= 3, \\ R_B^a(3) &= 2, \\ W_A^a(3) - R_B^a(3) &= 3 - 2 = 1. \end{aligned}$$

Due to the uniform data dependences in the example, $W_A^a(i) - R_B^a(i) = 1, \forall i \in [1, N]$ and consequently the size of FIFO channel $a = max(\ W_A^a(i) - R_B^a(i)\ ) = 1$. In the same way, we compute the buffer sizes of the remaining FIFOs, i.e.,

size of FIFO channel $b = max(\ W_B^b(i) - R_C^b(i)\ ) = 0$,
size of FIFO channel $c = max(\ W_C^c(i) - R_D^c(i)\ ) = 0$,
size of FIFO channel $d = max(\ W_A^d(i) - R_D^d(i)\ ) = 1$.

Because a buffer size can not be zero, buffer sizes 1 are assigned to all FIFO channels being the minimum buffer sizes that guarantee deadlock-free execution of the process network (see also Section 3.1.3).

Computing buffer sizes that guarantee maximum performance

If we look at Figure 3.11(a), we see that the scheduling algorithm scheduled processes B, C, and D for execution at the same iterations of the common iteration space. This means

that write and the corresponding read operations are scheduled at the same iterations where correctness is guaranteed by the lexicographical order and the blocking semantics. This leads to the minimal buffer sizes, however, as we showed in Section 3.1.3, these buffer sizes also lead to temporal blocking on write during the execution of the processes and therefore, to reduced performance. Recall that after process A writes data to channels a and d, it blocks on writing to (the full) channel d until process D reads data from it. In order to guarantee maximum performance, we need to compute buffer sizes such that the temporal blocking of the execution is avoided (as in the case of unbounded FIFOs). This easily can be achieved by modifying the scheduling algorithm in the following way: When processes are combined (in the common iteration space), the algorithm ensures that no write and read operations of a write-read pair are scheduled at the same iterations. That is, the algorithm “shifts” the read operations at further iterations with relation to the corresponding write operations.

For the process network we use as an example, the shifting is depicted in Figure 3.11(b). Once the schedule is found, the buffer sizes are computed in the same way as we already described: $FIFO\ size\ =\ max(\ W(i) - R(i)\ )$. For the example, this results in buffer size 1 for FIFO channels a, b, and c; and buffer size 3 for FIFO channel d. As we already discussed in Section 3.1.3, these buffer sizes avoid blocking on write. Consequently, these are the minimum buffer sizes that guarantee maximum performance determined by the bottleneck process.

3.4 Conclusion

In this chapter, we presented techniques to prune the design space by reducing the number of implementation possibilities of MPSoC instances where each MPSoC instance is defined by an application (KPN), a platform, and a mapping. In the presented approach, the design space is reduced by limiting the number of possible mappings to a set of MANY-TO-ONE mappings which deliver the same (maximum) performance as the ONE-TO-ONE mapping for the same application and platform. Also, in this chapter, we discussed the factors that affects system performance. Taking these factors into account and given the knowledge we have about our application and platform models, we proposed mapping rules that allow for creating MANY-TO-ONE mappings while keeping the performance of ONE-TO-ONE mapping. In addition, we discussed how the mapping rules can be applied in practice considering the KPN application model and the polyhedral descriptions we use to represent a KPN. An assumption of the presented discussion in this chapter is that the FIFO channels are bounded to sizes guaranteeing maximum performance. Consequently, for KPNs derived from static affine nested loop programs, we presented how such FIFO sizes are computed at design time.

In this chapter, we showed that the devised mapping techniques can effectively prune the design space without compromising the quality of the generated results (i.e., the design points representing MPSoC instances). Therefore, the presented mapping approach can be used to complement the techniques in the SESAME tool in order to improve the design space exploration in the DAEDALUS design flow.

Chapter 4

Case studies

In this dissertation, we proposed methods and techniques to close the implementation gap (introduced in Chapter 1) between the System and the RTL abstraction levels of description. These methods and techniques are implemented in the DAEDALUS framework presented in Section 1.2. With DAEDALUS, the implementation gap is closed in a particular way because we target only embedded multiprocessor systems that execute data-streaming applications in the domain of multimedia and signal processing using the KPN MoC as a programming model. DAEDALUS offers a fully integrated tool-flow for very fast exploration and implementation of alternative MPSoCs, where design space exploration (DSE), system-level synthesis, application mapping, and system prototyping of MPSoCs are highly automated. In this chapter, we present three case studies which demonstrate the potential and the efficiency of our methods and techniques for automated MPSoC design in terms of overall design time, achieved performance, and HW utilization. Also, we comment on the accuracy of the results obtained by performing high-level system simulations (during the DSE process) compared to real implementation numbers.

The first case study uses a JPEG encoder application to show the steps in DAEDALUS to close the implementation gap in the system-level MPSoC design. It illustrates a complete flow, starting from a sequential program, performing system-level DSE with SESAME, synthesizing design instances with ESPAM, and prototyping them by using commercial synthesis and compiler tools. In this case study, we illustrate the design time and the efficiency, in terms of HW resource utilization, of our approach for connecting processing cores using communication component, memories, and controllers. In addition, we comment on the accuracy of the models used in the system-level DSE process by comparing the achieved results with the results we have obtained by measuring actual numbers from real implementations.

In the second case study, we address heterogeneous MPSoCs where both programmable processors and dedicated IP cores are used as processing components. We illustrate the approach, discussed in Section 2.4, for integrating of predefined IP cores into heterogeneous systems by using automatically generated IP Modules. We show its efficiency by implementing three

applications, namely, the JPEG encoder application used in the first case study, a Sobel edge detection, and a Discrete Wavelet Transform (DWT). In this case study, we comment on the design time, IP core integration time, resource utilization of the prototyped systems, and the obtained performance results.

The purpose of the last case study is to push DAEDALUS “to the limit” in order to check how large and complex systems can be designed using the proposed methodology and considering the constraints imposed by the FPGA technology we use. In this case study, we are interested in the maximum performance that can be achieved, therefore, we consider MPSoCs with a point-to-point communication topology only. We report on the size, in terms of number of processing components, and the performance achieved by several alternative homogeneous and heterogeneous MPSoC instances. In this experiment, the MPSoC instances realize the JPEG encoder application exploiting both task and data parallelism.

4.1 Experimental setup

Currently, for fast prototyping in order to validate our approach, we use the Xilinx VirtexII and VirtexII-Pro FPGA technology. Therefore, our library of processing components include the two programmable processors supported by Xilinx. These are the $MicroBlaze$ [42] soft-core processor and the $PowerPC$ [43] hard-core processor. In addition, our platform (library of components) contains several dedicated predefined IP cores. Our approach for IP core integration imposes several requirements for these IPs discussed in Section 2.4. The communication part of our platform model contains several communication components, i.e., a point-to-point network, a crossbar switch, and a shared bus component with several arbitration schemes. These *communication* components are mutually exclusive and determine the communication topology of a multiprocessor platform instance.

In the experiments presented in this chapter, we used an FPGA prototyping board connected to a Pentium based personal computer (PC) through a PCI interface. The FPGA board contains 6 banks of static memory, 256K x 32bits each. The memory can be accessed either from the PC or the FPGA and it is used for data communication between the PC and the FPGA board. The PC serves only as a host to the FPGA board, i.e., the PC is used to configure the FPGA, and to organize the input and the output data transfers. The output generated by ESPAM is used to generate the bit-stream file that configures the FPGA for which we use a GCC compiler and a VHDL synthesizer provided by Xilinx [10].

4.2 Homogeneous MPSoCs design with DAEDALUS

To demonstrate the steps in the DAEDALUS system design flow, in this case study we use real-life example, namely a JPEG encoder application. The main objective of this experiment is to show that DAEDALUS successfully closes the aforementioned implementation gap. In this case study, we evaluate the effectiveness of the design flow for automated MPSoC synthesis, programming, and implementation in terms of total design time, i.e., how fast alternative multiprocessor systems can be synthesized, programmed, and implemented.

Also, we comment on the HW resource utilization of the implemented MPSoCs employing our approach to connect processors using a communication component, memories, and controllers. In addition, we validate, in terms of accuracy, the high-level simulation models used in SESAME to explore the design space targeting MPSoCs with $MicroBlaze$ (MB) and $PowerPC$ (PPC) processors, and crossbar communication topology. We present a comparison between the results obtained by running system-level simulations (during the design space exploration) and real implementations of the JPEG encoder application. The system design steps in DAEDALUS are outlined below.

1. **KPN generation.** Starting from a sequential C program of the JPEG encoder, an equivalent KPN specification is derived automatically by the PNGEN tool. Recall that the input of PNGEN are sequential programs restricted to the class of static affine nested-loop programs which were discussed in Section 2.3.1.
2. **System-level DSE.** The derived KPN specification is used by SESAME to perform system-level DSE using high-level models of the components from the platform model presented in Section 2.1.5. SESAME allows for quickly evaluating the performance of different application-to-MPSoC mappings, HW/SW partitionings, and target MPSoC topologies. The exploration results in a number of promising MPSoC design instances, candidates for implementation.
3. **System-level MPSoC synthesis.** A system-level description of an MPSoC is translated to RTL by the ESPAM tool, presented in Chapter 2. The input to ESPAM is the KPN specification and the high-level system specifications, i.e., a platform and a mapping for an MPSoC instance. Using these specifications and together with the RTL version of the platform components, ESPAM automatically generates synthesizable VHDL that implements each candidate MPSoC instance. In addition, ESPAM generates C code for these KPN processes that are mapped onto programmable cores.
4. **Final implementation.** The output generated by ESPAM is subsequently used by commercial synthesis tools and compilers to generate the final implementation of the MPSoC instances. Since with DAEDALUS we currently use the Xilinx FPGA technology, the MPSoCs are prototyped on a Xilinx FPGA using the Xilinx Platform Studio (XPS) tool [10].

