Tools and techniques for efficient system-level design space exploration

Thompson, M.

Publication date
2012
Document Version
Final published version

Tools and Techniques for Efficient System-Level Design Space Exploration

Mark Thompson

Tools and Techniques for Efficient System-Level Design Space Exploration

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor promoties ingestelde
commissie, in het openbaar te verdedigen in de Agnietenkapel
op woensdag 18 januari 2012, te 12:00 uur

door

Mark Thompson

geboren te Amsterdam

Promotor:	Prof. dr. C. R. Jesshope
Co-promotor:	Dr. A. D. Pimentel
Overige Leden:	Prof. dr. ir. Cees T. A. M. de Laat
	Prof. dr. Donatella Sciuto
	Prof. dr. ir. Henk Corporaal
	Dr. ir. Todor P. Stefanov

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The work described in this thesis has been carried out in the ASCI graduate school (ASCI dissertation series number 251) and was financially supported by Technologiestichting STW and Progress.

Typeset by LaTeX. Printed and bound by Ipskamp Drukkers Enschede.
ISBN: 978-94-6191-160-5

Acknowledgements

I have repeatedly heard people describe their PhD as an adventure or a journey. This seems to me an accurate description as I, like them, could not have imagined in advance what doing a PhD entails and how it may affect you in the long run. Like an adventure, it poses challenges of which the outcome is uncertain; like a journey it changes the way you view the world and your place in it. The PhD life has ups and downs, moments of joy and accomplishment when solving a problem, as well as moments of near-despair when the (typically ill-defined) goals seem so far out of reach. I wish I was eloquent enough to describe the experience more comprehensively, but instead I only want to mention one aspect of it. Looking back now, the thing that strikes me most about doing a PhD is the realization that none of it would be possible without the help and support of so many people. Here I would like to extend my gratitude to all my colleagues, friends and family who supported, stimulated and encouraged me.

I feel very lucky to have had Andy as my PhD supervisor. Apart from providing excellent support and guidance, I have greatly benefited from both his technical skills and professional knowledge, as well as from his inspirational attitude and optimistic character. In particular, he was always willing to make time for a discussion or a question: never by appointment, but simply by knocking on his office door. Constructive feedback on any work (and later thesis chapters), would always arrive very quickly. This was quite in contrast to the extended periods of time that I typically needed for writing. Although Andy encouraged me often to give him rough draft versions much earlier, I have always found this quite difficult. I appreciate very much Andy's help and patience while I learned, improved and finally adopted more efficient ways of working.

It seems like only a short time since I finished my Master's dissertation at UvA and started a PhD in the same group, but it is really already quite some years ago. It was just a small group then, and Simon and Cagkan were there from the beginning and by working with them I learned much about the methods and techniques that formed the basis of our research. I am very thankful for their kind support at that time, for the pleasant and friendly atmosphere in the office and for their continued friendship in the years to follow. I fondly remember the break times when we'd take one of the many RC cars, helis or planes which decorated our office for a spin outside or (with rain or wind) in one of the abandoned offices and hallways. In time, more and more new colleagues joined the group, which not only allowed me to glean a wider variety of research topics, but also had the added and much appreciated

benefit of increasing the number of birthdays and other occasions which (according to our well-maintained group tradition) were celebrated with cakes.

I specially want to mention my colleagues with whom, at various occasions, I worked closely together on a project or research topic: Toktam, Kamana, Roberta, and Peter, as well as Carlos and Abhinav who travelled from far to join our group for a short time. Working with you was a pleasure and I hope to have the opportunity to work with you on a paper or project again in the future. Outside of work, we also shared many fun times, such as delicious dinners and relaxing entertainment, often graciously hosted by Raphael. As I regain some spare time, I will be happy to return the favor by inviting you all to my place more frequently.

Furthermore, I thank my promotor Chris for his support of my research and for his lightning-fast response regarding some last-minute administration issues. I am also grateful to Cees de Laat, Todor Stefanov, Donatella Sciuto and Henk Corporaal for agreeing to be in my promotion committee and for reading my thesis. Moreover, I doubly extend my gratitude to Todor and Hristo for their co-operation within the Daedalus project and to Cees de Laat for making his group's computing resources available to me.

In the final stages of thesis writing and with the final deadline looming, several people helped me a lot by offering both practical and emotional support. Michiel helped me tremendously in the last few hours before submission by improving the readability of many graphs, Steven generously advised on some questions about statistics and Roberta talently designed an aesthetically pleasing thesis cover. I thank Erol, Gokhan, Andy, Michiel and Raphael for sharing their thoughts, providing encouragement and putting things in perspective when I was unable to do so myself. Thank you all so much!

I would like to thank my parents for always being there and giving me essential and wise advice when I needed it. Also, I don't know how I could have taken the last hurdles without Xun at my side: her love, intelligence and sense of humor brightens my life. Lastly, I am grateful to all my family and friends for their restraint to not ask too often that one question that is –at some point– dreaded by all PhDs: "By the way, how is it going with...?". To that question I can finally answer that, yes, now it is done!

Contents

Acknowledgements **3**

1 Introduction **9**
1.1 Design constraints and trade-offs 10
1.2 Current state of technology 11
1.3 Platform-based design 13
1.4 System-Level Design 14
1.5 Navigating the design space 15
1.6 Scope and contribution of this thesis 17
1.7 Thesis layout 19

2 Daedalus: design flow **23**
2.1 Introduction 23
2.2 The Daedalus framework 24
2.3 Parallelizing applications 26
2.4 Design Space Exploration 28
2.5 System-level Synthesis 29
2.6 The Daedalus Design-flow Infrastructure 30
2.7 Related Work 32
2.8 Conclusion 32

3 Sesame: modeling and simulation **33**
3.1 Introduction 33
3.2 Overview 33
3.2.1 The application model 35
3.2.2 The architecture model 36
3.2.3 Mapping 39
3.3 Implementation Aspects 43
3.3.1 Model specification 43
3.3.2 The application model 44
3.3.3 The architecture model 46
3.3.4 The virtual/mapping modeling layer 48
3.3.5 Graphical user-interface 49
3.4 Setting up the Design Space 50

3.5 Model and design space evaluation 54
3.6 Conclusion 56

4 Model calibration **57**
4.1 Introduction 57
4.2 Model calibration 58
4.3 Off-line model calibration 60
4.4 On-line trace calibration 62
4.5 Trace calibration experiments 65
4.6 Signature based model calibration 68
4.6.1 Application requirements 68
4.6.2 Processor capacity and performance estimation 70
4.7 Signature-based calibration experiments 71
4.7.1 Related work 73
4.8 Conclusion 75

5 Multi-application modeling **79**
5.1 Introduction 79
5.2 Multi-application workload modeling 80
5.2.1 Synthetic multi-application workload modeling 80
5.2.2 Realistic multi-application workload modeling 83
5.3 Multi-application modeling: a designer's perspective 84
5.4 Dynamic application behavior 85
5.4.1 Dynamic inter-application behavior 86
5.4.2 Dynamic intra-application behavior 88
5.5 A preliminary case study 90
5.6 Related work 91
5.7 Conclusion 93

6 Modeling dynamically reconfigurable systems **95**
6.1 Introduction 95
6.2 Modeling dynamically reconfigurable systems 96
6.2.1 Dynamic allocation of model components 96
6.2.2 Resource management 97
6.2.3 Mapping strategy 99
6.2.4 Event trace clustering 101
6.2.5 Reconfiguration points 103
6.3 The molen reconfigurable platform 104
6.4 Sesame molen model 105
6.4.1 Mutual exclusion of GPP and RP 106
6.4.2 Mapping strategies 107
6.4.3 Application pipelining 108
6.4.4 Component interaction 109

6.5 Experimental results . . . 109
6.6 Related work . . . 114
6.7 Conclusion . . . 115

7 Support for automatic DSE 117
7.1 Introduction . . . 117
7.2 Related work . . . 118
7.3 DSE as a GA search problem . . . 121
7.4 Initial case study . . . 124
7.5 Initial case study - parameters . . . 129
7.6 Distance metric . . . 134
7.6.1 Observations on the design space . . . 135
7.6.2 Distance metric details . . . 135
7.6.3 Example . . . 137
7.6.4 Performance improvement . . . 139
7.7 Initial case study with distance metric . . . 139
7.8 GA with integrated distance metric . . . 142
7.8.1 Reducing representation redundancy . . . 143
7.8.2 A distance-metric based cross-over operator . . . 144
7.8.3 Combination of approaches . . . 145
7.8.4 Experiments . . . 146
7.9 Conclusion . . . 151
7.10 Future work . . . 152

8 Case studies 155
8.1 Introduction . . . 155
8.2 Case study 1: Exploration and validation . . . 155
8.3 Case study 2: a tiled MPSoC architecture . . . 159
8.3.1 Simulation-level DSE . . . 160
8.3.2 Implementation-level DSE . . . 163
8.3.3 Evaluation . . . 167
8.4 Conclusions . . . 167

9 Conclusions 169
9.1 Evaluation of a single design point . . . 170
9.2 Traversing the design space . . . 173

References 175

Nederlandse samenvatting 188

Scientific output 190

Chapter 1

Introduction

In today's world, many devices that traditionally operated on purely mechanical or analog electro-technical principles, are enhanced and extended with small, integrated digital computer systems. These so-called *embedded systems* either replace or accompany traditional components as part of the updated design of the device, thereby extending its functionality or reducing the cost. Examples of such embedded systems are close at hand: modern TVs contain one or multiple computer systems in order to handle functionality such as decoding the input signal, performing various image enhancements techniques as well as displaying and updating live information (e.g., program guide or weather forecast). Cars depend on embedded systems to do anything from braking to fuel injection and deploying airbags. The use of embedded systems is however by no means restricted to consumer electronics: in industrial, medical or defense applications they are equally pervasive. In fact, it is estimated that embedded systems now outnumber more commonly known computer types (desktop PCs, game consoles, etc.) by two orders of magnitute. This can partially be explained by the fact that embedded systems bear the promise to improve existing products (in terms of functionality, usability, interoperability, etc.) and to enable the development of entirely new products and devices that were previously inconceivable. Additionally, the fabrication process technology for embedded computing systems is now at a point where they can be produced at relatively low prices.

In addition to the increasing demand for embedded systems, there is also a clear trend towards more complex systems that combine different functions into a single device. Consider for example mobile phones that integrate more and more functions such as (video) camera, GPS-based navigation and internet browsing capabilities. Moreover, new generations of these products have to be released in shorter time frames. By recognizing and extrapolating this trend, the notion of ubiquitous computing has been developed: small, yet powerful interconnected computer systems that are unobtrusively integrated in our everyday objects and activities, augmenting our natural cognitive, sensory and communication capabilities. In face of the excitement of such a prospect (be it positive or negative), one would almost forget the enormous technical challenges that need to be solved for even the current and next

generation of "common" embedded systems [1]. It is clear that improved methodologies and tools are needed in order to design the next generation of embedded systems that meet the requirements of the future. In the remainder of this chapter, we describe the background of the embedded systems field, discuss the motivation of the work presented in this thesis, and address the main research questions.

1.1 Design constraints and trade-offs

The design and engineering of embedded systems makes for an interesting field of study because it not only deals with the issues already present in commodity-computer system design (e.g., functionality, performance), but it deals with additional constraints as well. For example, the issue of power consumption is of relatively small concern for desktop computer systems. But for a mobile, battery operated embedded system, high power usage can mean complete design failure and render the device practically useless (e.g. a mobile phone that discharges within a day). Reliability is often also a concern for embedded systems, since they may be part of continuously operating devices (set-top boxes, surveillance systems), or safety-critical systems such as fly-by-wire systems of an airplane. Other design requirements that are typically associated with embedded systems are cost, physical size, redundancy and flexibility (the ability to use or reuse the system in multiple applications). These requirements result in a set of so-called *design constraints*: a list of requirements that have to be met by any candidate system design in order to be considered successful. Even the development time of an embedded system can be considered a design constraint, since updated or innovative systems have to enter the market before the competition. It is commonly accepted that design constraints are inherently non-orthogonal: improving the system according to one constraint may decrease the value of another. For example, performance can be increased by adding additional processing components, which typically reduces power efficiency and increases cost. Performance and power usage can be improved by using ASICs instead of programmable processor components, but this in turn reduces the flexibility of the system and increases design time in case of custom ASICs.

In the field of embedded systems, it has traditionally been the case that a system's functionality was quite fixed and therefore flexibility was not a major design concern. However, these days flexibility is a very important design criterium. As embedded systems are becoming increasingly complex, such as mobile communication and media devices that may be required to run software that was not envisioned at design time, or alternatively, the device was designed to be adapted and updated during its lifetime. The latter makes sense from a design perspective, since embedded systems are pervasive in all kinds of applications, and there may be great cost involved in updating or fixing such systems. Take as an example the embedded systems in use in the automotive industry: post-production errors may require a recall of all cars of a specific model in order to fix the problem. The total cost of the recall

[1] Not to mention the non-technical challenges which include human-computer interaction, privacy, environmental and safety concerns

can be reduced if the fix is a simple software or firmware upgrade by the brand dealer instead of an expensive repair that requires replacement parts and labor.

A different concern related to flexibility exists in those consumer product areas where a system is used for a relatively short time and new devices are released frequently, thus increasing pressure on the design process. A good example where product lifetime (and therefore design cycles) are reduced is the mobile consumer electronic market of mobile phones, media players, navigation systems, etc. The result is that there is too little time to re-design the system between generations, and therefore, large parts of the design have to be reusable. In these cases, but the design process itself needs to be flexible in order to be reused for the next generation of products.

There is no generic solution to the problem of non-orthogonal design constraints, so that trade-offs have to be considered carefully for each system individually. In traditional design methodologies, a system designer would often make trade-offs implicitly, guided only by his expert knowledge and experience ("the art of system design"). As the complexity of embedded systems increases, there is a trend towards more explicit declaration of design constraints, which enables methods and tools to (semi-)automatically help the designer to search for designs that meet the design constraints in the best possible way. The aim of such methods and tools is always to reduce the complexity of embedded system design, which in turn should increase quality and reduce design time.

1.2 Current state of technology

Modern embedded systems are built using the wide range of component, process and packaging technologies that are available today. Typically, a system consists of one or multiple CPUs (e.g., microprocessor, DSP, or ASIC), memory (ROM, RAM, etc.), interconnects, timing sources and counters as well as external interfaces (USB, Ethernet, UART, etc.). Many options are typically available for each type of component, each of which has different properties that will push the design constraints one way or another. For example, using an ASIC implementation to perform a certain (fixed) functionality generally improves performance and reduces power consumption as compared to execution on a microprocessor. On the other hand, the use of an ASIC reduces the flexibility as compared to a microprocessor and it may increase the cost of a system because of increased packaging cost or intellectual property (IP) licensing fees. In addition, there exists an entire range of processor options in between generic microprocessor and ASIC, each with different properties (consider for example ASIPs or DSPs). Determining which combination of components is suitable for a particular system is a non-trivial problem.

For many embedded applications a better trade-off in the design criteria can be found by combining different (*heterogeneous*) types of cores in system, where each core is optimized for a particular part of the operation of the whole system. This is the reason that embedded systems were the earliest mass-produced multi-processor systems, far before the first general purpose multi-processor PC systems (IBM Power4, 2001). The embedded system field was particularly suitable for this development, because initially embedded systems were not

designed with flexibility in mind and therefore design trade-offs could be fine-tuned to the limited set of requirements for a particular application. As we mentioned before, this is completely different for current and future generations of embedded systems.

There are different methods to combine the various components for a heterogeneous embedded system. In addition to the traditional printed circuit board (PCB), components can now be joined together as a System-in-Package (SiP), Package-on-Package (PoP), System-on-Chip (SoC), or using a combination of these techniques. Each technique has different pros and cons with respect to the manufacturing process, which again results in a system of trade-offs. For example, a processing/memory combination as a PoP allows a manufacturer to develop the components separately or, alternatively, to allow some waiting time to buy an off-the shelf component at a good price point. A System-on-Chip does not have this benefit, because for a SoC all components are manufactured onto the same silicon die. However, SoC has the added benefits of increased performance (lower latencies between components), lower system-assembly cost and may work out cheaper if produced in sufficient volume.

Improvements in lithographic process technologies continually increase the density of on-chip resources. The result is that many embedded systems now make use of multi-processor SoC (MPSoC) technology. Indeed, Moore's law seems to be alive and well, predicting a doubling of transistors on chip every two years. Next generations of high-end commodity processors will consist of a few billion of transistors and it is likely this will be tens of billions in the near future. The result is that the predominant research question in both the embedded and the commodity processor design fields is now: how to to put these enormous amounts of available resources to efficient use? There is a general consensus that there exists an *implementation gap*, that is: transistors are now so plentiful that traditional design methods fail to efficiently use them all. In the commodity processor field the (stop-gap) solution has been to use the resources to replicate existing designs by doubling the number of cores or by increasing the L2 cache size. However, such a homogeneous solution does not fit well with the embedded design field which (for the reasons mentioned in the previous section) looks towards inherently heterogeneous designs.

In recent years, MPSoC system development based on reconfigurable technologies (such as FPGAs) have received increasing attention from both research and industry. This is not surprising, as the cost of FPGAs goes down and gate count goes up, driven by the improvement of manufacturing technology. Modern FPGAs consist of hundreds of millions of gates, which is sufficient to implement complex MPSoC systems consisting of tens or hundreds of processing components as well as memories and on-chip interconnection networks. FPGA technology fills a niche in the (embedded) system design market as it has different cost and performance properties compared to traditional ASIC-based system design. System development on FPGA does not need the same time-consuming and expensive fabrication process as ASICs, making FPGAs an interesting solution for system prototyping, especially in research and low-volume applications. Compared to ASICs, the unit cost of FPGA solutions is relatively high, and the performance is generally rated to be lower than ASICs [59] for systems based on similar hardware design. But in some computational domains FPGA technology seems actually to be catching up; e.g., it has been reported in [108] that the increase

in peak performance per year is higher for FPGAs than for commodity CPUs. However, the major point in favor of FPGAs is their flexibility: the logic design of the FPGA system can be adapted to and optimized for a specific application or workload. We will frequently refer to and use this technology throughout this thesis.

1.3 Platform-based design

Platform-based design has become one of the major approaches in recent years to overcome the challenges for embedded system design. A platform is a partial definition of a system encompassing hardware and software components, interfaces, APIs and (sometimes) a tool suite with compilers and debugging tools or an integrated development environment (IDE). On the one hand, the hardware part of a platform (in contrast to a fully custom-designed system), is defined with flexibility in mind so that it is suitable for a range of applications or products. Flexibility comes from the inclusion of programmable cores (microprocessor, DSP) or reconfigurable hardware (FPGA) in the platform. On the other hand, platforms typically also contain more static architectural features (such as ASICs) which are optimized for specific applications. In this way, platforms aim to strike a balance by combining flexibility with application specific optimization, which can ultimately result in systems that meet e.g. power and performance requirements. The challenge for the designer is to make the remaining design decisions offered by the platform, in order to create the system as a *platform instance* for use in a given application.

There are different implementation and fabrication possibilities for platforms: on a single IC-die, as a collection of interoperable components, entirely on reconfigurable fabric, or even as a mix of these. Moreover, platform-based design can have a positive impact on the economic trade-offs that are inherent to manufacturing. For example, the increasing non-recurring engineering cost (e.g., mask creation) for custom-built ICs can be mitigated as multi-purpose platform ICs can achieve larger production volumes. Single-die IC-based platforms are typically released as a family of platform products, where each type offers different configurations (e.g., different memory size or integrated ASICs to accelerate particular applications). These platform families can reuse large parts of the platform design, which reduces design cost. Later on in this thesis we will consider (multi-processor) platforms implemented entirely on the reconfigurable fabric provided by an FPGA. Here, platform based design takes the form of composing the platform from a pre-defined library of components which contains both programmable and dedicated hardware cores. As we will see however, the methods and techniques presented in this thesis are equally applicable to all types of platforms, irrespective of implementation or manufacturing technology.

Another benefit of platform-based design is that it provides a certain level of standardization, which is conductive to the development of the software stack running on the hardware as well as the development of software tools. In this way, libraries, (real-time) operating systems, compilers and debugging tools can reach a level of maturity that benefits development on that platform.

In summary: a platform fixes most technology parameters (and some of the other design

parameters) and provides a stable development environment that may not be readily available for custom ICs. However, a platform is only the starting point of a design process, and many problems still need to be solved. Finally, we consider that the platform itself has to be designed, which is perhaps the most complex design problem of all, due to the requirement of having to be useful for many different purposes.

1.4 System-Level Design

In order to manage the increasing complexity of modern embedded systems, designers are forced to view the system from a higher level of abstraction. *System-level design* aims to provide a path from system specification to system implementation in such a way that the resulting system meets the design requirements and the design effort is efficient in terms of time and cost. The system level viewpoint considers the system as a modular collection of software and hardware components, without the need to define every detail of every component at the early design stages. In practice this means that the design process is divided in a number of (more or less) distinct phases that a designer traverses one by one. In each subsequent phase, the system specification is extended with additional details, so that the final phase results in a fully specified, implementable system specification.

An example design process that starts at the system level is given in Figure 1.4. The width of the pyramid shape indicates the relative number of design options in each stage of the design: the high-level stages have less options, since they omit the lower level details of the system; the base of the pyramid represents all the possible system implementations. The design process starts at the top with the high-level system requirements and the platform specifications.The first design stage includes the pen-and-paper designs and very simple spreadsheet models which are common design practice to confirm the designer's initial intuitive solutions and to define initial design space boundaries. It is unfortunately the case that the next design stage too often is the cycle accurate or RTL model, as they are currently the most available and best understood models. Particularly the use of RTL-level models is common, because it is offered by industrial design tools, which are geared towards implementation and debugging and provide very little system-level design support that support design decisions in the very early stages. This is in fact a manifestation of the aforementioned implementation gap from the perspective of system design: there exist no mature methodologies, techniques, and tools to effectively and efficiently convert system-level system specifications to RTL specifications. We propose a smoother transition by adding an intermediate stage consisting of abstract executable models. These models are less detailed, easy to construct and allow for evaluation (by fast simulation) to support early design decisions (Chapter 3).

Making design decisions in the early design stages is essential to reduce the number of implementation options and thereby reducing the total design effort. This process is called design space *pruning*. In every design phase, a subset from the non-pruned design options is selected and evaluated. The evaluation is typically performed by means of simulation, or (at the lower abstraction levels) by making prototype implementations. The feedback from

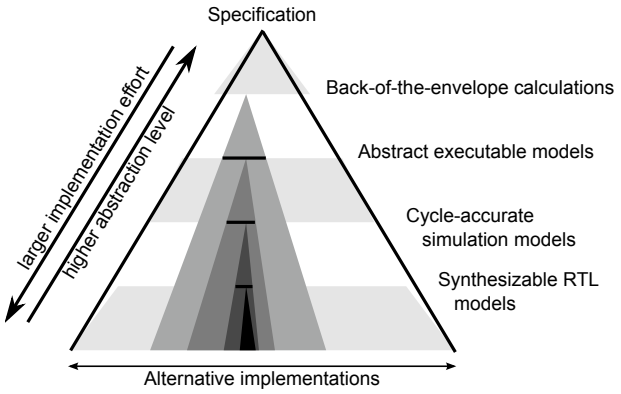


Figure 1.1: The system-level design process

the evaluation determines which of the candidates will be used in the next (lower) level of abstraction in the design process. In Figure 1.4, the set of evaluated design candidates is represented by the horizontal black line; the triangular shapes show the optional designs (at subsequent design phases) that can follow from a higher level candidate. Pruning at a higher level of abstraction has the potential to significantly reduce the design effort. For example, by selecting a good candidate at the abstract executable model (where evaluation is cheap), one may prevent multiple expensive re-implementations at the RTL-level.

System-level design is still very much an active area of research, since there are many unsolved problems in the design path from system-level specification to implementation. In particular, it is hard to offer a single, generic methodology, and furthermore, the transition from one level of detail to another typically requires various amounts of manual effort. This makes it infeasible to provide a fully automatic design process. Nevertheless, there is a clear need for such a methodology as it could help solve the aforementioned implementation gap problem. In the next chapter we will introduce the Daedalus system-level design tool flow that aims to do exactly that.

1.5 Navigating the design space

In order for an embedded system to meet the design criteria, the system designer is faced with the challenge of making the right design choices at every design stage and for every aspect of the system. Given the increasing complexity and delivery demands of embedded systems, it has become infeasible to perform this task by hand. The many design choices that have to be made at the system level, may include (but are certainly not limited to) the following:

- the number and type of programmable processors,
- HW/SW partitioning: deciding which tasks to implement in software and which tasks as fixed ASICs or reconfigurable hardware blocks,

- mapping of application tasks to architecture resources,
- choice of on-chip interconnect (e.g., bus, direct connection, crossbar, etc.),
- size, type and location of memory components
- et cetera ...

Indeed many of these design decisions in reality give rise to any number of additional choices, e.g., embedded soft or hardcore IPs may be parameterizable (or be available in variants) with different number and type of functional units, pipeline depth, specific ISA extensions or bus interfaces. Note that design choices can have inter-dependencies and are not necessarily fully orthogonal. For example, the mapping of application tasks depends on the number of available processors in the architecture, and the type of processor determines whether the task runs efficiently (e.g., a task that contains divide operations may not run efficiently on a processor without integer or floating-point FPU), or whether the processor can run the task at all (only specific tasks may be mapped to an ASIC).

However, in most cases, the number of possible parameter combinations far outweighs the dependent or impossible combinations and the design space grows exponentially with the number of parameters. For the purpose of illustration, let's assume p parameters, each with an equal number of c orthogonal, independent number of choices, then the design space consists of c^p possible designs. We can view the design space as a p-dimensional space where each axis represents one of the parameters. We can map each point in this *parameter space* to a point in the *objective space* (Figure 1.2), where each axis represents the design criteria in terms of specific objectives (performance, power, etc.). The values of the objectives have to be obtained with an appropriate evaluation mechanism, which may consist of some kind of estimation algorithm, a simulator, or measurements on a real (prototype) system implementation. If the objective space were a given, then a designer could easily select the candidate design and build a system with the given parameters. However, in real design problems, the design space is large and complex and the objective space can not be trivially derived from the parameter space (at least not in reasonable time).

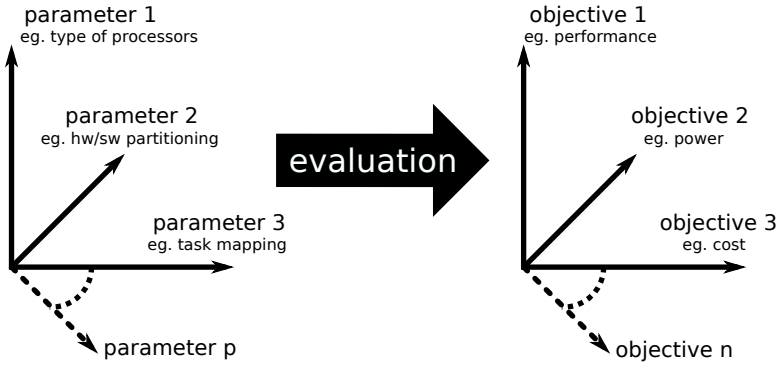


Figure 1.2: The design space broken down in parameter and objective space

In practice, the designer will attempt to navigate the design space using a partial understanding of the objective space. In many cases, the one design point that is better than all other design points (the *true optimum*), may never be found. However, finding a design point in the known design space that meets the design requirements as best as possible, is often sufficient. In general, we identify two ways of navigating the design space:

- *Design space pruning* (discarding unsuitable design points)
- *Design space searching* (looking for optima in the design space)

Both cases attempt to bring into focus the part of the design space that is of interest to the designer (containing the optima): either by slashing unsuitable parts of the design space (pruning) or by design space traversal based on an algorithm that finds incrementally better design points (searching). Note that these methods are often combined to obtain the best result. In general we observe that for successful DSE there are two requirements:

- the ability to evaluate a single design point
- the ability to use the evaluation to traverse the design space in search of optima

Both requirements are non-trivial and in practice many trade-offs have to be made. For example, to obtain the objective values of a single design point with high accuracy, detailed (and therefore slow) evaluation is necessary. Such evaluation mechanisms are unsuitable for use in the early design stages, where most of the design options are still undecided and evaluation feedback should be quick. Note however, that low-level simulations are typically very useful in later design stages. As we will see later in this thesis, giving up some of the accuracy in exchange for speed is a useful strategy.

However, faster evaluation speed goes only so far when we consider the exponentially growing design spaces of real-world design problems. Efficient pruning and searching algorithms are required that make appropriate use of the evaluation mechanism. This is the area of design space exploration that is currently the least well understood. Various attempts have been made and, in some particular cases, have been shown to be quite successful, but it is still an open research question to find generic, scalable methods that work for a wide variety of design problems.

1.6 Scope and contribution of this thesis

The work in this thesis has been performed in the context of the Daedalus design tool flow. The tools in Daedalus enable system-level design space exploration and help the designer to (semi-)automatically traverse the entire design process, including parallelizing applications, design space exploration (DSE), and finally resulting in a prototype system implementation (see Chapter 2 for more details). In this thesis we focus on the modeling and simulation part of the tool flow and its implications for efficient design space exploration. Within Daedalus, the Sesame tool (Chapter 3) is used for modeling and simulation at the abstract executable

level in order to provide feedback about design decisions in the early design stages. The abstract modeling methodology used by Sesame delivers high-performance models that are require only a small implementation effort compared to traditional, lower level models.

The main contributions of this thesis concern the following topics:

- model calibration and validation: improving and verifying model accuracy
- extending modeling scope: modeling capabilities for different system types
- implementation and analysis of evolutionary search algorithms for DSE

These topics have been selected on the basis of an ideal DSE scenario, where the methodology and tools support a designer by offering fast, modular models that can be easily modified to accurately model many different systems. These models then serve as input for one or more efficient design space search algorithm that can quickly find optimal design points. Recognizing the many problems that are still open in this field, we try to approximate the ideal scenario as much as possible. Therefore, we expand the two requirements for efficient DSE of the previous section into a classification of what we consider the most important characteristics on respectively the modeling and exploration side (Figure 1.3).

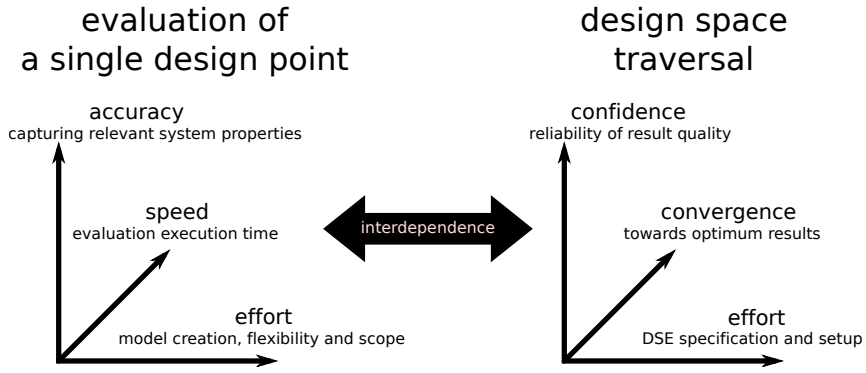


Figure 1.3: The two-part taxonomy of concerns for efficient DSE

Naturally we need models that are both as fast and as accurate as possible (or at least sufficiently so for our purpose). But it is just as important that the early design stages are not burdened by model implementation effort: modularity and reusability of models is important in this respect. Lastly, the models need to be able to model a wide range of systems, covering both old and new technologies.

On the traversal/exploration part, *convergence* denotes the speed of evaluating a range of design points, and, more specifically, the rate at which the DSE search algorithm manages to converge to an optimum. The *confidence* characteristic denotes how certain we are that the design points returned by the DSE includes the true optimum, or alternatively, how close they are to the true optimum. In many search algorithms confidence is obtained by avoiding local optima and ensuring sufficient design space coverage. Completely analogous with

effort in the case of evaluating a single design point, the *effort* (for design space traversal) should again be minimized. In the latter case *effort* refers to implementation of the search method and setting its parameters, as well as setting up, running and evaluating the results of the exploration experiment. Effort reduction can be accomplished by finding generic search methods that perform well for a wide range of design spaces; and by automating many of the exploration steps that would otherwise have to be done manually. In the conclusions of this thesis (Chapter 9), we return to the above classification and discuss the various contributions in the context of the ideal DSE scenario.

1.7 Thesis layout

The chapters in this thesis can be roughly divided into three parts:

- Part One: background, design flow and tools (Chapters 1, 2 and 3)
- Part Two: techniques and methods (Chapter 4, 5, 6 and 7)
- Part Three: experiments and case studies (Chapter 8)

The introduction provided the background of the work and puts the work in a global context.

Chapter 2 describes the Daedalus tool flow, which provides the specific context in which the subsequent chapters of Part Two and Part Three have been performed. The tools section is continued in Chapter 3 with the modeling and simulation part from the toolflow (Sesame). Sesame is used for almost all of the experiments in this thesis, either in its basic form, or with specific adaptations that support a particular technique. Specifically, we describe the various parts that compose a Sesame model: the model description, the application and architecture model and the application-to-architecture mapping. Furthermore, we show how to perform a design space exploration case study using Sesame. The aim of this chapter is to provide sufficient information about the Daedalus toolflow (and the Sesame tool in particular) to promote a good understanding of the topics in the remainder of this thesis. Additionally, we give enough details to serve as a user introduction to Sesame and to recreate any of the experiments.

The second part of the thesis presents various advanced modeling and simulation techniques. These techniques can aid a designer to traverse the design space by improving the accuracy of the model, or to capture different systems and behavior that could not be represented in the standard Sesame model. Each chapter focuses on a specific method or technique, discusses its benefits, and shows how the Sesame model can be adapted to support it. Most of these methods and techniques are not restricted for use in Sesame and can be applied in other modeling and simulation environments as well. Care will be taken to indicate which parts of the techniques and methods are Sesame-specific, and which are more generally applicable. Each chapter concludes with an experimental section to demonstrate how the specific method or technique can be applied in a practical case study. Two detailed and industrially relevant case-studies are given in the last part of the thesis.

The first chapter of Part Two (Chapter 4) describes techniques to calibrate the performance estimation of a model to a specific platform or implementation. Sesame's high-level models capture both the behavior of each component in the system as well as the interaction between model components. The description of each component's behavior (or component interaction) is usually maintained separately from its quantitative performance impact. Quantitatively, the performance is described as a list of latency values, which are typically part of the parameter list of a model component. These parameters influence greatly Sesame's ability to evaluate the performance of a modeled system. Chapter 3 gives an overview of latency parameters used by various standard Sesame components. In Chapter 4, a method will be shown to obtain and refine these model latency parameters.

Chapter 5 deals with multi-application workloads in Sesame. The increasing complexity of modern embedded systems is partially due to the fact that they need to be able to execute multiple applications concurrently. There are different ways such multi-application workloads can be described and implemented. Naturally, as Sesame is intended for use in the early design phases (where important design decisions are made), it should be capable of modeling such workloads. In addition to real workloads, we describe a method to integrate synthetic application workloads in Sesame. These can be particularly useful for speculative exploration of a system's properties as well as supporting simultaneous development of application and architecture models by using synthetic workloads to develop the architecture while the application is still in development.

Chapter 6 demonstrates a technique to model platforms that contain reconfigurable components. Reconfigurable components, such as FPGAs, can be integrated into embedded systems to improve performance and flexibility. Performance is obtained by implementing application-dependent accelerators in logic on the reconfigurable element(s) in order to speed up execution. The ability to change part of the reconfigurable logic on-the-fly (partial dynamic reconfigurability) introduces the flexibility to re-use the reconfigurable resources for different accelerators at runtime. In this chapter it is shown how this feature can be modeled in an environment (such as Sesame) that does not have native (built-in) support for it. An additional challenge is that dynamically reconfigurable systems greatly increase the number of design choices, which results in a more complex design space. Furthermore, some suggestions and an example are given of how design space exploration can be performed for these systems. A case study looks at the evaluation of different reconfiguration policies for an extended processor-coprocessor platform.

Chapter 7 deals with the problem of fully automated design space exploration. Whereas the topic of design space exploration has been around for many years, *automated* DSE is a relatively young field of research. Relatively little is known about the structure of design spaces and the best algorithms to automatically search them. In this chapter we take a closer look at the use of evolutionary algorithms in this context (particularly genetic algorithms: GAs). Such algorithms have been used successfully for combinatorial search problems in other research domains. Here we consider the benefit of GAs for some of the most challenging design space parameters available: task mapping. In an attempt to improve the results we introduce a metric that may provide a handle on the otherwise chaotic organization of the

mapping subspace. The metric also gives rise to a new implementation of GA primitives.

In the third and final part of the thesis, Chapter 8 reports on several larger case studies. Firstly we show the results of a validation case study where we validate the Sesame simulation results as compared to measurements on prototype systems from the Daedalus toolflow. Next, an industrial case study is presented where an image processing application is mapped onto a tiled architecture. We discuss the modeling and simulation aspects and show the design space exploration process, which is partially automated. Finally, the thesis concludes with Chapter 9, where we revisit the contributions of the thesis in light of the requirements for efficient DSE as summarized in Figure 1.3.

Chapter 2

Daedalus: design flow

2.1 Introduction

As mentioned in the previous chapter, the complexity of modern embedded systems, which are increasingly based on heterogeneous MultiProcessor-SoC (MP-SoC) architectures, has led to the emergence of system-level design. To cope with this design complexity, system-level design aims at raising the abstraction level of the design process. Key enablers to this end are, for example, the use of architectural platforms to facilitate re-use of IP components and the notion of high-level system modeling and simulation [53]. The latter allows for capturing the behavior of platform components and their interactions at a high level of abstraction. As such, these high-level models minimize the modeling effort and are optimized for execution speed, and can therefore be applied during the very early design stages to perform, for example, architectural Design Space Exploration (DSE). Such early DSE is of paramount importance as early design choices heavily influence the success or failure of the final product.

System-level design for MP-SoC based embedded systems typically involves a number of challenging tasks. For example, applications need to be decomposed into parallel specifications so that they can be mapped onto an MP-SoC architecture [66]. Subsequently, applications need to be partitioned into HW and SW parts since MP-SoC architectures often are heterogeneous in nature. To this end, MP-SoC platform architectures need to be modeled and simulated to study system behavior and to evaluate a variety of different design options. Once a good candidate architecture has been found, it needs to be synthesized, which involves the synthesis of its architectural components as well as the mapping of applications onto the architecture. To accomplish all of these tasks, a range of different tools and tool-flows is often needed, potentially leaving designers with all kinds of interoperability problems. Moreover, there typically remains a large gap (the so-called *implementation gap* [69]) between the deployed system-level models and actual implementations of the system under study. Currently, there exist no mature methodologies, techniques, and tools to effectively and efficiently convert system-level system specifications to RTL specifications.

Daedalus' main objective is to bridge the aforementioned implementation gap for the

design of multimedia MP-SoCs. It does so by providing an integrated and highly-automated environment for system-level architectural exploration, system-level synthesis, programming and prototyping. In this chapter we will show how the different components fit together as the pieces of a puzzle, resulting in a system-level design environment that addresses the entire design trajectory with an unparalleled degree of automation.

The next section provides a birds-eye overview of Daedalus, after which the three subsequent sections present the three core tools that constitute Daedalus in more detail. More specifically, Section 2.3 explains how multimedia applications are automatically decomposed in parallel specifications. Section 2.4 describes how – given the parallel application(s) – promising candidate architectures can be found using our system-level modeling, simulation and exploration methodology and toolset. In Section 2.5, we explain how selected candidate architectures can be automatically and rapidly synthesized, programmed and prototyped. Section 2.6 describes in some more detail how the different components of the tool-flow have been linked together. Finally, Section 2.7 discusses related work. As the main focus of this thesis concerns design space exploration, the next chapter will take a closer look at the Sesame simulation and modeling environment, which we already introduce shortly in this chapter (Section 2.4 In Chapter 8 of this thesis a case study demonstrates the entire Daedalus design flow in action.

2.2 The Daedalus framework

In Figure 2.1, the design flow of the Daedalus framework is depicted. As mentioned before, Daedalus provides a single environment for rapid system-level architectural exploration, high-level synthesis, programming and prototyping of multimedia MP-SoC architectures. Here, a key assumption is that the MP-SoCs are constructed from a library of pre-determined and pre-verified IP components. These components include a variety of programmable and dedicated processors, memories and interconnects, thereby allowing the implementation of a wide range of MP-SoC platforms. The remainder of this section provides a high-level overview of Daedalus, after which the subsequent sections zoom in on its core components and how they interact with the rest of the design flow.

Starting from a sequential application specification in C or C++, the KPNgen tool [116] allows for automatically converting the sequential application into a parallel Kahn Process Network (KPN) [51] specification. Here, the sequential input specifications are restricted to so-called static affine nested loop programs, which is an important class of programs in, e.g., the scientific and multimedia application domains. By means of automated source-level transformations [94], KPNgen is also capable of producing different input-output equivalent KPNs, in which for example the degree of parallelism can be varied. Such transformations enable application-level design space exploration.

The generated or handcrafted KPNs (the latter in the case that, e.g., the input specification did not entirely meet the requirements of the KPNgen tool) can subsequently be used by our Sesame modeling and simulation environment [79] to perform system-level architectural DSE. To this end, Sesame uses (high-level) architecture model components from the

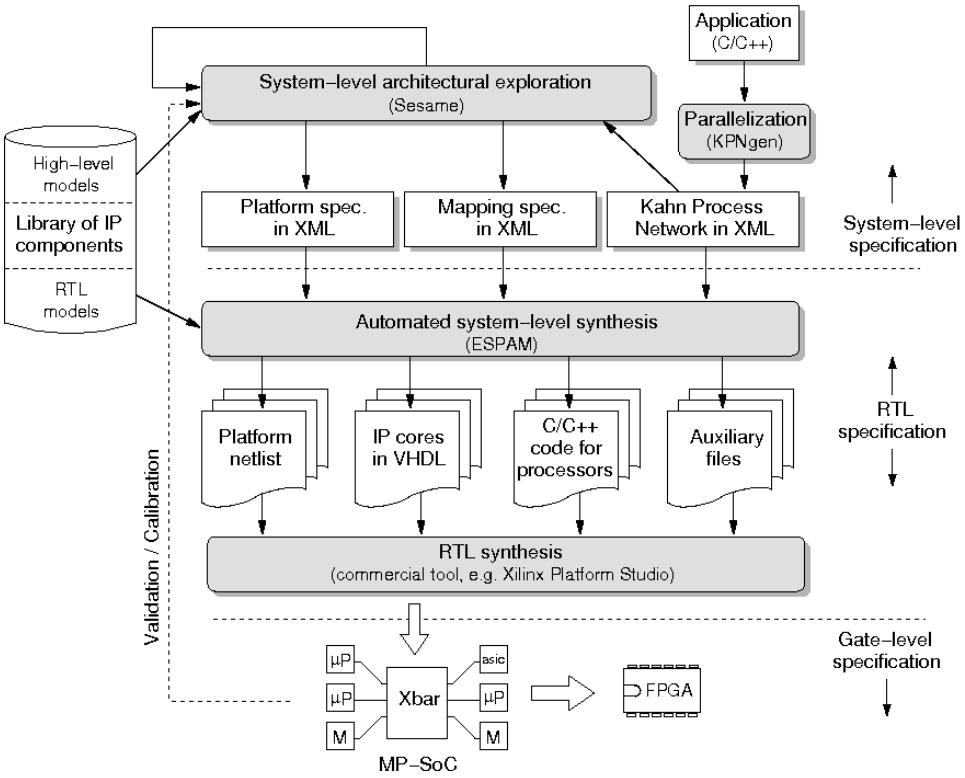


Figure 2.1: The Daedalus design flow.

IP component library. Sesame allows for quickly evaluating the performance of different application to architecture mappings, HW/SW partitionings, and target platform architectures. Such DSE should result in a number of promising candidate system designs, of which their specifications (system-level platform description, application-architecture mapping description, and application description) act as input to the ESPAM tool [74]. This tool uses these system-level input specifications, together with RTL versions of the components from the IP library, to automatically generate synthesizable VHDL that implements the candidate MP-SoC platform architecture. In addition, it also generates the C/C++ code for those application processes that are mapped onto programmable cores. Using commercial synthesis tools and compilers, this implementation can be readily mapped onto an FPGA for prototyping. Such prototyping also allows for calibrating and validating Sesame's system-level models, and as a consequence, improving the trustworthiness of these models.

Ultimately, Daedalus aims at traversing an entire design flow – going from a sequential application to a working MP-SoC prototype in FPGA technology with the application mapped onto it – in a matter of hours. Evidently, this would offer great potentials for quickly experimenting with different MP-SoC architectures and exploring design options during the early stages of design.

In the following sections we describe each of the main components in Daedalus: appli-

cation parallelization, design space exploration and system-level synthesis.

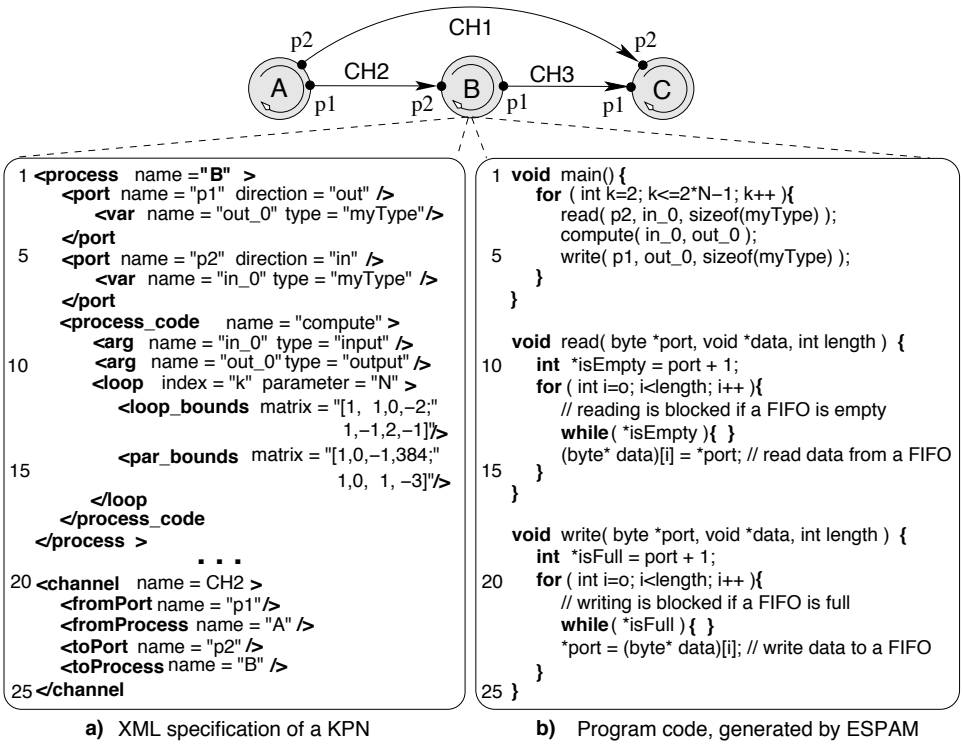
2.3 Parallelizing applications

Today, traditional imperative languages like C or C++ are still dominant with respect to implementing applications for SoC-based architectures. It is, however, difficult to map these imperative implementations, with typically a sequential model of computation, onto MP-SoC architectures that allow for exploiting task-level parallelism in applications. In contrast, models of computation that inherently express task-level parallelism in applications and make communications explicit, such as CSP [43] and Process Networks [51], allow for easier mapping onto MP-SoC architectures. However, specifying applications using these models of computation usually requires more implementation effort in comparison to sequential imperative solutions.

In Daedalus, we start from a sequential imperative application specification (C/C++) which is then *automatically* converted into a Kahn Process Network (KPN) [51] using the KPNgen tool [116]. This conversion is fast and correct by construction. In the KPN model of computation, parallel processes communicate with each other via unbounded FIFO channels. Reading from channels is done in a blocking manner, while writing to channels is non-blocking. We use KPNs for application specifications because this model of computation nicely fits the targeted media-processing application domain and is deterministic. The latter implies that the same application input always results in the same application output, irrespective of the scheduling of the KPN processes. This provides complete scheduling freedom when, as will be discussed later on, mapping KPN processes onto MP-SoC architecture models for quantitative performance analysis and design space exploration.

As mentioned before, KPNgen's input applications need to be specified as so-called static affine nested loop programs to allow for automatic parallelization of applications. As a first step, KPNgen can apply a variety of source-level transformations to these specifications in order to, for example, increase or decrease the amount of parallelism in the final KPN [94]. Subsequently, the C/C++ code is transformed into single assignment code (SAC), which resembles the dependence graph (DG) of the original nested loop program. Hereafter, the SAC is converted to a Polyhedral Reduced Dependency Graph (PRDG) data structure, being a compact mathematical representation of a DG in terms of polyhedra. Finally, a PRDG is converted into a KPN by associating a KPN process with each node in the PRDG. The parallel KPN processes communicate with each other according to the data dependencies given in the DG.

In Figure 2.2, a Kahn Process Network example is given in which three processes (A, B and C) are connected using three channels (CH1-3). Figure 2.2(a) shows the XML description of Kahn process B as generated by KPNgen. The XML describes both the topology of the KPN (i.e., how the processes are connected together, see e.g. lines 20-25) as well as the communications and computations performed by processes. In our example, process B executes a function called *compute* (line 8). The function has one input argument (line 9) and one output argument (line 10). The relation between the function arguments and the



a) XML specification of a KPN **b)** Program code, generated by ESPAM

Figure 2.2: A Kahn Process Network example.

communication ports of the process is given in lines 3 and 6. The function has to be executed $2 * N - 2$ times as specified by the polytope in lines 12-13. The value of N is between 3 and 384 (lines 14-15).

From the XML specification, Daedalus allows for automatically generating the C/C++ code implementing the behavior of each KPN process. This is done by the ESPAM tool, which will be discussed later on. Figure 2.2(b) shows, for example, the generated C code for process B (some variable declarations have been omitted). The code contains the main behavior of a process, together with the read/write communication primitives. In accordance with the XML specification in Figure 2.2(a), the function *compute* – which is derived from the original sequential application specification – is part of a loop that iterates $2 * N - 2$ times. For synthesis purposes, Daedalus also allows for generating the code for the read and write communication primitives, as shown in Figure 2.2(b). Currently, these primitives are implemented using polling and memory-mapped I/O. Note that the implementation of the write primitive is blocking since at implementation level FIFO channels are bounded in size.

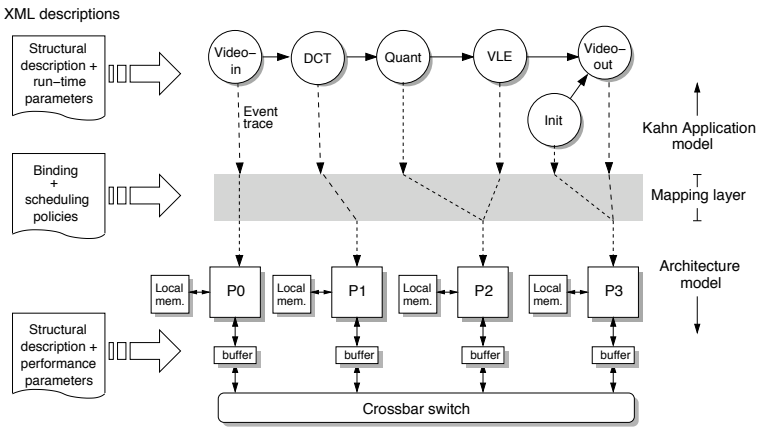


Figure 2.3: Sesame's layered infrastructure.

2.4 Design Space Exploration

Given a (set of) KPN application specification(s) – as for example generated by KPNgen or devised by hand – and the components in Daedalus' IP library, the Sesame system-level simulation framework [79] addresses the problem of finding a suitable and efficient target MP-SoC platform architecture. Figure 2.3 illustrates Sesame's layered infrastructure for the case in which a Motion-JPEG application is studied with a crossbar-based distributed-memory MP-SoC as target architecture. Sesame deploys separate application and architecture models, where an application model describes the functional behavior of an application and an architecture model defines architecture resources and captures their performance constraints. After explicitly mapping an application model onto an architecture model, they are co-simulated via trace-driven simulation. This allows for evaluation of the system performance of a particular application, mapping, and underlying architecture. Essential in this methodology is that an application model is independent from architectural specifics and assumptions on hardware/software partitioning. As a result, a single application model can be used to exercise different hardware/software partitionings and can be mapped onto a range of architecture models, possibly representing different architecture designs or modeling the same architecture design at various levels of abstraction.

For application modeling, the computational and communication behavior of the KPN application specifications are captured using *application event traces*. The computation and communication events in these traces typically are coarse grained, such as *Execute(DCT)* or *Read(channel_id, pixel-block)*. To generate the application events, the C/C++ code of each Kahn process is instrumented with annotations that describe the application's computational actions. In addition, Sesame provides read and write communication primitives that generate communication events as a side-effect. So, by executing the KPN model, each process generates its own trace of application events, representing the workload that is imposed on the underlying MP-SoC architecture model.

An architecture model simulates the performance consequences of the computation and

communication events generated by an application model. To this end, each component in the architecture model is parameterized with performance parameters specifying the latencies of computation events like *Execute(DCT)*, communication transactions, and memory accesses. This approach allows to quickly assess, e.g., different HW/SW partitionings by simply experimenting with the latency parameters of processing components in the architecture model: a low computational latency refers to a HW implementation while a high latency mimics a SW solution.

To bind application tasks to resources in the architecture model, Sesame provides an intermediate *mapping layer*. It controls the mapping of Kahn processes (i.e. their event traces) onto architecture model components by dispatching application events to the correct architecture model component. The mapping also includes the mapping of Kahn channels onto communication resources in the architecture model. The mapping layer has two additional purposes. First, the event dispatch mechanism in the mapping layer provides a variety of static and dynamic policies to schedule application tasks (i.e., their event traces) that are mapped onto shared architecture model components. Second, the mapping layer is also capable of dynamically transforming application events into (lower-level) architecture events in order to facilitate flexible refinement of architecture models [79].

The output of system simulations in Sesame provides the designer with performance estimates of the system(s) under study together with statistical information such as utilization of architectural components (idle/busy times), the contention in a system (e.g., network contention), profiling information (time spent in different executions), critical path analysis, and average bandwidth between architecture components. Such results allow for early evaluation of different design choices, identifying trends in the systems' behavior, and can help in revealing performance bottlenecks early in the design cycle. Here, the exploration process is also facilitated by the fact that system configurations (bindings, scheduling and arbitration policies, performance parameters, and so on) are specified using XML descriptions. Hence, different system configurations can be rapidly simulated without remodeling and/or recompilation.

As a result of the design space exploration with Sesame, a small set of promising MPSoC platform instances can be selected for automatic synthesis (see next section). Each selected platform instance is specified using two XML files. One describing the architectural platform at the system level, i.e. which IP components are used in the platform and how they are interconnected. And the other describing how application tasks are mapped onto the platform components.

2.5 System-level Synthesis

The system-level specifications that result from DSE – describing (the structure of) the application and platform architecture as well as the mapping of the former onto the latter – are given as input to the ESPAM tool for system-level synthesis [74]. To guarantee correctness-by-construction, ESPAM first runs a consistency check on the provided platform instance. This includes finding impossible and/or meaningless connections between system-level plat-

form components as well as parameter values that are out of range. Subsequently, ESPAM refines the abstract platform model to a 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, ESPAM generates program (C/C++) code for each programmable processor in the multiprocessor platform in accordance with the application and mapping specifications. To this end, it uses the XML specifications generated by KPNgen. In addition, ESPAM also provides the support for scheduling the code in the case multiple application processes are mapped onto a single processor in the platform. Currently, this code scheduling is performed statically.

The output of ESPAM, namely an RTL specification of the MP-SoC platform, is a model that can adequately abstract and exploit the key features of a target physical platform at the register transfer level. It consists of four parts (as shown in Figure 2.1): 1) a *platform topology* description defining in greater detail the structure of the multiprocessor platform; 2) *hardware descriptions of IP cores* containing predefined and custom IP cores used in 1). These IP cores, which are selected from Daedalus' IP component library, include programmable as well as dedicated processors, various memory components (FIFO buffers, random access memory, etc.), and different interconnects (point-to-point links, shared bus with various arbitration mechanisms, and a crossbar switch). For programmable processors, ESPAM currently uses PowerPCs and Microblazes since it targets the Xilinx VirtexII-Pro family of FPGA technology for prototyping the synthesized MP-SoCs. ESPAM also automatically generates custom IP cores needed as a glue/interface logic between components in the platform; 3) the *program code for processors* — as mentioned before, to execute the software parts of the application on the synthesized multiprocessor platform, and 4) *Auxiliary information* containing files which give tight control on the overall specifications, such as defining precise timing requirements and prioritizing signal constraints.

With the above descriptions, a commercial synthesizer can convert an RTL specification to a gate-level specification, thereby generating the target platform gate-level netlist (see the bottom part of Figure 2.1). At this moment, ESPAM facilitates automated MP-SoC synthesis and programming using the Xilinx VirtexII-Pro family of FPGAs and therefore uses the Xilinx Platform Studio (XPS) tool as a back-end to generate the final bit-stream file that configures the FPGA. However, our framework is general and flexible enough to be targeted to other physical platform technologies as well.

2.6 The Daedalus Design-flow Infrastructure

As discussed in the previously, the heart of Daedalus consists of the three core tools KPNgen, Sesame and ESPAM. In addition, Daedalus also features several supporting tools to improve the user-friendliness and deployability of the framework. This section provides a brief overview of the supporting infrastructure.

In Daedalus, most design information (e.g., structural descriptions of the application, architecture, and the mapping of the former onto the latter) as well as experimental results are described using XML-based descriptions. Daedalus therefore contains the Oracle Berkeley

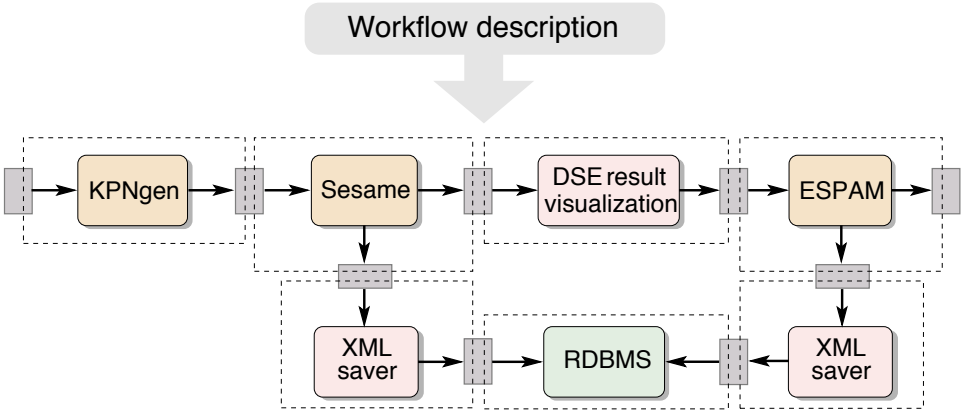


Figure 2.4: Daedalus' customizable work flow.

DB XML relational database management system (RDBMS) to store all information (models, parameters and results) related to designs and experiments. This RDBMS, together with its GUI, provide the designer with a powerful tool to e.g., explore and visualize the large amounts of data generated by Daedalus design space exploration. Moreover, it guarantees the reproducibility of experiments at all times.

The vision behind the Daedalus software infrastructure is that it should be open for integration of new tools as well as that it should allow for customization of the design flow. Therefore, the design flow (or tool flow) in Daedalus is composable and constructed from 'design-flow blocks'. These design-flow blocks, which are illustrated as the dashed boxes in Figure 2.4, are the tools that take part in the design flow together with their input- and output descriptions. The latter descriptions, illustrated by the grey boxes in Figure 2.4, provide information about what input/output data a tool consumes/produces and from/to where it reads/writes this data. This allows us to describe a design flow as a simple composition of the design-flow blocks, specified in the *workflow description*. For example, Figure 2.4 shows a design flow which includes a visualization block to visualize Sesame's DSE results and which stores both the DSE and ESPAM's prototyping results in the RDBMS (using the so-called 'XML saver' tool). Evidently, this composability of the design flow allows for easily adding new design steps to a design flow, or to customize design flows for specific design domains.

Control and monitoring software utilities have been developed to facilitate the process of setting up and executing experiments on the FPGA-based prototypes of MP-SoCs generated by Daedalus. Such utilities are necessary and very useful for: (i) conducting an effective and efficient design space exploration at implementation level on a narrow design space defined by Sesame; (ii) measuring real performance and cost numbers used for calibration of the Daedalus' high-level architecture models [80]; (iii) preparing real HW/SW demonstrators. The control and monitoring utilities include a configuration manager, an execution control panel, and an on-line monitoring console, all supported by a GUI which allows users unfamiliar with the FPGA prototyping board to perform experiments with the MP-SoCs.

2.7 Related Work

Systematic and automated application-to-architecture mapping has been widely studied in the research community. The closest to our work is the Koski MP-SoC design flow [52] and the SystemC-based design methodology presented in [40]. Koski provides a single infrastructure for modeling of applications, automatic architectural design space exploration, and automatic system-level synthesis, programming, and prototyping of selected MP-SoCs. The methodology in [40] supports automated design space exploration, performance evaluation, and automatic platform based system generation. But unlike Daedalus, [52] and [40] do not allow for automated parallelization of applications, nor design space exploration at application level. Both [52] and [40] require applications to be specified by hand in UML and SystemC, respectively.

Other examples of related work can be found in [95, 64, 20, 33]. However, these efforts are limited to processor-coprocesor architectures [95], only provide a limited degree of automation [64, 20], or do not provide an automated step towards RTL [33].

Companies such as Xilinx and Altera provide design tool chains attempting to generate efficient implementations starting from descriptions higher than (but still related to) the register transfer level of abstraction. The required input specifications are still so detailed that designing a single processor system is still error-prone and time consuming, let alone designing alternative multiprocessor systems. In contrast, Daedalus raises the design to an even higher level of abstraction allowing the exploration, design, and programming of multiprocessor systems in a short amount of time. For a more detailed discussion of related work and a classification of system-level design and synthesis tools we refer to [34].

2.8 Conclusion

In this chapter we presented the Daedalus framework that tries to bridge the so-called implementation gap between system-level platform specifications and the actual physical implementations of these platforms. To this end, Daedalus focuses on the design of multimedia MP-SoC platforms. As such, it provides an integrated and highly-automated environment for system-level architectural exploration, system-level synthesis, programming and prototyping. Such a framework offers remarkable potentials for quickly experimenting with different MP-SoC architectures and exploring system-level design options during the very early stages of design. In Chapter 8 we illustrate Daedalus' design steps and demonstrate its efficiency using a case study with a Motion-JPEG encoder application.

Chapter 3

Sesame: modeling and simulation

3.1 Introduction

As was shown in the previous chapter, the Sesame tool is used within Daedalus for efficient system-level modeling and simulation. Most of the topics discussed in the second part of this thesis (*Techniques*: Chapters 4, 5 and 6) either use Sesame directly, or with the modifications proposed in those chapters. In the current chapter we will give an overview of the Sesame environment to give the reader a general insight into the methodology and scope of Sesame. Although Sesame is the tool of choice in the context of the Daedalus toolflow, it may be equally useful in other contexts where system-level modeling and simulation is required. Therefore, in this chapter, Sesame will be presented as a generic modeling and simulation tool, but where appropriate, its connection to the Daedalus environment will be discussed.

The structure of this chapter is as follows. In the next section a general overview of the Sesame environment will be presented. A Sesame model consists of three parts: the application model, the architecture model and the mapping model: shown in Figure 3.2 as the top, bottom and middle model layer respectively. The discussion in the next section will follow the natural order from modeling the functional behavior of the system to modeling the system platform and hardware aspects with respect to non-functional behavior. Section 3.3 revisits these topics with a focus on the implementation details of different parts of the model in Sesame. Some modeling examples will be given to make the previously discussed topics more concrete. The final section discusses issues related to the periphery of Sesame: methods for efficient specification of the design space (Section 3.4) and interpretation of simulation results (Section 3.5).

3.2 Overview

One of the main design principles of the Sesame environment is to apply separation of concerns where possible. In this way, complicated problems are decomposed in smaller, easier to solve sub-problems, which (combined) still represent the original complicated problem. The most prominent separation of concerns in Sesame is the separation of functionality and

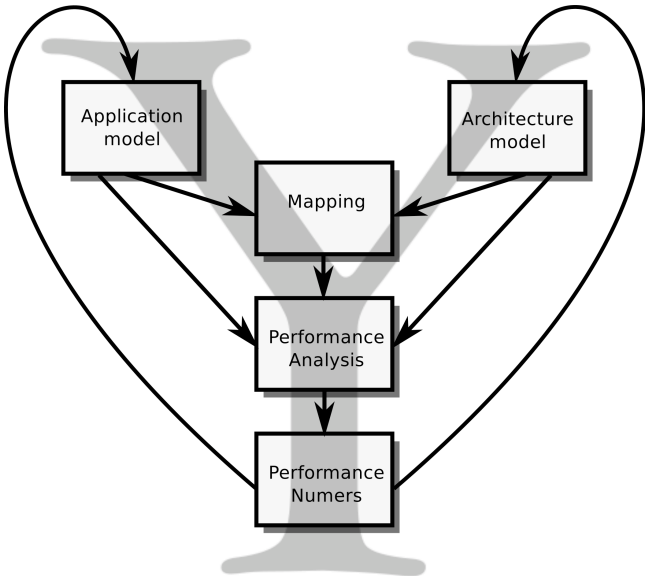


Figure 3.1: Y-chart design methodology

implementation, which is also proposed by the Y-Chart design principle ([54]). Sesame adheres to this principle by providing separate application and architecture models: the former captures the functional behavior of the system while the latter captures non-functional behavior. The non-functional behavior modeled by Sesame principally concerns system performance, but the model can be extended to account for additional objectives such as energy [81] or cost. The Y-Chart principle is schematically depicted in Figure 3.1 and consists of three stages: 1) application and architecture model creation, 2) mapping application to architecture and 3) model evaluation and feedback. The application model describes the functional behavior of an application in an architecture-independent way and it does not contain any architectural specifics such as the resource or performance constraints of architectural components. To obtain a rough estimate of its performance requirements, the application model can typically be studied independently from the architecture model using traditional software analysis tools such as software profilers. However, for the purpose of more detailed analysis, an architecture model is created that models for example system performance, power or cost. Sesame uses trace-driven cosimulation where the application model "drives" the architecture model by providing it with dynamic application information. A trace consists of events, and each event describes a single atomic action by the application model. The information contained in an event is typically a high-level description of a communication or computation action of the application: detailed functional information is omitted. For example, a data communication is described by its source, target and data size, but the actual values of the data are not included because they are not needed in Sesame's high-level non-functional architecture models. In the remainder of this section we will discuss how the three Y-Chart stages are represented in Sesame and give further details of the

interaction between the application and architecture models. Section 3.3 will give more details towards the implementation of Sesame.