4.2.1 Design time

As explained in Section 1.2, ESPAM needs three input specifications, namely an application specified as a KPN, a platform specification, and a mapping specification. Table 4.1 shows that the KPN specification of the JPEG application was derived in 22 seconds from sequential C code using our PNGEN tool [7]. A small manual modification (taking no longer than 30 minutes) to the initial C code was necessary to meet the PNGEN tool input requirements. Generating the KPN specification is a one-time effort since the same specification is used for all subsequent exploration and implementation steps.

The platform and the mapping specifications were generated by SESAME after performing design space exploration using the derived KPN specification of the JPEG application as an

Table 4.1: Processing Times (hh:mm:ss).

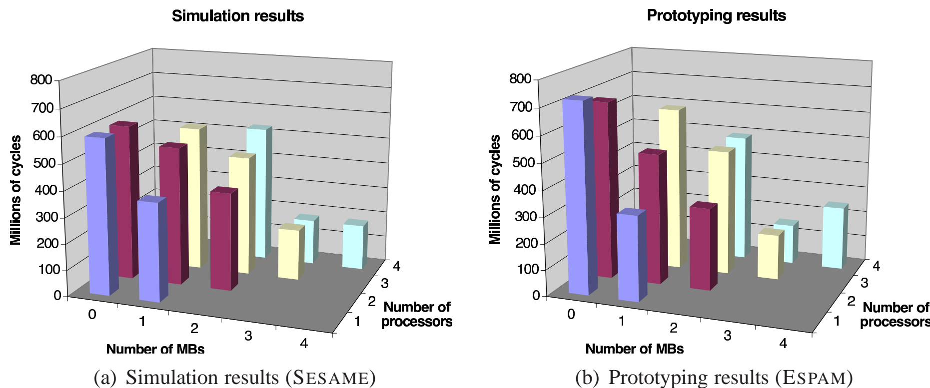
Design steps	Tools in DAEDALUS	KPN Deriv.	Plat./Map. Deriv.	System to RTL conv.	Physical Implement.	Manual Modific.
Step 1	PNGEN	00:00:22	—	—	—	00:30:00
Step 2	SESAME	—	01:26:00	—	—	—
Step 3	ESPAM	—	—	00:25:00	—	—
Step 4	XPS	—	—	—	18:29:00	—

input. We explored heterogeneous, crossbar-based MPSoC platforms with up to 4 processors (MB or PPC). In our design space exploration, we used three degrees of freedom, namely the number of processors in the platform (1 to 4), the type of processors (MB versus PPC), and the mapping of application processes onto the processors. This resulted in a design space consisting of 10148 design points. In this experiment, the mapping rules presented in Chapter 3, are not applied. Instead, by using SESAME and performing system-level simulations, we exhaustively explored the resulting design space. The reason of exploring the whole design space is to verify the proposed mapping rules which we do in the following way: Evaluate all design points from the design space, select the best found design points, and check whether they actually comply with the proposed mapping rules.

As illustrated in Table 4.1, the complete design space sweep took 1.5 hours. We selected 11 design points that represent 11 alternative MPSoCs with the best found application-to-MPSoC mappings in terms of performance of the application executed on these MPSoCs. Then we checked, and the results confirmed that the mapping of the selected 11 best design points comply with the mapping rules presented in Chapter 3. The generated platform and mapping specifications for each of the selected 11 design points, together with the application specification (KPN), are used by ESPAM to synthesize, program, and generate 11 multiprocessor systems at RTL. This process took 25 minutes, see Table 4.1. The generated files were automatically imported to the Xilinx Platform Studio (XPS) tool for physical implementation, i.e., mapping, place, and route onto the FPGA. For prototyping in this experiment we used an FPGA board with the Xilinx VirtexII-Pro-20 device. It took the XPS tool more than 18 hours to implement the 11 MPSoCs. All tools ran on a Pentium IV machine at 1.8GHz with 1GB of RAM. The figures in Table 4.1 show that a complete implementation and programming of all 11 MPSoCs starting from high abstraction system-level specifications can be obtained in just about 22 hours using our system design flow. So, a significant reduction of design time is achieved.

4.2.2 Performance results and accuracy of the DSE numbers

Performance results are shown in Figure 4.1. The performance numbers obtained during design space exploration by simulations of system-level models for the selected MPSoCs are shown in Figure 4.1(a). The real performance numbers for the same MPSoCs implemented and run on the FPGA are shown in Figure 4.1(b). In both figures the left axis shows the performance numbers (in clock cycles) of each alternative MPSoC. The right axis shows how many processors an MPSoC contains and the bottom axis shows how many of them are $MicroBlaze$ processors. For example the bar with right coordinate 4 and bottom coordinate



(a) Simulation results (SESAME) (b) Prototyping results (ESPAM)

Figure 4.1: Validation experiment: simulation results vs. actual measurements.

2 (4,2) represents the performance of a system that has 4 processors where 2 of them are $MicroBlazes$. It means that the other 2 are $PowerPC$ processors. The empty points in the figures represent *non-implementable* design points, i.e., point (3,4) means a system with 3 processors and 4 of them to be $MicroBlazes$.

The performance numbers in Figure 4.1 show that with a 4-processor MPSoC, a performance improvement close to the theoretical maximum (4x) is attainable. An important observation, however, is that the performance degrades with increasing the number of $PowerPC$ processors. This is because on the Xilinx FPGAs, a $PowerPC$ processor uses a single (shared) memory for storing instructions and data, although, the $PowerPC$ architecture allows for using separate memories. The shared memory needs arbitration which amortizes the efficiency and the higher working frequency of the $PowerPC$ processor compared to a $MicroBlaze$ processor. Comparing the simulation numbers with the implementation numbers, we see that the system-level simulations adequately show the correct performance trends, with an average error of 13% and worst-case error of 28%. The inaccuracies in terms of absolute cycle numbers are caused by the high-level modeling of the processors' behavior (mainly of the $PowerPC$ shared memory) and the request-based communication mechanism. Actually, accuracy is the price we have to pay in order to achieve very fast simulations and design space exploration.

Using ESPAM and the XPS tools, we implemented, ran, and measured the performance of the alternative MPSoCs described above in approximately 2 days. This fact indicates that in a relatively short amount of time we were able to explore the performance of alternative multi-processor systems through real implementations and measurements of actual numbers. These numbers are 100% accurate. Gathering these numbers is faster than running cycle accurate simulations of the MPSoCs. We do not know how much time is needed for an experienced designer to verify an RTL simulation of several hardwired components and several processors running in parallel and executing different programs. However, we know that only setting up and performing such simulation may take days. Of course, performing simulation at a higher level of abstraction is faster than implementation and measurement of real performance but the 100% accuracy of the numbers cannot be achieved as we showed above.

4.2.3 Synthesis results

In Table 4.2 we present the overall resource utilization of the multiprocessor systems with 4 processors we consider in this experiment. We also present the utilization results for the communication controllers (CC) and a 4-port crossbar component (CB). The FPGA resources are grouped into slices that contain 4-Input Look-Up tables and Flip-Flops. The first row in the table show that the multiprocessor systems utilize around 40% of the slices in the FPGA. Also, the last three rows show that our communication component (CB) together with the CCs in each system utilize a minor portion of the FPGA slices – only 5%. These numbers clearly indicate that the approach to connect processors through communication components and communication memories, proposed in this dissertation, is very efficient in terms of slice utilization. The last column in Table 4.2 shows a relatively high overall utilization (60%)

Table 4.2: Resource Utilization.

	#Slices	#4-Input LUT	#Flip-Flops	#BRAMs
4 Proc. & Crossbar	3653 (39%)	4748 (25%)	2357 (12%)	85 (60%)
4 CCs & CMs	288 (2%)	468 (2%)	116 (1%)	9 (7%)
4 Port Crossbar	397 (3%)	587 (3%)	56 (1%)	—

of the on-chip memory. This high utilization is not related to inefficiency in our approach to connect processors via communication memories because for each JPEG system we use a maximum of 9 BRAM blocks (out of 85) to implement FIFO buffers, distributed over 4 communication memories. The high BRAM utilization is due to the fact that almost all BRAM blocks are used for the program and data memory of the 4 processors in our MPSoCs.

4.2.4 Conclusions

In this case study, we used the JPEG encoder application targeting crossbar-based MPSoCs. We illustrated the design time and the efficiency of our approach, in terms of HW resource utilization, for connecting processing cores using communication component, communication memories (CMs), and communication controllers (CCs). In addition, we commented on the accuracy of the models used in the system-level DSE process. Based on the experiments conducted in this case study, we conclude that the automation achieved with DAEDALUS significantly reduces the design time starting from system-level specification and going down to complete implementation. That is, we are able systematically, automatically, and quickly to implement and to program a multiprocessor system within 2 hours. Moreover, the presented results show that the proposed approach of connecting processors through CCs and CMs is efficient in terms of HW utilization and performance speed-up. For the JPEG encoder application implemented with four processors, the communication logic utilizes only 5% of the resources and the achieved speedup is close to the theoretical maximum as compared to a single-processor system. Based on these results, we conclude that the main limitation on the size of a multiprocessor system that can be built on a single FPGA chip still remains the amount of on-chip memory. Using the FPGA on-chip memory instead of external memories is crucial for the high-performance multiprocessor systems we target because external memories are slower than on-chip BRAMs and usually the external memories have

to be shared between multiple processors which further limits the performance. In addition, the system-level simulation results adequately show the correct performance trends. For the JPEG encoder application, the average error is 13% and the worst-case error is 28%.