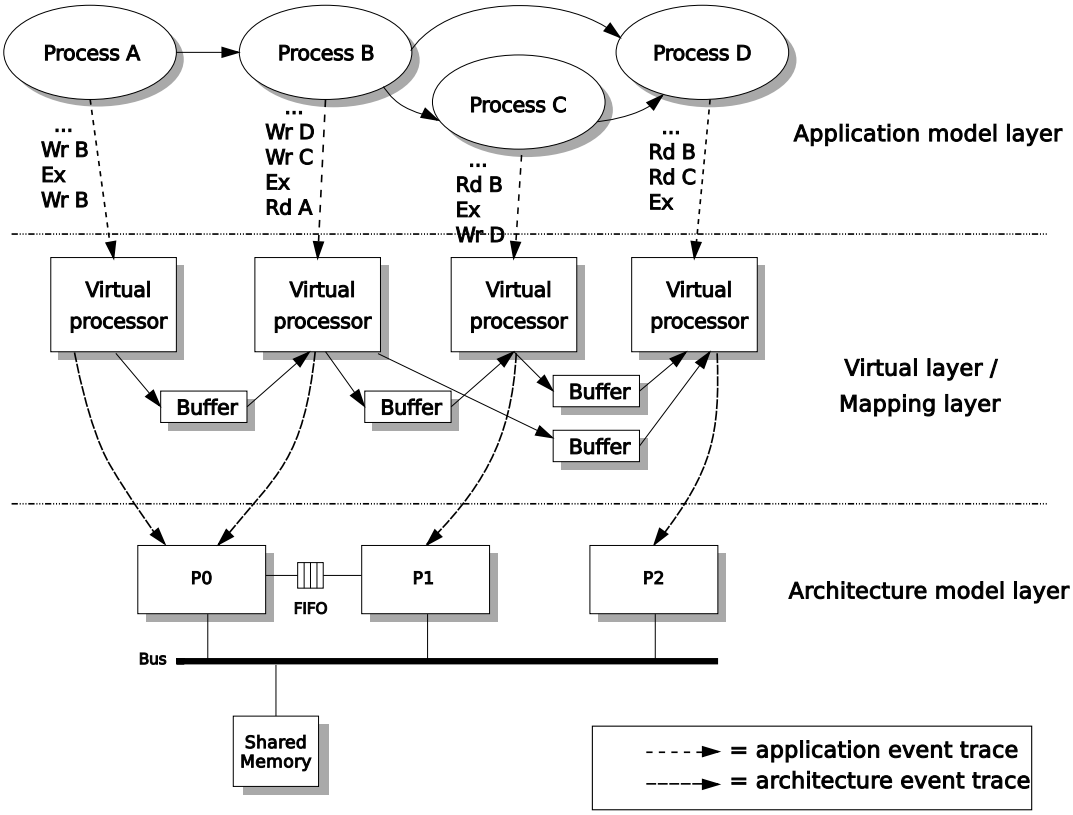


Figure 3.2: Overview of a Sesame model: the application, architecture and virtual layer

3.2.1 The application model

In Sesame we use Kahn Process Networks (KPN) as the preferred Model of Computation (MoC) to represent the application. This choice is motivated by the application domain that is targeted by Daedalus: multimedia and signal processing applications are often data-flow dominated and can be thought of as streams of data passing through a network of actors performing computations on individual data items (which may represent, in the example of an imaging application, pixels, lines, pixel blocks or image frames). A KPN is defined as a network of concurrently executing processes without access to shared memory, which communicate exclusively over Kahn channels. Kahn Channels are one-directional FIFO buffers of unbounded capacity without restrictions on the type of data that can be sent. In the top section of Figure 3.2, an example is given of a simple KPN with 4 processes and 4 channels. Internally, Kahn processes may be defined in any high-level language as long as the Kahn semantics are observed.

The KPN MoC is defined mainly by the semantics that are enforced on the channel communication. Processes are allowed to read to and write from any channel at any time. Write operations always succeed, since the channel buffers are of infinite capacity. However, reading from an empty channel causes a process to stall ("block") until data is written to this channel. A further condition is that there is no "test" operator to check data availability and therefore process execution is independent of the state of the channel. There are no restrictions on what a Kahn process is allowed to do internally, except (as mentioned before) that it is not a allowed to access memory that is shared with other processes: all inter-process communication has to occur explicitly over Kahn channels. Similar to the Synchronous Data Flow MoC (SDF), KPNs fall in the untimed MoC category: there is no concept of time passing, neither in terms of clock cycles nor in (simulated) time.

The result of the above semantics is that the KPN MoC is deterministic: the order of the tokens that are communicated over the channels does not depend on the execution order or execution time of individual processes. Determinism is a highly desirable property for the kind of models that we are considering: the system model (the combination of application and architecture) will produce the same output (for given input) regardless of the specific process scheduling or architectural (timing) characteristics. It also guarantees the validity of the event traces between the independently executing application and architecture models. Although determinism guarantees the functional behavior of the system, no assertions can be made yet about the system's non-functional behavior. For this purpose we need either an implementation on a real system or an architectural simulation that captures certain non-functional behavior.

3.2.2 The architecture model

The architecture model in Sesame (see the bottom of the three layers in Figure 3.2) is responsible for modeling latencies associated with the physical properties of the system. The Sesame architecture model is specified using the Pearl discrete event simulation language. The Pearl language was designed from the ground up to efficiently model multi-processor heterogeneous computer architectures at a high level of abstraction. On the one hand it achieves very good runtimes for entire system-level simulations running an application with a representative input workload. On the other hand, it sports an easy and compact concurrent programming style with powerful semantics that enables designers to express complex interactions between model components with relative ease. This means that complete system level models can be defined while the model development time remains modest, which is especially important in the early stages of development as architectural design decisions may be taken at the highest level and models may frequently need to be changed to test and evaluate new designs. The choice of programming language is therefore more than a trivial choice, since it can directly affect the overall design time of the system. In the following we discuss some of the main language features of Pearl. In Section 3.3.3, some more details and a simple model example will be given.

The Pearl programming language is characterized by two main features: 1) an object oriented approach for defining model components and 2) integrated primitives for commu-

nication and synchronization between model components. The functionality and behavior of model components is specified as a *class*, similar to a class definition in object oriented languages such a Java or C++. Also similar to object oriented languages is that each class has a private namespace for variable and functions in order to avoid variable name clashes. A light-weight class hierarchy scheme is available to group related classes into a family with a common superclass. The Pearl syntax will be familiar to C users, but the language has been designed to include only those features that are necessary for architecture modeling. In particular this means that there is no support for advanced data structures (other than classes for model components), no pointer support, and only a limited set of basic data types consisting of integers, floats and strings and one-dimensional arrays of these basic data types. In rare cases where a programmer needs more advance language features, it is possible to incorporate C functions which will be linked into the final simulation executable. At simulation time, classes can be instantiated as simulation components (also referred to as *modules*). Class instantiation is done automatically by the simulator prior to the start of the actual simulation according to an XML-based specification which is defined by the user (see Section 3.3.1). Since traditionally the components in a hardware system are fixed, Pearl does not support instantiation of classes after the initialization of the simulator. This assumption does not generally hold anymore with the advent of dynamically reconfigurable hardware platforms such as FPGAs. Indeed this topic is addressed in Chapter 6 where we show how Pearl can still be used to model this emerging type of systems.

The main difference with traditional object-oriented languages, however, is that each module maintains its own execution context, or thread, so that modules can easily express the naturally occurring parallelism between components in hardware. Modules can not modify data items in each other's address space. Instead, Pearl has Remote Procedure Call (RPC)-like primitives for communication and synchronization between modules. For example, the following synchronous function call statement:

```
moduleref ! functionname(parameters);
```

calls a method with the name `functionname` on a module pointed to by `moduleref` with the given `parameters`. The Pearl runtime system translates the call to a message which is sent to the remote module. The calling module will now be halted until the remote module has completed the function (using a `reply()` statement, which can optionally be used to return a value to the calling module). Note that in addition to this synchronous method call (that halts the calling module), there is also an asynchronous method call (that gives control immediately back to the caller) using the operator "`!!`". For both synchronous and asynchronous calls, the function will execute in the context of the remote module *and* at a time that is specified by the remote module (note the difference with a method call in a normal Object Oriented language). Therefore, the runtime system will store (completely transparent to the user) all function calls issued on a certain module in a dedicated message queue. The remote module is itself in charge of accepting calls to certain functions by issuing a matching `block` statement:

```
block(functionname);
```

The result of the `block` statement is that the remote module takes the function call message from its queue and executes it in its own thread context. When this function call exits, the issuing module is unblocked and both modules continue with the next statement in their code listing. If a module issues a `block(functionname)` statement before the `functionname` has been called, then the module will block until such a call is made.

The final important Pearl statement is used to indicate that a module is busy for a given number of clock cycles (eg. a processor component is busy executing some computational kernel or a bus is busy transferring some data):

```
blockt(time);
```

During the specified amount of `time` the module is suspended and can not perform other actions such as remote function calls; it will resume after `time` clock cycles have passed. The simulation runtime will execute without priority and non-preemptively any runnable module with simulated concurrency. Modules are runnable until they yield by blocking on remote function calls or with `blockt`. The Pearl simulation runtime maintains simulation time using the globally shared clock. Note that a system with multiple clock domains can be modeled by relative scaling of latencies (small errors introduced by scaling are insignificant at Sesame's high level of abstraction). When there are no more runnable modules, then the discrete-event simulation engine will increment global clock to the earliest resume-time for a suspended module and resume execution of that module. Valid termination occurs when either all modules have finished execution, or when all modules are blocking in unmatched communication primitives: a remote function call primitive without a matching `block`, or vice versa.

In Pearl, the remote procedure call primitives perform – in a completely transparent way – relatively complex communication and synchronization transactions between modules. Furthermore, the compact syntax of the primitives ensures that simulation code in Pearl is easy to write, which is an essential property for early model development. As we will see in the example in Section 3.3.3, Pearl code is also easy to read, since the combination of RPC primitive and function name completely determines the state a module at any given time. Code modularity (and thereby code reuse) is further promoted by the use of classes to define simulation components. The set of remotely callable functions in a class acts as an explicit module interface: a module can be easily be replaced by another module as long as it implements the same functions. All in all, Pearl is a suitable language for the creation of simulation models that support design decisions in the early stages of MPSoC design where models typically have to be created and modified in short time frames. We shortly refer to the work in [104] where we transferred the functionality of the compact Pearl communication primitives to the more widely-used and standardized SystemC modeling language. However, this approach suffered from performance limitations of the reference SystemC implementation, thus favoring the use of the custom Pearl language within Sesame to improve model execution speed.

3.2.3 Mapping

Due to the separation of application and architecture, an explicit interface is needed to connect the two models. Sesame performs trace driven simulation where the events collected in the untimed application model are streamed to the timed domain of the architecture model. The events are abstract representations of the actions performed by the application model and are initially chosen to represent only coarse grain events. At this coarse grain level of abstraction only channel communication and computation of large kernels or functions are represented by events. In Sesame, these event traces are not immediately consumed by the architecture model, but instead they pass through a so-called "virtual layer" (sometimes called *mapping layer*). The virtual layer serves multiple functions towards mapping, synchronization and scheduling of events before they are passed on to the architecture model. The virtual layer exists in the same timed domain as the architecture model and is in fact part of the same simulation model, however, we treat it here as a separate entity from the architecture layer as its functionality is distinct from the architecture model. The virtual layer is composed of virtual processors and virtual channels that are connected according to the same topology as the application model (see Figure 3.2). For this reason the virtual layer can initially be generated automatically (this process will be further explained in Section 3.3.4). The virtual processors read the event traces that are generated by the application model and forward them (according to the mapping specification) onto architectural model components such as processors. Before forwarding events, however, the virtual layer takes care of modeling several important issues with regard to synchronization of the events as well as some issues that are optional for certain types of models. In the following we will shortly discuss the different functions of the virtual layer.

Synchronization modeling

According to the Kahn MoC, communication takes place in a world where there is neither time, nor a limitation on the channel size. Of course, in any practical implementation of a system this assumption does not hold and (apart from physical wire and switching delays) the speed of communication is limited by the storage capacity of the transport medium (bus width, packet size or memory size, etc). This limited capacity may lead to problems when pairs of producing and consuming nodes are not producing and consuming data at the same rate. For example consider the case where a receiving node consumes packets slower than the production rate of packets; eventually the (limited) buffer capacity will fill up and the producing node will be forced to wait until storage space becomes available. Similarly, a consuming node may be blocked on an empty channel when the production rate is slower than the rate of consumption. Within the context of Sesame, the time spent waiting for empty or completely saturated communication channels is referred to as *synchronization latency* and it is modeled by the virtual layer. Note that latencies associated with architecture specific parameters such as bus width, throughput or latency are modeled in the architecture model, as will be shown in the next subsection.

The virtual processors and virtual channels can be seen as the real-world representatives

of the Kahn applications and channels: they *do* in fact model timing consequences of waiting for data (when a channel is empty) or waiting for free space (when a channel is full). In the case of a virtual processor this works a follows: for each communication event (RD or WR) the virtual processor checks whether the data is available or whether there is sufficient storage space for a write event to complete. If necessary, the virtual processor will halt until the respective condition is met. The amount of time the virtual processor is halting is referred to as a synchronization latency. When the condition is met, only then is the communication event forwarded to the architecture model to model physical latencies that may be associated with communicating data over a certain medium (communication network or memories for example). Note that the synchronization latency indirectly includes architectural latencies (which are modeled in the architecture layer). For example, when a write is blocked on a full channel, then the total write synchronization latency includes both waiting for the initiation of the corresponding read event *and* the latency modeled by the architecture model to complete the read (including e.g., bus and memory latencies).

This reveals a second case of separation of concerns in Sesame: synchronization latencies are modeled separately from the hardware communication latencies. This turns out to be particularly useful when the synchronization behavior of certain tasks needs to be changed, as we will see in Section 3.3.4.

Deadlock prevention

For completeness, we list the prevention of deadlock as one of the tasks of the virtual layer, although this is actually a side-effect of the synchronization modeling in the previous chapter. Deadlocks could occur by chance when two dependent application processes are mapped onto the same processor, depending on the order of the trace events. Such deadlocks need to be prevented, since a deadlocked model in Sesame satisfies the model termination condition and thus no useful information can be derived from a deadlocked simulation run. For example in Figure 3.2, assume process A writes to process B, but the communication buffer in the virtual layer is full. If the write event from A had already been sent to the architecture model, then this write event would block processor P0 and deadlock would occur, since the read event from process B (which would free the required buffer space) has to execute on P0 too. The synchronization mechanism from the previous section resolves this problem by delaying forwarding the write event to P0 until sufficient space is available in the buffer.

Model Refinement

An important consideration in Sesame is to support mixed-level modeling and gradual model refinement. In a typical design case study it may be necessary to model some parts of a system in more detail, for example because their effects on global system performance can not be captured accurately by the high-level model. Sesame enables a designer to start with a high level model and gradually refine certain parts of the model as needed. The resulting model mixed model contains components specified at different layers of abstraction, but still runs as a single simulation entity. Moreover, by gradually refining more and more

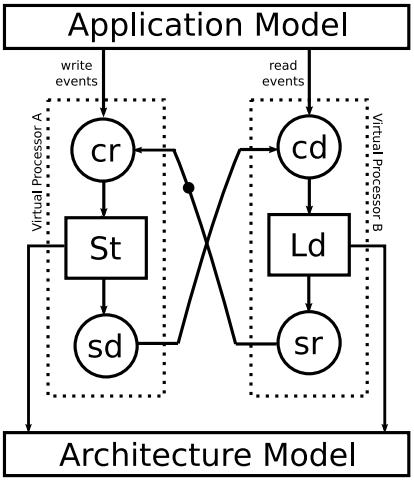


Figure 3.3: Virtual processors with refined synchronization behavior

components towards a lower level of abstraction (specifying ever more details about the system), the model will come closer to a level of specification which is a sufficient starting point for synthesis using traditional methods. As we explained in the previous chapter, Daedalus is more geared towards system synthesis based on high-level specifications, but the refinement to synthesizable specification is still an often used design trajectory.

However, when architecture model components are refined, the level of granularity of the events produced by the application model may not be sufficient to "drive" the underlying refined architecture model components. Because of the relatively high abstraction level of the atomic application events (a sequence of Rd, Wr, Ex-events which is executed in strict trace order), information about the application model is lost that can be needed for the refined architecture components, eg: control flow information, event dependencies, etc. An example is an image processing component that works on frames, but the refined architecture component works on pixel lines and has only local memory to contain a few pixel lines. The processing component should now read the frame data using multiple data transfers, which in turn need multiple synchronization events. Such information was not present in the original application event trace. One option is to refine the application model together with the architecture model to generate events containing this additional information, but this would break the separation of application and architecture.

To resolve this problem (while maintaining the benefits of separate application and architecture models), Sesame provides dataflow based trace transformation techniques in the mapping layer. For this purpose, the Virtual Processors in the mapping layer are extended with automatically synthesized, modified versions of Synchronous Dataflow (SDF) and Integer Data Flow (IDF) graphs which act as executable, yet abstract representations of the original application processes. The nodes in these data flow graphs represent synchronization events such as Sr (Signal Room), Sd (Signal Data), Cr (Check Room), Cd (Check Data) or architectural events such as Ld (Load data) or St (Store data). While synchronization nodes

are typically connected to each other to express dependencies and available parallelism, Ld and St nodes directly connect to architectural model components (Figure 3.3). When a Ld or St nodes fires, events are forwarded to the architecture model, where the latency of the event is modeled, which in turn completes the firing action of the node. Note that this so-called token-exchange mechanism between the virtual and architecture layer executes in the timed Pearl simulation domain, which yields timed dataflow models, resulting (in contrast to pure SDF-models) in a semi-static scheduling. In order to support more complex refinements, eg. for Kahn processes containing loops and conditional code within Kahn processes, special IDF models have been defined.

As an example we use the default synchronization behavior that was explained in the previous section. It is in fact also a form of trace refinement: the Rd and Wr events have been refined in a sequence of smaller communication and synchronization events:

$Rd : Cd \rightarrow Ld \rightarrow Sr$

$Wr : Cr \rightarrow St \rightarrow Sd$

Figure 3.3 shows the SDF corresponding to this refinement for two virtual processors A and B. Note the dependencies between the synchronization events and the initial token that is required for the first Cr (check room) event. The token exchange mechanism (consisting of the refined trace events) with the architecture model is depicted as a line that connects the Ld and St SDF nodes to model components in the architecture layer. This type of synchronization behavior is currently the default for any Virtual Processor in Sesame and further model refinement is optional.

To conclude, we can say that the combination of SDF and IDF refinement models support a smooth transition between abstraction levels both for communication (and synchronization) and execution events while keeping the application model unchanged. In this way refined architecture models can be used that capture intra-task level parallelism, various communication policies as well as pipelined processor components. For more details and examples we refer to [78].

Scheduling

As explained previously, virtual processors are the "timed representatives" of application processes and the application mapping definition determines to which architectural component the virtual processors forward trace events. Simultaneous events from different processes will compete for resources when they have been mapped onto the same architectural component (e.g., Processes A and B in Figure 3.2). In the standard non-refined Sesame system model, virtual processors (after modeling synchronization (Section 3.2.3) directly forward events to the architectural component. As a result of Pearl's inter-process communication semantics based on message passing, simultaneous events will be queued at the processor and architectural timing consequences are modeled in a first-come first-serve (FCFS) manner by the processor component. The FCFS scheduling policy for application events is a suitable model for systems supporting dynamic scheduling of tasks on the execution units. This may for example be a processor running an operating system that implements pre-emptive threads (e.g., POSIX pthreads), user threads (light weight processes), OS-level

processes or some kind of job scheduling middleware. Note that the particular scheduling technology or scheduling unit (thread, process, etc.) is irrelevant at the high level of abstraction of the unrefined Sesame model: we simply assume the resource will schedule a task as soon as all earlier tasks have been processed.

There are cases, however, where the scheduling behavior of architectural resources differs significantly. For this purpose, a scheduler model component has been defined that can be used in the virtual layer to enforce specific user defined scheduling policies. The scheduler consults an external policy component to identify when tasks are to be forwarded to the resource. In this way a wide range of different scheduling policies can be implemented; examples include fixed predefined (*static*) schedules, time slicing or priority scheduling. The only limitation of the scheduler is that it can not split the application events, e.g. in order to implement a pre-emptive scheduling after an event has already been forwarded to the architecture. Therefore, application events have to be considered atomic, unless they have been explicitly refined (Section 3.2.3). The scheduling policies are defined as separate components to promote reusability and to separate the event scheduling from policy definitions. Schedulers in Sesame will be further discussed in Chapter 5, where they are used, for example, to model multi-application workloads.

3.3 Implementation Aspects

In the previous sections we showed Sesame's distinct application and architecture model layers, their important characteristics and their interaction. In this section we focus more on the implementation aspects of the models and we again follow the structure of the layers in a Sesame model: application model, architecture model and finally the one connecting both: the virtual (or mapping) layer. But first we discuss Sesame's model description language called YML (Y-chart modeling language), which is used to describe both the structure (and *topology*) of the various models and to specify the initialization values of each component.

3.3.1 Model specification

YML is a specialized dialect of the more commonly known XML (Extensible Markup Language). YML has the advantage of being both a machine readable as well as human readable format (at least for smaller specifications). Each YML element defines a component or topological relationship in the model. For example we use elements with tag-names `node`, `network` or `link` to describe respectively a component, a group of components or a relationship between components. XML attributes specify the additional required details for each type of element. All element types and their optional and obligatory attributes have been formally defined in a schema (XSD) specification. By using the schema, it is easy to use existing parsing solutions, which not only simplifies the implementation of the Sesame tools, but also greatly enhances the possibility to exchange specifications with external tools which may have been developed by third parties.

XML is defined as a hierarchical specification language where certain elements (according to the relevant schema) may contain other elements, which in turn contain other elements, etc. This way, what is essentially a tree-like structure is defined, which is very suitable to describe applications or architectural topologies. In Sesame's YML, the starting element of a model is always a `network` element in which all other model components are contained. As an example, see Figures 3.4 and 3.6, where a very compact application and architecture model is specified. The attributes of the root-level network specify the URI for the YML schema definition, a name and the type (`class`) of the network to indicate whether this is an application (`KPN`) or architecture model specification (`net`). Additional `property` elements describe additional information such as the location of the compiled application code (Fig. 3.4 line 8-10, or Fig. 3.6 line 14-15) which will be used respectively by the application or architecture simulation runtime. In the following we will describe the implementation-level details of model components at each of the Sesame layers and their specification in the corresponding YML file.

3.3.2 The application model

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <network xmlns="http://sesamesim.sourceforge.net/YML"
3           name="application"class="KPN">
4  <property name="library" value="libmjpeg.so"/>
5  ...
6  <node name="ND_2" class="CPP_Process">
7   <property name="class" value="ND_2"/>
8   <property name="header" value="aux_func.h"/>
9   <property name="header" value="ND_2.h"/>
10  <property name="source" value="ND_2.cpp"/>
11  <port name="IG_1" dir="in">
12   <property name="type" value="struct TBlocks"/>
13  </port>
14  <port name="OG_1" dir="out">
15   <property name="type" value="struct TBlocks"/>
16  </port>
17 </node>
18 <node name="ND_3" class="CPP_Process">...</node>
19 ...
20 <link innode="ND_2" inport="OG_1" outnode="ND_3" outport="IG_1"/>
21 <link innode="ND_3" inport="OG_1" outnode="ND_4" outport="IG_1"/>
22 ...
23 <property name="operation:op_initVideoIn" value="0"/>
24 <property name="operation:op_mainVideoIn" value="1"/>
25 <property name="operation:op_mainDCT" value="2"/>
26 ...
27 </network>
```

Figure 3.4: Sample application XML

Figure 3.4 lists part of the YML specification for a MJPEG application that has been automatically derived from sequential code by the Daedalus PNGen tool. The node elements represent Kahn processes of the class `CPP_Process` (as indeed we will see shortly they

are implemented in C or C++), and a `name` attribute. The `port` elements (line 11 and 14) indicates that the process contains an in-bound port and an outgoing port. The ports form the anchor points for Kahn channels which communicate data of the type as specified by the port (in this case 8x8 pixel macroblocks: `struct TBlocks`). The `link` property (line 20) connects the out-port of process `ND_2` with an in-port of process `ND_3`, thus connecting the Kahn channel between the two processes. Figure 3.5 lists the C++ implementation of the

```
1 #include "ND_2.h"
2
3 ND_2::ND_2(Id n, ND_2_Ports *ports) :  ND_2_Base(n, ports) {}
4
5 void ND_2::main() {
6   struct TBlocks in_0ND_2;
7   struct TBlocks out_1ND_2;
8
9   for( int c0 =  ceil1(0); c0 <=  floor1(7); c0 += 1 ) {
10    for( int c1 =  ceil1(0); c1 <=  floor1(15); c1 += 1 ) {
11      for( int c2 =  ceil1(0); c2 <=  floor1(7); c2 += 1 ) {
12        ports->IG_1.read( in_0ND_2 );
13
14        mainDCT(&in_0ND_2, &out_1ND_2) ;
15        execute("op_mainDCT");
16
17        ports->OG_1.write( out_1ND_2 );
18      }
19    }
20  }
21 }
```

Figure 3.5: Sample application process code

`ND_2` process. Its regular structure of affine nested loops and slightly odd variable names is due to the fact that this code generated by PNGen. In general, a Kahn process can have many in- and outgoing channels and there is no restriction on the number or ordering of reads and writes, the number of produced execute events, or the naming conventions for nodes and ports. Channels are accessible from the process code by means of the ports structure which is inherited from the (automatically generated) base class `ND_2_Base`.The runtime system will automatically instantiate and initialize the processes and channels before it proceeds to execute the process network. The ports connect processes by means of Kahn channels according the topology defined by the `link` elements in the YML. Input ports and output ports have, respectively, `read()` and `write()` functions defined on them, where a read is blocking if the channel is empty and the write is always non-blocking due to infinite capacity of the channels.

Note that the process network is a *functional* application: in the case of `ND_2` the DCT function operates on real data and the result of the given KPN network is a sequence of compressed MJPEG images. The DCT function (line 14) is called on the input macroblock (line 12) and writes the result to the next process which will perform quantization (line 17). Important to note is the execute statement at line 15 which has the sole purpose of generating a trace event (which has identifier 2 as specified in the YML (Fig. 3.4 line 25). The port

`read()` and `write()` functions generate trace events as a side-effect of their communication actions, so these do not have to be specified explicitly. The trace events from each process are captured as an ordered stream of events which is forwarded –first to the components in the virtual layer– and then to the architecture model, where timing consequences of the events are modeled.

3.3.3 The architecture model

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <network xmlns="http://sesamesim.sourceforge.net/YML"
3           name="architecture"class="net">
4  <node name="processorX" class="pearl_object">
5   <property name="class" value="processor"/>
6   <property name="header" value="processor.ps"/>
7   <property name="source" value="processor.pi"/>
8   <property name="init" value="out0,[359,0,0,0,154,1,45,0,4,154]"/>
9   <port name="out0" dir="out">...</port>
10  <property name="template" value="vproc">
11   <network name="processorX" class="net">
12    <node name="vproc" class="pearl_object">
13     <property name="class" value="vproc"/>
14     <property name="header" value="vproc.ps"/>
15     <property name="source" value="vproc.pi"/>
16     <property name="init" value="trace,#&lt;vchannel&gt;"/>
17     <port name="out0" dir="out">...</port>
18     <port name="trace" dir="out">...</port>
19     <port name="channels" dir="out">...</port>
20     <link innode="vproc" inport="out0" outnode="this" outport="channels"/>
21    </node>
22    <port name="channels" dir="out">...</port>
23   </network>
24  </property>
25 </node>
26 <node name="bus" class="pearl_object">
27  <property name="class" value="bus"/>
28  <property name="header" value="bus.ps"/>
29  <property name="source" value="bus.pi"/>
30  <property name="init" value="out0"/>
31  <port name="out0" dir="out">...</port>
32 </node>
33 ...
34 <link innode="processorX" inport="out0" outnode="bus" outport="in1"/>
35 <link innode="bus" inport="out0" outnode="memory" outport="in0"/>
36 </network>
```

Figure 3.6: Architecture YML

In Figure 3.6 we list a part of the YML specification for an architecture description of a multi-processor system connected to a shared memory. The `node` elements in the YML specify which simulation components will be instantiated in the Pearl simulation language. In Figure 3.7 a slightly simplified version of the Pearl code for each of the main components is given. In the YML specification, the element on line 4 starts the definition of `processorX` and subsequent lines list its source code files. The `init` string on line 8 gives the initial-

ization values with which the variables b and operations in the processor object will be initialized (Fig. 3.7 line 3-4). The first of these variables is a reference with the name b that points to a bus object. According to the init string (Fig. 3.6), this reference is initialized to whatever the out1 output port is connected. On line 34 the link element specifies that it connects to an input port on the bus object. Thus, a synchronous or asynchronous Pearl method call on b, means a call to the bus object defined on line 26 of the YML. This is a clear example that the model topology is specified separately (in the YML file) from the model behavior (in the Pearl source files). This facilitates model component reuse and makes it easier to re-run experiments (without re-compilation) by simply changing the parameters in the YML files. Lines 10-24 in the YML specify a virtual processor template, from which the components in the virtual layer are constructed (we discuss this further in the next section).

```
1  class processor
2
3  b: bus
4  operations: [10] integer
5
6  read: (address: integer,
7         size: integer) -> integer
8  {
9    data: integer
10   data = b ! read(address, size);
11   reply(void);
12 }
13
14 write: (address: integer,
15          size: integer) -> integer
16 {
17   b ! write(address, size);
18   reply(void);
19 }
20
21 execute: (id: integer) {
22   blockt(operations[id]);
23 }
24
25 {  // class main routine
26 while(true) {
27   block(any);
28 }
```

```
1  class bus
2
3  m: memory
4
5  read: (address: integer,
6         size: integer) -> integer
7  {
8   blockt(5*size);
9   reply(m ! read(address));
10 }
11
12 { // class main routine
13  while(true) { block(any) }
14 }
```

```
1  class memory
2
3  read: (address) -> integer
4  {
5   blockt(3);
6   reply( get_value(address) );
7  }
8
9  { // class main routine
10  while(true) { block(any) }
11 }
```

Figure 3.7: Pearl model component code samples for a simple architecture

We offer the following short description of the model behavior that is expressed by the Pearl code in Figure 3.7. All components are indefinitely waiting for calls to their methods (shown in the code listings starting from lines 25, 12 and 9 respectively). Whenever the processor component receives a read-event from the application model, a virtual processor from the mapping layer will call the processor read() method to model the timing consequences of the event. The read() method will pass (line 10) the call to the bus, which models some transfer latency (which depends on the size of data being communicated before passing the call to the memory (bus class line 8). The memory will retrieve the data located at address

and the cascade of `reply` statements will return the data to the processor. Contention on the bus component is implicitly modeled: if two (or more) processor components simultaneously call methods on the bus, then the bus will first accept one of the calls and the others are stored in a message queue. The continuous wait-loop (bus class line 11) will process the queued calls one after the other, and the calling processors will incur a delay, since they made synchronous calls to the bus. Note that since the Pearl architecture model typically does not model functional behavior (just the timing consequences of the events mapped onto the architecture components), the `data` items passed between the components do not need to contain real values. To model an execute event (a virtual processor calls the `execute` method on the processor), the processor will simply block for an amount of time as was specified by the array in the YML file (Fig. 3.6 line 8). See Chapter 4 for more information on how these latency numbers are obtained.

3.3.4 The virtual/mapping modeling layer

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <mapping xmlns="http://sesamesim.sourceforge.net/YML_Map"
3           side="source"name="application">
4  <mapping side="dest" name="architecture">
5   <map source="A" dest="processorX" dtmpl="vproc"/>
6   <map source="B" dest="processorY" dtmpl="vproc"/>
7   <map source="A.out0->B.in0" dest="memory" dtmpl="vmembus"/>
8  </mapping>
9 </mapping>
```

Figure 3.8: Mapping YML

As was discussed before in Section 3.2.3, the virtual layer in Sesame resides between the application and architecture model layers and provides important functionalities to the model. Here we focus on the implementation aspects and show how the virtual layer can be automatically created using the template mapping mechanism. Figure 3.8 shows the YML file that specifies the (user-defined) application-to-architecture mapping for the example architecture in the previous section. The `mapping` elements construct a context where the `source` attribute refers to the application and the `dest` attribute to the architecture. Subsequently, a `map` element maps maps processes and channels to resources in the architecture model (lines 5-7). Note that the `map` element has an additional `dtempl` attribute which specifies one of the architectural *templates* associated with an architectural component.

The automatic procedure to generate the mapping layer will instantiate (for every Kahn process and channel) an object in the virtual layer corresponding to the template mapping. This is clarified in Figure 3.9 where the three Sesame modeling layers are shown. Note that every processor in the architecture model has at least one template associated with it (e.g. in architecture YML Fig. 3.6 lines 10-24). For each Kahn process, a Virtual processor (in the virtual layer) is automatically instantiated from the template as specified by the application mapping. So, although Process B and C are mapped onto the same processor P1, they ac-

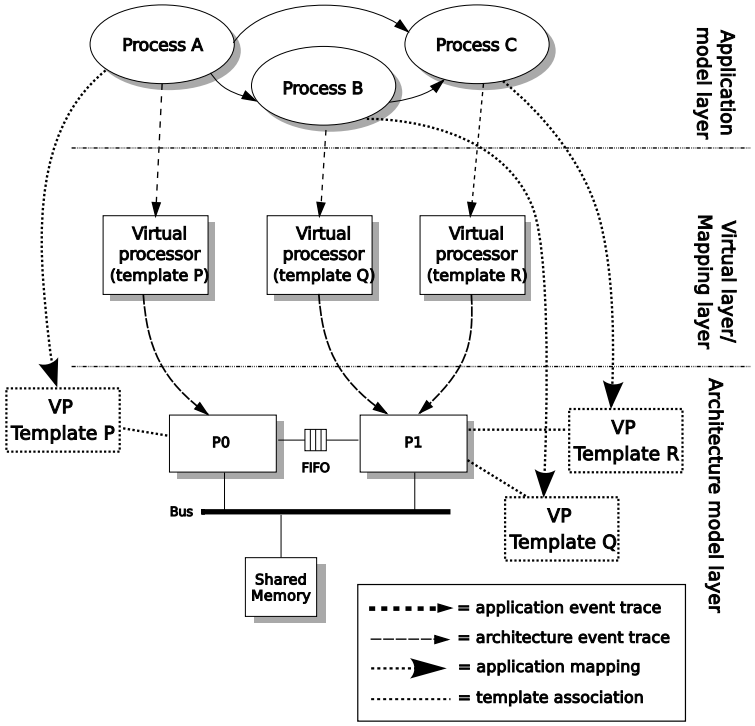


Figure 3.9: Template mapping strategy in Sesame (templates and mapping for Kahn channels omitted)

tually will be represented by different virtual components (Q and R respectively). Similarly (although not shown in the figure), Kahn channels map onto virtual channel templates, so that the virtual layer can be fully automatically derived from the information in the mapping and architecture YML files. The template mapping mechanism makes the model more flexible, since each template may express different behavior at the virtual layer, such as for example different synchronization behavior or different model refinements (as discussed in Sections 3.2.3 and 3.2.3).

3.3.5 Graphical user-interface

A GUI has been developed in order to simplify the process of creating application and architecture YML specifications and to provide an integrated development environment (IDE) for use with Sesame. The current GUI is implemented as a plugin to the widely-used Eclipse IDE: a screenshot is shown in figure 3.10. The middle window panes show the application model (top) and the architecture model (bottom) as a graph. A model designer can change the models by selecting a tool from the palette (on the right of the graph) and drawing new nodes or connections. Pre-defined modeling components can be dragged from a library (just under the palette) and newly defined components can be stored in the library for later use. The virtual layer models and the templates from which the virtual layer is built can be edited

from a separate view (not shown in the figure). On the left side is the project manager (top) and a window to specify application-to-architecture mapping (bottom). The bottom right window shows all the properties of the currently selected node or link from the application or architecture windows. Properties can be edited and new ones can be added. The Eclipse environment can also be used to call the back-end tools to build and run the model, such as the automatic virtual layer generator, compilers and the simulator. The application shown is the MJPEG application, which we have used as an example in the previous sections. The architecture is a 4-processor crossbar based architecture: the crossbar component is shown on the right. It connects to 4 processors (note the CPU picture), by means of a bus and a local storage memory. The two components on the left of each processor are a scheduler and its scheduling policy object.

3.4 Setting up the Design Space

In the previous sections we have shown how Sesame can be used to evaluate single design points: systems consisting of an application and an architecture model with a given configuration as specified in YML. In order to make design decisions and trade-offs, designers are typically interested in comparing results between different systems and system configurations. As we have shown before, it is easy to manually modify Sesame models and their parameters – either directly in YML, or using the graphical editor. However, even a small amount of manual effort per design point is infeasible for large design spaces with thousands or millions of design points. A designer is typically interested in a specific part of the design space as delimited by some boundary parameters such as a the total number of processors in the system, a finite set of alternative components, the total amount of distributed memory, etc. In order to provide a direct path from the parameter space to the objective space (as shown in Figure 1.2 of Chapter 1), we should be able to automatically generate model specifications for all design points that fall within the part of the design space delimited by the boundary parameters. In this section we will discuss two methods in Sesame to generate a set of YML specifications that represents a specific (area of) the design space. Sesame can then be iteratively run for each specification, thus evaluating all design points. We note that such techniques are not only useful for exhaustive design space search, but also for other design space search algorithms such as heuristic search algorithms (which is the topic of Chapter 7.

The design space generation approaches discussed here are quite generic and similar approaches have probably been used in the context of other modeling and simulation tools. Since each tool provides its own methods to enumerate parts of the design space, much time and effort is unnecessarily spent to create such scripts and programs. Unfortunately, standardization of even single system-level design point specifications has not yet been realized yet by the EDA community. And as far as we know, there are no standards at all for specifying a range of design points (or parts of a design space). This is unfortunate, since design space specification and generation is required in many embedded system design tools, both commercial as well as research. Lack of such standards hinders the development of new

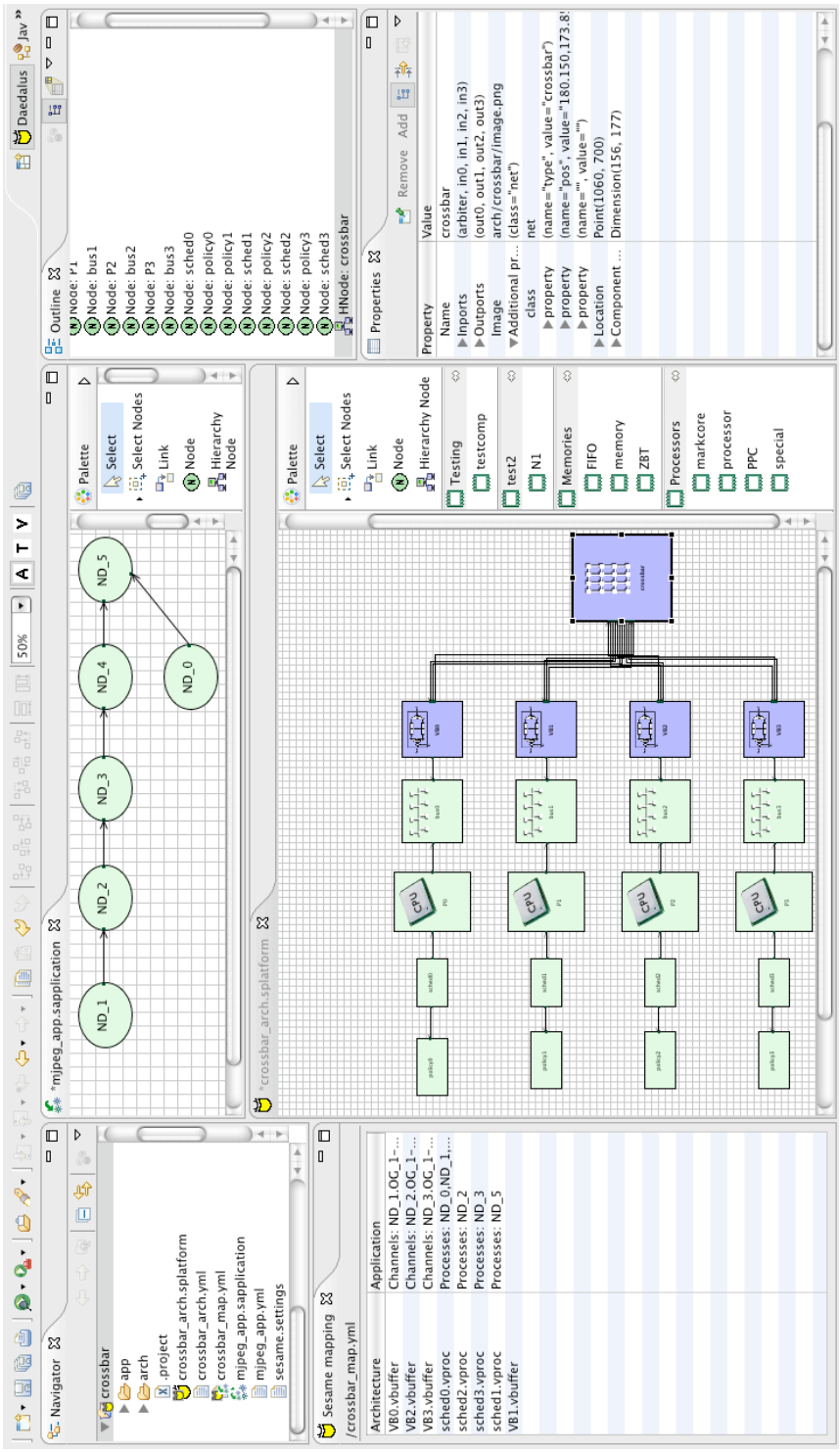


Figure 3.10: Eclipse plugin for creating Sesame model specifications

methods and tools and also makes comparison of existing work more difficult. However, in recent years, things are slowly changing with e.g. the emergence IP-XACT from the Spirit Consortium. While different tools typically require a different specification format, a separation can be made between front-end and back-end specification formats: the front-end uses the design space parameters to generate abstract representations of design points; the back-end translates the abstract design point to the required format (in our case YML). This separation is recognized by IP-XACT in the sense that it is being promoted both not only as a standard to be used natively by tools, but also as an abstract interchange standard that needs to be translated to/from proprietary specification formats.

In the remainder of this thesis we will show various case studies where small and large design spaces are to be evaluated. In the following we show two design point enumeration methods that are usable with Sesame: the meta-platform approach (which we commonly use) and the generator-approach (which is still under development). A hybrid approach, which combines some of the advantages of both methods, was developed in [48].

Meta-platform approach

The first approach towards instantiating a design space is based on the idea of a *meta-platform* specification. In such a specification all components and parameters that could be used by any of the design points are merged into a single system specification. Individual design points are then defined uniquely by the application-to-architecture mapping specification. This is possible since in Sesame it is typically the case that architecture model components onto which nothing has been mapped (and therefore are not receiving trace events), do not influence the simulation results. Consider for example a design space which studies an architecture which has 1 to 4 processors which can be of 2 different types connected by point-to-point, crossbar or a bus network. In this case we could instantiate an architecture specification containing 8 processors (one of each type) and the 3 different interconnects. It is now relatively easy to generate automatically an exhaustive list of mapping specifications that map application processes to all possible combinations of processors and application channels to different interconnects. If, for example, the mapping specification puts all application tasks on one processor, then the remaining processor model components will simply remain inactive. If part of the exhaustively generated mappings constitute invalid design points, then they can easily be filtered out. The result is that the problem of generating the design space has been reduced to the simpler problem of generating mapping specifications and a meta-platform specification.

This approach has a number of drawbacks, the most notable of which is that it scales badly: even in small exploration case studies the number of components that have to be included in the meta-platform may be very large. Consider an example where the objective is to find optimal memory sizes within the range of 2Kb to 64Kb: with a discretization of 2Kb, a system with 4 memories would need to instantiate $(32+1)*4 = 132$ memories. The number of components multiplies further if the same component has additional properties, eg. if there are memories with different speeds, different number of ports, etc. For real case studies where exploration may cover many types of components and properties, creating a

meta-platform becomes infeasible. Another problem is that the meta-platform specification always puts hard limits on the design space: when another DSE experiment requires a larger meta-platform, then a new meta-platform has to be designed manually. For example a meta-platform that contains 4 processors requires manual modification before it can be used in explorations with up to 8 processors. Therefore, in some cases, it is better to derive the YML system specification directly from the parameters: we call this the generator-approach.

Generator approach

There are many different ways to implement this approach, but here we sketch one possibility that is currently being considered for implementation with Sesame. In our case we assume that each design space parameter is linked to an operator that modifies directly the structure of the YML specification. The operator takes YML as input an produces YML as output and by putting the operators together in various ways, different design points can be generated according to the boundary parameters. An example is a processor-operator which can instantiate a processor with certain properties; more processors are added by iteratively calling the operator. Different operators add different components and properties to the architecture in such a way that a valid YML specification results after all operators have been applied. Operators may have different input requirements: for example, an operator that instantiates a bus may be able to operate on an empty input, whereas an operator that instantiates a fully connected peer-to-peer network requires the processors already to be instantiated. The order in which the YML operators are specified therefore has to occur according to a priority value associated with the operator.

```
designpoint =
  emptyYML
  .addBus()
  .addProcessor(ARM)
  .changeProcFreq(200)
  .addProcessor(PPC)
  .addProcessor(MB)
  .changeProcFreq(100)
  .changeProcFreq(400, 2)
eval( designpoint )
```

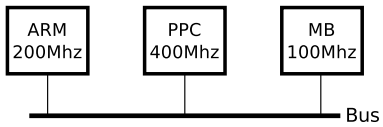


Figure 3.11: Instantiating a single design space specification

The advantage compared to the meta-platform approach is that the generator approach does not suffer from the same problems with respect to scaling. Furthermore, the approach is extensible: YML operators can be reused between experiments and new ones need to be defined only for any additional parameters types that need to be explored in the new experiment. Operators can be implemented as functions that work on a parsed YML tree or as programs or scripts that take YML files as input and produce YML as output. Thanks to the fact that XML is widely supported by most major programming languages (either natively or using libraries), implementing operators is not complicated and requires a fixed,

one time effort. Three operators are given a pseudo-definition in Figure 3.13. Each definition lists what requirements need to be present in the input YML, the type of operator (e.g., insertion or replacement) and the YML code to be inserted or the pattern to be substituted. For insertion operators it is specified where the inserted YML will be connected to the input YML. The example in Figure 3.11 shows how a single design point can be generated. Note the use of pseudo object-oriented function calls as operators that operate in sequence on an `emptyYML` object. The `eval()` operator is where the design point can be evaluated using (for example) the Sesame architecture simulator. Using such operators it is not hard to generate a design space according to some boundary parameters. Figure 3.12 shows a small loop that iteratively generates design space specifications consisting of a bus based system with 1 to 20 processors. Naturally, more complex combinations are possible to generate specifications for a wide variety of design spaces.

```
for(num: 1 2 .. 20) {
  designpoint = emptyYML
  designpoint.addBus()
  while(num--) {
    designpoint
    .addProcessor(ARM)
    .changeProcFreq(200)
  }
  eval( designpoint )
}
```

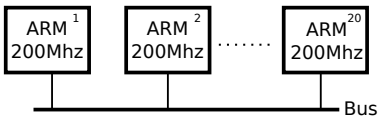


Figure 3.12: Instantiating a range of design space specifications

3.5 Model and design space evaluation

As we have seen in Section 3.3, a completely defined Sesame model consists of application and architecture YML files as well as source code for the model components. It is then ready to be run as an application-architecture co-simulation, or the architecture model can be run stand-alone if the application traces have been generated and stored previously. After the model has been executed, the Pearl architecture simulator will output a large amount of statistics that have been collected during simulation time. These statistics provide system-level as well as detailed component-level information about the performance of the model. They represent an estimation of the performance of the modeled system with a certain accuracy (depending on the quality of the model). Perhaps the most frequently used statistic is that of total simulated run-time, which is given as the value of the simulated clock when the model terminates. For a model without errors or deadlocks, this is equal to the amount of clock-cycles that the modeled system needs to process all application trace-events. More detailed information includes for example the utilization of architectural components, which indicates the ratio of busy and idle times of each component. Another statistic measures system contention (indicating how many components are waiting for some other component at the same time). As Sesame uses a transparent message passing mechanism as the ba-

operator name	`addProcessor(type)`
requirements	primary network
operation type	insertion

addition	`<node name="mp" class="pearl_object">`
	`  <property name="class"`
	`  value="processor"/>`
	`  <property name="type"`
	`  value="$type"/>`
	`  ...`
	`</node>`

connection point	to primary network
operator name	`addBus()`
requirements	none
operation type	insertion

addition	`<node bus="mp" class="pearl_object">`
	`  <property name="class"`
	`  value="bus"/>`
	`  ...`
	`</node>`

connection point	none
operator name	`changeProcFreq(freq, procId)`
requirements	processor *procId* or last added
operation type	replacement

pattern	`<node bus="mp" class="pearl_object">`
	`  ...`
	`  <property name="freq"`
	`  value="XXX"/>`
	`  ...`
	`</node>`

replacement	freq

Figure 3.13: Definition of YML generator operators

sis of its (a)synchronous communication primitives, system contention can be measured by the number of message waiting in a component's message queue. Other statistics include profiling information (e.g., time spent in each method of each component), bandwidth information and critical path analysis (which sequences of remote method calls block other method calls). By default, the statistics are given as a text file, which can easily overwhelm the designer. For this purpose a visualization of the statistics has been developed in the work of [99], that gives a concise summary of all the objective properties of a design point in a single view.

For real design space exploration case-studies, many design points typically need to be evaluated. We have discussed in Section 3.4 how to generate larger sets of XML specifications representing a range of system design candidates. Here we emphasize the problem of dealing with the large amount of output data that results from such experiments. In the context of the Daedalus framework, all the statistics of the Sesame simulation models will be stored in an XML-database. It also saves all input-data, so that it is possible to recreate each design point (the model complete with code and XML specification). Using the database, well-formed queries can be expressed in an XML query language which reduces the effort of writing custom scripts to analyze large volumes of simulator output statistics. In addition, we further exploit the human visual cognitive capacities in aid of design space exploration. By the same author of [97], another visualization method has been proposed which is particularly useful for evaluating large, multi-dimensional design spaces. It visualizes the design space as a large tree shape for which the leaves represent different design points. Different colors and shapes for leaves and nodes summarize important objective information about the design space (which is taken from the aforementioned XML database). Various selection and refinement mechanisms are available to the designer to re-shape the tree to interactively manipulate the visualization and to focus on design points of interest. Different metrics for visualizing (multi-objective) properties of the design points are also available (for more information we refer to [98]).

3.6 Conclusion

This chapter focused on the Sesame modeling and simulation tool which we use extensively throughout the work presented in this thesis. We have shown that Sesame uses a strict separation of concerns and a relatively high level of modeling abstraction by default. Application and architecture models are highly modular which facilitates component reuse and which further reduces the design and development time of a Sesame model. Some examples show some of the details that are involved in the creation and specification of a Sesame model. A graphical user interface has been made available to make model composition and specification even easier. We have discussed the difficulties of specifying models for multiple design points and have shown some solutions that can be used in the Sesame context and beyond. Finally we have discussed the type of information that can be obtained from the standard Sesame model.

Chapter 4

Model calibration

4.1 Introduction

In the previous chapters we have seen how models are defined in Sesame. The behavior of the model as a whole is –of course– defined by all its component parts as well as by the interconnections of the components. The behavior of a model component is in turn specified by its simulation code and its interactions with other components. Additionally, the behavior may depend on application input data (such as video input streams), initialization of random seeds, the initialization of its variables, etc. Sesame helps the model designer with all modeling aspects, for example by offering separation of concerns, reusable model components, an easy-to-use simulation language, deterministic model execution, a graphical model editor, etc. In this way the designer can quickly and easily create functionally correct models for many different systems and applications. These models can then be quickly and easily evaluated thanks to good model run time performance and automatically generated evaluation reports.

However, functional correctness does not by itself mean that the model displays the correct non-functional behavior as well. Non-functional properties that are important to a system designer include system performance, power consumption, production cost as well as physical characteristics such as die size, thermal hot-spots, etc. For example, in Sesame, we can define a video encoding platform by trivially mapping the application onto an architecture model for which no realistic properties (eg. processor timings, clock speed, memory sizes) have been defined. In this case the (application) model will be functionally correct since it produces a correctly encoded video stream, but performance behavior of the architecture model will be unrealistic. Therefore, such a model is likely unusable to predict non-functional behavior of any realistic system. We say that a model is behaviorally correct if it is able to represent (with a certain level of accuracy) those non-functional properties that are of interest to the designer.

To determine the behavioral correctness of a model, one has to consider (among other things) the following. Firstly there is the domain of (non-functional) behavior that is important for the designer for example: performance, power, cost, physical characteristics, etc.

Secondly, the designer may be interested only in the behavior of part of the system, such as network behavior or contention of processing or memory resources. Sesame supports this by offering mixed-level modeling where parts of the system that are of particular interest may be defined at a more detailed level of abstraction than the rest of the model. Choosing the right level of abstraction is the third main consideration that may influence behavioral correctness. The latter is particularly important, since it is well known that the level of detail used by a simulation model can have major impact on the run-time performance of the model.

With the above considerations in mind, the challenge of system-level design can be summarized as follows: to create a functionally and behaviorally correct model that accurately and quickly approximates the important parts of the system under evaluation.

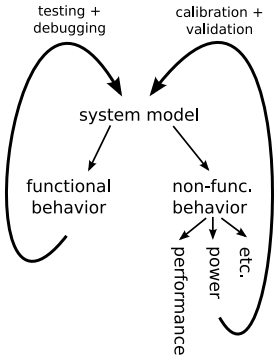


Figure 4.1: Modeling overview

When a system designer constructs a system-level model, then this model rarely displays the correct non-functional behavior from the beginning. Analogous to the traditional method of software debugging to achieve functional correctness of application software, there exists a need to tune the non-functional behavior of (parts of) the model in order to achieve behavioral correctness. This process is called *model calibration*, and it is one of the most difficult problems involved in creating a good system-level model (for a conceptual overview see Figure 4.1).

In this chapter we report on three methods to perform calibration of system-level simulation models, focusing on the performance objective. In Chapter 8 we will show an an extensive case study using one of these techniques and there the simulation results will be validated against prototype system implementations in the context of the Daedalus design flow.

4.2 Model calibration

In Chapter 3 the Sesame modeling methodology was discussed in detail. The workload that is modeled by the application model is forwarded to the architecture model by means of event traces. The architecture model consumes these events and models their performance behavior as a delay (or *latency*) on the relevant model components. This may be done by simply associating a latency to that event, but the event may also trigger complex interactions between components. Using special primitives in the simulation language, it is easy to describe the (possibly complex) interaction between different components in the model. Moreover, the propagation of latencies as a consequence of component interaction is automatically performed by the simulation language's runtime system. However, determining the correct latency value at any place in the model can pose quite a challenge depending

on the type of model, the type of component and even the particular interest and requirements of the designer. In general, correct latency values can be obtained from many sources: component specification and documentation, instruction set simulators, measurements on prototypes, and even rough estimations may be sufficient in some cases. It may be necessary to recalibrate the latency values during the different stages of development of the system model as more details become known and more design decisions are taken. In the end, only validation of the model (eg. by comparing the behavior with a (partially) implemented prototype) can provide certainty about the quality of the model. Therefore, calibration does not supersede, but rather supplements model *validation*, as a method to make the non-functional behavior of the model match the behavior of the actual system. Figure 4.2 schematically details the modeling process and the calibration and validation techniques.

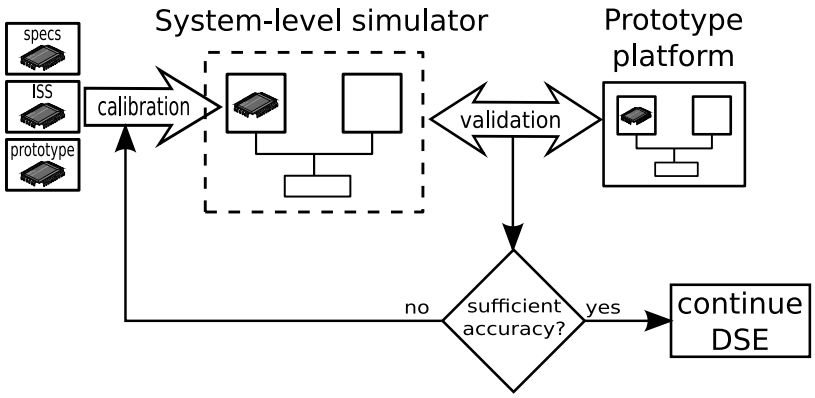


Figure 4.2: The calibration and validation process

For some types of components latencies are relatively easily obtained because their behavior is known and predictable. Take for example an on-chip network component, such as a bus or crossbar: a performance model could assume some latency for setting up a new connection and subsequently add a certain latency per amount of communicated data. The latencies for the setup time and transfer rate are part of a component's specification. Note once more that these latencies do not need to include any additional latency caused by contention of components (such as a shared bus), since this is will be automatically captured by the simulation runtime system (e.g. using Pearl's synchronous method calls: Section 3.2.2).

Another possibility is that the designer is making a model that contains components which are yet to be developed. For example, the designer needs a component to process a certain task X, but still does not know whether X should be implemented in software or hardware. Creating a dedicated hardware component represents a significant development cost, but may speed-up the overall system performance. Since the designer will not be able to obtain accurate latencies for X, he could estimate a ball-park value based on his experience. Using the estimated value he can the use the system-level model to support such design choices in the very early design stages.

Latencies can be much more difficult to estimate for some other types of components.

E.g., programmable processor components associate a latency to any application execution events. In our initial, high-level system models such an execution event typically represents a large chunk of code. The exact latency to be associated with that event may vary according to data dependencies, but in our simplest and most abstract models, the latency for an execution event of a certain type is set to an average value. In this case, the calibration process is focused on finding usable average values.

In general, the design of MPSoC embedded systems heavily depends on the reuse of existing components and therefore it is often the case that data sheets, tools, and prototypes are available for at least parts of the system. Since all system-level models need calibration it is crucial to have generic and feasible methods to calibrate those models by finding sufficiently accurate latencies in a structural and easy way. In the next section we will present the *off-line* model calibration method, which does just that, by measuring these values using an Instruction Set Simulator (ISS). We present also the *on-line* trace calibration method, which is a form of mixed-level co-simulation which has the potential to provide more accurate latency values even when latencies are data-dependent.

A possible disadvantage of the off-line and on-line calibration methods is that the calibration is performed for a specific application. Therefore, in the second part of this chapter a method will be presented that attempts to associate a "signature" to an architecture component that describes the architecture component's capabilities in an application-independent way. Using this signature, the relevant latencies can be derived automatically within the component. We will present the results from a small DSE case study.

4.3 Off-line model calibration

In Figure 4.3 a code fragment is shown from the basic processor model component available in Sesame. The processor component accepts read, write and execute events from the virtual layer and models processing time of an event as being busy or *blocked* for a certain number of cycles (line 19). We focus here on the modeling of execution (EX) events and in particular those that are mapped onto programmable processor cores. In this very simple model, a table is used to associate a latency to a particular type of event (line 18). For non-programmable processor cores that are dedicated to process certain fixed tasks, these latencies can often simply be taken from their respective documentation. However, for general purpose programmable cores the cycle count for certain events may vary depending for example on input data.

Here, we integrate a lower-level simulator in order to statically (i.e., before system-level simulation) calibrate the values in an operation latency table according to code-fragment performance measurements on the ISS. To explain this in more detail, consider Fig. 4.4. In this example, we assume that model component p2 onto which application process B is mapped needs to be calibrated using the ISS. This means that the code of Kahn application process B is crosscompiled for the ISS (indicated by B' in Fig. 4.4a). We note that Figure 4.4a focuses on the application model level, and only abstractly depicts the mapping and architecture model levels. Also, throughout the remainder of this chapter, we take the

example of incorporating an ISS into Sesame. However, other types of simulation models (like RTL models) can also be used with trace calibration.

The cross-compiled code is further instrumented such that it measures the performance of the code fragments that relate to the computational application events generated by the application process. For example, if process B can generate an execute(DCT) event, then the performance of the code in process B that is responsible for the DCT calculation is measured. To this end, we instrument the code at assembly level (currently done manually) to indicate where to start and stop the timing of code fragments. In the case of the SimpleScalar ISS, we use its annote instruction field for this purpose. To perform the actual code fragment timings for application process B, the code of this process is executed both in the Kahn application model and on the ISS (the cross-compiled B'). This allows us to keep the application model to a large extent unaltered, where B' runs as a "shadow process" of B to perform code fragment measurements.