4.3 Heterogeneous MPSoCs design with DAEDALUS

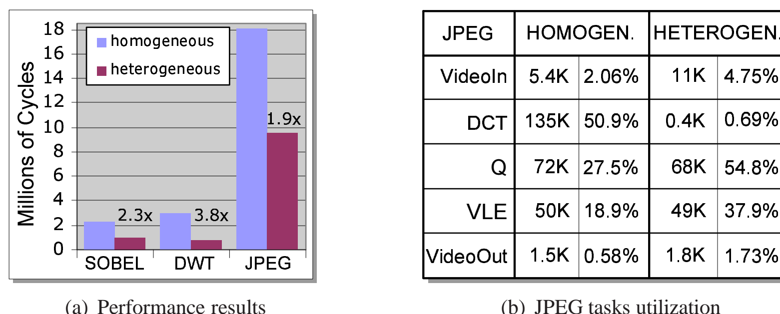
In this section, we present some of the results we have obtained by implementing and executing three applications, namely the JPEG encoder application used in the previous case study, a Sobel edge detection, and a Discrete Wavelet Transform (DWT), onto homogeneous and heterogeneous multiprocessor systems. The main objective of this experiment is to evaluate the approach, discussed in Section 2.4, for integrating of predefined IP cores into heterogeneous systems by using automatically generated IP Modules. More precisely, we evaluate the effectiveness of the proposed HW IP core integration in terms of design time, achieved performance, and HW resource utilization of the generated IP Modules. For prototyping purpose we use an FPGA board with one VirtexII-6000 device.

4.3.1 Design time

In this case study, we started with the three applications (Sobel, DWT, and JPEG) given as sequential C programs and automatically derived the *Application Specifications*, i.e., KPNs using the PNGEN tool in 5 minutes. Details about the derived KPNs are presented in [7]. For each application, the system-level *Platform* and *Mapping Specifications* were written by hand in XML format in 10 minutes. In this experiment, each of the three homogeneous MPSoCs contains 5 $MicroBlaze$ processors connected via crossbar communication component. Having all three input specifications for each application, the ESPAM tool generated and programmed a homogeneous multiprocessor system at RTL, which was imported to the Xilinx XPS tool for physical implementation onto the prototyping FPGA. The overall design and implementation time of each homogeneous system was about an hour.

We have performed similar actions as described above in order to generate three *heterogeneous* multiprocessor systems using our design flow. We had to modify only the initial system-level *Platform* and *Mapping Specifications* for each application in order to replace some of the $MicroBlaze$ processors with dedicated HW IP cores. This took us less than 5 minutes. For the Sobel application, we used 3 $MicroBlaze$ processors and 2 dedicated IP cores. The IP cores estimate the first derivative of an image intensity function. For the DWT application, we used 1 $MicroBlaze$ processor and 4 dedicated IP cores. The IP cores are 2 Low and 2 High Pass filters. For the JPEG application, we used 4 $MicroBlaze$ processors and 1 Discrete Cosine Transform (DCT) IP core. Again, the overall design and implementation time of each heterogeneous system was about an hour.

As explained above, in the heterogeneous systems we used several dedicated HW IP cores. They were written in synthesizable VHDL. For the Sobel and DWT applications, the IP cores have a simple structure, i.e., they implement convolution based operations. These IP cores have been developed and added to the library of platform components in about one working



(a) Performance results

JPEG	HOMOGEN.		HETEROGEN.	
VideoIn	5.4K	2.06%	11K	4.75%
DCT	135K	50.9%	0.4K	0.69%
Q	72K	27.5%	68K	54.8%
VLE	50K	18.9%	49K	37.9%
VideoOut	1.5K	0.58%	1.8K	1.73%

(b) JPEG tasks utilization

Figure 4.2: Experimental performance and synthesis results.

day. For the JPEG application, we used an IP core that performs a DCT operation. We have downloaded this IP core from the Xilinx website [85]. In order to add this IP core to the component library in DAEDALUS, we had to make small modification related to the control (`Enable`/`Valid`) interface discussed in Section 2.4.3. The DCT IP core provided by Xilinx has `Valid` signal but it does not have `Enable` signal. This signal was added to the IP core and the IP core to the library within 30 minutes.

4.3.2 Performance results

The performance numbers we have obtained for the implemented multiprocessor systems are shown in Figure 4.2(a). For each multiprocessor system we measured the exact number of clock cycles needed to process an image of size 128x128 pixels. As one may expect, the numbers in the figure show that the heterogeneous systems achieve better performance. This is because the dedicated HW IP cores we use, work more efficiently than the $MicroBlaze$ processors for the same functionality. What is more important to discuss here is the achieved speed-up depicted in Figure 4.2(a) above the bars of the heterogeneous systems in order to show the efficiency of our approach for IP Module generation and IP core integration. Consider the performance results of the JPEG systems. The JPEG application consists of 5 tasks, i.e., VideoIn, DCT, Quantization (Q), Variable-length encoding (VLE), and VideoOut. The left part of column *HOMOGENEOUS* in Figure 4.2(b) shows how many thousands of clock cycles it takes for a $MicroBlaze$ processor to execute each task by processing one data token – an image block of 8x8 pixels. The numbers in the next column show the same information in percentage of the overall processing time utilized by each task. It can be seen that the DCT is the bottleneck of the system taking more than 50% of the whole processing time for one block and consequently, for the whole image. These 50% mean that if the DCT is substituted with more efficient implementation, theoretically, the overall performance of the system can be increased at most 2 times. The column *HETEROGENEOUS* in Figure 4.2(b) shows the clock cycles and the percentage of each task performed by the heterogeneous JPEG system where the DCT is implemented by a very fast dedicated HW IP core and integrated using our IP Module generation approach. In this system, the DCT utilizes less than one percent of the whole processing time. In this case, Figure 4.2(a) shows that the overall speed-

up compared to the homogeneous system is 1.9x which is close to the theoretical maximum 2x for the heterogeneous system where only the DCT is a dedicated IP core. This clearly shows the efficiency of our approach for IP core integration by generating IP Modules.

4.3.3 Synthesis results

Recall that an IP Module generated by the ESPAM tool consists of an IP core and a wrapper around it where the IP core is given and only the wrapper is generated by ESPAM, see Section 2.4. Therefore, we present only the HW resource utilization of our generated wrappers in order to show how efficient our wrappers are in terms of utilized HW. In Table 4.3, we present the resource utilization of the IP wrappers of six IP cores that we used in our experiments. Each row (*Wrapper1–Wrapper6*) in Table 4.3 corresponds to an IP wrapper. The utilized FPGA resources are grouped into Slices that contain 4-Input Look-Up tables and Flip-Flops – see columns 2, 3, and 4, respectively. The numbers show low HW resource utilization which on average is 241 slices. Moreover, the resources utilized by a wrapper does not depend on the size of the IP core it integrates, i.e., a larger IP core does not require a larger wrapper. For example, Wrapper3 of the DCT core utilizes only 208 slices whereas the DCT IP core itself utilizes 1369 slices. Wrapper2 of the IP core that estimates the first derivative in Sobel utilizes 240 slices, whereas the IP core itself utilizes 424 slices.

Table 4.3: HW resource utilization of the IP wrappers.

	#Slices	#4-Input LUT	#Flip-Flops
Wrapper1	221	371	190
Wrapper2	240	371	192
Wrapper3	208	361	147
Wrapper4	274	412	173
Wrapper5	269	390	173
Wrapper6	236	351	157

In general, the HW complexity of our wrappers is determined mainly by the number of MUX and DeMUX components, the number of counters implementing *for*-loops of a KPN process, and the number of behavioral parameters of a KPN process. The three applications we used in our experiment process images. We specified the applications with two nested *for*-loops that iterate through an image and we used two behavioral parameters as loop bounds, i.e., image width and height. Since the number of *for*-loops and behavioral parameters is the same for all wrappers in our experiment, the difference in the resource utilization of our wrappers is caused by the different input/output ports of the wrappers and by the different input/output ports of the IP cores they integrate.

4.3.4 Conclusions

The purpose of the case study presented in this section was to illustrate the method and techniques implemented in ESPAM for automated integration of dedicated hardwired IP cores into heterogeneous multiprocessor systems where both programmable processors and dedicated

IPs are used as processing components. The integration is based on an IP Module generation that consists of predened dedicated IP core and a wrapper around it. The proposed IP core integration approach was applied on three real-life applications, i.e., a Sobel edge detection, a Discrete Wavelet Transform, and a JPEG encoder. Based on the obtained results, we conclude that the IP core integration in ESPAM is efficient in terms of achieved performance and HW resource utilization.

4.4 Putting DAEDALUS to work

The previous two case studies confirm that the methods and techniques implemented in the DAEDALUS design flow successfully close the implementation gap introduced in Chapter 1. Moreover, both case studies show that the DAEDALUS methodology is efficient in terms of design time, HW resource utilization, and achieved results for both homogeneous and heterogeneous MPSoC designs. Subsequently, the purpose of the experiment conducted in this case study is to push DAEDALUS ”to the limit“ in order to check how large and complex systems, in terms of number of processing components, we can design and implement given the constraints imposed by the FPGA technology we currently use for prototyping. In addition, we are interested to find out what is the maximum performance that can be achieved given the application (in this case study, the JPEG encoder), the design methodology, and the constraints of the used FPGA technology. Metrics in this case study are the speed-up achieved by the considered MPSoCs compared to the JPEG application executed on a single-processor system and the efficiency of these MPSoCs, where

$$efficiency = \frac{speed - up}{\#\ processing\ components}.$$

Based on the performance results obtained in the previous two case studies, we realized that given the JPEG application and the components in the IP components library, by exploiting task-level parallelism only, the maximum attainable speed-up is around 5x. Therefore, in order to achieve higher speed-up, we need to consider data parallelism as well. Data parallelism means that several identical tasks (e.g., DCT) process different data. Unfortunately, the JPEG algorithm does not allow multiple VLE tasks to work in parallel which becomes the potential bottleneck of the application when considering data parallelism. Therefore, the only way to achieve higher performance is to split the input image in tiles and each tile to be processed independently. The JPEG KPN for a single tile can exploit task-level parallelism by pipelining tasks as well as data-level parallelism by performing multiple DCT and Q tasks in parallel. This requires a modification (transformation) of the initial sequential program, which we performed manually before using the PNGEN tool to generate the corresponding KPN. By processing the input image in tiles, the performance can increase linearly with the number of tiles processed simultaneously, which means also, with the number of processing components in an MPSoCṪhe latter is limited by the available resources in the target (FPGA) technology which is considered in the high-level design space exploration step in DAEDALUS performed by SESAME.