```
1  class processor
2
3  // latency table initialized in yml:
4  operations_t = [] integer
5  operations :  operations_t
6
7  read:(...)  -> void {
8    ...
9  }
10
11 write:(...)  -> void {
12   ...
13 }
14
15 execute:(operindx:integer)
16 -> void {
17    latency :  integer =
18      operations[operindx];
19    blockt(latency);
20    reply();
21 }
22
23 {
24   while (true) {
25     block(any);
26   }
27 }
```

Figure 4.3: Extract of simulation code for a processor component

The two executions of B are synchronized by means of data exchanges implemented by an API using an underlying IPC mechanism needed to provide B' (on the ISS) with the correct application input-data. These data exchanges, which will be discussed in detail in the next section, only occur when the Kahn application process taking part in the calibration (process B in our example) performs communication. For example, when Kahn process B reads data from its input channel, it forwards the data to process B' on the ISS, i.e., process B' reads and writes its data from/to process B instead of a Kahn channel.

During execution, the ISS keeps track of the code fragment timings. From these measurements, timing averages are then used for calibrating the latency values of the architecture model component in question: i.e., the average value is used in the latency table of a processor model component. Off-line calibration is a relatively efficient mechanism since it is performed before system-level simulation and basically is a one-time effort. However, if there is high variability in the demands of computations (i.e., data-dependent computations) then the measured average timings may be of moderate accuracy. In this case it may be beneficial to perform calibration of trace events in a dynamic way as shown in the next section.

4.4 On-line trace calibration

The *on-line* trace calibration technique accomplishes mixed-level co-simulation by means of dynamic calibration of the event traces that are generated by an application model. Rather than using fixed values in the latency tables of processing components in Sesame's architecture models (see Section 2), trace calibration dynamically computes – using lower-level simulators – the latency values of computational tasks. Subsequently, instead of generating fixed computational execution events, like *Execute(DCT)*, the Kahn application process generates *Execute(*Δ*)* events, where Δ equals to the actual cycle count taken by, for example, a DCT computation (or any other computation in between communications). This is illustrated in Figure 4.4b, where an instruction set simulator (ISS) is used for calibrating the trace from application process B.

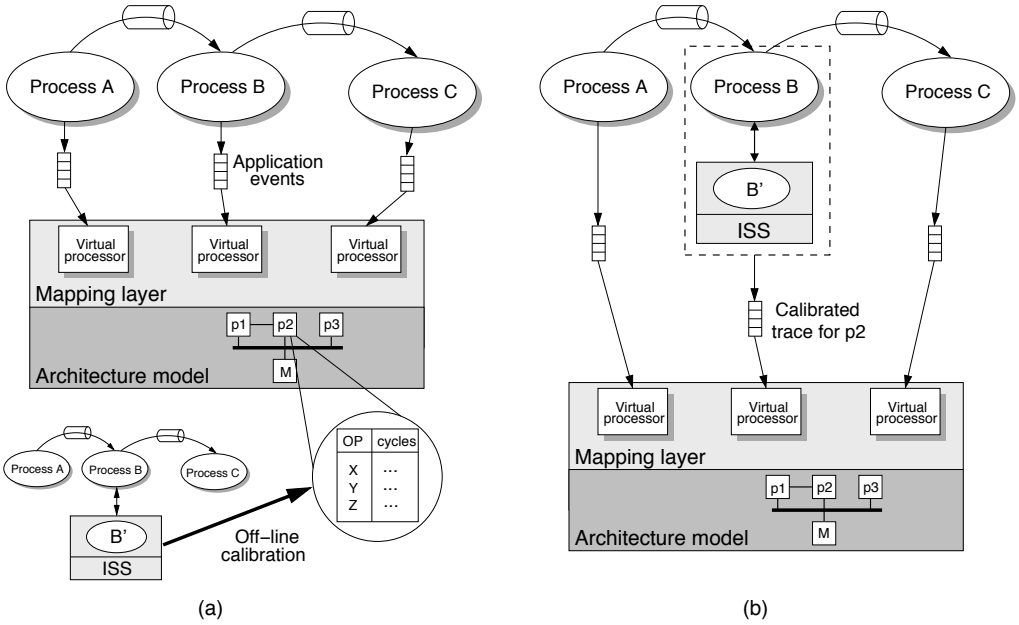


Figure 4.4: Incorporating an instruction set simulator using *off-line* model calibration (a) and *on-line* trace calibration (b)

Compared to the original static operation-latency table method , a calibrated trace can more accurately represent the computational behavior of an architecture component. For example, in Figure 4.4(b), statistics on pipeline stalls and cache behavior for process B can be retrieved from the ISS. Also, Sesame's dataflow-based event refinement methodology [79] can still be applied to calibrated traces to, e.g., perform communication refinement, i.e., refine the *Read* and *Write* application events. The system-level effects of this improved accuracy can subsequently be measured within Sesame's architecture performance model which accounts for the global timing consequences of all (calibrated as well as non-calibrated) event traces generated by the application model. Efforts to quantify the improved accuracy,

should however be performed with great care since their outcome heavily depends on the quality (i.e., accuracy) of the initial abstract performance models.

The increased accuracy comes however at the cost of higher execution times due to the inclusion of (slow) lower-level simulators. But, as will be demonstrated in the next section, these performance overheads due to wrappers (i.e., data and control exchange between simulation components) and time synchronizations are very small in our trace calibration technique. This is because time synchronization occurs at the highest level of abstraction, namely within Sesame's trace-driven architecture model, and data and control exchanges via our API only take place at (Kahn) communication points. In some occasions, it is even possible to eliminate almost all overheads related to trace calibration. For example, if no architectural exploration is performed on the architecture components that are simulated by the lower-level simulators, then the (calibrated and non-calibrated) traces could be generated once, storing them on disk, and can be re-used in the exploration process of the remaining parts of the system without rerunning the lower-level simulators.

To explain the details behind the trace calibration technique, consider Figure 4.5. This figure renders the dashed box from Figure 4.4(b) in more detail. The code in the Kahn application processes typically consists of alternating periods of communication and computation, as illustrated by the small code fragment for process B in Figure 4.5. In this fragment, some data is read from Kahn channel `c_in`, followed by some computational code (which may also be discarded, as will be explained later), after which the resulting data is written to Kahn channel `c_out`. The two boxes on the right of this code fragment indicate what the run-time system of the application model executes when it encounters the Kahn read and write communications. Note that these run-time system actions are automatic and transparent: the programmer does not need to add or change code. First, the run-time system queries the ISS via the API, using `API_get_cycles()`, to retrieve the current cycle count from the ISS. As will also be described later on, the ISS provides this cycle information by executing a matching `API_put_cycles()` call. The run-time system then generates an *Execute(*Δ*)* application event for the architecture model, where $\Delta = n_{cur} - n_{prev}$, i.e., Δ equals to the time between the previous cycle query and the current one. Hence, the *Execute* event models the time that has past since the previous communication. Subsequently, a *Read* application event is generated for the architecture model. Hereafter, the actual read from Kahn channel `c_in` is performed. Finally, the data that has been read is copied, using `API_write`, to process B' running on the ISS.

Figure 4.5 also shows how the ISS side (process B') is handled. First, it sends the current cycle count of the ISS to the application model (`API_put_cycles`) to service the `API_get_cycles()` query from process B. Then, it reads the data that was sent by process B, i.e., the `API_read` from process B' matches up with the `API_write` from process B. After receiving the data, process B' executes the computational code shown in grey in Figure 4.5. This computational code is finished by a communication (a write to `c_out`), which again causes a cycle count query by the run-time system of the application model. The generated *Execute(*Δ*)* application event that follows, represents a detailed timing of the computational code on the ISS. Figure 4.5 also shows that process B' on the ISS first writes

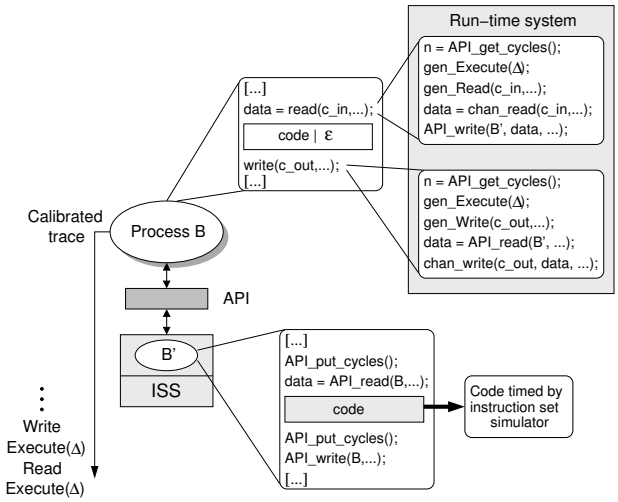


Figure 4.5: Interaction between application model and instruction set simulator.

back the resulting data to process B in the application model before the latter forwards this data to Kahn channel `c_out`. This allows for discarding the computational code between the communications in process B in the application model. In that case, only process B' simulates computational functionality, while process B only communicates data with its neighboring application tasks. From the above, it should be clear that the `API_get_cycles` and `API_read` calls are blocking.

With trace calibration, it is relatively easy to incorporate any external low-level simulator into Sesame's system-level architecture models. Only three API functions need to be introduced in the low-level simulator or in the code that runs on it (in the case of an ISS): `API_put_cycles()`, `API_read()`, and `API_write()`. Here, `API_put_cycles()` is the only function that needs a hook into the simulator to retrieve its cycle count. Most simulators provide such a hook, otherwise it can be created by a small modification to the simulator (as will be briefly explained in the next section). Using the API calls, the code to run on an ISS simulator (B') can be trivially derived from the application code (B). Therefore, the total coding effort to enable trace calibration is small.

Moreover, for trace calibration, the execution of the lower-level simulators is location independent. That is, it is straightforward and completely transparent to place the lower-level simulators on different hosts, yielding *distributed co-simulation*. To this end, the implementation of the API between application process and lower-level simulator simply features different communication adaptors (e.g., shared memory or named pipes for local communication, and sockets for remote communication). As will be shown in the next section, the distributed co-simulation support considerably improves the scalability of the co-simulations in the case multiple lower-level simulators are incorporated.

As a side note, we need to mention that a source of inaccuracy, of which a similar situation is reported in [55], occurs when mapping multiple application tasks to a single (programmable) architecture component. In this situation, the traces from these application

tasks would be calibrated by *different instances* of an ISS. The scheduling of the (calibrated) application events from the different traces is subsequently performed at Sesame's mapping layer. This approach has two major advantages. First, there is no need for running an OS-scheduler on an ISS since ISSs always execute a single task. Second, the ISS instances, representing a single processor in the architecture, can be executed in parallel on different hosts. However, since context-switching is not modeled by the ISSs, the simulated cache (performance) behavior in this situation may be inaccurate.

4.5 Trace calibration experiments

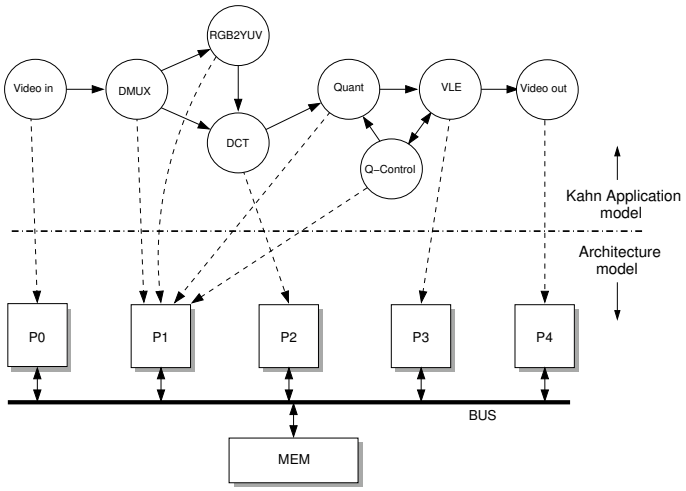


Figure 4.6: Modeling a Motion-JPEG application on an MP-SoC architecture.

To demonstrate that the performance overheads of trace calibration are low, we present a case study with a Motion-JPEG encoder application and a shared-memory MP-SoC architecture consisting of five processing elements. Sesame's application model, architecture model and mapping for this case study are shown in Figure 4.6. In this experiment, we have incorporated SimpleScalar's sim-outorder ISS [5] in Sesame's high-level model of the MP-SoC architecture. This required only one small extension of the ISS: a system-call that retrieves the cycle count, which is needed by `API_put_cycles()`. Small C macros implement the functions `API_read()`, `API_write()`, and `API_put_cycles()` and are compiled together with the application code executing on the ISS.

For the experiments, we have used a small cluster of unloaded Pentium M 1.7GHz (Debian) machines connected by 100 Mbit ethernet. In all simulation runs, we have simulated the encoding of 11 CIF frames with a resolution of 128x128 pixels. Table 4.1 shows the wall-clock times (averaged over several runs) for three different simulation configurations executed on a single host machine: a Sesame-only system-level simulation (without trace calibration), a trace-calibrated (mixed-level) co-simulation where the DCT applica-

Table 4.1: Co-simulation performance.

	Sesame only	DCT on ISS	DCT+VLE on ISSs
Time (secs)	8.1	157.1	279.6
Kcycles/sec	8,000	414	233

tion process is executed on the ISS (representing P2 in Figure 4.6), and a trace-calibrated co-simulation where both the DCT and VLE processes are executed on different ISSs (representing P2 and P3). For each simulation configuration, the number of simulated Kcycles/sec is also given. Table 4.1 clearly shows the performance drop when incorporating lower-level simulators in Sesame's architecture models. The results also show the efficiency of the Sesame-only simulation (8 Mcycles/sec when simulating a 5 core MP-SoC).

Figure 4.7: Overhead for one or two ISSs.

To demonstrate that the performance decrease of our mixed-level co-simulations is almost entirely due to the lower-level simulators themselves and not due to co-simulation overheads, Figure 4.7 shows the execution time breakdowns of the two trace-calibrated co-simulation configurations from Table 4.1. The breakdowns clearly prove that the total execution times are totally dominated by the ISS execution times. The measured overheads are respectively 5% (DCT on ISS) and 1% (DCT+VLE on ISSs) of the total execution time. We suspect that the latter has lower overheads because of the scheduling of the two ISSs that allows for hiding some of the overheads caused by communications between the DCT/VLE processes and the ISSs.

By means of distributed co-simulation, the system-level simulation slowdown due to the incorporation of lower-level simulators can be effectively reduced. To illustrate this, Table 4.2 presents the performance effect of distributing the ISSs over different hosts. Here, we use the terms 'local' and 'remote' to indicate if an ISS is executed on respectively the same or a different host as Sesame's application and architecture models. The results show that distributing the lower-level simulators over multiple hosts is certainly beneficial. For

example, by placing the ISSs for the DCT and VLE processes on two different hosts (see Table 4.2), a speedup of 1.86 is achieved in comparison to execution on a single machine (see Table 4.1). In this case, only 6.5% of the total execution time is due to overheads (including network overhead).

Table 4.2: Distributed performance.

	DCT on remote ISS	DCT on local ISS VLE on remote ISS	DCT+VLE on remote ISSs
Time (secs)	144.2	160.6	150.2
Kcycles/sec	452	405	433

We also performed experiments in which less computational intensive application processes, like RGB-to-YUV and Quant, are trace-calibrated as well (these results are not shown in table form). A fully distributed co-simulation with ISSs for the DCT, VLE and RGB-to-YUV processes takes 158.3 seconds. Adding an ISS for the Quant process to the previous distributed co-simulation results in a wall-clock time of 165 seconds. These results indicate that, with distributed co-simulation, trace calibration scales well.

For comparison, [55] lists the performance of similar case studies using their own work, Seamless CVE and Synopsys System Studio. With respect to the latter two, our co-simulations are one to two orders of magnitude faster, while for our Sesame-only simulations this is even three orders. The reported performance of the state-of-the-art technique proposed in [55] approximates our co-simulation performance, but they have used the ARMulator ISS which is significantly faster than the SimpleScalar ISS we have used.

To get some indication of the accuracy gains of trace calibration, we have also performed several simple experiments that compare a fully trace-calibrated model to a partially uncalibrated model. In this case, the uncalibrated model components have to use estimated latency values for application events. Statically estimating the performance of code executed on a particular processor is a well-known problem as it may be hard to deduce the correct number and types of executed instructions due to data-dependent behavior and to determine the CPI for the processor. Our experiments show, e.g., that if for one processor (onto which the DCT is mapped) we know the right number and types of executed instructions but mispredict the CPI by only 0.07 (where the actual CPI is 0.43) to calculate the latencies for the application events, then the estimated performance for the *whole system* is off by 13%. If the misprediction is bigger or if more components are uncalibrated then the system-level accuracy is reduced even further. For a more extensive validation case study of our calibrated models compared to an implemented system, we refer to [80]. Moreover, in Chapter 8, another validation case study will be shown.

4.6 Signature based model calibration

As we mentioned before, the off-line calibration method in Section 4.3 has a possible disadvantage that calibration is dependent on the application, requiring re-iteration of the calibration process to obtain new values for the latency table for each application. In the case of on-line calibration these values could be obtained automatically, provided that the KPN code is sufficiently platform independent to cross-compile it without problems for the ISS simulator. In both cases the (calibrated) event traces can be stored when the application workload has been fixed. The stored traces can subsequently be reused for further architectural design space exploration, resulting in a model performance improvement for the on-line calibrated model. This way, any additional effort and model execution time can be considered only a minor manual effort in the early stages of design for a system that runs a small number of applications. However, this effort may become a problem for a system targeting wider range of applications, since the calibration overhead would increase. In those situations it is desirable to be able to describe the behavioral (performance) properties of system components in an application-independent way.

In the following we use a general decomposition of the problem of determining performance values (eg. for use in a system-level model) in three sub-problems:

(1) determine workload requirement

(2) determine component capacity

(3) compute performance estimate from (1) and (2)

More specifically, we can say that for a Kahn process k_1 and processor p_1, the performance $\mathcal{T}$ is defined by a function f:

$$\mathcal{T}_{p_1} = f(k_1, p_1) \tag{4.1}$$

The performance of a certain architectural component naturally depends on both its computational capacity and the computational requirement of the workload that is mapped onto it. The aim is then to define a function f that calculates the performance of p_1 as a number of cycles taking as input some workload requirement of k_1 and some capacity of p_1. We will refer to k_1 and p_1 respectively as the application signature and the architecture signature. In this section we will describe the signature-based calibration method as proposed in [46] and show a simple case study to demonstrate the usefulness of this approach. Furthermore, we show that the accuracy of the signature-based method provides good relative (though not absolute) performance numbers.

4.6.1 Application requirements

To characterize application requirements a description is proposed that captures a Kahn process's computational requirement at a high abstraction level. Towards this end we define an execution profile as a vector of instruction counts for the individual execution events generated by a process. In order to keep this so-called application signature architecture

`read`	f_2
`execute`	op_1
`write`	f_1
`read`	f_2
`execute`	op_2
`write`	f_1
`execute`	op_1
`write`	f_1
`write`	f_1

(a)

Signature index	AIS opcode	Description
1	`AIS_BMEM`	Block memory transfers
2	`AIS_MEM`	Memory transfers
3	`AIS_BRANCH`	Branches
4	`AIS_COPROC`	Co-proc. instructions
5	`AIS_IMUL`	Int. multiplications
6	`AIS_ISIMPLE`	Simple Int. arithmetic
7	`AIS_OS`	Software interrupts
8	`AIS_UNKNOWN`	Non-mappable instruction

(b)

ARM instruction	AIS opcode
bl 0x81c4*;*	`AIS_BRANCH`
mov ip, **sp***;*	`AIS_ISIMPLE`
stmdb sp, fp, ip, lr, pc;!	`AIS_BMEM`
sub fp, **ip**, #4*;*	`AIS_ISIMPLE`
sub sp, **sp**, #12*;*	`AIS_ISIMPLE`
ldr r2, [**fp**, #−16]*;*	`AIS_MEM`
ldr r3, [**fp**, #−20]*;*	`AIS_MEM`
add r2, r2, r3*;*	`AIS_ISIMPLE`
ldr r3, [**fp**, #−24]*;*	`AIS_MEM`
rsb r3, r3, r2*;*	`AIS_ISIMPLE`
str r3, [**fp**, #−24]*;*	`AIS_MEM`
ldr r2, [**fp**, #−16]*;*	`AIS_MEM`
ldr r3, [**fp**, #−20]*;*	`AIS_MEM`
add r2, r2, r3*;*	`AIS_ISIMPLE`
ldr r3, [**fp**, #−24]*;*	`AIS_MEM`
··· ⇒	
··· ⇒	
mul r3, r2, r3*;*	`AIS_IMUL`
str r3, [**fp**, #−16]*;*	`AIS_MEM`
ldr r2, [**fp**, #−20]*;*	`AIS_MEM`
ldr r3, [**fp**, #−16]*;*	`AIS_MEM`
mul r3, r2, r3*;*	`AIS_IMUL`
str r3, [**fp**, #−24]*;*	`AIS_MEM`
ldr r2, [**fp**, #−16]*;*	`AIS_MEM`
ldr r3, [**fp**, #−24]*;*	`AIS_MEM`
add r2, r2, r3*;*	`AIS_ISIMPLE`
ldr r3, [**fp**, #−20]*;*	`AIS_MEM`
mul r3, r2, r3*;*	`AIS_IMUL`
str r3, [**fp**, #−16]*;*	`AIS_MEM`
sub sp, **fp**, #12*;*	`AIS_ISIMPLE`
ldmia sp, **fp**, **sp**, **pc***;*	`AIS_BMEM`
mov ip, **sp***;*	`AIS_ISIMPLE`
stmdb sp, **fp**, **ip**, **lr**, **pc***;*	`AIS_BMEM`

(c)

Figure 4.8: Table (a) lists the event trace of process k_1, Table (b) shows the currently defined AIS instructions with their index in the vector-based process signatures and Table (c) shows an execution trace of op_1 as obtained by an ARM ISS (left column) and the corresponding AIS classification (right column).

independent, we use an abstract instruction set (AIS), see Figure 4.8b. For each type of processor that uses the signature based calibration method a mapping has to be provided that maps its particular instructions to one of the AIS categories. Next we use an instruction set simulator to generate an execution profile for those parts of the code that are represented by Sesame's execute events. As an example Figure 4.8a shows a Kahn process event trace containing two op_1 and one op_2 events. Note that here we discard the description for communication events (read and write) for simplicity, more details can be found in [47]. In the source code we annotate the beginning and end of each of the execute events in a very similar way as with the trace calibration technique. This enables the ISS to know the start and end points to count instructions according to our AIS categories. For the example execution trace in Figure 4.8a; one of the two op_1 executions has the detailed execution profile as listed in Figure 4.8c. Measurements for each op_x results in the following derived signatures:

$$\begin{aligned} op_1.\text{signature} &= [7, 17, 8, 0, 2, 31, 2, 0] \\ op_2.\text{signature} &= [3, 15, 1, 0, 3, 9, 0, 0] \\ op_1.\text{signature} &= [8, 15, 8, 0, 3, 29, 2, 0] \end{aligned}$$

Note that for the purpose of the example we only consider two execution events, but in general we would aim to measure as many signatures as possible. Also note that when an op_1 is measured repeatedly, its signature may consist of different AIS instruction counts because of data-dependencies or pseudo-random behavior.

4.6.2 Processor capacity and performance estimation

Next we will show how to derive the signature p_1 that describes the computational capacity of a processor. Using this signature we will then be able to estimate the number of cycles required for any workload by defining f from Equation 4.1.

$$\mathcal{T}_{p_1} = f\left(k_1.\text{signature}, p_1.\text{signature}\right) \tag{4.2}$$

While the ISS is executing (and counting instructions of the application signatures), we also register the increase of simulated clock cycles within the ISS. In this way we not only obtain the instruction count for each op_x, but also the measured latencies for each op_x. If there was an exact linear relationship between a signature and its execution performance, then the following would hold:

$$\begin{pmatrix} op_1.\text{signature} \\ op_2.\text{signature} \\ op_1.\text{signature} \end{pmatrix} \cdot p_1.\text{signature} = \begin{pmatrix} \mathcal{T}_{op_1} \\ \mathcal{T}_{op_2} \\ \mathcal{T}_{op_1} \end{pmatrix} \tag{4.3}$$

Of course it is not to be expected that such an exact linear relationship exists, but instead we approximate a solution p_1.signature for the matrix equation using the least squares method. The vector of application signatures and its measured timings will serve as our benchmark

(training set) for the approximation. The performance estimation of the processor signature in the previous example the previous example would for example result in:

$$\begin{pmatrix} 7 & 17 & 8 & 0 & 2 & 31 & 2 & 0 \\ 3 & 15 & 1 & 0 & 3 & 9 & 0 & 0 \\ 8 & 15 & 8 & 0 & 3 & 29 & 2 & 0 \end{pmatrix} \cdot p_1.\text{signature} = \begin{pmatrix} 185 \\ 369 \\ 196 \end{pmatrix} \tag{4.4}$$

The processor's signature can be considered vector of 8 weight-factors, where each value represents the average estimated contribution of each AIS instruction to the total processing time. Now we can define the function f from Equation (4.2) as the inner product of both signatures. This allows us to estimate a cycle count for an arbitrary operation op_x that was not in our original training set:

$$\mathcal{T}_{op_x} = p_1.\text{signature}^T \cdot op_x.\text{signature} \tag{4.5}$$

In general, the training set should be made as large as possible, containing op_x measurements from a range of different applications in order to provide a good approximation of the processor signature for a particular processor. Also note that the training process may measure the same operation signature op_x with differently measured cycle counts, due to the previously mentioned data-dependencies or micro-architectural effects on pipelining and caching. All these effects will therefore be included in the approximation of the processor signature.

4.7 Signature-based calibration experiments

In this section, we present an experiment using a Motion-JPEG (M-JPEG) encoder application in which the possibilities of mapping application tasks to architecture components are explored. We compare the results of an off-line calibrated model with a model using signature based calibration. The signature-based performance models are calibrated using the Mediabench benchmark suite [61] as an external training set. Finally we discuss the impact of the choice for the training set on the exploration results.

To validate our signature-based analytic performance model, we studied the mapping of a Motion-JPEG (M-JPEG) encoder application onto an MP-SoC architecture. This is illustrated in Figure 4.9. The target MP-SoC consists of four ARM processors with local memory, FIFO buffers for streaming data, and a crossbar interconnect. The design space we considered for this experiment consists of all possible mappings of the M-JPEG tasks (i.e. processes) on the processors in the MP-SoC platform. Without consideration for any duplicate mappings caused by the symmetry in the target platform, the size of the design space can be calculated as: $4^6 = 4096$ (6 tasks onto 4 processors).

Before the M-JPEG application model was mapped on the architecture model, the application was compiled using an ARM C++ compiler, and executed within the SimIt-ARM instruction set simulator environment [82]. The generated ARM instruction traces were used to create the application *and* architecture signatures as described in the previous section. Note

that determining the application and architecture signatures is one-time (pre-exploration) effort, and that the resulting execution latencies can be used to explore the full range of different mapping options. To train our performance model, we first constructed the (application) operation signatures for each separate Mediabench program[1] using the approach as discussed in Section 4.6.1. Since the execution time (in terms of simulated cycles) of the `mpeg2enc` program is quite high, we split up the execution of this program into four chunks, and generated a separate operation signature for each of these chunks. Figure 4.10 shows the histogram of the resulting AIS opcode counts for each Mediabench program. This graph clearly shows that most programs are dominated by `AIS_ISIMPLE` instructions, followed by `AIS_MEM` and `AIS_BRANCH` instructions respectively.

Similar to Figure 4.10, Figure 4.11 shows the AIS opcode histogram for the application processes in our target M-JPEG application. Here, we excluded the AIS opcodes with a zero or insignificant contribution. At first sight, the trends in both Figures 4.10 and 4.11 are similar, which thus appears to be confirming that Mediabench is a representative training set for M-JPEG.

As a next step, we determined the processor signatures for our performance model using the Mediabench operation signatures. Using this Mediabench-trained performance model, we again performed the DSE experiment with the M-JPEG application. Figure 4.12 shows a comparison of the DSE results of the Mediabench-trained ("Full Mediabench" in the graph) and the reference simulation model. The reference model uses “exact” latencies for the various computational events as directly obtained by ISS measurements (no latencies that were obtained using our signature-regression model). Again, the graph only shows the performance results of unique mappings, and all mapping instances are sorted based on the performance order of the mappings from the reference model. Comparing the Mediabench-trained model ("Full Mediabench") to the reference model, it is clear that there is a significant absolute error between the results of these models (an average error of 29.6%, see Table 4.3). But the trends between the graphs of the two models still is highly similar. This is especially true

[1] The `pgp`, `ghostscript`, and `sphere` benchmarks were excluded due to execution problems on the SimIt-ARM simulator.

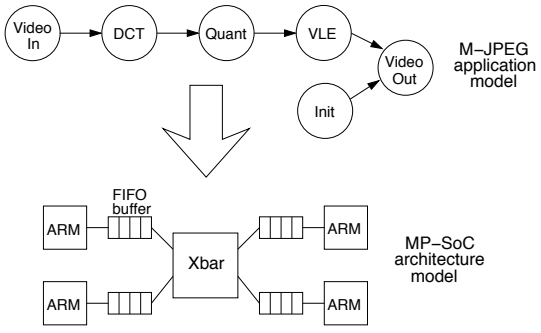


Figure 4.9: Mapping an M-JPEG application to a crossbar-based MP-SoC architecture.

for the better-performing mapping instances (lower mapping indices). This again implies that both models find exactly the same optimal mappings and that the model is quite good at determining the relative performance of the mappings (which is an important property for early DSE).

To study the sensitivity of the selected benchmark programs that are used for training, we performed a number of experiments in which we clustered the Mediabench programs according to some measure, after which we trained our performance model with the programs from such a cluster only. As a first experiment, we applied our performance model that was trained with all Mediabench programs to the operation signatures from the Mediabench programs themselves. Then, we clustered only those programs that show a good fit with the performance model (i.e., removing the outliers). Training the performance model again with this cluster, gives the results that are tagged with "Outliers removed" in Figure 4.12 and Table 4.3. The results of this new performance model are slightly better (average error of 24.9%, see Table 4.3) than the model that was trained with all Mediabench programs.

We selected the program size as the second means to cluster our Mediabench training set. Programs with more than 500 million executed instructions are clustered as "Large programs", while the remaining programs are clustered as "Small programs". Figure 4.12 again shows the DSE results when training our performance model with one of these clusters. The cluster with large programs again shows an accuracy improvement, lowering the average error to 18.6%. Clearly, the cluster with small programs only yields poor results, both in terms of average error (79.8%) and trend behavior. The latter can even be seen at the lower mapping indices where some optimal mappings (according to the reference model) are not considered optimal according to the model trained with small programs only.

	Full Mediabench	Outliers removed	Large programs	Small programs	DCT-similar programs	MJPEG-self trained
Av. error	29.6%	24.9%	18.6%	79.8%	7.0%	9.2
Std.dev.	3.8	4.4	5.1	2.1	4.5	4.5

Table 4.3: Average error and standard deviation of the various trained models as compared to the reference model.

Since the DCT process in the M-JPEG encoder is dominant in terms of computational intensity, our final clustering is based on similarity with the DCT process (in terms of AIS opcode distribution). We again trained our model with these DCT-similar programs. The DSE results of this cluster show again considerable improvement with an average error of only 7%, which is even slightly better than a model that is trained the MJPEG application itself (last column of 4.3).

4.7.1 Related work

Model calibration is a well-known and widely-used technique in many modeling and simulation domains. In the computer engineering domain, the calibration of performance models

is mostly applied in cycle-accurate modeling of system components like processor simulators, e.g., [13, 72]. So far, the calibration of high-level performance models that aim at (early) system-level design space exploration has not been widely addressed yet. The work in [67] proposes a so-called vertical simulation approach that shows similarities with our calibration approach. It is unclear, however, whether or not vertical simulation has ever been realized. In [14], a high-level communication model is discussed which is calibrated using a cycle-true simulator.

The *back annotation* technique is closely related to model calibration. In back annotation, performance latencies measured by a low-level simulator are back annotated in a higher-level model. For example, an un-timed behavioral model could be back annotated such that it tracks timing information for a specific implementation. So, rather than calibrating a fixed set of performance model parameters, back annotation *adds* architecture-specific timing behavior (usually by means of code instrumentation) to a higher-level model. Back annotation is a widely-used technique for (high-level) performance modeling of software [38]. In the context of system-level modeling, various research efforts (e.g., [7, 16, 15]) also refer to back annotation as a technique for adding more detailed timing information to higher-level models in the case lower-level models are available. But these efforts generally do not provide much insight of how back annotation is applied during the early stages of design where lower-level models typically are not abundant. In a way, our calibration methods can be considered as a form of back annotating the latency tables in Sesame's architecture models using results from ISS simulation and/or automated component synthesis.

Related to our on-line calibration technique, much work has been performed in the field of mixed-level HW/SW co-simulation, mainly from the viewpoint of co-verification. This has resulted in a multitude of academic and commercial co-simulation frameworks (e.g., [2, 1, 3, 42, 32, 10]). Such frameworks typically combine behavioral models, ISSs, bus-functional models or HDL models into a single co-simulation. These mixed-level co-simulations generally need to solve two important problems: i) making the co-simulation functionally correct by translating any differences in data and control granularity between simulation components, and ii) keeping the global timing correct by synchronizing the simulator components and overcoming differences in timing granularity. The functionality issue is usually resolved using wrappers, while global timing is typically controlled using either a parallel discrete-event simulation method [31] or a centralized simulation backbone using e.g. SystemC [32, 10]. Synchronization between simulation components usually takes place with the finest timing granularity (i.e. lowest abstraction level) as the greatest common denominator between components. E.g., system-level co-simulations with cycle-accurate components are typically synchronized at cycle granularity, causing high performance overheads. Besides the performance overheads caused by wrappers and time synchronization, the IPC mechanisms often used for communication between the co-simulation components may also severely limit performance [55], especially when synchronizing at cycle granularity.

In the mixed-level co-simulation that results from our on-line (trace-)calibration technique, we take the opposite direction with respect to maintaining global timing. Instead of

synchronizing simulation components at the finest timing granularity, it maintains correct global timing at the highest possible level of abstraction, being the level of Sesame's abstract architecture model components. As shown in [105], the performance overhead caused by wrappers and time synchronizations is in that case reduced to a minimum. Our on-line calibration technique shows some similarities with the trace-driven co-simulation technique in [55]. However, the latter operates at a lower abstraction level and is applied in a classical HW/SW co-simulation context.

In particular with respect to the signature-based calibration technique, we note that much work has been performed in the area of software performance estimation [8], including methods that use profiling information, typically gathered at the instruction level. For example, in [9] a static software performance estimation technique is presented which uses profiling at the instruction level and which includes the modeling of pipeline hazards in the timing model. In [38], a source-based estimation technique is proposed using the concept of "virtual instructions". These are similar (albeit a bit more low level) to our AIS instructions, but which are directly generated by a compiler framework. Software performance is then calculated based on the accumulation of the performance estimates of these virtual instructions. The idea of convolving application and machine signatures, where the signatures contain coarse-grained system-level information, has also been applied in the domain of performance prediction for high-performance computer systems [92].

As In [24], a workload modeling approach based on execution profiles is discussed for statistical micro-architectural simulation. Because they address micro-architectural simulation, their profiles include much more details (such as pipeline and cache behavior), while we address the system level at a higher level of abstraction. In [50], the authors suggest to derive a linear model from a small set of simulations. This method tries to model the performance of a processor at a mesoscopic level. For example, cache behavior and pipeline characteristics are taken into account. The significance of all cache and pipeline related parameters is determined by simulation-based linear regression models. This may be comparable with the 'weight' vector discussed in Section 4.6.2. Another interesting approach is presented in [93], in which the CPI for in-order architectures is predicted using a Monte Carlo based model. The Milan framework [70] deploys a design pruning approach using symbolic (instead of analytic) analysis methods to reduce the design space that needs to be explored with simulation.

4.8 Conclusion

In this chapter, we have presented an efficient mixed-level co-simulation technique, called trace calibration, which improves Sesame's abstract models by poviding more accurate values for use in the architectural model components' latency tables. This technique has been prototyped within our Sesame modeling and simulation framework, which targets efficient system-level design space exploration of embedded multimedia systems. To evaluate trace calibration, we have used a Motion-JPEG case study in which we incorporated up to four external instruction-set simulators into Sesame's abstract performance models. These ex-

periments show that trace calibration only requires minor modification of the incorporated simulators and that performance overheads due to co-simulation are very low. It was also demonstrated that distributed co-simulation – which is easy and transparent in trace calibration – allows for effectively reducing the slowdown due to the incorporation of lower-level simulators.

A disadvantage of the trace calibration method is that it has to be performed anew for each application. The signature-based calibration technique is aimed at finding a characteristic "signature" that summarizes in an application independent way the computational capacity of a processor component. Experiments showed that although signature-based performance models accurately represent relative performance behavior, achieving correct absolute performance behavior is more difficult. The latter requires very careful selection of benchmarks to train the model. This is, however, not a major objection, since in the early stages of system design, determining relative performance of different design options is often more important than absolute performance.

In summary, both trace-calibration (on-line and off-line) and signature based calibration techniques can be effectively used to create system-level performance models with relative ease. Choosing the right technique for a given design problem depends on the required speed-accuracy trade-off, the availability of suitable lower-level simulators and the ability to create good training sets. Encouraged by the results of the on-line trace-calibration method, we have started work on a co-simulation technique where event trace measurements are performed on actual hardware, instead of an ISS. The technique is completely analogous to the on-line trace calibration method, but now the API functions transfer data and cycle measurements (Section 4.3) directly to a process running on an FPGA prototype platform. An advantage of this "hardware-in-the-loop" co-simulation is that the measured execution latencies are 100% accurate, whereas an ISS-simulator often is still an approximation of the actual target processor. However, our initial results showed that synchronization overhead is larger than in the case of on-line trace-calibration with ISS, due to the communication mechanism with the stand-alone FPGA board. Nevertheless, we will report on this method in future work, since it is likely to provide yet another valuable option for the designer with respect to the speed-performance trade-off.

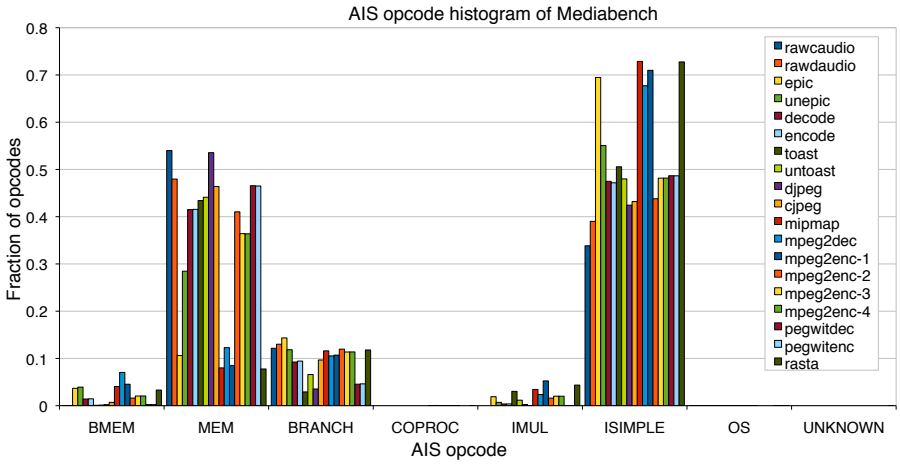


Figure 4.10: Histogram with the various AIS opcode counts of the Mediabench training set.

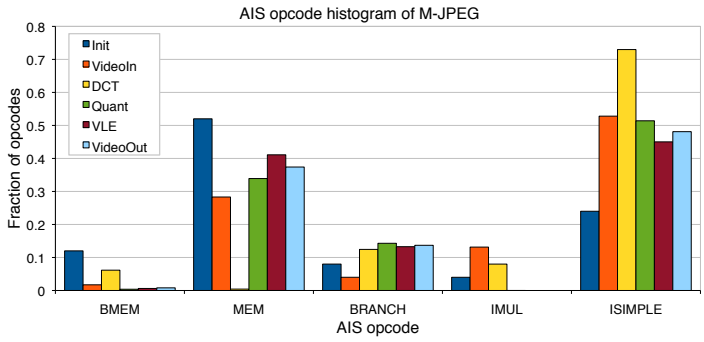


Figure 4.11: Histogram with the various AIS opcode counts of M-JPEG.

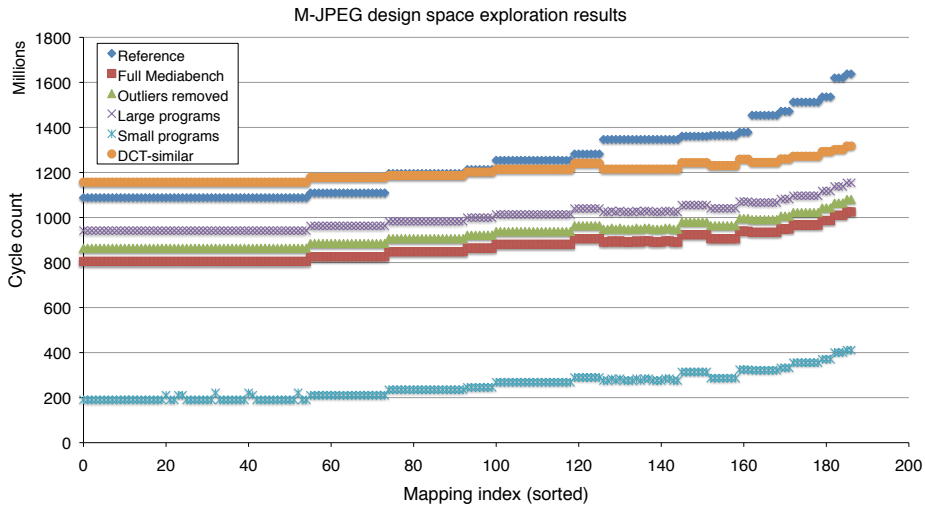


Figure 4.12: Comparing the DSE results from our Mediabench-trained models

Chapter 5

Multi-application modeling

5.1 Introduction

In Chapter 1 we described the many ways in which modern embedded systems are becoming increasingly complex. It can be argued that this trend is driven for the most part by the increasing requirements of the end-user of the system. Apart from increasing reliability, performance and quality requirements, possibly the most challenging requirement is the integration of many functionalities on the same device. Where previously embedded systems were dedicated to a single task or a small set of tasks, modern embedded systems need to support an increasing variety of tasks. A good example is the modern mobile phone, which in addition to its primary communication function now also supports functions such as photo and video capturing, music playing, gaming, as well as browsing and office applications. The latter functions previously belonged to the domain of dedicated devices such as cameras, mp3 players, or the domain of console or desktop computing. It is a major challenge to combine all these functionalities together with additional non-functional design requirements for (mobile) embedded systems, such as power usage, cost and form factor.

So far, Sesame has only supported the mapping of a single application onto an architecture model at the time. But since modern multimedia embedded systems are increasingly multi-tasking, we need to address the modeling of effects of executing multiple applications concurrently in our system-level architecture models. In this chapter, we present two multi-application workload modeling techniques in Sesame. One technique is based on the use of synthetic application workloads while the second technique deploys only real application workloads to model concurrent execution of applications. Synthetic application workloads are particularly useful in the early design stages, since they enable (partially) parallel development of the application and architecture model. For example, the architecture model can already be tested for functional correctness while the application model is still being finalized. As will be shown in later sections, another benefit of synthetic workloads is that their parameters can be easily adapted to test the behavior of the system under specific workload conditions.

Additionally, in this chapter we will propose some ideas to combine multi-application

workloads in such a way that they contain dynamic behavior. Dynamic behavior within application models may exist at two levels: at the level of applications, or within the application at the level of processes. We refer to the former as inter-application scenarios and the latter as intra-application scenarios. Both types of scenarios need to be studied, since they can have a great impact on the workload which is to be processed by the underlying system architecture.

This chapter is organized as follows. Section 5.2 presents the two proposed multi-application workload modeling techniques for Sesame. First a synthetic multi-application model is introduced, followed by a multi-application model consisting of real applications. In Chapter 3 we discuss how various features in Sesame help the designer to create multi-application models with relatively little effort. Subsequent Sections 5.4.1 and 5.4.2 deal with the modeling and representation of dynamic inter and intra-application workloads in Sesame. Experiments showing the various techniques presented throughout the chapter are in Section 5.5. Related work is in Section 5.6 and we conclude the chapter in Section 5.7.

5.2 Multi-application workload modeling

As mentioned before, Sesame has up to now only supported the mapping of a single application onto an architecture model at the time. Modern multimedia embedded systems are however increasingly multi-tasking. Therefore, we need to address the modeling of effects of executing multiple applications concurrently in our system-level architecture models. To this end, we propose two multi-application workload modeling techniques. One technique, which we will discuss first, is based on the use of synthetic application workloads while the second technique deploys only real application workloads to model concurrent execution of applications.

5.2.1 Synthetic multi-application workload modeling

Multi-application modeling using synthetic application workloads is illustrated in Figure 5.1. Note that the FIFO buffers between virtual processors are not depicted in Figure 5.1 for the sake of simplicity. On the left-hand side, a Sesame system-level model with a single, primary application is shown. The three processes in this application are mapped onto two processing cores (P0 and P1) in the underlying architecture. Since processes A and B are mapped onto the same resource, a scheduler named Local-Scheduler (or L-Scheduler) is used for scheduling the workloads (i.e., application events) from both processes. However, a second level of scheduling hierarchy is added by introducing so-called Global-Schedulers (or G-Schedulers). These global schedulers are basically equivalent to local schedulers in terms of functionality but instead of intra-application events they schedule application events from different applications. Evidently, the local and global schedulers can also deploy different scheduling policies. When, for example, the interleaving of processes inside an application is statically determined at compile time, the local scheduler can model this by ‘merging’ the events from the event traces according to this given static schedule. At the same time, the

global scheduler can schedule application events from different applications in a dynamic fashion based on, for example, time slices, priorities, or a combination of these two. Here, we would like to note that although the schedulers support preemptive scheduling, this can only be done at the granularity of application events. The simulation of a single application event is atomic and thus cannot be preemted in Sesame. Furthermore, we currently do not model any overheads caused by the context switching itself (e.g., OS overhead, cache misses, etc.). This is considered as future work.

In synthetic multi-application modeling, the application events external to the primary application (see Figure 5.1) are generated by a stochastic event generator. Hence, this event generator mimics the concurrent execution of one or more application(s) besides the primary application. Based on a stochastic application description, which will be discussed later on, the application generator generates traces of EX(ecute), READ and WRITE application events and issues these event traces to special virtual processors, indicated by VP_S in Figure 5.1. Multiple instances of these event generators, each with their own stochastic application description, can be used to model concurrent execution of more than two applications.

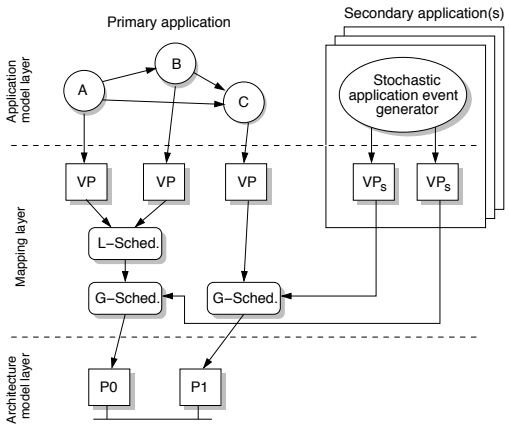


Figure 5.1: Multi-application modeling using synthetic application workloads.

The virtual processors (VP_S) used for the trace events from the stochastic event generator are special in the sense that they, unlike normal virtual processors, are not connected to each other according to the application topology (see Section 3.2.3 in Chapter 3). Rather than explicitly modeling communication synchronizations, a VP_S models synchronization behavior stochastically. To illustrate the interactions between the event generator, a VP_S and a global scheduler of a system-level model, consider Figure 5.2. The figure shows these interactions in the case an "EX(A) , EX(B) , READ , WRITE " event sequence is generated by the event generator. At (simulation) time t_0, the EX(A) event is consumed by the VP_S. The VP_S immediately forwards this event to the global scheduler it is connected to, and waits for an acknowledgment from the scheduler. After the EX(A) event has been scheduled for execution on the architectural resource (taking T(sched) time units) and the actual execution (taking T(A) time units), control is returned to the VP_S by sending it an acknowledgment. Hereafter, the VP_S can consume another application event again. In the case of the example in Figure 5.2, the VP_S now consumes the EX(B) event which is handled in an identical fashion as the EX(A) event. However, VP_S handles the READ and WRITE events, which are consumed at times t_2 and t_3 respectively, in a slightly different way. Instead of directly forwarding these events to the global scheduler, like is done with EX events, VP_S now first models a synchronization latency. This latency refers to the time the read and write transactions need to wait for data or room in the

buffer from/to which is read/written. The synchronization latency, indicated by T(sync) in Figure 5.2, is a stochastic parameter of VP_S, as discussed below.

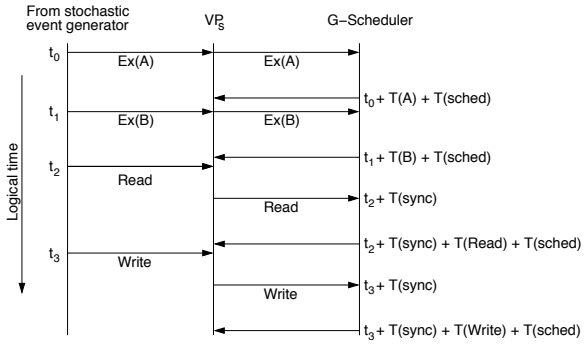


Figure 5.2: Interaction between Virtual Processor (VP_S) and G(lobal)-Scheduler in synthetic multi-application modeling.

Table 5.1: Parameters for the synthetic application workload generation.

Stochastic event generator parameter	Description
A_{Ex}	Set of possible Ex(ecute) application events
P_{Ex_i}, with $\sum_{i \in A_{Ex}} P_{Ex_i} = 1$	Probabilities of the different events in A_{Ex}
r_{comp}:r_{comm}	Computation to communication ratio
r_{read}:r_{write}	Read to write ratio
M	Set of possible message sizes
P_{M_i}, with $\sum_{i \in M} P_{M_i} = 1$	Probabilities of the different message sizes
NP	Number of communication ports
P_{port_i}, with $\sum_{i=0}^{NP} P_{port_i} = 1$	Probabilities of the different port usages
VP_S parameter	Description
$Sync_{Read}$	Mean synchronization latency for reads
σ_{Read}	Standard deviation of read latencies
$Sync_{Write}$	Mean synchronization latency for writes
σ_{Write}	Standard deviation of write latencies

Table 5.1 lists the parameters used by the stochastic event generator as well as a VP_S. These parameters can be specified both globally – describing the behavior for all traces (for the event generator) or ports (for a VP_S) – and on a per-trace/per-port basis. Descriptions on a per-trace/per-port basis overrule global descriptions, in the case there is an overlap of both types of descriptions. The parameter A_{Ex} specifies the set of possible EX events that can be generated. For example, $A_{Ex} = \{DCT, VLE\}$ specifies that EX(DCT) and EX(VLE) events can be generated. P_{Ex_i} describe the probabilities of the events in A_{Ex}.

The ratio's r_{comp}:r_{comm} and r_{read}:r_{write} specify the computation to communication ratio and read to write ratio, respectively. So, for example, by increasing the r_{comp}:r_{comm} ratio, the application behavior can be made more computationally or communication intensive. The parameter M specifies the set of possible message sizes that can be used in communications. In multimedia applications, application data is often communicated in fixed data chunks (e.g. pixel blocks) from one application phase to the other. P_{M_i} specify the probabilities of the different message sizes. NP denotes the number of communication ports for which read and write transactions can be generated. P_{port_i} are the probabilities of the different port usages. Again, all of the above parameters can be specified globally (valid for all event traces) or on a per-trace basis.

The VP_S parameters $Sync_{Read}$ and $Sync_{Write}$ specify the mean synchronization latency for read and write transactions, respectively. σ_{Read} and σ_{Write} contain the standard deviations of the two aforementioned means. By default, a VP_S uses an Erlang distribution to determine synchronization latencies. These VP_S parameters can again be specified globally (valid for all communication ports of a VP_S) or on a per-port basis.

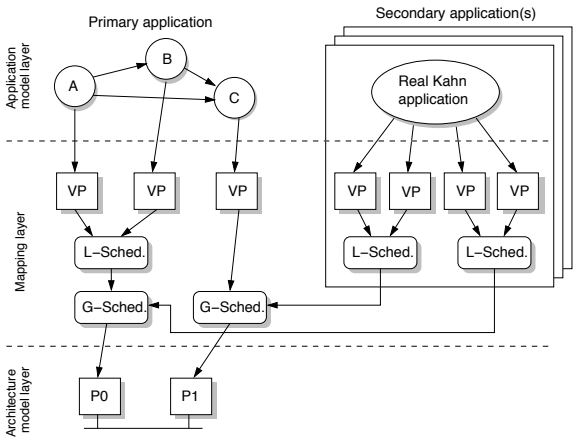


Figure 5.3: Multi-application modeling using realistic application workloads.

5.2.2 Realistic multi-application workload modeling

In our second multi-application workload modeling technique, we realistically model the concurrent execution of multiple applications. That is, multiple Kahn application models are actually executed concurrently, as shown in Figure 5.3, and produce realistic event traces that are again scheduled on the underlying architectural resources using the global schedulers. In contrast to synthetic workload modeling, the secondary KPNs use normal virtual processors in the mapping layer. Hence, synchronization behavior in the parallel applications is modeled explicitly for all participating KPN applications (i.e., there is no difference between primary and secondary applications). This implies that, when considering Figure 5.2, the T(sync) now refers to the actual synchronization times between application processes.

Moreover, the secondary KPNs also require L-schedulers to 'merge' (i.e. schedule) event traces when multiple application tasks are mapped onto a single architecture resource. Naturally, the policies of the L-schedulers can vary between the different KPN applications taking part in the system simulation. When considering Figure 5.2, we now have T(sched) = T(L-sched) + T(G-sched) for all participating KPNs.

5.3 Multi-application modeling: a designer's perspective

In the previous sections (as well as the remainder of this chapter) different modeling techniques are discussed that extend Sesame's modeling capabilities. In this section we discuss how these modeling extensions can be implemented in Sesame in such a way that the implementation effort is minimal. As we are targeting models for use in the early stages of system design, it is important that all techniques can be used in an easy and straightforward way, in order not to slow down the design process. In this section we will describe some of the ways our tools can support reduction of the modeling effort for the previously proposed techniques.

In Chapter 3 the different layers within a Sesame model and their respective role in the system model were shown in detail. It was described in Section 3.2.3 that the virtual layer is automatically instantiated before the start of the simulation. This instantiation process uses *templates* which are associated with each processing or communication component in the architecture model. The application mapping (which maps tasks and channels to their respective targets: processing and communication resources in the architecture layer) has an additional parameter to specify which template to use for each component in the virtual layer. In this way, different synchronization behaviors can be contained in different templates and the virtual processors will be automatically instantiated with the right synchronization behavior. Since they implement a special kind of alternative synchronization behavior, stochastic virtual processors (i.e. VP_S in Figure 5.1) are implemented as templates too. In Figure 5.1, the details of the stochastic event generator are not shown, but it does in fact consist of multiple (stochastic) processes (just like a regular KPN). Therefore, mapping stochastic processes is as easy as mapping ordinary processes, with the only addition that the VP_S parameters need to be specified (see Table 5.1. Moreover, when the designer uses the model specification GUI (Section 3.3.5), the designer can simply drag-and-drop a generic processor component from the library, which already includes a VP_S template.

Secondly, we discuss the process of creating a stochastic application model. In Figure 5.1, the stochastic application event generator is shown without details such as individual stochastic processes and their channels. Creating the internal topology of the stochastic application event generator is currently a manual process: using the GUI, the designer can drag stochastic processes onto the canvas and connect them with channels in any topological structure. Again, the designer then only needs to fill in specific values for the latency tables which can be done either globally or on per-port/per-trace basis (Table 5.1). There can be situations where the designer wants to quickly instantiate a KPN without detailed control over the topology. For this purpose we envision a new tool for the GUI that instantiates a

topology according to a few parameters, such as the number of synthetic processes and the average number of channels per process. Using only a few parameters, various (a)cyclic, low or fully connected KPN topologies can be instantiated which the designer can further tailor to need. Instantiation of stochastic virtual processors is again done automatically, using the template mapping as mentioned before.

The final tool-related issue that we address here concerns the parameters for the stochastic virtual processors (Table 5.1). In particular we propose a way to automatically derive initial values for the VP_S parameters. As observed before in Section 3.2.3, the synchronization latencies of virtual processors consist of all kinds of delays. Eg. when a READ event is blocked this may be due to the fact that the writing process is waiting for some processes that are occupying shared resources (eg. processors or memories). A possible initial setting for the VP_S parameters is to base the mean synchronization latency on the workloads of all other processes that share the same resource. For example, we could use the average of all EX event latencies for processes that share the same resource. As an alternative to the stochastic virtual processor, we propose an auto-tuning stochastic virtual processor that "learns" an acceptable set of synchronization latencies during a trial-run simulation. Initially, for each channel connected to the virtual processor an initial synchronization latency of 0 is used. This value is automatically adjusted at model runtime by counting the workload and frequency of EX events that share the same resources. For example, the $Sync_{Read}$ value of a process is auto-tuned to the average workload of the process that writes to that channel. Note that there are various sources of inaccuracy in the auto-tuning method. For example, there may be other latencies in the architecture model (other than the EX events) that contribute to the synchronization latency. Despite such inaccuracies, the auto-tuning synthetic virtual processors provide initial values for the VP_S parameters, which can be refined by the designer.

We conclude that the designer has a range of options available to create a stochastic application model in the early stages of design with a relatively small engineering effort. Additional effort is necessary only in cases where if the designer wants more detailed control of the stochastic application's properties or when the designer wants the stochastic application to match the behavior of a primary (real) application model. An automatic topology generator and auto-tuning synthetic virtual processors were introduced as future Sesame extensions to reduce even further the effort to create synthetic models.

5.4 Dynamic application behavior

In this section we discuss how Sesame's real and synthetic application models can be adapted to represent dynamic application behavior. We distinguish two types of dynamic application behavior: inter-application behavior (between applications) and intra application behavior (between processes within a single application KPN). A variation of the dynamic inter-application behavior presented in the first subsection below, has been used in a case study that considers a partially dynamic reconfigurable architecture (see Section 6.5). Dynamic intra-application behavior is already available in some of the realistic application models

that we use in Sesame. However, the synthetic application generator from Section 5.2 does not explicitly support this. In Section 5.4.2, we propose a method to add dynamic intra-application behavior to stochastic Sesame models.

5.4.1 Dynamic inter-application behavior

A multi-application model in Sesame consists of two or more disjunct KPNs, which have no dependencies other than being mapped onto shared resources. Although this is a perfectly valid Sesame application model, it is unable to capture important dynamic inter-application behavior that is particularly relevant to modern embedded systems where tasks may enter and leave the system at any given time. For example: when a mobile-phone user takes a picture, the camera application tasks will put a temporary additional load on the system's resources. In this way, application loads can be generated with a huge dynamic range: an additional application can for example double the workload on the underlying architecture. These types of workloads are sometimes called user-scenarios, since the arrival of a new application is often (but not necessarily) initiated by the user or the environment. We will refer to them as inter-application scenarios to distinguish them from intra-application scenarios (the topic of the next section). In the following, we will shortly discuss two methods that allow the modeling of dynamic inter-application workloads in Sesame. Some of the problems that need to be solved are 1) KPNs are not naturally suited for modeling dynamic/reactive behavior at the inter-application level and 2) KPN is an untimed model of computation which complicates the specification of the arrival *time* of a sporadic application.

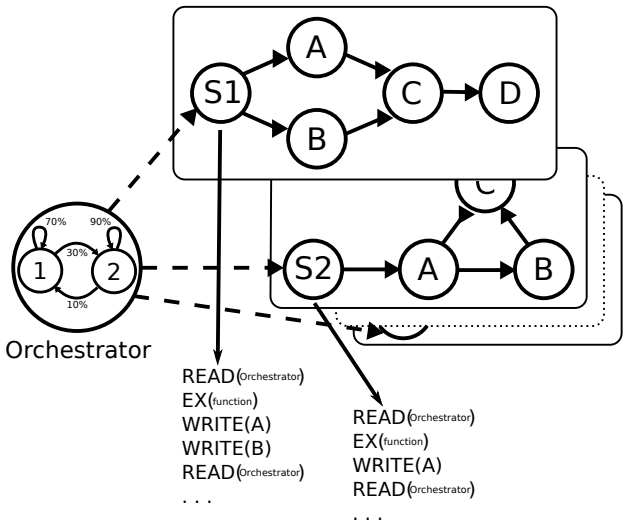


Figure 5.4: Example of a multi-application workload with dynamic inter-application behavior using a Markov-model orchestrator node.

Here we will assume that each KPN in our multi-application model has one or more "source" nodes as this is common in our targeted streaming media application domain.

Source nodes have no initial dependencies on other nodes and are therefore immediately runnable. An example source node is a node that feeds raw input data into the application (either from a framebuffer memory, a file, or a raw data stream: camera, or microphone, etc.). Therefore, the KPN is usually alive for as long as the source node feeds data into the KPN.

Now, suppose we create a multi-application model in Sesame consisting of two or more KPNs with source nodes as described above. Sesame's application simulator will start running all KPNs concurrently and therefore, by default, we can only model the single scenario where all applications start running at the same time (though possibly with different priorities as specified in the G-schedulers in the virtual layer. In the first method to model dynamic multi-application workloads, we remedy this situation by introducing an *orchestrator* node as shown in Figure 5.4. The purpose of the orchestrator node is to put an artificial (read) dependency on each source node such that each of the KPNs can only start according to a scenario defined by the orchestrator. Note that in this way the orchestrator can not only introduce applications to the scenario, but also remove them. For example, let us assume that the second application in Figure 5.4 is a video encoder and that source node S2 (after the orchestrator dependency READ (Orchestrator) reads out a framebuffer (represented by the EX(function) event) and passes it as macroblocks to node A. The orchestrator can include the application in the current scenario by sending a token on its corresponding output control channel. Vice versa, it can exclude the application by not sending the token. Using the control tokens, the orchestrator can create those scenarios and scenario transitions that are of interest to the designer. We propose here to use a Markov chain to represent the transitions between scenarios where each state represents a scenario (each with its respective control tokens) and transitions are given by probabilities in the normal way (see Figure 5.4).

One problem with the orchestrator approach described above, is that it is impossible to define transitions between user-scenarios at certain time intervals. This would be necessary to implement the behavior where the system reacts to externally timed events such as the occurrence of interrupts (e.g., a user presses a button on the TV's remote control after which teletext is started as a picture-in-picture application on the screen). Since KPNs are an untimed model of computation, it is not possible to express the behavior "wait n time units and then start application X". For this purpose, we define a special 'SLEEP(N)' application event, which basically indicates that an application process is not active during a period of N time units. The SLEEP event is created by a special event annotation in a Kahn process and it takes n as its only argument. As with all events, it is passed to the virtual layer, where it is consumed by a virtual processor (it is not passed on later to the architecture model). The virtual processor exists in the same timed simulation domain as the architecture model, so that we can implement the desired behavior. In the virtual processor, the SLEEP event causes the virtual processor to sleep (i.e. block in virtual time) for the specified period. While the virtual processor is blocked in this way, it will be unable to continue its normal processing of EX, READ , and WRITE events, thus effectively suspending the application process. Note that by suspending one process in the KPN of an application, the other processes will soon stall because of (in)direct unmet dependencies on the suspended process. We can prevent

suspending the application mid-stream (with unprocessed data before or after the suspended process) by issuing sleep events only to the source nodes (this situation is shown in Figure 5.5).

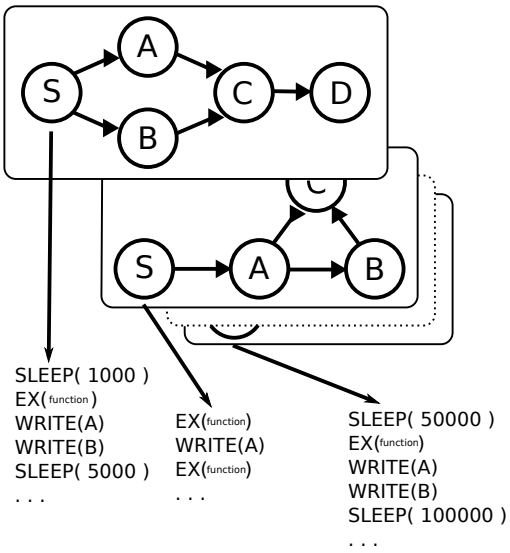


Figure 5.5: Example of a multi-application workload with dynamic inter-application scenarios implemented using the SLEEP application event.

Evidently, the SLEEP events provide the opportunity to freeze the issuing of application events for a while, which basically mimics sporadic or periodic execution behavior of applications. The sleep event technique will work transparently for both realistic and stochastic application workloads alike. Compared to the sleep-event technique, the orchestrator technique has the advantage that the scenarios and use-cases can be defined in a single location in the model. These two techniques enable the designer to assess a variety of different scenarios or use cases [36]. A technique based on the orchestrator approach will be used in Chapter 6 to create a pseudo-dynamic workload for an architecture with dynamic reconfiguration capabilities.

5.4.2 Dynamic intra-application behavior

The stochastic application workload proposed in Section 5.2 consists of stochastic processes (and stochastic virtual processors) which use a parameterizable, yet fixed event generation distribution. In realistic non-trivial applications, it is often the case that the application (and its separate processes) move through different execution phases. We would like the synthetic trace event generator to create events according to a unique probability distribution in each of those phases. We refer to each possible combination of simultaneously occurring process phases as an intra-application scenario. They differ from user scenarios (or inter-application scenarios) as they occur at the process (task) level within applications and are typically not directly related to any action by the user.

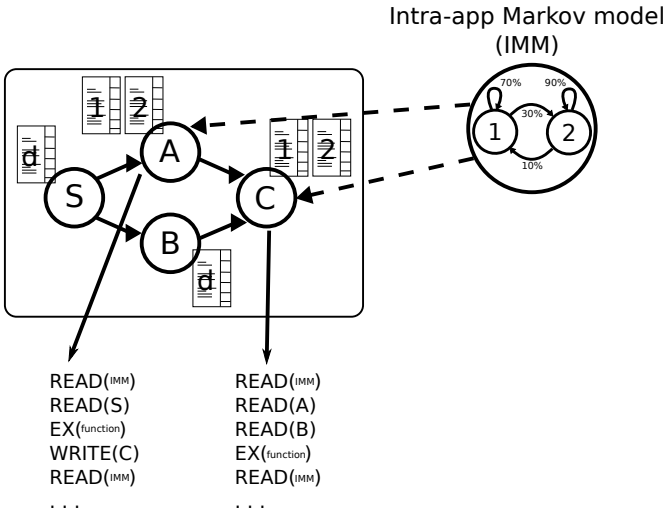


Figure 5.6: Example model and traces for a stochastic application with dynamic intra-application behavior.

In Sesame, there are multiple ways in which intra-application scenarios can be specified for the purpose of stochastic modeling. We propose here a solution analogous to the previously discussed orchestrator process by using a Markov chain to traverse different process phases. In this case, each stochastic process contains multiple probability tables: one for each scenario in which it can be involved. Note that if a stochastic process is not part of an explicit scenario, then it simply uses its default probability table. The Markov model exists outside of the application graph and each of its states represents a scenario. The node containing the Markov model is connected by control channels to all stochastic processes that contain multiple probability tables. The granularity with which the Markov model changes state is to be determined by the designer. For example, if a node is a typical streaming process containing a Read-Execute-Write loop, then it makes sense to advance the Markov model at the beginning of the loop.

In Figure 5.6, an example is given where the application can run in two distinct scenarios (1 and 2). Processes A and C can take part both in scenario 1 and 2; depending on the value from their respective control channel they use either probability table 1 or 2. The Markov model lists a high probability for scenario 2, but approximately 10% of the time, the application jumps back to scenario 1. Processes S and B have the same behavior in each scenario and therefore contain only a single (default) table. In order to synchronize the switching between scenarios, it may be necessary that the Markov model (IMM) is implemented outside the KPN, not adhering to normal KPN communication rules. Note that an implementation of the proposed intra-application method has not yet been implemented and remains as future work.

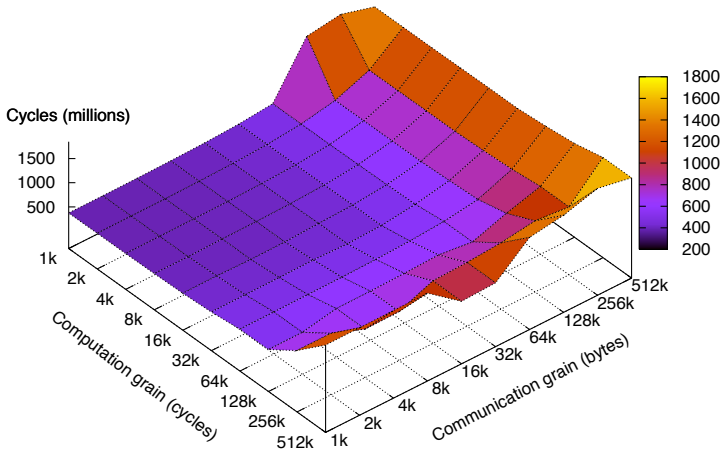


Figure 5.7: Estimated execution times of concurrent execution of M-JPEG and producer-consumer applications. The latter is parameterized in both computation and communication grain size.

5.5 A preliminary case study

For illustrative purposes, we performed a small experiment using the multi-application workload modeling support now available in Sesame. More specifically, we modeled two Kahn applications that execute concurrently. The first (and primary) application is a Motion-JPEG (M-JPEG) encoder, and the other one is a synthetic 'producer-consumer' application transferring data from producer to consumer. The M-JPEG application encodes 8 consecutive 128x128 resolution frames, while the producer-consumer application is parameterizable in both computational and communication load. That is, the producer iteratively models a parameterizable computing latency after which it sends a parameterizable chunk of data to the consumer. In our system-level model, both applications are mapped onto a multi-processor SoC, containing 4 processors with distributed memory and connected through a crossbar switch. We applied a simple round-robin policy for scheduling tasks from both applications at the G-schedulers (see Section 5.2.2).

Figure 5.7 shows the estimated system-level execution times (combined for both applications) when varying the computation and communication grain sizes of the producer-consumer application. As can be seen from Figure 5.7, the results show a quite predictable behavior, which helps to gain trust in our multi-application modeling method. That is, the system performance is only marginally affected for small computation and communication grains of the producer-consumer application. But after a certain threshold, the producer-consumer application starts to dominate the system performance (computation-wise, communication-wise, or both).

As future work, we plan to actually validate multi-application modeling results against a Daedalus prototype implementation. Note that for a useful validation, we may have to extend the model to capture latencies associated with the (embedded) operating system that

schedules the tasks from multiple applications. This would be necessary, for example, if there is a significant context switching overhead. We also note that the L-schedulers have been frequently used in single-application workloads to enforce a specific scheduling of application tasks. For example, in the validation case studies in Chapter 8, the modeled MPSoC uses a static scheduling of tasks. Therefore, a scheduler is associated with each processor in order to schedule events according to a fixed, static scheduling policy.

5.6 Related work

The modeling of (parallel) workloads for the purpose of performance analysis is a well-established research domain, both in terms of realistic and synthetic workload modeling (see e.g. [56, 28, 91]). A recent focus area is, for example, statistical simulation for micro-architectural evaluation [23]. In this technique, a stochastic program description, which is a collection of distributions of important program characteristics derived from execution profiles, is used to generate synthetic instruction traces. These synthetic traces are subsequently used in trace-driven processor and/or memory-hierarchy simulations. This work focuses on generation of applications at the instruction level, since it is targeted towards execution on instruction set simulators. Although we target a different domain (system-level multi-processor models), it is interesting to compare the similarities and differences between the approaches. The work of [22] is motivated by the need to find workload representations (a stochastic model) that have smaller storage requirements than instruction traces from actual applications (since reading large traces slows down simulation time significantly). On the other hand, our high-level event traces for a single application are fairly small and trace sizes are only problematic in very special cases (see the work of [111] described below). Additionally, compared to [22] our work is not focused on finding stochastic models that accurately represent workloads of actual applications. Instead, our focus is on providing the designer with tools to generate stochastic workloads for use in the very early design stages when the final workload of the system may still be (partially) unknown. Where [22] produces traces with syntactically correct read and write dependencies, the stochastic event generators presented in this chapter do not. Instead, we include the read/write dependencies in the stochastic model itself to give the designer maximal flexibility to experiment with different workload characteristics.

Another area in which synthetic workload modeling has recently received a lot of attention is network workload modeling for network-on-chip simulations [113, 103, 65]. In [62, 68], multimedia application workloads are described and characterized analytically using so-called variability characterization curves (VCCs) for system-level performance analysis of multi-processor systems-on-chip. These VCCs allow for capturing the high degree of variability in execution requirements that is often present in multimedia applications. A fair number of research efforts addressed the high-level modeling of a RTOS to be used in system-level models for early design space exploration [35, 41, 60]. Rather than focusing on how to model multi-application workloads, these efforts mainly address abstract modeling of RTOS functionality, efficient simulation of this functionality, and refinement of these

abstract RTOS models towards the implementation level.

In the work of [77] design space exploration case studies using a modified simulated annealing technique are performed on randomly generated KPN graphs. A closer look at their KPN generator reveals that it takes only a few simple parameters to create KPNs with random topology, communication patterns and execute annotations. These parameters specify global properties of the KPN: the number of processes, number of communication events, size of communicated data, execution time and avg. number of channels per process, etc. In contrast to our stochastic event generator, their approach produces syntactically correct event traces, but there is very little control on the properties of specific processes and channels. In this way, some properties that are typical streaming media applications are not captured, eg. the fact that communication or execution events for a specific process often have a limited choice respectively for data size or event type. Of course, there may be situations where the designer does not require those workload properties. Therefore the KPN generator from [77] is a useful addition to the Sesame toolbox (in fact, we use it in Chapter 8).