4.4.1 Simulation-level DSE

The design space considered in the following DSE experiments is determined by the target MPSoC implementations and it is currently constrained by:

1. **The amount of the available memory.** In order to achieve high performance, in our MPSoCs we use on-chip memory for processors' program and data segments, including buffers for inter-processor data communication. We do not consider using external (off-chip) memory because of its large latency compared to the on-chip memory. Moreover, usually there is a limited number of available external memory banks which requires the external memory to be shared between several processors. This fact significantly limits the overall MPSoC performance. We use external memories only for communication with the environment (source of data and destination of the generated results). An average size FPGA nowadays has around 200–300KB of on-chip memory distributed on several blocks. In our experiments, we use a Xilinx VirtexII-6000 FPGA, and therefore, we constrain the total MPSoC memory to be up to 288KB, being the amount of on-chip memory of this FPGA.

2. **The type of the processing components.** The MPSoCs are built of components from our library. The library is under development and currently contains two programmable processors: $PowerPC$ (IBM) and $MicroBlaze$ (Xilinx). In addition, the library contains several dedicated HW IP cores. However, for the JPEG encoder we can use only one, i.e., the Discrete Cosine Transform (DCT) IP. For the considered FPGA, $PowerPC$ processors can not be used. Therefore, the processing components of the MPSoCs are limited to $MicroBlaze$ processors and DCT HW IP cores only.

In this experiment, we assume that the image that needs to be compressed is tiled, and that multiple JPEG encoders can process these tiles in parallel. This is illustrated in Figure 4.3, which also shows the corresponding KPN of the JPEG encoder application for one tile. The number of KPN processes and the constraints discussed above result in a design space consisting of a huge number of design points which makes the approach applied in the first case study, i.e., evaluating the whole design space, infeasible in reasonable time. Therefore, in this experiment we consider the subset of the design space defined by the mapping rules presented in Chapter 3. For example, we do not consider design points in which process V_{IN} is merged with any Q, VLE, or V_{OUT} processes, and design points in which any DCT process is merged with VLE or V_{OUT} processes. In addition, we found that increasing data-parallelism beyond 4 parallel DCT-Q streams (see Figure 4.3) will not improve performance as the VLE becomes the bottleneck. Note that the proposed mapping rules do not consider any physical constraints, e.g., the amount of the available memory. This was taken into account in the performed SESAME-based exploration, in which we also varied the type of processors in the MPSoC instances: All KPN tasks to be executed on a $MicroBlaze$, while for the DCT, Q and VLE tasks we also assessed dedicated HW IP implementations. Evidently, the simulation-level DSE also explores *non-implementable* design instances. That is, design instances that cannot be further implemented by ESPAM since these instances use HW IP components that are not (yet) available in the library of RTL IP components.

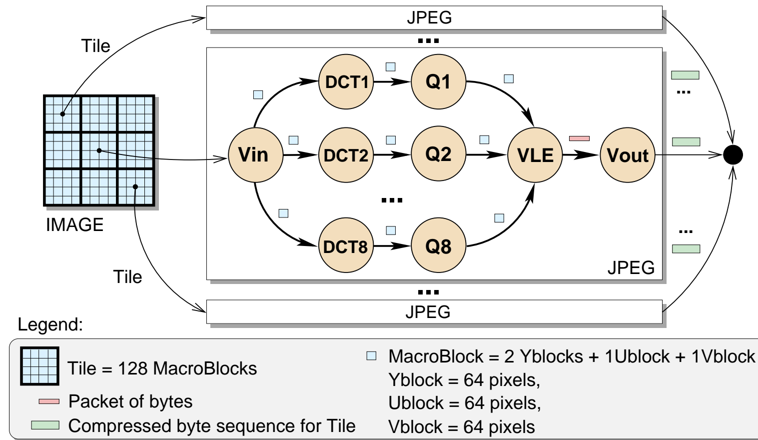


Figure 4.3: The JPEG application KPN.

Figure 4.4 shows a scatter plot with the performance results of the explored design instances plotted against the expected memory utilization of each design instance once implemented on the targeted FPGA. The memory utilization of the design instances was estimated using a simple accumulative model that has been calibrated with numbers from implementation-level experiments (see Section 4.2). Since the memory utilization of all design points in Figure 4.4 is below 288KB, they will all fit memory-wise on the targeted FPGA. But, as will be shown further on, the real MPSoC will consist of a combination of multiple of these (single JPEG encoder) design instances working in parallel, which, of course, may not necessarily fit on the FPGA. The points in Figure 4.4 can be classified as three types of design instances:

1. Design instances that are *implementable* (do not use HW IPs for Q and VLE), but are not part of the Pareto front;

2. *Implementable* design instances that are part of the Pareto front;

3. Design instances that are *non-implementable*, i.e., contain HW IP components for the implementation of Q or VLE.

Moreover, the homogeneous design instances, i.e., the platforms only using $MicroBlaze$ processors, are tagged with circles.

A number of observations can be made from Figure 4.4. The (implementable) Pareto optimal solutions are all heterogeneous designs, containing one or two DCT HW IP components. Two of these Pareto optimal solutions are shown in Figures 4.6(b) and 4.6(d). Clearly, the (non-implementable) design instance in which the DCT, Q and VLE tasks are all implemented by a HW IP core is the fastest and most memory efficient. When considering the homogeneous design points in Figure 4.4, another observation can be made: The design points with a memory utilization less than 75KB are the designs that exploit task-level parallelism only. The speed-up due to task-level parallelism levels off at a performance of around 18

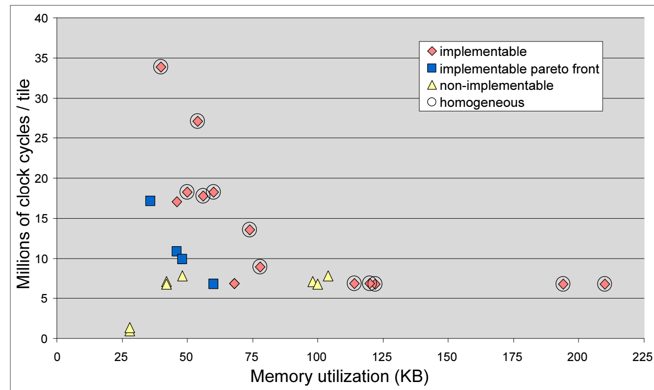


Figure 4.4: DSE for performance/memory utilization trade-offs.

Mcycles/tile. But, when data-parallelism is also exploited, the speed-up levels off at around 7 Mcycles/tile at the cost of increased memory utilization.

As mentioned and as will be illustrated in Section 4.4.2, the design points in Figure 4.4 are the building blocks for the entire system, in which multiple of these instances, possibly in a hybrid constellation, are encoding image tiles in parallel. For example, the most optimal, but (currently) non-implementable, system would consist of multiple JPEG encoders with HW IPs for the DCT, Q and VLE tasks. The projected performance of this system, considering the targeted FPGA, equals to an execution time of about 6 Mcycles to encode an image with a 1 Mpixel resolution. For implementable solutions, the Pareto optimal design instances from Figure 4.4 are obvious candidate building blocks for the MPSoC. Four of these building blocks are depicted in Figure 4.6.

Figure 4.5 shows the estimated maximum performance – in terms of speed-up over a single JPEG encoder executed on one $MicroBlaze$ – for different JPEG compression MPSoCs realized with a combination of implementable design instances from Figure 4.4. The x-axis indicates the number of processing cores (either $MicroBlaze$ or HW IP) in the MPSoC, and the y-axis shows the estimated speed-up for the optimal combination of design instances for a specific number of cores in the MPSoC which still adheres to the memory constraints of the targeted FPGA. Furthermore, a distinction is made between homogeneous systems (i.e., only $MicroBlazes$) and heterogeneous systems (i.e., containing also DCT HW IP components). For example, the optimal homogeneous 4-core system is a combination of four sequential JPEG design instances, i.e. a system containing four $MicroBlazes$ that all perform a full JPEG on different image tiles in parallel. In Section 4.4.2, more examples of, sometimes hybrid, combinations of design instances will be discussed.

Essentially, Figure 4.5 provides a projection of the feasible system performance, given the constraints of the targeted FPGA. For homogeneous solutions, the high-level simulations predict that a speed-up of around 10x to 12x is attainable. The memory utilization model indicates that scaling the homogeneous system beyond 24 cores is not possible because of the memory constraints. For heterogeneous systems, the memory model indicates that the system can be scaled to 30 cores since the HW IP components only use a fraction of the memory

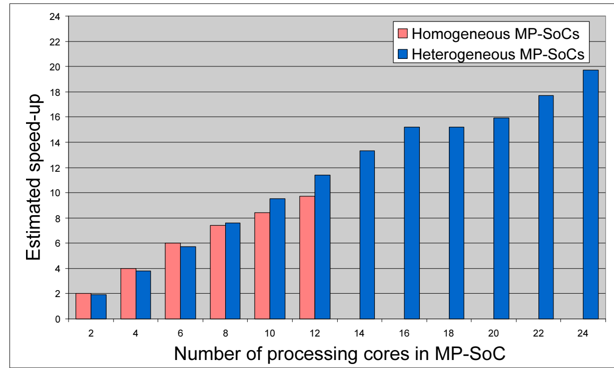


Figure 4.5: Estimated speed-ups.

used by a $MicroBlaze$. Here, the predictions show that a speed-up of around 20x to 22x is feasible. The results from the simulation-level DSE, as displayed in Figures 4.4 and 4.5, are used in the next section for steering the implementation-level DSE. These implementation-level experiments will also provide a validation of the simulation-based predictions.

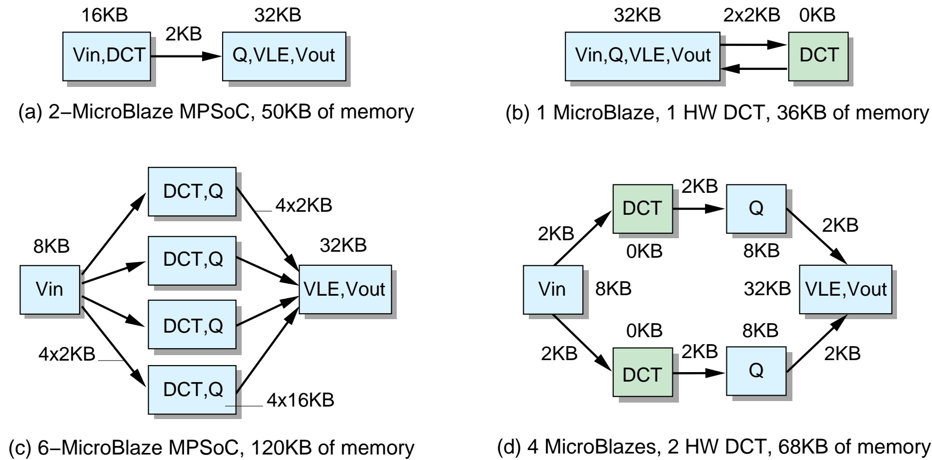
Performing DSE at a high-level of abstraction by simulation can not deliver 100% accurate performance/cost numbers but it can rapidly narrow down the design space to a few promising design points. Thus, we perform 100% accurate exploration in the narrowed design space by generating real MPSoC prototypes and we measure the actual performance/cost in order to select the optimal MPSoC designs given a set of physical implementation constraints. Below, we present our implementation-level DSE results for MPSoCs implemented on a Xilinx FPGA.

4.4.2 Implementation-level DSE

Due to the aforementioned implementation-level constraints, some of the best design points found by the simulation-level DSE (see Figure 4.4) could not be implemented, e.g., all application tasks to be realized as HW IPs. Therefore, we considered the implementable design instances depicted in Figure 4.5. From them, we selected only the instances that have $efficiency$ above 0.8. This selection resulted in implementations of homogeneous MPSoCs consisting of up to 13 $MicroBlaze$ processors and heterogeneous MPSoCs with up to 24 cores. Evidently, better performance is delivered by the heterogeneous systems, however, the homogeneous systems add more flexibility when, for example, trade-off between performance and cost is needed.

Homogeneous systems

The implementation results for the homogeneous MPSoCs are depicted in Figure 4.7. The x-axis represents the number of $MicroBlaze$ processors in an MPSoC and the y-axis depicts the number of clock cycles (in millions) to compress one image consisting of 32 tiles of



(a) 2–MicroBlaze MPSoC, 50KB of memory

(b) 1 MicroBlaze, 1 HW DCT, 36KB of memory

(c) 6–MicroBlaze MPSoC, 120KB of memory

(d) 4 MicroBlazes, 2 HW DCT, 68KB of memory

Figure 4.6: Alternative Pareto front design instances to process one tile.

128x128 pixels each. Above the bars, we indicated the achieved speed-up of the particular MPSoC compared to a single $MicroBlaze$ system (the leftmost bar). At the top of the figure, we present the amount of memory utilized by each MPSoC.

As mentioned before, our JPEG encoding MPSoCs process the input image in tiles. We started with a single $MicroBlaze$ system (processing all the tiles) and then we increased the number of processors by selecting the best points found by the simulation-level DSE. These points exploit data-level parallelism, i.e., several $MicroBlazes$ process different tiles. This is the most efficient way to increase performance because if we increase the number of processors that process independent tiles, then the speed-up increases linearly with the number of processors. To execute the JPEG application, a single $MicroBlaze$ processor system requires 40KB of memory. Therefore, we were able to implement systems with up to 7 processors on the considered FPGA (7x40=280KB), achieving speed-ups (see the first 7 bars in the Figure 4.7) up to 7x.

By exploiting only data-level parallelism, with 7 $MicroBlaze$s processing 7 tiles in parallel, we reached the limit of the available memory in our FPGA. Then, the question is whether there are design points that give even better performance (with more processors) and still match the resource constraints. We were able to increase the number of processors to more than 7 by selecting points that exploit both data-level parallelism between tiles and also task-level parallelism within the tiles. For this purpose, we used the 2-$MicroBlaze$ architecture depicted in Figure 4.6(a), where the Vin and all DCT processes (see Figure 4.3) are executed on the first processor and the remaining processes on the second one. By exploiting task-level parallelism, reaching linear speed-up is not possible due to data dependencies between the tasks. However, the total memory requirement of the system is reduced because the application tasks are distributed, and each processor executes a portion of the initial application. As a result, larger systems can be built, and consequently, larger overall speed-up can be achieved. For instance, a single-processor system needs 40K to execute the JPEG encoder, while a two-processor system – exploiting task-level parallelism – needs a total amount of 50KB for the

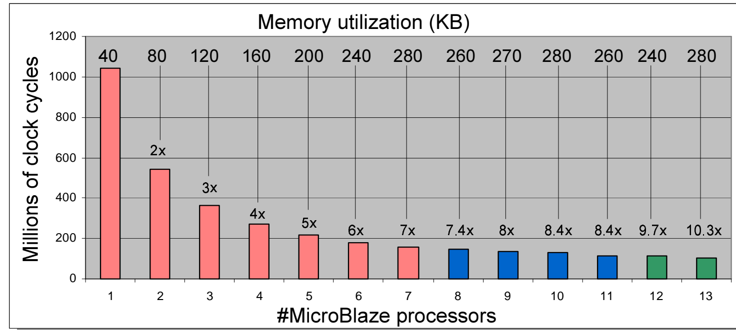


Figure 4.7: Performance results: homogeneous MPSoCs.

same application, on average 25KB per processor. Thus, by exploiting the reduced memory requirement, we were able to increase the number of processors and to implement systems with up to 11 $MicroBlaze$ processors. The selected points are actually combinations of a 1-$MicroBlaze$ system per tile and a 2-$MicroBlaze$ system per tile. The MPSoCs with 8 to 11 $MicroBlazes$ process 6 tiles in parallel. The achieved speed-ups are not linear, e.g., 7.4x for an 8-processor MPSoC and 8.4x for an 11-processor MPSoC, but they are higher than the speed-up of the 7-processor system.

In order to implement even larger systems, we exploited data-level parallelism between the tiles and data- and task-level parallelism within the tiles. We selected and implemented points representing 12 and 13 processor systems with total memory requirements that match our physical constraints. The 12-processor system processes 2 tiles in parallel where each tile is processed by a 6-$MicroBlaze$ architecture depicted in Figure 4.6(c). This architecture requires 120KB of memory. The 13-processor MPSoC utilizes an additional $MicroBlaze$ processor (additional 40KB), therefore, increasing the number of tiles processed in parallel to 3. The results are shown at the right part (the two rightmost bars) of Figure 4.7. The achieved speed-up of 12- and 13-$MicroBlaze$ systems is 9.7x and 10.3x respectively, compared to a 1-$MicroBlaze$ system.

Heterogeneous systems

The implementation results for the heterogeneous MPSoCs are depicted in Figure 4.8. The notation is the same as in Figure 4.7 with the only difference that the x-axis of Figure 4.8 indicates how many of the used cores are $MicroBlaze$ processors and how many DCT HW IPs. By exploiting data- and task-level parallelism, we implemented heterogeneous MPSoCs consisting of up to 24 cores. As a reference number to estimate the speed-up of each MPSoC, we again used the number of clock cycles of the 1-$MicroBlaze$ system (see the leftmost bar in Figure 4.7). We started with a 2-core system consisting of 1 $MicroBlaze$ and 1 DCT IP as depicted in Figure 4.6(b). It exploits task-level parallelism within a tile, which affects the achieved speed-up. Although the DCT IP core is very efficient and fast in terms of performance, the overall speed-up is only 1.9x (see the leftmost bar in Figure 4.8), which actually is in line with Amdahl's law. Similarly to the experiments with the homogeneous systems,

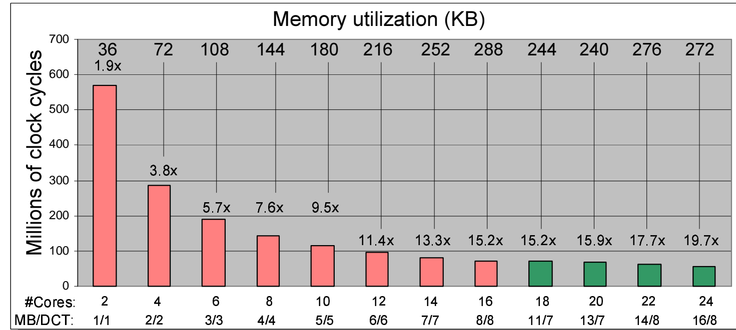


Figure 4.8: Performance results: heterogeneous MPSoCs.

we continued with points that exploit data-level parallelism between the tiles, increasing the number of tiles processed in parallel. The 2-core MPSoC requires 36KB of memory, i.e., the DCT IP core reduces the $MicroBlaze$ memory requirement to 32KB but with an additional 4KB used for communication buffers, see Figure 4.6(b). Therefore, with 288KB of memory, we were able to implement systems with up to 8 $MicroBlaze$s and 8 DCT IPs (16 cores, processing 8 tiles in parallel). The achieved speed-up linearly scales from 1.9x for 2 cores to 15.2x for 16 cores as illustrated in Figure 4.8.