The idea to represent transitions between different scenarios or application process phases as a Markov model is not new (see for example [102]). Markov chains provide a clear and natural way to model transitions between a finite number of states with (simulated) non-deterministic behavior. In our approach the Markov model determines the stochastic event generator's probability functions: each state corresponds to a different set of probability functions for certain processes in the KPN. In [102], the Markov model is integrated in an extended data-flow (SDF) model called Scenario Aware Dataflow (SADF). SADF integrates special control nodes (detectors) into the application graph, which use control channels to steer the current scenario to each of the nodes that participate in a certain scenario. In each firing, the detector determines the new scenario according to one of its integrated Markov models. This is quite similar to our approach, except that in our case the control channels "select" the process' probability table, whereas in their approach it selects a specific token production and consumption rate. Another difference is that in SADF the state advancement of the Markov model occurs with every firing of the detector node, whereas in our case it is user-defined. We note that the SADF models in are more suitable for static analysis, but are not as expressive as KPN models. Another difference is that the SADF work in [102] has a focus on approximating realistic application workloads, which in our case is not the primary concern. For an extensive review of application modeling approaches based on the integration of finite state machines with dataflow models, we refer to [26].

A very interesting continuation of multi-application modeling performed in the context of the Sesame simulation environment is in [112, 110]. From this work we borrowed the intra and inter-application scenario terminology used throughout this chapter. The main contribution of [111] consists of a method to efficiently identify intra and inter-application scenarios from a multi-application workload as well as a co-exploration technique based on genetic algorithms. Scenarios for a certain workload are identified by executing a multi-application model (may contain both real and stochastic applications) and analyzing the generated event traces. The scenarios are then stored in a tree-structured database where the leaf nodes consist of all unique trace segments that can be produced by a single process. One level higher

in the tree stores a representation of the intra-application scenarios: it lists (for each application) combinations of communicating trace segments of individual processes that occur in the workload. At the root of the tree the inter-application scenarios indicate which of the applications in the workload were identified to execute simultaneously. By efficient loop-compression of the trace segments in the leaf nodes, this hierarchical scenario database has minimum storage requirements and is subsequently used for the co-exploration to identify solutions (architectural design points/mappings) for the problem space (all possible scenarios as stored in the database). The problem to be solved is that evaluating all possible scenarios for all possible design points is infeasible and that it is very hard to find a smaller subset of scenarios that represents all scenarios. The proposed co-evolution technique is shown to produce good results by simultaneously searching for optimal design points *and* a representative subset of scenarios.

Note that the work in [111] and the work in this chapter are complementary, but are geared towards different phases in the design process. Where [111] has a focus on full-blown design space exploration (when the system's workload is known), the work in this chapter is geared towards early model development (when the final workload is typically unknown). Moreover, the work in [111] does not consider the dynamic inter-application phase transitions other than those that were present during scenario identification. In order to create new inter-application scenarios, the orchestrator-technique from this chapter could be applied.

5.7 Conclusion

In this chapter we have shown the mechanisms and techniques provided by Sesame to model multi-application workloads. This is required, since multi-application workloads are increasingly becoming standard in modern embedded system design. Furthermore, we have shown that Sesame has basic support for modeling synthetic in addition to the realistic application workloads. Initial support is available to model application workloads with dynamic inter- and intra-application behavior. The case study illustrated that stochastic modeling using synthetic trace generators is a useful tool to answer various "What if..?" questions about a system design candidate. For example: what happens to primary application performance if (after a certain time) an additional task with certain properties enters the system? Furthermore, an important benefit of stochastic application models is that they enable simultaneous co-development of application and architecture models. For example, the architecture model can be tested with one or more stochastic application models while the application model is still under development. This is a realistic case, since significant effort is involved in making an application model suitable for mapping onto an MPSoC by transforming it (for example) in a KPN. Such a co-development of application and architecture model can result in a significant reduction in the early stages of system design. In order not to increase development time at these early design stages, we consider it important that the techniques introduced in this chapter do not penalize the designer with a lot of additional implementation time. Therefore, we put quite some emphasis on the user-friendliness of the various stochastic

multi-application modeling techniques. We mentioned the possibility to use the library to contain standard scheduling components, scheduling policies and standard stochastic application generator components. Furthermore, the template mapping strategy proved to be helpful to reduce the effort that is otherwise associated with creating a stochastic application model. However, we have not yet deployed the techniques from this chapter (in particular the stochastic application models) in a large scale case study. This remains as future work.

Chapter 6

Modeling dynamically reconfigurable systems

6.1 Introduction

In recent years, MPSoC system development based on reconfigurable technologies (such as FPGAs) have received increasing attention from both research and industry. This is not surprising, as the cost of FPGAs goes down and gate count goes up, driven by the improvement of manufacturing technology. Modern FPGAs consist of hundreds of millions of gates, which is sufficient to implement complex MPSoC systems consisting of tens or hundreds of processing components as well as memories and on-chip interconnection networks. A major point in favor of FPGAs is their flexibility: the logic design of the FPGA system can be adapted to and optimized for a specific application or workload. Because of the flexibility of reconfigurable systems, the implementation gap (as described in Chapter 1) becomes even more apparent. As the amount of configurable resources continues to scale with technology improvement, tools and methods are needed to support the designer to make efficient use of those resources. This, as we explained before, is one of the main motivations of the Daedalus approach to system design (Chapter 2).

However, a newly emerging feature of modern FPGA hardware exacerbates the problem even further. *Partial dynamic reconfiguration* is the ability to change the logical configuration of parts of the FPGA at runtime, while other parts are still actively processing. This feature gives rise to a whole new dimension of performance optimization where tasks can be off-loaded onto hardware components that are instantiated at runtime. For example, commonly executed functions or compute kernels can execute on dedicated logic components in the FPGA. Also, thinking beyond the traditional processor/co-processor model, one can imagine the possibilities of an MPSoC on FPGA that can dynamically adjust its configuration according to design requirements and runtime conditions. Such a system could for example make trade-offs between performance and power by migrating tasks to fewer cores depending on the power supplied by the environment. Similarly it could allocate extra cores when dynamic variation in the workload require more processing power. However, from the

design perspective, this means that in addition to static, design-time optimization problem we now need to solve the dynamic, run-time optimization of the system as well.

In this chapter we will address the design problems associated with dynamically reconfigurable systems: in particular the modeling and simulation aspects. We will do so by extracting important characteristic features common to partial dynamic reconfigurable systems and consider issues related to modeling these features. In Section 6.2 it will be discussed how Sesame can be extended to support basic dynamically reconfigurable behavior. These extensions should be as generic as possible so as to be useful for modeling a wide range of dynamically reconfigurable systems. Subsequently, by means of an example, we will apply the extensions to an existing platform (the Molen polymorphic processor), which is described in Section 6.3. The resulting model is discussed in detail in Section 6.4. In Section 6.5, some of the results of the model will be presented as an illustration of the usefulness of modeling these kinds of system with Sesame. Finally, Section 6.6 will discuss related work and conclusions and recommendations for future work are given in Section 6.7.

6.2 Modeling dynamically reconfigurable systems

This section will discuss some of the specifics of modeling dynamically reconfigurable systems in Sesame. It will be shown in which areas the standard Sesame model (as explained in Chapter 3) has to be extended. For some extensions, it will prove necessary to make assumptions about the particular reconfigurable system that is to be modeled. However, the extensions listed in this section are designed to be as generic as possible and, where appropriate, design choices will be motivated. In part, this section will serve as an introduction to the techniques that will be used to create a model for the Molen reconfigurable architecture (Section 6.4). Additionally, this section aims to provide some practical insight into the process of reusing basic Sesame components and adapting the standard Sesame model in order to model different kinds of systems. Finally, it will be illustrated here (and in the remainder of this chapter) that Sesame is quite capable of exploring a wide variety of systems.

6.2.1 Dynamic allocation of model components

The reconfigurable fabric of FPGAs can be configured into many different types of logical components: processors, memories, interconnects, etc. Sesame is able to model MPSoC systems that are completely implemented on FPGA (such as for example the Daedalus prototype platforms 2), or MPSoC systems that contain on or more FPGA components. They are in fact not so different from traditional MPSoCs: from the modeling perspective, only the properties (e.g., performance latencies) of the model components may need to be changed. However, *dynamically* reconfigurable systems have the additional property that functional units may be configured and unconfigured at runtime.

The architectural simulation language in Sesame (Chapter 3) has been developed to model traditional MPSoC systems, where architectural resources are persistent. This translates to a practical limitation in the Sesame's Pearl language: model components ("mod-

ules") can only be instantiated once, at the initialization time before the actual start of the simulation. The model structure consisting of various model components is then static throughout the simulation runtime. Dynamically creating model components to model the functional units being configured and retired from the FPGA fabric, is therefore not supported by the simulation language. This limitation is in fact common in many kinds of computer architecture simulation languages (e.g., SystemC). It would be cumbersome to change the Pearl simulation language to support dynamic component creation: not only would the necessary changes pervade through the whole language implementation, but the semantics of the language would be affected as well, since the communication and synchronization primitives between dynamically emerging and disappearing model components is not well defined.

The solution proposed here is base on the fact that, although the set of logical components that is configured on the FPGA at any given time can change dynamically, the set of all possible logic components is finite. The reason for this is that the logic components themselves are defined (and their bitstreams compiled) at design time[1]. So without loss of generality, we can design our model by using statically defined model components for the dynamically reconfigurable logic components. In Figure 6.1 a heterogeneous system model is shown consisting of a General Purpose Processor (GPP) and an FPGA with up to n logic components.

In the work presented here we assume the logic components to be functional units: e.g., soft-core processors (the FPGA-equivalent of programmable processors) or dedicated hardware-IP processing blocks (the FPGA-equivalent of ASICs) that perform processing on certain input data, possibly resulting in output data for other tasks or other side-effects. This assumption corresponds to a majority of research on dynamically reconfigurable systems, where the (re)configurable logic components are typically used to accelerate specific functions or kernels. It is however possible to consider dynamically configurable interconnects or memory components. Although such reconfigurable components are not supported by the model presented here, similar techniques could be applied to extend it towards this goal.

6.2.2 Resource management

On dynamically reconfigurable FPGAs, each logic component will allocate a certain amount of FPGA resources when it is configured. The configured logic can then be used for as long as necessary and may subsequently be removed when the logic component is no longer needed, thus freeing up the used resources. In our model, the FPGA is represented as a number of distinct functional units, so we introduce a dedicated model component to maintain the state of the FPGA: the availability of resources and a list of currently configured functional units. This resource manager (RM in Figure 6.1) is tightly coupled with the functional units as shown in Figure 6.2 where the nodes indicate the possible states of a functional unit and the arrows indicate the interaction with the RM. The interaction works as follows. An

[1] There is a practical limitation to do on-the-fly creation of logic components (bitstream generation): synthesis and place-and-route are computationally intensive operations and therefore infeasible in the context of embedded systems.

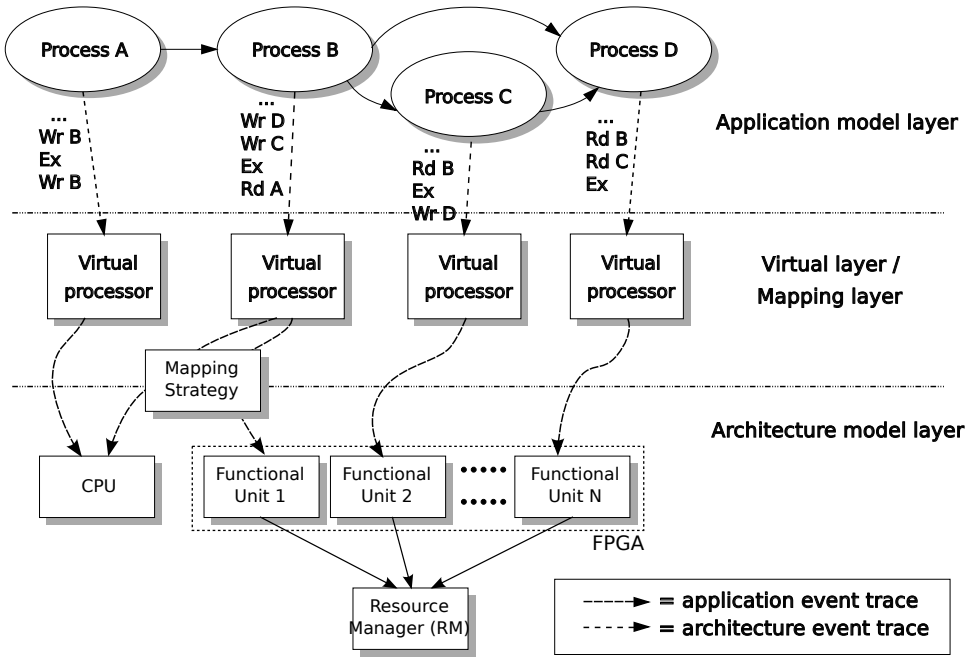


Figure 6.1: Overview of a generic Sesame-framework model for dynamically reconfigurable systems

idle functional unit that wants to execute (because it receives events from the application model), first needs to be configured. The RM may deny the request (e.g., when no resources are available), or it can be accepted. Configuring the functional unit (loading the bitstream on the FPGA) may take some time, after which its state is configured, but still idle. The functional unit enters the *busy* state when it starts processing event traces. If the functional unit has no more events to process, it simply reverts to the idle-and-configured state (*task finished 1*). At certain times, the functional unit may be requested to yield its resources to other functional units, at which point its state changes back to idle/unconfigured (*yield resources*). The decision when to unconfigure a functional unit may be performed by the RM, or by an advanced scheduling mechanism as will be explained in the next section.

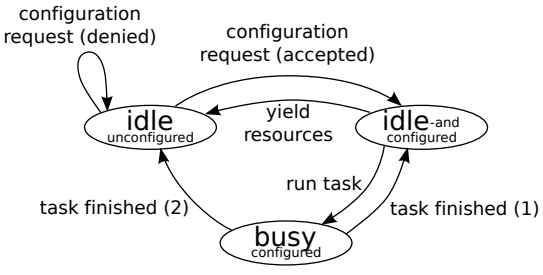


Figure 6.2: State diagram for a dynamically configurable functional unit

There are many different ways to count the resources that are available in FPGAs: FPGA slices, gates and complex logic blocks (CLBs) are commonly used units of measurement. However, they are device and manufacturer dependent and moreover, they are not always homogeneous: e.g., Xilinx FPGAs often consist of both general purpose slices as well as 'DSP'-slices that contain for example pre-defined multiply-accumulate components for efficient implementation of DSP-style systems. As we aim to make our high-level model as generic as possible, we simply denote the resource usage of a functional unit as a percentage of the total available FPGA resources. If this simple abstraction is unsuitable for certain systems, then the model can easily be extended to include more refined resource accounting. This is also true for modeling other resource-related issues such as resource fragmentation where the physical allocation of resources is taken into account. Synthesis tools typically allocate resources for a logical component in close physical proximity to optimize signal propagation and clock frequency. Frequent dynamic reconfiguration may separate the sum of unused resources in small, unusable chunks in disconnected physical locations of the FPGA. Again, this is not within the scope of the model presented here, but extensions and refinements are possible to model these effects.

6.2.3 Mapping strategy

As we described before, one of the design issues with dynamically reconfigurable systems is to find the most efficient use of the FPGA resources. There are a number of issues to take into consideration. Firstly, not all processes benefit equally from execution on the FPGA: this depends both on the type of code as well as the quality of the hardware IP implementation. Secondly, depending on the application and input-data, some processes may contribute more to the total application load or prove to be more of an application bottle-neck than other processes. A general rule of thumb is to execute processes on the FPGA that 1) get the most benefit from execution on the FPGA (highest ratio between FPGA and GPP execution times) *and* 2) represent a large fraction of the application load (e.g., computational kernels and frequently called functions). However, there may be more such tasks available at any given time than the FPGA can accommodate for. Therefore it is important to find efficient strategies to swap out functional units and use them in the most effective way.

A major consideration is the fixed cost associated with every reconfiguration: the time spent to load a bitstream of a certain functional unit. In some cases this reconfiguration penalty means that it can be beneficial to keep a functional unit on the FPGA even though 1) it is currently unused and 2) it occupies resources that could be used towards other functional units. This trade-off becomes particularly interesting when an application task can be executed both by some functional unit on the FPGA *and* on a programmable processor. In this case, the system should contain at least one programmable core and one FPGA and for a certain task X both a software code and a bitstream should be available. Then the workload generated by task X may execute on either resource, depending on what is more efficient given the state of the system at that time. Of course, we are also assuming here that the runtime of the system (be it in hardware, operating system or user-level) supports the functionality capable of dynamic mapping and decision making. Note that in this work we

are not promoting one runtime system or mapping strategy over another, but rather we focus on providing the designer with the necessary tools to support such design decisions. In the experimental section some optimization strategies for dynamic mapping will be discussed as an example.

To illustrate the potential effects of different mapping strategies, we present a small example in Figure 6.3. Each column represents a time-diagram (time increments in the downward direction), where two non-parallel tasks A and B are executing on a processor/co-processor architecture. Task B is dependent on input from task A and we consider three iterations of A-B executions. Task A runs on the reconfigurable co-processor, while task B can run on either the co-processor *or* the CPU. In column 1, we assume that the co-processor has sufficient resources to contain both A and B. Loading the bitstream for A and B only needs to be done once (the shaded area indicates this *configuration time*). In column 2, the FPGA does not have sufficient resources, so that A and B run alternating (each time reconfiguring the FPGA). The third column shows the same mapping, but with a more efficient task scheduling. The last column has task B running on the CPU, while A runs on the FPGA. This figure shows the different trade-offs that may need to be considered by the simulation model depending on the implementation details of the target architecture. Column 1 has the shortest execution time, but requires more reconfigurable resources. Columns 2-4 require the same amount of FPGA area, but 2 performs a lot of reconfigurations, making it slower. Column 3 reuses the configuration of the functional units, which possibly requires some storage to store intermediate results of A before being processed by B. Column 4 executes task B as a software task on the CPU, which may result in extra power consumption if the CPU does not have fine-grained switching control. Of course an ideal mapping strategy will consider each of these trade-offs and switch efficiently between strategies depending on the (design time) system requirements and various run-time conditions of the system. In Section 6.5 we show the results of modeling different heuristic mapping strategies.

In Sesame, the application workload is represented by the events that are passed down from the application, through the virtual layer, to the architecture model. The virtual processors in the virtual layer have been linked to architectural components (typically a processor component) according to the statically defined application mapping. For tasks that have both a software and a hardware implementation, we extend the virtual processor such that it is now linked to both the GPP and the functional unit on the FPGA as shown for process B in Figure 6.1. In Sesame this can be easily accomplished: using the application mapping we map this kind of task to its corresponding functional unit on FPGA, but in the virtual processor template we specify an additional link to the general purpose processor (see Section 3.3.4). When the virtual processor is instantiated by the automatic virtual layer generator, process B will now have two links (to the FPGA functional unit *and* to the GPP), so that it can decide at runtime to which of the resources the events will be forwarded. The final mapping decision may depend on many system factors, including resource availability and the current configured/running state (Fig. 6.2) of other tasks. Therefore it is not convenient that each virtual processor makes the mapping decision locally, but instead a global mapping decision entity is required. This issue will be discussed in detail in Section 6.4.2.

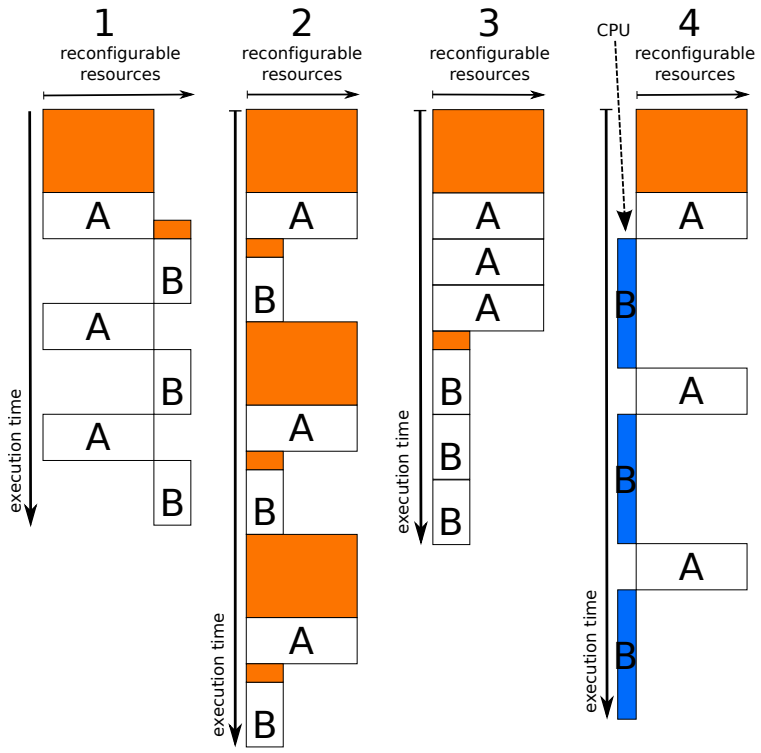


Figure 6.3: Effects of different mapping strategies

6.2.4 Event trace clustering

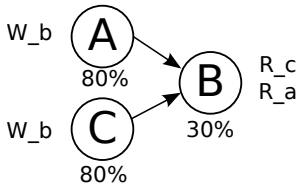


Figure 6.4: Resource based deadlock

In the basic Sesame model (as explained in Chapter 3), R, W and X events that arrive in the processor components of the architecture model, may or may not have been explicitly scheduled. If there is no scheduler component in the model, then the default behavior for events from (possibly different) processes will be interleaved, thus implicitly modeling a pre-emptive process runtime system on the CPU. We call this default behavior of the (virtual) processors *self-schedulable*, since it works with all possible application to architecture mappings and without using global scheduling information. Self-schedulability is a desirable property for initial models, which can be easily extended when specific behavior is required. For dynamically reconfigurable systems we require the self-schedulability condition to hold

even when the mapping of tasks is to be determined at runtime. In this case, we have to consider that R,W,X events are no longer modeled by just the simulated time they consume on the processor, but also by the reconfigurable area that is consumed on the FPGA. Simply adding one or more reconfigurable units to the basic Sesame model can therefore easily introduce deadlocks and break the self-schedulability property. For example, assume that a functional unit for task B is occupying reconfigurable fabric while waiting for data from task A. However, the functional unit for task A may require the same reconfigurable resources to produce this data item. Such a situation is shown in Figure 6.4 for a very simple application model of three processes with the reconfigurable area requirement as specified. If we assume process C starts before A, then its W_b event prompts B to configure (after C has unconfigured). B is then waiting for a write event from A, but since A needs more than the remaining 70% of area, the R_a event of C will never complete and a deadlock has occurred. To avoid many such cases of deadlock, while maintaining self-schedulability, we introduce a mechanism that forces a process to configure its functional unit as late as possible so that any processes on which it depends get a chance to execute first.

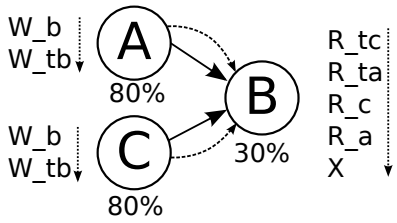


Figure 6.5: Token channels resolving deadlock

The mechanism that we propose here works by making sure that functional units are configured only when they are ready to perform some computation (typically after reading some input data), so that it does not block reconfigurable resources unnecessarily. A token notification mechanism is introduced that lets a process know when the data is available on its input channels. These tokens are sent using ordinary Kahn channels which have been added to the application model; see the dashed arrows in Figure 6.5 for an example based on the KPN in Figure 6.4. Processes A and B write to the token channel after having written the actual data for process B. Process B first attempts to read both tokens before it performs the reads for the actual data. The virtual layer components do not forward the token read and token write events to the architecture layer. Effectively, token-events only synchronize the availability of data, but do not initiate configuration of a functional unit as described in Section 6.2.2. The data is guaranteed to be available after process C (Figure 6.5) has completed the two token read events. Subsequent read event (R_c) is therefore safe to proceed and configure the functional unit for process B (if necessary).

Extending the application model with token channels is easy for KPNs with a regular, data-independent communication pattern. For example, the KPNs used in Daedalus (as produced by the PNGen tool) typically consist of a loop that first reads some data, then performs execution and finally writes output data. The PNGen tool could easily be extended

to output the KPN with the token channels in place. However, KPNs may also have highly irregular communication patterns as even data-dependent communications are permitted (in contrast to other types of dataflow models such as SDF). The token channel-mechanism can be applied to all KPNs, but KPNs with cyclic communication patterns have to be considered specially. Because of the cyclic communication pattern of those KPNs it is not possible to guarantee data availability before reads. In these cases deadlock can not be prevented unless sufficient resources are available to accommodate simultaneously all functional units that are part of the cycle.

Finally we note that the token channels that we have added to the application model will be represented by suitable channels in the virtual layer (this is done automatically by the virtual layer generator 3.3.4). The token channels in the application can be added manually, or could be generated automatically for KPNs derived by PNGen for most application models. It is also possible (although we did not take this approach here) for the token channels to exist only in the virtual layer, thus keeping the original application model clean.

6.2.5 Reconfiguration points

Next we address the issue of how and when reconfiguration should take place. Consider that the functional units in the FPGA are receiving a stream of R,W and EX events from the virtual layer. What happens if a functional unit is swapped out of the FPGA after it has just performed some function (EX event), but it has not yet written that data out. This actually is very much dependent on the actual system that is being modeled: some systems may save the state of the functional unit (including output data), while others may just re-run the events that are lost. In our basic model we offer a solution that is based on the simplification that no functional unit can be swapped out (*un*configured) while it has state. In this case it means that a functional unit that has read data can not be unconfigured until it has performed its operation on the data and written the result back into the system. Note that we use this approach for the remainder of this chapter, but that one of the other behaviors could be modeled instead.

Mapping decisions will be taken at the level of the virtual layer, however, virtual processors can not know when (in the stream of application events), a process is state-less. Therefore we annotate an application process with a so-called *pragma* event (named after a reconfiguration instruction in the Molen ISA). The pragma event is an EX-event that serves only the purpose to communicate to the virtual and architecture layer when the instruction-stream from a task reaches a state-less point. Such pragma events therefore do not model any latency on processor components in the architecture model. Instead, these *reconfiguration points* will be utilized by mapping strategy components to determine safe points to change the mapping or to unconfigure a functional unit.

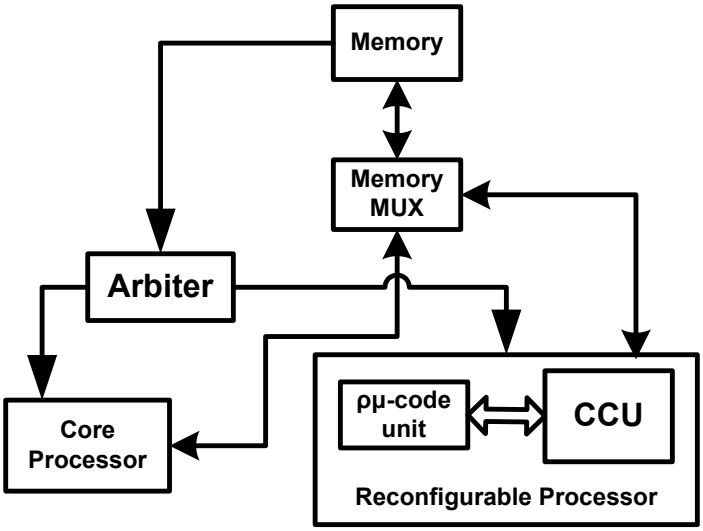


Figure 6.6: The Molen Architecture

6.3 The molen reconfigurable platform

In this section some background will be provided for the Molen processor architecture. In the next section we will use the previously discussed reconfigurable Sesame extensions to create a model for an existing reconfigurable architecture. The Molen polymorphic processor ([115][114]) is established on the basis of the tightly coupled co-processor architectural paradigm. As is typical in these kinds of architectures, there are two main components: the *Core Processor* and a *Reconfigurable Processor* (RP) such as an FPGA. The Core Processor is a General Purpose Processor (GPP) which provides the flexibility to the system to run any kind of compiled code. The Reconfigurable Processor (RP) is used to process computationally intensive kernels and functions (if hardware implementations have been made available), in such a way that the FPGA acts as an accelerator for those functions and kernels. The final performance of such heterogeneous architectures may depend on many factors: the application workload, the quality of the hardware IP implementations as well as the details of the reconfigurable platform itself, including memory organization, methods to hide the reconfiguration overhead and scheduling and prediction methods. Many of these issues are still relevant for different applications of the Molen platform as well as for future extensions to the platform. In the remainder of this section we discuss the basic Molen platform and some design extensions that are currently being developed, or that are under consideration by the Molen developers. The Molen platform will serve as an example to show how Sesame can be applied to support design decisions in the field of reconfigurable architectures.

Figure 6.6 provides a high-level overview of the basic platform of the Molen reconfigurable architecture. The two main processing components (GPP and RP) communicate through a shared memory. The RP is subdivided into the $\rho\mu$-code unit and one or more *Cus-*

tom Configured Units (CCU) (see Figure 6.6). The GPP and RP are connected to an arbiter [115][114] which controls the co-ordination of the GPP and the RP by directing program instructions to either of these processors. The code to be mapped onto the RP is annotated with special *pragma* directives. When the arbiter receives the *pragma* annotated instructions, it initiates the CCU unit, gives data memory control to the RP and drives the GPP into a wait state. An operation executed by the RP is divided into two distinct phases: *set* and *execute*. In the *set* phase, the CCU is configured (i.e. a configuration bitstream is loaded onto the FPGA if it was not still present) and in the *execute* phase the CCU(s) starts its execution. When the RP finishes its execution, it acknowledges the arbiter, which releases the data memory control back to the GPP so that it can resume its normal program execution.

This Molen platform is basically performing sequential execution of a sequential code: the GPP is halted while code annotated by pragmas is executing on one of the CCUs. There is one source of parallelism on the RP, which is provided by a *prefetch* operation which allows prefetching the bitstream and configuring a CCU on the RP while another CCU is in the execute phase. This way, the considerable latency of loading bitstreams on the RP can be hidden by the execution phase of another CCU (provided there are sufficient FPGA resources to contain both CCUs at the same time). Support for multi-threading on the Molen and other reconfigurable platforms has been discussed in [119], which mentions that Molen allows the use of multiple threads on the GPP and a single additional thread on a single CCU on the RP. A logical extension to this would be to completely unlock the parallel potential of the RP by enabling multiple CCUs to run different tasks in parallel. In the next section we consider this feature as a future extension to the current Molen design and build our Sesame model accordingly. The experimental section (6.5) will use this model to evaluate how the additional parallelism can be used efficiently by optimizing the allocation and scheduling of CCUs on the RP.

6.4 Sesame molen model

In this section a Sesame model will be presented for an extended Molen platform. The extended platform supports the use of multiple, parallel co-processors on the reconfigurable processor. Towards this end we use the Sesame reconfigurable model extensions as presented in Section 6.3, but some additional Molen-specific changes are needed to complete the model. An overview of the Sesame Molen model is given in Figure 6.6, which contains three additional components (indicated by the labels 1, 2 and 3) as compared to the general model in Figure 6.1. The first additional component is an arbiter component (label 1) that determines whether the GPP and CCU can be used in parallel. The second is the Runtime Mapping Manager (RMM, label 2), a centralized system component that attempts to optimize the mapping of tasks that have both a GPP and an RP implementation. And finally a token channel (label 3) has been added that optionally removes software pipelined parallelism from the application. In the remainder of this section these additions to the basic reconfigurable model will discussed in detail.

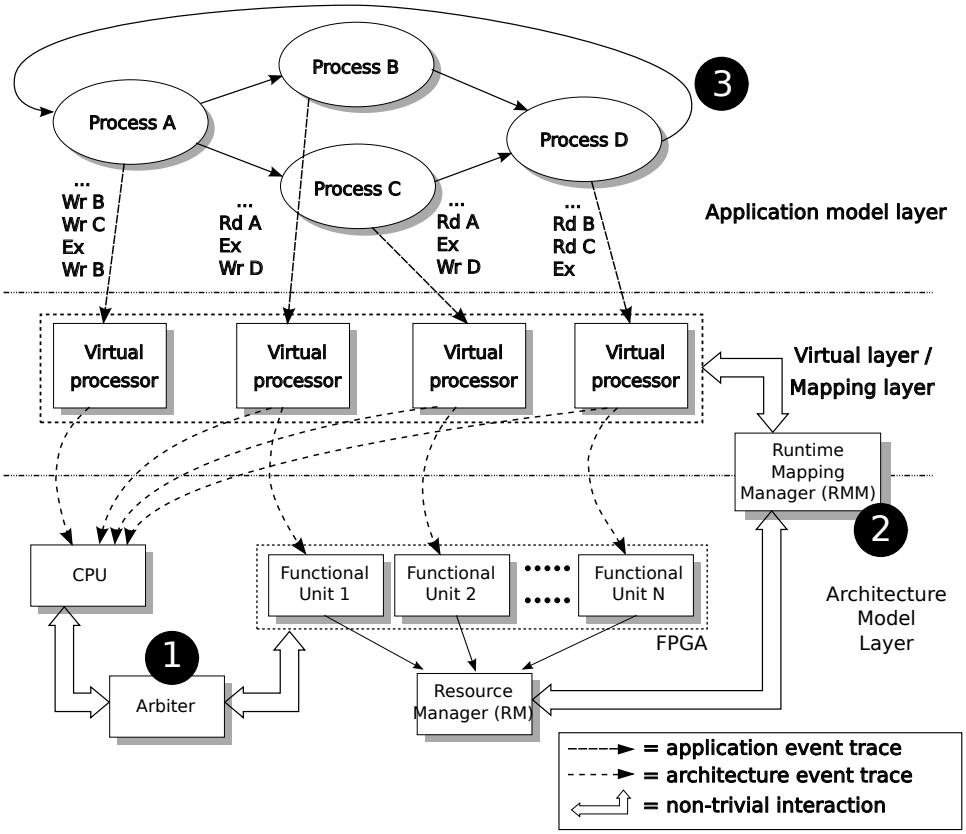


Figure 6.7: Sesame model for the Molen platform

6.4.1 Mutual exclusion of GPP and RP

Sesame was traditionally developed to model MPSoC systems that exploit all available parallelism: e.g., functional and data parallelism as exposed by the KPN application model, as well as software pipelining (even lower-level parallelism with refined models, see Section 3.2.3). Modeling concurrent behavior is therefore a default assumption in Sesame models: processor components are assumed to be able to execute in parallel (provided that dependencies and resource requirements are resolved as explained in Chapter 3). However, platforms that do not support parallel execution on some of their functional units can still be modeled by explicitly removing parallelism between model components. This is indeed the case for the particular instance of the Molen platform that we choose to model here, which uses a traditional processor-coprocessor paradigm between the GPP and the RP and only exploits parallelism on the various CCUs of the RP. This also serves as an illustration that the self-scheduling ability (Section 6.2.4) keeps the model error-free even for mutual exclusive model components (which would otherwise be a common source of deadlock in the model). In order to model a Molen platform that *does* allow parallel execution on both GPP and CCU, it would be sufficient to simply remove the arbiter from the model, or to modify it

such that it indiscriminately grants locks to everyone.

The arbiter is shown in Figure 6.6 as a component between the GPP and the RP (label 1) with block-arrows indicating a non-trivial interaction with either component. In order to implement mutual exclusive execution behavior between the GPP and the RP, the arbiter has been implemented as a locking mechanism with a semaphore. Before a either one of the main functional units (GPP or RP) can execute a R, W or X event, it first has to request the lock from the Arbiter. If a component already has a lock, then a subsequent request will simply grant the lock again. If the lock has been taken by the other component, then the request will block until the other component releases it. All CCUs on the RP ask for the lock individually, but a special condition applies in that case: when one CCU has the lock, then a request from another CCU will also be granted. CCUs also need to release the lock individually, but the GPP will not be able to obtain the lock until all CCUs have released the lock. This locking mechanism implements non pre-emptive mutual exclusion of the GPP and the RP while maintaining concurrency between different CCUs in the RP.

The processor model components (both for GPP and CCU) have been extended so that they call a function `lockAndConfigure()` on the arrival of a R, W, or X event. This function first attempts to acquire the lock and subsequently models the configuration of the CCU on the RP (only for events sent to the RP and only if the CCU is not configured already). This function blocks (possibly increasing simulation time) until both locking and configuring are successful, after which modeling of the event continues as normal. If the locking and configuring have been done previously, then this function does not model any additional delay.

6.4.2 Mapping strategies

As mentioned before in Section 6.2.3, the Virtual Processors can forward events to either GPP or CCU, if a task has both a hardware and a software implementation. Finding the optimal mapping and scheduling strategy for reconfigurable systems is a known NP-hard problem ([12]). Successful approximation algorithms need access to run-time system information such as the state of system components, the available resources, the state of the other CCUs, etc. It is possible to let each Virtual Processor make scheduling and mapping decisions in isolation by distributing the global decision-making strategy. However, this means that not only the state of the system needs to be distributed to all Virtual Processors, but also synchronization has to be performed for VPs that are making decisions simultaneously. Therefore, from the modeling perspective, it makes sense to use a centralized model component taking care of the mapping decisions for all tasks with multiple mapping options (GPP or RP). We call this component the Runtime Mapping Manager (RMM), shown with label 2 in Figure 6.6. An additional benefit of a global implementation is modularity: different RMMs, implementing different mapping strategies can be exchanged and evaluated more easily.