Like in the previous experiment, with the given constraints larger MPSoCs can be implemented (and higher speed-ups can be achieved respectively) by exploiting data-level parallelism between the tiles and data and task parallelism within the tiles. The most efficient heterogeneous MPSoC instance found by the simulation-level DSE to exploit data- and task-level parallelism within a tile is depicted in Figure 4.6(d). It consists of 4 $MicroBlaze$ processors and 2 DCT IP cores. The total memory requirement of this system is 68KB. We selected and implemented the 18-, 20-, 22-, and 24-core systems in Figure 4.5 which actually are combinations of 2-cores per tile (2-CPT) and 6-cores per tile (6-CPT) platform instances illustrated in Figure 4.6(b) and Figure 4.6(d) respectively. The 18-core system consists of 11 $MicroBlazes$ and 7 DCT IPs. It processes 5 tiles in parallel: 3 tiles are processed by three 2-CPT instances and 2 tiles are processed by two 6-CPT instances. The speed-up of this MPSoC is 15.2x. The 20-core system processes 4 tiles in parallel: 1 tile is processed by one 2-CPT instance and 3 tiles are processed by three 6-CPT instances. In total, 13 $MicroBlazes$ and 7 DCT IPs achieve a speed-up of 15.9x. The speed-up of the 22-core system is 17.7x. This MPSoC consists of 14 $MicroBlazes$ and 8 DCT IPs that process 5 tiles in parallel: 2 tiles are processed by two 2-CPT instances and 3 tiles are processed by three 6-CPT instances. The 24-core MPSoC, consisting of 16 $Microblazes$ and 8 DCT IPs, processes 4 tiles in parallel utilizing four 6-CPT instances. The achieved speed-up by this system is 19.7x compared to a 1-$MicroBlaze$ system.

4.4.3 Conclusions

In this section, we presented a case study demonstrating the efficiency of the system design methods and techniques proposed in this dissertation for automated multiprocessor sys-

tem synthesis, implementation, and programming. The level of automation achieved with DAEDALUS significantly reduces the design time starting from system-level specifications and going down to complete implementation. All presented DSE experiments and the real implementations of 25 MPSoCs on FPGA were performed in a short amount of time, 5 days in total. Around 70% of this time was taken by the low-level commercial synthesis and place-and-route FPGA tools. The obtained results confirm that the high-level MPSoC models used in SESAME are capable of accurately predicting the overall system performance. By exploiting the data and task parallelism in the JPEG application, DAEDALUS can deliver scalable MPSoC solutions in terms of performance and cost. We were able to achieve a performance speed-up of up to 20x compared to a single processor system. The MPSoC performance was limited by the available on-chip FPGA memory resources and the available IP cores in DAEDALUS RTL library. To achieve higher performance speed-up, the RTL library has to be extended with more dedicated HW IP cores.

Chapter 5

Summary and Conclusions

In this dissertation, we have presented a methodology implemented in the DAEDALUS tool-flow (see Section 1.2), for automated design, programming, and implementation of MPSoCs starting at a high level of abstraction. The methods and techniques in DAEDALUS bridge the gap between the system level and the register-transfer level of design abstraction introduced in Section 1.1, which was the main objective and it is the main contribution of this dissertation. With DAEDALUS, this (implementation) gap is closed in a particular way because we target only embedded multiprocessor systems that execute data-streaming applications in the domain of multimedia and signal processing. DAEDALUS offers a fully integrated tool-flow for very fast exploration and implementation of alternative MPSoCs, where design space exploration (DSE), system-level synthesis, application mapping, and system prototyping of MPSoCs are highly automated. The main idea is starting from a functional specification of an application and a description of an MPSoC at system level, to refine and translate them to lower RTL descriptions in a systematic and automated way. This is achieved by applying a model-driven, platform-based approach, where

- We use a parallel model of computation, namely the Kahn Process Network (KPN) MoC [6], to represent an application as a set of (concurrent) application tasks. Having an application in a parallel form allows for mapping it onto the processing components of an MPSoC which can be programmable (ISA) processors as well as non-programmable, dedicated IP cores. We proposed techniques for programming the ISA processors in an automated way based on the KPN MoC. Moreover, in case of non-programmable processing components, we proposed an approach for automated integration of predefined (third-party) dedicated IP cores;
- By carefully exploiting and efficiently implementing the simple communication and synchronization features of a KPN, we have identified a platform model which captures very well the operational semantics of the KPN MoC. This allows system-level descriptions of platform instances to be refined and translated to detailed RTL descriptions in an automated way. The good match between the KPN MoC and our platform model results in efficient implementations when KPNs are executed on such platforms;

- We use a mapping model to express the relation between the processes and the communication channels in the application (KPN) and the processing and memory components of the platform. In the proposed approach, the communication mapping is implicit, i.e, ESPAM analyses the mapping of processes to processing components and automatically finds an optimal mapping of communication FIFO channels of the application onto memory components of the platform.

DAEDALUS provides new embedded system design/programming paradigm. It includes platform specification in a way that the designer does not have to deal with platform specific details. Yet, DAEDALUS gives system designers the opportunity to control platform specific issues in order to enforce performance and cost metrics. DAEDALUS uses a data-flow (KPN) MoC to specify the application in a parallel form. This allows the exposed parallelism to be exploited in different ways as processes can be mapped on different processing cores and the cores can be arranged in different communication topologies. Designing an MPSoC with DAEDALUS includes essentially an MPSoC instance generation (see Chapter 2) and mapping (assignment) of application tasks to processing components of that instance, where the parallel KPN representation of an application is automatically derived from a sequential program by the PNGEN tool. More specifically, a system designer can specify a multiprocessor platform instance and a mapping specification at a high level of abstraction in a short amount of time, say a few minutes. Then, ESPAM refines these specifications to a real implementation, i.e., it generates a synthesizable (RTL) HW description of the MPSoC and it generates SW code for each processor, in an automated way. This reduces the design and programming time from months to hours. As a consequence, an accurate exploration of the performance of alternative multiprocessor platform instances becomes feasible at implementation level in a few hours.

In the design process, different number and type of platform components can be used to construct an MPSoC instance as well as different mappings can be considered. This leads (usually) to large and complex design space which represents a large number of different MPSoC implementation possibilities. Then, the key issue is to reduce the number of different implementation possibilities to a subset, consisting of the most promising design points from which, based on certain criteria, the designer can choose the best one. Traversing the whole design space or applying general techniques for design space exploration may not be always feasible (in reasonable time) for large and complex design space. This motivated us, and in Chapter 3 of this dissertation, we proposed techniques to narrow down the design space in a systematic way by exploiting the properties of the application and the platform models we use. More precisely, we defined (mapping) rules for mapping of application tasks to processing components in the target MPSoCs such that less number of processing components are used without compromising the achieved system performance. The proposed approach can be used to complement the techniques in the SESAME tool for reducing the design space that need to be considered in the design space exploration process in the DAEDALUS design flow.

In this dissertation, we have presented three case studies in order to validate and evaluate the proposed design methodology. That is, we have used the case studies to demonstrate the potential and the efficiency of our methods and techniques for automated MPSoC design in terms of overall design time, achieved performance, and HW utilization. Also, we have

commented on the accuracy of the results obtained by performing high-level system simulations (during the DSE process) compared to real implementation numbers. The case studies have clearly shown that our research work presented in this dissertation can be applied successfully on real-life industrially relevant applications. The case studies, the corresponding experiments, and the obtained results have been reported in Chapter 4.

With the first case study, we have illustrated a complete design flow with DAEDALUS for a JPEG encoder application, starting from a sequential program, performing system-level DSE with SESAME, synthesizing design instances with ESPAM, and prototyping them by using commercial synthesis and compiler tools. The second case study has been used to illustrate the method and techniques for automated integration of dedicated hardwired IP cores into heterogeneous MPSoCs where both programmable processors and dedicated IPs are used as processing components. The proposed IP core integration approach has been applied on three real-life applications, i.e., a Sobel edge detection, a Discrete Wavelet Transform, and a JPEG encoder. The purpose of the last case study was to push DAEDALUS "to the limit" in order to check how large and complex systems can be designed using the proposed methodology and considering the constraints imposed by the FPGA technology we currently use for prototyping.

Based on the experience we have gained by conducting the three case studies and the results we have obtained, we draw the following conclusions:

- The level of automation achieved with DAEDALUS significantly reduces the design time starting from system-level specifications and going down to complete implementation. That is, starting from a sequential application and going down to complete implementation, e.g., to an MPSoC prototyped on an FPGA, is only a matter of hours;

- The high level of abstraction of the input specifications allows a system designer easily to construct many alternative platforms instances which are automatically implemented by ESPAM. This, and the reduced design time, enables fast exploration of design points at implementation level with 100% accuracy during the early stages of design;

- The proposed approach of connecting processing components through communication controllers and communication memories is efficient in terms of HW resource utilization and performance speed-up;

- The proposed techniques for automated integration of dedicated IP cores by generating IP Modules (wrappers around the IP cores) lead to efficient integration in terms of achieved performance and HW resource utilization;

- The obtained results by using DAEDALUS are as good as the components in the IP component library. Recall that in the third case study, we could not implement the best design points found by the design exploration process because the required dedicated IP cores were not available in the DAEDALUS IP component library;

- The devised mapping techniques can effectively prune the design space by preserving the MPSoC instances that deliver highest performance.

The name DAEDALUS

Finally, we conclude this dissertation with the story of Daedalus from the ancient Greek mythology, which motivated us for the name of the proposed system design flow. DAEDALUS means cunning worker (in Latin) and he was an innovator in many arts. The myth goes that Daedalus built a labyrinth for King Minos, but afterward lost the favor of the king, and was shut up in a tower on island of Crete. Daedalus contrived to make his escape, but could not leave Crete by sea because the king kept strict watch on all the vessels leaving the island. "Minos may control the land and sea," – said Daedalus – "but not the regions of the air. I will try that way." Daedalus set to work to fabricate wings for himself and his young son Icarus. He tied feathers together, from smallest to largest as the larger ones he secured with thread and the smaller ones with wax. When both were prepared for flight, Daedalus warned Icarus not to fly too high, because the heat of the sun would melt the wax, nor too low because the sea foam would make the wings wet and they would no longer fly. "Keep near me and you will be safe." – said Daedalus to Icarus, and the father and son flew away. However Icarus, exulting in his ability to fly, began to leave the guidance of his father and rose upward into the air. The blazing sun softened the wax which held the feathers together, they came off and Icarus fell into the sea.

Daedalus and Icarus,
by Charles Paul Landon, 1799.

The analogy: with the DAEDALUS system design flow, we propose new, disruptive technology which is based on the following assumptions:

- It is meant for data-flow (streaming) applications;
- Applications specified in the form of static affine nested loop programs;
- Targets distributed memory MPSoC implementations, utilizing communication memories and communication controllers;
- The results are as good as the components in the IP library.

The DAEDALUS design flow makes system-level design "take-off". However, the assumptions need to be well understood and respected in order to avoid "falling into the sea"!

Bibliography

[1] Gordon E. Moore. Multidimentional synchronous dataflow. In *Eletronics*, volume 38, April 1965.

[2] G. Martin. Overview of the MPSoC Design Challenge. In *Proc. DAC*, July 2006.

[3] 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 *In Proc. 45th ACM/IEEE Int. Design Automation Conference (DAC'08)*, pages 754–579, Anaheim, USA, June 8-13 2008.

[4] Daedalus system-level design, http://daedalus.liacs.nl/.

[5] Alberto Sangiovanni-Vincentelli and Grant Martin. A Vision for Embedded Systems: Platform-Based Design and Software Methodology. *IEEE Design and Test of Computers*, 18(6):23–33, 2001.

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

[7] S. Verdoolaege, H. Nikolov, and T. Stefanov. pn: A Tool for Improved Derivation of Process Networks. *EURASIP Journal on Embedded Systems*, 2007:Article ID 75947, 13 pages, 2007. doi:10.1155/2007/75947.

[8] A. Pimentel et. al. A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels. *IEEE Transactions on Computers*, 55(2), 2006.

[9] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. In *IEEE Trans. on CAD of Integrated Circuits and Systems*, volume 27, March 2008.

[10] Xilinx, Inc. Xilinx Platform Studio and the Embedded Development Kit, EDK version 8.1i edition. www.xilinx.com/ise/embedded_design_prod/platform_studio.htm.

[11] Synfora Inc. PICO Technology. http://www.synfora.com/.

[12] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, and M. Sgroi. Benefits and challenges for platform-based design. In *Proc. DAC*, pages 409 – 414, June 2004.

[13] Edsger W. Dijkstra. *Selected writings on Computing: A Personal Perspective*. Springer-Verlag, New York, USA, 1982. ISBN 0-387-90652-5.

[14] Kurt Keutzer, Sharad Malik, Richard Newton, Jan Rabaey, and Alberto Sangiovanni-Vincentelli. System-Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 19(12):1523–1543, 2000.

[15] Edward Lee and Alberto Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[16] G. Kahn and D.B. MacQueen. Coroutines and Networks of Paralel Processes. In *Proc. IFIP Congress 77*, pages 993 – 998, Toronto, Canada, August 1977. North-Holland Publishing Co.

[17] T. Stefanov et al. System Design using Kahn Process Networks: The Compaan/Laura Approach. In *Proc. DATE*, pages 340–345, February 2004.

[18] A. Nieuwland et al. *C-HEAP: A Heterogeneous Multi-processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems*. Kluwer Publishers, 2002.

[19] Erwin de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proc. ISSS*, pages 68–73, October 2002.

[20] K. Goossens et. al. Guaranteeing the Quality Of Services in Networks On Chip. In *Networks on Chip*, pages 61–82. Kluwer Publishers, 2003.

[21] B. Dwivedi et. al. Automatic Synthesis of System on Chip Multiprocessor Architectures for Process networks. In *Proc. CODES+ISSS*, September 2004.

[22] B. Kienhuis et al. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proc. CODES*, May 2000.

[23] T. Stefanov and E. Deprettere. Deriving Process Networks from Weakly Dynamic Applications in System-Level Design. In *Proc. CODES+ISSS*, pages 90–96, October 2003.

[24] A. Turjan et al. Translating Affine Nested-loop Programs to Process Networks. In *Proc. CASES*, September 2004.

[25] T. Parks. Bounded Scheduling of Process Networks. Technical report, University of California, EECS Dept., Berkeley, CA, 1995. PhD Thesis.

[26] J. Buck and E. Lee. Scheduling Dynamic Data Flow Graphs with Bounded Memory using the Token Flow Model. In *Proc. IEEE Conf. on Acoustics, Speech, and Signal Processing*, pages 429–432, April 1993.

[27] M. Geilen and T. Basten. *Requirements on the Execution of Kahn Process Networks*. LNCS, Springer, vol. 2618/2003, pp. 319-334, 2003.

[28] E. de Kock et al. YAPI: Application modeling for signal processing systems. In *Proc. DAC*, pages 402–405, June 2000.

[29] E.A. Lee et al. Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java. Technical report, University of California at Berkeley, 1999. UCB/ERL M99/40.

[30] Andy Pimentel, Paul Lieverse, Pieter van der Wolf, and Ed F. Deprettere. Exploring Embedded-Systems Architectures with Artemis. *IEEE Computer*, 34(11):57–63, November 2001.

[31] Erwin de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proc. 15th Int. Symposium on System Synthesis (ISSS'2002)*, pages 68–73, Kyoto, Japan, October 2-4 2002.

[32] Pieter van der Wolf, Paul Lieverse, Mudit Goel, David La Hei, and Kees Vissers. An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology. In *Proc. 7th Int. Workshop on Hardware/Software Codesign (CODES'99)*, Rome, Italy, May 3-5 1999.

[33] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere. System Level Design with SPADE: an M-JPEG Case Study. In *Proc. ICCAD*, pages 31–38, November 2001.

[34] Paul Lieverse, Pieter van der Wolf, Kees Vissers, and Ed Deprettere. A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems. *Int. Journal of VLSI Signal Processing for Signal, Image and Video Technology*, 29(3):197–207, 2001.

[35] Martijn J. Rutten, Jos T.J. van Eijndhoven, and Evert-Jan D. Pol. Design of Multi-Tasking Coprocessor Control for Eclipse. In *Proc. 10th Int. Symposium on Hardware/Software Codesign (CODES'02)*, pages 139–144, Estes Park, Colorado, USA, May 6-8 2002.

[36] Cagkan Erbas, Selin C. Erbas, and Andy D. Pimentel. A Multiobjective Optimization Model for Exploring Multiprocessor Mappings of Process Networks. In *Proc. IEEE-ACM-IFIP Int. Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'03)*, pages 182–187, Newport Beach, California, USA, October 1-3 2003.

[37] Edwin Rijpkema, Ed F. Deprettere, and Bart Kienhuis. Deriving Process Networks from Nested Loop Algorithms. *Parallel Processing Letters*, 10(2):165–176, 2000.

[38] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubuhr, A. Deyhle, A. Hadert, and J. Teich. A SystemC-Based Design Methodology for Digital Signal Processing Systems. *EURASIP Journal on Embedded Systems*, 2007:Article ID 47580, 22 pages, 2007. doi:10.1155/2007/47580.

[39] J. Falk, C. Haubelt, and J. Teich. Efficient representation and simulation of model-based designs in SystemC. In *Proc. of the International Forum on Specification & Design Languages (FDL 06)*, pages 129–134, Darmstadt, Germany, September 2006.

[40] M.J. Rutten et al. A Heterogeneous Multiprocessor Architecture for Flexible Media Processing. *IEEE Design & Test of Computers*, 19(4), 2002.

[41] Akash Kumar, Shakith Fernando, Yajun Ha, Bart Mesman, and Henk Corporaal. Multiprocessor systems synthesis for multiple use-cases of multiple applications on fpga. *ACM Trans. Des. Autom. Electron. Syst.*, 13(3):1–27, 2008.

[42] The Xilinx's Microblaze Soft Core Processor. http://www.xilinx.com/products/design_resources/proc_central/microblaze_arc.htm.

[43] IBM PowerPC Wite Paper. http://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_405_ Embedded_Cores.

[44] A. Jerraya, A. Bouchhima, and F. Ptrot. Programming Models and HW-SW Interfaces Abstraction for MultiProcessor SoC. In *Proc. DAC*, July 2006.

[45] D. Lyonnard et al. Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip. In *Proc. DAC*, June 2001.

[46] L. Gauthier, S. Yoo, and A. Jerraya. Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software. *IEEE Trans. on CAD*, 20, November 2001.

[47] F. Balarin et al. *Hardware-Software Co-design of Embedded Systems – The POLIS approach*. Kluwer Academic Publ., 1997.

[48] A. Gerstlauer and D. Gajski. System-level abstraction semantics. In *Proc. ISSS*, pages 231–236, October 2002.

[49] P. Paulin et al. Parallel Programming Models for a Multiprocessor SoC Platform Applied to Networking and Multimedia. *IEEE Trans. on VLSI Systems*, 14(7), July 2006.

[50] Celoxica Web Page:. http://www.celoxica.com.

[51] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, New Jersey, 1985.

[52] J. Zhu, R. Domer, and D. Gajski. Syntax and Semantics of the SpecC Language. In *Proc. of the Synthesis and System Integration of Mixed Technologies*, Osaka, Japan, December 1997.

[53] H. Yu, R. Domer, and D. Gajski. Embedded Software Generation from System Level Design Languages. In *Proc. ASPDAC*, pages 463–468, January 2004.

[54] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink. Design and Programming of Embeded Multiprocessors: an Interface-Centric Approach. In *Proc. CODES+ISSS*, September 2004.

[55] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer. An Automated Exploration Framework for FPGA-based Soft Multiprocessor Systems. In *Proc. IEEE-ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, New Jersey, USA, September 19-21 2005.

[56] K. Ravindran, N. Satish, Y. Jin, and K. Keutzer. An FPGA-based Soft Multiprocessor System for IPv4 Packet Forwarding. In *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, Tampere, Finland, August 24-26 2005.

[57] Altera, Inc. Quartus II Handbook Volume 4: SOPC Builder, Dec 2005. www.altera.com/literature/quartus2/lit-qts-sopc.jsp.

[58] Multiprocessor Solutions with FPGAs. White paper, FPGA and Programmable Logic Journal, 2005, www.fpgajournal.com/whitepapers_2005/altera_20050224.htm.

[59] SPIRIT Web Page:. http://www.spiritconsortium.org.

[60] VSIA Web Page:. http://www.vsi.org.

[61] OCP Web Page:. http://www.ocpip.org.

[62] Algorithms, complexity analysis and VLSI architectures for MPEG-4 estimation. Kluver, 2004.

[63] S. Wuytack, F. Catthoor, L. Nachtergaele, and H. De Man. Power Exploration for Data Dominated Video Applications. In *In Proc. of the 1996 international symposium on low power electronics and design (ISLPED'96)*, pages 359–364, Piscataway, NJ, USA, 1996.

[64] D. Moolenaar, L. Nachtergaele, F. Catthoor, and H. De Man. System-level Power Exploration for MPEG-2 Decoder on Embedded Cores: A Systematic Approach. In *IEEE workshop on signal processing systems (SIPS'97)*, pages 395–404, Leicester, UK, 1997.

[65] H. Nikolov, T. Stefanov, and E. Deprettere. Efficient External Memory Interface for Multi-processor Platforms Realized on FPGA Chips. In *17th Int. Conference on Field Programmable Logic and Applications (FPL'07)*, Amsterdam, The Netherlands, August 27-29 2007.

[66] T. Stefanov. Converting Weakly Dynamic Programs to Equivalent Process Network Specifications. Leiden Institute of Advanced Computer Science (LIACS), Leiden University, The Netherlands, December, 2004.

[67] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. *Realizations of the Extended Linearization Model*. in Domain-Specific Embedded Multiprocessors (Chapter 9), Marcel Dekker, Inc., 2003.

[68] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A Technique to Determine Interprocess Communication in the Polyhedral Model. In *Proc. Int. Workshop on Compilers for Parallel Computers (CPC'03)*, Amsterdam, The Netherlands, January 8-10 2003.

[69] H. Nikolov, T. Stefanov, and E. Deprettere. Efficient Automated Synthesis, Programming, and Iplementation of Multi-processor Platforms on FPGA Chips. In *16th Int. Conference on Field Programmable Logic and Applications (FPL'06)*, pages 323–328, Madrid, Spain, August 28-30 2006.

[70] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[71] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 79–103, 1996.

[72] Paul Feautrier. Dataflow Analysis of Scalar and Array References. *Int. Journal of Parallel Programming*, 20(1):23–53, 1991.

[73] Paul Feautrier. Parametric Integer Programming. Operations Research, 22(3):243-268, 1988.

[74] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *Proc. FPL*, September 2003.

[75] H. Nikolov, T. Stefanov, and E. Deprettere. Automated Integration of Dedicated Hardwired IP Cores in Heterogeneous MPSoCs Designed with ESPAM. *EURASIP Journal on Embedded Systems*, 2008:Article ID 726096, 15 pages, 2008. doi:10.1155/2008/726096.

[76] Edward A. Lee and Thomas M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.

[77] H. Nikolov and E. Deprettere. Parameterized Stream-Based Functions Dataflow Model of Computation. In *6th Int. Workshop on Optimizations for DSP and Embedded Systems (ODES-6)*, Boston, USA, April 6 2008.

[78] B. Kienhuis and E. Deprettere. Modeling stream-based applications using the sbf model of computation. *Journal of VLSI Signal Processing*, 34(3), July 2003.

[79] Eric Cheung, Harry Hsieh, and Felice Balarin. Automatic Buffer Sizing for Rate-Constrained KPN Applications on Multiprocessor System-on-Chip. In *IEEE Workshop on High Level Design Validation and Test (HLVDT'07)*, pages 37–47, November7 9 2007.

[80] D. Kearney and N. Bergmann. Performance evaluation of asynchronous logic pipelines with data dependant processing delays. In *Proc. 2nd Working Conference on Asynchronous Design Methodologies (ASYNC'95)*, May 1995.

[81] 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. Technical report, Laboratoire Image et Calcul Parallle Scientifique, 1997. Rapport Technique RR 97-05, URL: http://icps.u-strasbg.fr/PolyLib.

[82] S. Verdoolaege et al. Multi-dimensional incremental loop fusion for data locality. In *Proc. ASAP*, pages 17–27, June 2003.

[83] S. Verdoolaege et al. Analytical computation of Ehrhart polynomials: Enabling more compiler analyses and optimizations. In *Proc. CASES*, pages 248–258, September 2004.

[84] P. Clauss et al. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. ICPS Research Report 06-04, Université Louis Pasteur, October 2006.

[85] Xilinx' DCT HW IP Core. www.xilinx.com/bvdocs/appnotes/xapp_610.zip.

Samenvatting

De dissertatie gaat over methoden en middelen voor het ontwerpen van multiprocessor systemen die zijn geïntegreerd in een enkele chip, 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 geheugen infrastructuur voor de uitwisseling van data.

De complexiteit van zulke ingebedde multiprocessor systemen heeft een niveau bereikt waarbij het noodzakelijk is geworden om het programmeren van deze systemen methodologiën te onderbouwen met het oog op een systematische en automatische uitvoering van deze belangrijke stap in het proces van systeemontwerp.

De dissertatie beschrijft een nieuwe ontwerpmethodologie, evenals de methoden en technieken voor de praktische uitvoering ervan. Deze zijn geïntegreerd in het ontwerppakket DAEDALUS dat het onderwerp is van het eerste hoofdstuk. Met dit pakket kan een ontwerp – inclusief de programmering en de implementatie – automatisch worden uitgevoerd, uitgaande van een abstracte specificatie. Met DAEDALUS wordt de afstand tussen abstracte en gedetailleerde specificatie automatisch overbrugd. De methoden en technieken in het DAEDALUS ontwerp pakket omvatten *exploratie* van de ontwerpruimte, *synthese* op systeem niveau, *afbeelden* van functionele specificatie modellen (toepassingen), op extrafunctionele implementatie modellen (architecturen), en *prototypereen* van het ontworpen multiprocessor chip-systeem.

De toepassingen worden gespecificeerd in termen van datastroom procesnetwerken, in het bijzonder *Kahn Proces Netwerken* die goed passen bij de beoogde datastroom applicaties. De multiprocessor architecturen worden gespecificeerd in termen van componenten die beschikbaar zijn in een bibliotheek van componenten voor de evaluatie en synchronisatie van functies, en de communicatie en opslag van data. De organisatie van de processornetwerk is zo gekozen dat procesnetwerken met de hoogst mogelijke prestatie kunnen worden doorg-

erekend. Abstracte specificaties van toepassing (procesnetwerk) en architectuur (processornetwerk), en de abstracte relatie tussen deze twee specificaties is alles wat nodig is om het chip-systeem – software en hardware – te implementeren; kennis van specifieke details is niet nodig.

Het tweede hoofdstuk behandelt de specificatie van een multiprocessor architectuur organisatie, de relatie tussen processen en processoren, en de afleiding van het procesnetwerk uitgaande van een specificatie van de toepassing in de vorm van een traditioneel sequentieel programma. Deze drie componenten vormen samen de abstracte chip-systeem specificatie. De DAEDALUS methode en de ESPAM techniek zorgen daarna voor een verfijning van deze specificatie tot een implementatiespecificatie op het niveau van synthetiseerbare *register transfer* code, en processor code. Met ESPAM kan de ontwerp- en programmeertijd van multiprocessor chip-systemen worden gereduceerd van maanden tot uren.

Een gegeven toepassing kan op vele manieren abstract worden gespecificeerd als een procesnetwerk, een processornetwerk, en de relatie tussen deze twee netwerken. Het is daarom noodzakelijk de verzameling van mogelijke specificaties te herleiden tot een paar specificaties die veelbelovend zijn in termen van gekozen optimalisatie criteria. Het derde hoofdstuk stelt methoden en technieken voor het achterhalen van deze *beste* specificaties op een systematische manier. Voor de voorbeelden die in de dissertatie zijn gegeven werd als criterium gekozen het minimaliseren van het aantal processoren zonder de prestatie van het resulterend chip-systeem te compromitteren.

Het laatste hoofdstuk geeft gevalsstudies aan de hand waarvan de methoden en technieken worden gevalideerd. Omdat het ontwerptraject in korte tijd kan worden doorlopen is het mogelijk een relatief groot aantal alternatieve fysieke implementaties te evalueren en de resultaten daarvan te vergelijken met deze die gedurende de abstracte exploratie van de ontwerpruimte zijn verkregen. Op die manier kan het exploratieproces verder gekalibreerd worden.

Curriculum Vitae

Hristo Nikolov was born on 8th of January, 1974 in Gabrovo, Bulgaria. In 1993, he received his high-school diploma at The High Technical School of Microprocessor Technology in Pravetz, Bulgaria. After the military service he did in the Bulgarian army between 1993 and 1996, Hristo Nikolov started his study in computer engineering at the Technical University of Sofia (TU Sofia), Bulgaria. In 2001, he successfully defended his M.Sc. thesis entitled ”IP core for real-time edge detection applications” and received his Dipl.Ing. and M.Sc. degrees in Computer Engineering from the Technical University of Sofia. During his M.Sc. study, Hristo Nikolov worked at Innovative MicroSystems, Ltd., Sofia, Bulgaria on designing application specific microprocessor IP cores targeting the FPGA technology. After obtaining his M.Sc. degree, up until 2004, Hristo Nikolov worked as a Research and Development Engineer at Fabless, Ltd., Sofia, where he had been involved in the development of a reconfigurable MicroSystems-on-Silicon In-Circuit Emulator based on FPGAs.

In 2004, Hristo Nikolov joined the Leiden Embedded Research Center (LERC) which is part of the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University where he was appointed as a research assistant (Ph.D. student). He was involved in the ARTEMISIA project which deals with Architecture, Programming, and Exploration of Network-on-Chip based Embedded System Platforms. As a member of the ARTEMISIA project, he conducted research in the context of modeling of stream-oriented media applications and mapping them onto parallel architectures. In particular, he worked on defining a multiprocessor platform for efficient execution of applications specified as Kahn process networks and on devising methods and techniques for systematic and automated multiprocessor system design, programming, and implementation. The research work culminated in the writing of this Ph.D. dissertation in 2009.

Since January 2006, Hristo Nikolov is an IEEE member. His primary research interests include video and image processing, embedded multiprocessor systems-on-chip (MPSoCs), system-level design of MPSoCs, design automation for MPSoCs, Hardware/Software co-design, computer architectures.