The large block-arrows in Figure 6.6 indicate that the RMM communicates with both the Virtual Processors and the Resource Manager (RM). The RMM interacts with the those model components in the following way. Before Virtual Processors forward events to a

functional unit (but after modeling synchronization latencies, see Section 3.2.3), they query the RMM to ask what is the new mapping target. The RMM returns a target identifier, which can be either the GPP or a CCU belonging to that process. The Virtual Processor, which has connections to both targets simply forwards to the appropriate one. Note that configuration of a CCU is separated from the mapping procedure and will be performed by the CCU itself (if necessary) after receiving the first event.

In order to perform a mapping decision, the RMM will sometimes need to query the current status of the RP, which is maintained by the RM. The RM provides an interface for the RMM to obtain information on the current status of the RP: the amount of free resources, the state of various CCUs (Figure 6.2, etc. Information about individual CCUs, such as resource requirements and execution latency tables (see Section 3.2.2), do not need to be obtained from the RM, since they are constant throughout the simulation and will be communicated to the RMM at initialization time. For a discussion of different mapping strategies we refer to Section 6.5.

In Figure 6.6 the RMM is shown as a component in between the mapping and the virtual layer. This is to emphasize that in a real system implementation the scheduling and mapping functionality of the RMM may be provided by hardware, some kind of middleware or even the operating system or in user space: for our model this distinction is not important.

Lastly we note that ,in our model, the RMM is a non-blocking component: queries to the RMM and the decision process by the RMM do not consume simulation time. However, such latencies can be added if needed to model specific systems. The RMM needs to available (an can not be blocked) at any time in order to be able to accept requests from VPs. Therefore, such latencies can be computed by the RM, but would have to be modeled by another (possibly new) component in the model.

6.4.3 Application pipelining

In Figure 6.7, an additional token channel has been added to the application (label 3). It goes from the application's sink node (Process D) to the source node (Process A). It guarantees that Process A can not restart immediately after it has completed one iteration (and written data to its output channels) of its internal data-streaming loop. Instead, it has to wait until Process D has finished one iteration and has written to the token channel. In this way the token channel effectively removes pipeline parallelism from the application. The result is that the Kahn process networks as used by Sesame (and which can be generated with the PNGen tool from Daedalus (Chapter 2), fit better with the processor/co-processor model of Molen. Note that this is simply a design choice, we could remove the channel and the model still works, although the RMM should now also have to consider starvation, e.g., it has to prevent that only source process A gets to run. We note that the only remaining source of parallelism in the application is task-parallelism, which can be exploited by parallel execution on different CCUs (e.g., Process B and C in Figure 6.7).

6.4.4 Component interaction

Finally we demonstrate the interaction between the various Sesame model components for the Molen model using the time-sequence diagram given in Figure 6.8. We follow the sequence of interactions starting from when an application event is forwarded from the virtual processor to the architecture model layer (note that for R and W events communication synchronization latency has already been modeled, see Section 3.2.3). The virtual processor always first asks the RMM where to map the event (to GPP or to RU). If the RMM is considering a mapping decision (typically only after a reconfiguration point, see Section 6.2.5), then it may request information from the RM (e.g. the amount of free resources). Then the mapping decision is forwarded back to the VP, which then forwards events accordingly. Note that for both GPP or CCU execution, the lock has to obtained from the Arbiter (not shown in the figure) using the `lockAndConfigure()` function described in Section 6.4.1. Then finally, the event is forwarded to the GPP or CCU processor component where further architectural latencies will be modeled.

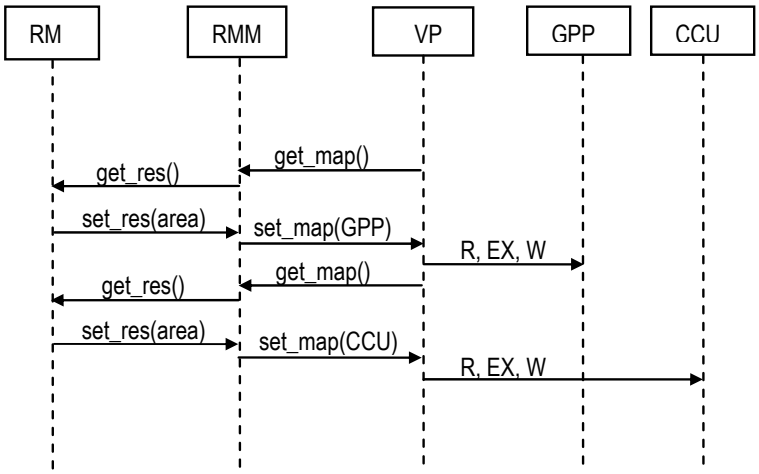


Figure 6.8: Interaction between different model components

6.5 Experimental results

The model that was presented in this chapter has been used in various case studies related to the Molen reconfigurable platform. In the works of [90, 89, 86, 88, 87] different versions of the model have been demonstrated showing increasingly sophisticated models and modeling aspects of the Molen system. In [90] the initial Molen model was introduced (a simple experiment based on this model will be shown below as **Selected experiment 1**).

In [89] the model is extended and applied to study runtime mapping exploration, where the Runtime Mapping Manager (RMM) optimizes the mapping at runtime to accommodate

an application with a dynamically varying workload. In this case we assume that the dynamic variation is caused by two or more applications sharing the system, where the main application is sporadically interrupted by tasks from the auxiliary applications. A two-staged exploration is proposed where the first stage evaluates each application in isolation and identifies three task types: hardware, software and *pageable* tasks (the latter can be mapped both on the GPP or the RU). The second stage then models the combined application model and the RMM makes mapping decisions at runtime for all pageable tasks from each of the applications. The first stage works as a pruning step by assigning some tasks to either HW or SW, reducing the complexity of the RMM in the second stage. The second stage is iterative: analysis of simulation results may prompt a re-assigning of task types (HW, SW or pageable), after which the second stage is repeated.

The generic Sesame components that have been developed in order to allow system-level modeling in Sesame as well as a complete Molen model have been collected in a framework under the name rSesame. This framework was introduced in [86] and it is shown there that rSesame fills a gap in a classification of existing frameworks and approaches based on modeling capabilities and evaluation method (abstract system-level simulation). Furthermore, the rSesame framework is evaluated from a designer's perspective, facilitating modularity, component-reuse, modeling flexibility, a high level of abstraction and general ease-of-use.

In [88], the runtime mapping manager is improved by using a range of heuristic strategies in order to improve the spatial and temporal behavior of pageable tasks. The former determines *where* to map a task (on GPP or on a CCU), whereas the latter determines *when* to configure a CCU, when to run the task on a CCU and when to retire the CCU (as shown in Figure 6.2). The result of the improved RMM strategies are analyzed by showing in detail the dynamically changing mapping targets of tasks and a quantitative evaluation of the different strategies is performed. This work is continued in [87] where a new mapping heuristic is proposed and compared against the other heuristics. We shortly introduce the various heuristics and show a sample of the obtained design space exploration results in **Selected experiment 2**.

Below we will show here a small selection of results as an example the experiments supported by rSesame and to provide insight in the kind of modeling results that can be obtained.

Selected experiment 1

We first show some results that were obtained using the first incarnation of the Molen model. The application is a MJPEG encoder that has been transformed into a KPN by the PNGen tool from Daedalus. In this case PNGen has been configured to output a data-parallel version of MJPEG so that the DCT and Quantizer tasks are divided over four parallel streams. The architecture model consists of a GPP and up to 8 CCUs and the RMM uses a static mapping policy (SW or HW mapping is determined at design time).

In the model we use estimated latencies for tasks executing on the GPP and in FPGA hardware. The latencies are based on observations in the prototype platforms generated by Daedalus, but we emphasize that they have not been validated and are used for the purpose of

illustration only. Execution of tasks on the reconfigurable hardware is assumed to be an order of magnitude faster than GPP execution, but CCUs are modeled to suffer a configuration overhead which is proportional to the size (FPGA area) of the CCU configuration bitstream. In Figure 6.9 the simulation results are shown for an experiment where the CCU area is estimated as 100% of the FPGA area, effectively forcing sequential execution of tasks on the FPGA. The left table in Figure 6.10 lists the simulation results (one simulation experiment per row) where in each simulation run, an additional task is marked for HW execution. In the 1st row all tasks are mapped on GPP; each subsequent row maps the previous HW set plus the indicated additional task to HW. Simulated runtime is expressed as the number of simulation cycles for the MJPEG to finish processing the sample input. We see that significant speedup can be obtained by off-loading DCT processes to the FPGA, but that the addition of the Quantizer tasks do not lead to significant speedup. The reason is that more tasks on the FPGA means more reconfiguration overhead, effectively making the FPGA the new bottleneck in the system.

Therefore, we experiment with smaller CCU sizes to see how much more speedup can be obtained. The right table in 6.10 shows the results of reducing CCU area and their proportionally reduced reconfiguration delay; the third column shows the number of required reconfigurations and speedup is compared to the last row in the left table (all DCT and Q tasks in HW, the remaining tasks in SW). The small speedup gain by reducing the CCU area from 95 to 75% is solely caused by the reduction of the reconfiguration delay (the number of reconfigurations is the same). When the area is 50%, two CCUs can execute in parallel, showing a larger performance gain, which turns out to be the maximum possible speedup. Reducing the area to 30% (up to 3 parallel CCUs), does not improve speedup, since system performance is now limited by the application tasks executing on the GPP.

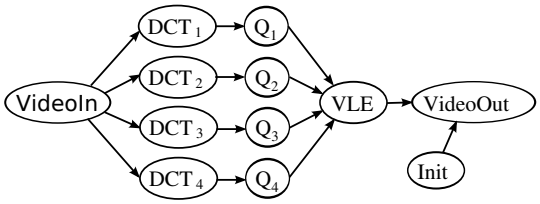


Figure 6.9: Task-parallel MJPEG KPN

Finally we note that modeling results will be different depending on the application, any available parallelism in the application as well as architectural parameters: the number of CCUs, quality of hardware implementations, reconfiguration technology (and hence reconfiguration delays), etc. Since these parameters can be captured by this model, we can see how the model helps exploration of different design options.

Selected experiment 2

As we discussed in Section 6.1, the real benefit of dynamic partial reconfiguration lies in the possibility to optimize at runtime the functional units that are on the FPGA. In the case of the Molen platform model, the runtime mapping manager RMM decides whether a task is mapped onto the GPP or a CCU: the mapping decision can change any time a pragma event (Section 6.2.5) is sent to the architecture model. Here we will show the results of an

Exp#	HW tasks	Cycle Time ($\times 10^6$)	Speedup
1	-	371	1.00
2	prev+DCT1	331	1.12
3	prev+DCT2	292	1.27
4	prev+DCT3	253	1.46
5	prev+DCT4	217	1.70
6	prev+Q1	199	1.86
7	prev+Q2	200	1.85
8	prev+Q3	194	1.90
9	prev+Q4	188	1.96

Area (%)	Reconf. Delay	Slow Reconf	Cycle Time ($\times 10^6$)	Speedup
95	25000	1792	189	1.97
75	18750	1792	176	2.10
50	12500	1536	138	2.70
30	7500	1280	140	2.64
10	2500	1280	138	2.69

Figure 6.10: Results for **Selected experiment 1**

experiment where a newly proposed strategy (RBH) is compared against several existing mapping strategies (AMAP and CBH). We shortly describe each of these strategies, but for the details we refer to [88, 87].

The AMAP ("As Much As Possible") strategy configures a CCU for any pageable task, but if the FPGA resources are exhausted, then the task will execute on the GPP. CCUs will be unconfigured (and FPGA resource released) when a tasks finishes, so hardware configuration reuse is dependent on the availability of resources. The Cumulative Benefit Heuristic (CBH) maintains (for each task) a benefit value that represents how much time would be saved if the task had always been executed on the RP (assuming execution on the RP is faster). When a mapping decision is made for a task, its benefit value is compared to the benefit of the tasks already on the RP. If it has a higher benefit value, then a minimal set of the lowest-benefit tasks is unconfigured from the RP to accommodate the CCU for the new task. Finally, the newly introduced Resource Based Heuristic (RBH [87]) focuses on efficient re-use of configured CCUs by avoiding unnecessary configurations. RBH makes mapping decisions (and selects tasks to unconfigure) based on a dynamically updated list of global and per-task statistics, the current state of a task, estimated speedup on CCU and its frequency of occurrence. Each of these heuristics has been implemented as plug-in policies to the RMM.

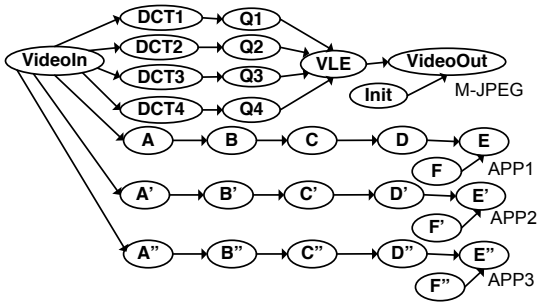


Figure 6.11: Multi-application model: MJPEG with auxiliary applications

A modified application model is used as shown in Figure 6.11 where the previously shown data-parallel MJPEG is combined with applications APP1, APP2 and APP3. The additional applications become active based on triggers in the VideoIn process (using the

Hardware	Area (Slices)
XC4VFX20	8544
XC4VFX40	18624
XC4VFX60	25280
XC4VFX100	42176
XC4VFX140	63168

Table 6.1: Indication of resources in the Xilinx Virtex4 FX family [4]

dynamic intra-application behavior scheme discussed in 5.4.2) and so represent a (synthetic) pseudo-dynamic application. In a real application, APPx could represent for example the extra workload from an intermittent background task or picture-in-picture functionality. The architecture supports up to 30 CCUs and each task is considered pageable; execution latencies for tasks on the CCUs are estimated by expert knowledge based on the properties of a Xilinx Virtex 4 FX family of FPGAs. We now consider the performance of each heuristic for different resource conditions for different Virtex4 FX implementations (see Table 6.1). The simulation results are shown in Figure 6.12 for each of the heuristics: the bars show the simulated execution time (left axis), while the lines show the speedup compared to a static all-to-GPP mapping (right axis).

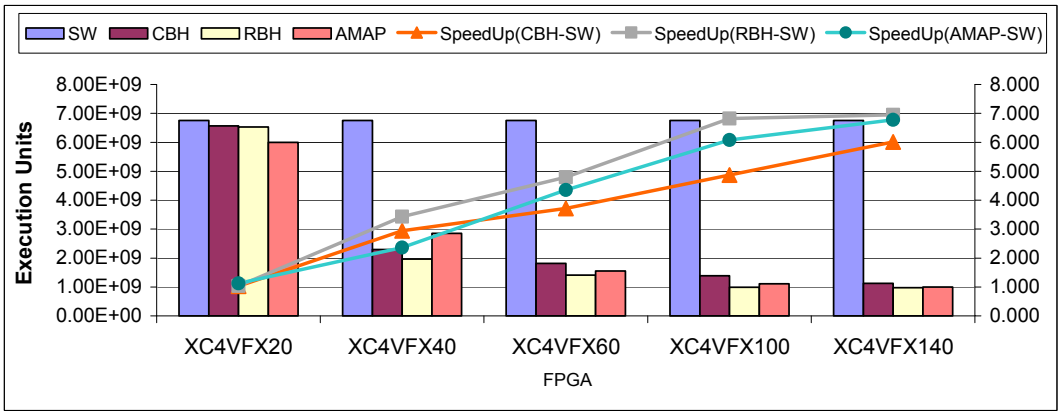


Figure 6.12: Simulation results for different mapping heuristics

For the smallest FPGA (FX20) we see that there is very little improvement compared to the SW mapping since there are simply too few resources available to be used in a significant way. Interestingly, the least sophisticated heuristic (AMAP) performs best under these conditions: apparently AMAP's greedy approach outperforms the dynamic learning capacities of the other heuristics. In the larger FPGAs ($\geq$FX40), we see major performance improvements for all heuristics. However, we can see that speedup does not really improve much for RBH and AMAP after FX100. For example, in the case of RBH, the performance improvement between FX40 and FX20 is 69% with 54% area increase, whereas the improvement between FX140 and FX100 is only 2% with 33% area increase. The reason is

that performance becomes increasingly limited by the inherent parallelism in the application rather than the FPGA resources. For the larger FPGAs, it is hard to distinguish the relative performance of the heuristics in Figure 6.12, this is more clear in Figure 6.13. The graph compares heuristic performance of RBH in contrast to CBH and AMAP. It clearly shows that RBH is better than the other heuristics except in the case of FX20 as noted earlier. For a more detailed evaluation of the results, including metrics to determine the efficiency of configuration reuse, we refer to [87].

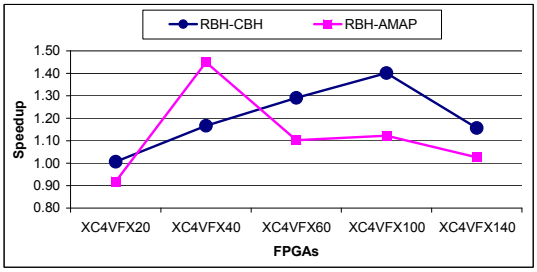


Figure 6.13: Relative comparison of different mapping heuristics

6.6 Related work

The technology that is behind the feature of partial dynamic reconfigurability is driven by advancements in FPGA technology and is supported in one form or another by major vendors such as Xilinx and Altera. These vendors provide extensive toolsets to support their hardware platforms and include hardware specification tools, compilers and synthesis tools. However, there is not much support for high-level system specification or system-level design space exploration, which is the target of the Daedalus design flow. Such support is needed, because the traditional DSE problems of HW/SW partitioning, application-to-architecture mapping, task scheduling and resource allocation are typically even more complex on dynamically reconfigurable systems. This topic has rapidly gained interest in (academic) research activities and has resulted in an extensive body of work, of which we can only reference a small portion here.

First of all we notice the wide variety of optimization algorithms and techniques that have been applied to the problem of design space exploration for dynamically reconfigurable systems, e.g.: dynamic programming, branch and bound, integer linear programming, graph partitioning, simulated annealing, genetic algorithm, ant colony optimization and tabu search ([84], [18], [19], [63], [49]), etc. However, most of these works take a design time approach where design candidates are evaluated for fixed system constraints and conditions. As such, dynamic behavior in the application, architecture or environment is not being considered at all, or the proposed design candidate is not optimized to deal with such dynamic behavior. In general it is hard to perform accurate design space exploration at design time for systems that are capable to dynamically adapt to changing conditions. For dynamically reconfig-

urable systems that contain both general purpose processing cores as well as (dedicated) hardware IP blocks, HW/SW partitioning decisions are typically considered together with task-mapping (the task is either mapped to a GPP or to hardware). Therefore, task mapping is one of the most important design parameters and one that *has* to dynamically take into account system conditions at run-time. Such efforts have been reported in [96, 58, 71, 45, 30].

One possible disadvantage that the previously mentioned works have in common is that they have been developed for a particular context and solve specific DSE problems. In other words, they have not been developed for use in a generic modeling and simulation framework for targeting a wide range of design problems. This typically makes it harder to reuse implementations of each of the proposed methods and greatly complicates comparisons between approaches. For example, the CBH heuristic was based on [45] and we previously used the Interval Based Heuristic ([88]) based on [30]). In each case we had to re-implement and modify these heuristics. Once such methods (together with many others) have been made available in a generic framework, then such efforts can be minimized and recurring comparison experiments can be done routinely for each new contribution (e.g. a competing method or a different DSE problem). Contributions that were performed inside a framework can be found in [75, 44, 85] and [83]. However, these works again are mostly using the design-time approach. Exceptions are [118] and [76] where a design time exploration is combined with runtime management to make a trade-off between fast exploration (at design time) and accuracy (run-time).

An related work that considers both design-time and run-time optimization of task mapping onto partially dynamic reconfigurable architectures is in [11]. Formal definitions of the mapping problem are given, a run-time mapping optimization based on Boolean satisfiability (which makes use of configuration reuse) is proposed and shown to benefit several media-related applications. Evaluations are performed on a NoC-based platform architecture which has a number of partial reconfiguration slots that is fixed at design time. We note that we also fix the number of CCUs in our simulation model, although we could choose a higher number, without reaching the practical limits that are inherent on the actual hardware (thus enabling speculative modeling).

We conclude that there is still a great need for more research and development into standardized simulation and modeling infrastructures that operate at the abstract system-level and support both design time as well as run-time modeling and optimization. The rSesame framework that was the result of the efforts reported in this chapter is an initial step in that direction.

6.7 Conclusion

In this chapter we introduced Sesame model components and techniques to model a recently emerging type of system that uses partially dynamic reconfiguration (typically implemented using FPGA technology). We have shown that such systems have some different modeling requirements, for which Sesame can be adapted. A model based on the Molen reconfigurable platform has been presented, as this was our initial modeling target. However, we empha-

size that the components and techniques in this chapter can be applied to a wider range of reconfigurable systems. Moreover, now that these techniques have been developed, modeling variants of such systems is relatively easy. The case studies show what kind of results can be obtained using the Sesame model. The first case study simply evaluated a few simple architectural parameters, but the second case study showed that more advanced components can be modeled, which in our case for example, incorporate advanced dynamic mapping heuristics. The feedback provided by such models allows a designer to evaluate different design options in the very early stages of system design. This is also the purpose of the presented Molen model, which is able to model both the current Molen platform architecture, as well as possible future extensions.

We remark that the Molen platform presented here has not yet been validated against the real system; instead we used very rough latency estimates for the purpose of demonstration. A non-validated model can still be useful to a designer, for example to answer speculative "what-if" design questions that require relative comparison of design candidates. For example, a designer may want to know if it is worth to optimize a certain HW implementation of a task, given that implementing a high quality HW-IP typically represent a significant design effort. The model may show no significant benefit of the optimization if other bottlenecks are present in the system. Depending on the result, the designer can decide to forfeit the optimization effort, or to spend it on the newly identified bottleneck. Another such speculative question we saw in the second case study: where the effect of different mapping heuristics was considered for FPGAs with varying resource constraints. In the end, it depends on the system designer's interpretation and requirements whether the measured differences between the various heuristic methods is significant. In order for the Molen model to produce accurate absolute performance numbers, it should first be calibrated (possibly using techniques presented in Chapter 4). Then in a second step it needs to be validated against a real Molen implementation; possibly the calibration step has to be repeated until the required accuracy is reached. Note that such a validation step is likely to improve both absolute performance numbers, as well as increase confidence in the model's ability to compare designs candidates relative to each other. The validated model can then be used to explore future extensions on the Molen platform design.

Chapter 7

Support for automatic DSE

7.1 Introduction

In the previous chapters we have demonstrated various tools and techniques to evaluate the different aspects of system modeling in the context of the Daedalus design flow. In the introduction (Chapter 1) it was stated that efficient design space exploration requires 1) a method to evaluate a single design point and, 2) a method to efficiently navigate a design space. Given that design spaces grow exponentially in the number of parameters, exhaustive search (evaluating every possible design point) is infeasible. The number of design points can easily outgrow any practical limit on execution time, given a minimum evaluation time of a single design point (limited by the evaluation algorithm or simulator). In this chapter we look at methods to navigate the design space using genetic algorithms (GAs).

A genetic algorithm is a well-known metaheuristic optimization algorithm from the evolutionary algorithm family. Genetic algorithms have previously been used successfully towards solving a wide range of combinatorial problems, such as the bin packing problem or the graph coloring problem [117, 107, 27, 109] as well as many combinatorial problems ([39, 6]). Moreover, GAs can be used in their basic (domain-independent) form, or with custom extensions that incorporate domain-dependent knowledge in order to improve search performance even further. For these reasons, GAs are a promising solution for the traversal of the combinatorial exponentially growing design spaces in the field of embedded system design.

Many attempts in this direction have already been made (these will be discussed in more detail in section 7.2). Although the authors of these works generally report positive results, no authoritative literature on this topic exists yet and –as far as we know– adoption of the methods in commercial tools is currently non-existent.[1] There may be several reasons why this may be the case. First of all there may be practical reasons that restrict the use of GAs, for example due to the previously mentioned limit on evaluation time, or because it is difficult to find a suitable "genetic" representation for the design problem. Secondly, it may

[1] Here we refer explicitly to the use of evolutionary algorithms for the purpose of design space exploration. For example, evolutionary algorithms have been used successfully in commercial place-and-route tools.

be the case that the effectiveness of evolutionary heuristic search algorithms simply has not been sufficiently proven (at least not for generic design problems). Unfortunately, the efforts in this area have been rather fragmented and there is no consensus over which methods are best for certain DSE problems, or alternatively, which has the best generic performance for a wide range of different problems. In any case, further research in this area is required to enhance and solidify our knowledge and understanding in this area.

In this chapter we report our efforts to mitigate some of the common problems and find solutions by evaluating both standard, well-known genetic algorithm techniques as well as some newly developed ones. We present the results of applying genetic algorithm techniques to a few typical Sesame case studies. The focus is on evaluating and improving the exploration of the application-to-architecture mapping. We consider this to be one of the most challenging dimensions of the design space, since it is a non-trivial design parameter with no apparent internal structure and its effects on the design objectives are hard to predict using for example analytical models. Furthermore, as we will explain in Section 7.3, the mapping parameter can be viewed as a "meta-parameter" that incorporates a range of other important design parameters such as the number and type of processors in a system. Although the mapping design space exploration parameter is important, it has not received as much attention as other, more straightforward design parameters (such as strictly architectural parameters related to component properties: component types, clock-timings and memory and buffer sizes). Based on some observations about the mapping design (sub-)space, we propose and evaluate an extension to a standard genetic algorithm search method. As an additional contribution, we propose, motivate and apply a simple, yet rigorous graphical method to compare different heuristic DSE search methods.

7.2 Related work

The combined DSE search problem that uses simulation or analytical estimation models (to evaluate individual design points) together with a heuristic search method (to traverse the design space) is a relatively new area of research. The need to address this problem is reinforced by the current state-of-the-art of implementation technologies that makes manual system design infeasible (the aforementioned implementation-gap). Heuristic search methods typically do not guarantee finding global optima in the design space, but they are nevertheless able to *prune* the design space: to reduce the design space to a set of design candidates that meet certain requirements (or are close to the optimum with respect to certain objectives). Evolutionary algorithms are already being used in many different areas in the field of system design, e.g. for place-and-route, netlist generation and for reliability testing and validation purposes. In the following however, we limit ourselves to applications for the purpose of system-level design space exploration. Moreover, we focus on those approaches that 1) make a clear separation between the parameter and objective spaces, 2) use analytical estimation methods or simulators to evaluate design points and 3) traverse the design space by means of an evolutionary algorithm or a related method. Works that fall in this category still have a striking variety of contexts, application domains and types of architectures.

Moreover, there is a large disparity in the types of design problems, the design parameters and objective metrics, design space sizes, evaluation mechanisms and optimization methods. And finally, many different qualitative evaluation methods are used to evaluate and compare proposed search methods. In the following we will not give an exhaustive comparison of related work, but rather emphasize some of the most important differences and similarities.

To our knowledge, the earliest work that meets the previously mentioned criteria is that of [101]. A method is proposed to optimize the system-level design (although referred to as *synthesis*) of a dataflow-based application implemented by an MPSoC architecture. Application and architecture are represented as specification graphs, which are tied together with edges representing expert knowledge indicating the (in)feasibility of mappings. A multi-objective fitness function (performance and cost) is defined that adds extra penalties to steer the evolutionary algorithm away from infeasible population individuals. A case study reports that the proposed method is able to efficiently find pareto-optimal points.

We also refer to earlier work from our research group ([25]), proposing a solution towards solving the design space search problem of mapping Kahn process networks (KPNs) onto heterogeneous MPSoCs. A mathematical model expresses the three objectives (performance, power, cost) of a design point as an analytically solvable non-linear mixed integer programming problem. This is then used as the fitness function for two elitist evolutionary algorithms (SPEA2 [120] and NSGA-II [21]). A case study shows that (according to three newly defined metrics) they perform similarly, but that NSGA-II is preferred due to its less computationally intensive fitness assignment scheme.

Also closely related to the work presented in this chapter is [77], where a standard simulated annealing algorithm is extended with automatic parameter selection. The design problem consists of mapping tasks from a synthetically generated KPN network onto a 2-4 processor architecture; with typical design spaces consisting of 10s of millions of candidates. The proposed method is compared against random search, group migration and standard simulated annealing by looking at the relative quality of results in terms of the objective space (performance). It shows that for a given class of KPNs, the proposed method finds better results for the same number of mapping evaluations (if that number is sufficiently high).

In the work of [37], the design space search problem is described as a Markov Decision Problem (MDP) and design space traversal is defined as a sequence of movement vectors between states. Movement vectors change states in parameter space (number of processors, I/D cache size) and approximate analytically the impact on the objectives (power and performance). A major advantage of this approach is that simulation only needs to be applied when repeated application of movement vectors exceeds a predefined level of estimation error. A case study is presented where MJPEG4 and Ogg-Vorbis applications are mapped onto a 2-8 ARM-processor MPSoC with varying I/D cache sizes. It shows an 80% reduction in evaluation time compared to tabu search or simulated annealing, or conversely, it finds better results given the same evaluation time. Note that exploration of explicit task-to-architecture mappings is not supported, probably because it would be hard to define a sufficiently accurate movement vector for this design space parameter.

In the work of [29] a method based on Ant Colony Optimization (ACO) is proposed to optimize mapping and scheduling of an application onto a 4-processor MPSoC platform. The application is specified as a parallel task graph and mapping concerns both task mapping and communication mapping. The proposed method is compared against other metaheuristic algorithms such as simulated annealing, taboo search and genetic algorithms. Case studies consisting of both synthetic and a JPEG application show that the proposed ACO method finds better average results and reaches optimal solutions faster (though at the cost of a higher execution time) compared to the other methods.

While these previous works typically focused on specific DSE methods that were very much tied to a specific context (simulation tools, architectural platforms or application domain), the work in [48] makes a commendable effort to propose a generic infrastructure for system-level DSE. The NASA framework aims to be modular, extensible, flexible and reusable. For example: the search method(s), feasibility checker, platform generation, evaluation (e.g. a simulator) and fitness functions have been de-coupled as separate parts and use well-defined interfaces so that substitute components can be used in a plug-and-play fashion. Components that conform to the interfaces are much easier to compare and code for setting up and evaluating experiments can be reused. An example of the flexibility and modularity of the system is illustrated by the framework's proposed *dimension-oriented* DSE approach, where different design dimensions are co-explored using the same or (optionally) different search algorithms. A further innovation is that instead of using a fixed architectural topology, NASA constructs the topology from so-called Basic Topology Units (BTUs), which consist of a network container and a number of element containers. According to a custom user-specification, these BTU building blocks will be automatically connected and the containers will be instantiated with components such as buses, bridges and links (network containers) or processors and memories (element containers). This allows for automatic exploration of a much wider range architectural topologies than is commonly the case.

To conclude, we will shortly clarify the position of the work presented in this chapter relative to the mentioned related works. As in the previous chapters, we consider streaming multimedia applications (as in [25, 77, 101, 17]), and more specifically we use Kahn process networks (as do [25, 77]). Since the focus of this chapter is to study DSE search algorithm behavior rather than optimization of a specific application or platform instance, we use synthetically generated KPN workloads (using a modified version of the KPN workload generator of [77]). This is also the reason why (in contrast to most other works [17, 101, 37, 25, 48]), we propose a way to compare and evaluate different algorithms independently of objective space metrics. Design points are evaluated using our Sesame system-level simulator: [17, 48, 77] also use simulation-based evaluation, whereas other approaches use analytical methods ([25, 101]) or a mix of both ([37]). Simulation-based evaluation can typically more accurately determine characteristics of actual design points, but may be slower than analytical approximations. Although we use the Sesame simulator, we aim for our methods to be equally applicable to other contexts (possibly using different simulators) while still being able to provide a useful comparison (e.g. by measuring search method efficiency in the number of required evaluations instead of time).

Our method of choice for searching the design space is a genetic algorithm (like [101, 25, 48]) using a mapping based genetic representation (like [101, 25]). However, we propose methods to optimize the standard GA using newly developed techniques that (as far as we know) have not been used previously in the field of system-level DSE (see Section 7.8). Our methods for optimization are not only relevant since the mapping-based representation is commonly used (e.g. [25, 101, 48]), but also because they are highly compatible with many other optimizations (so that different optimizations can be applied simultaneously). The work in [101] and [17] report that their genetic algorithms manage to find the true optimum in the design space. In both cases, however, the identification of the true optimum seems to be given by means of some (not further specified) analytical method. However, it is not addressed if and how the presence of such an analytical solution influences their respective evolutionary search algorithms. For example, it is possible that a genetic algorithm is able to converge more easily if there are true-optimal solutions which are (apparently) not too hard to derive. We clarify in advance that the search problems from this chapter have been set-up in such a way that there is no known method to derive the true optimum of the design space, other than by exhaustive search (which, as we will see, is often infeasible).

7.3 DSE as a GA search problem

In this section we discuss the representation of a DSE problem as a search problem which is suitable for solving by a GA. GAs operate by searching through the solution space where each possible solution has an encoding as a string-like representation (often referred to as the *chromosome*). A (randomly initialized) population of these chromosomes will be iteratively modified by performing a fixed sequence of actions that are inspired on their counterparts from biology: evaluation and selection, crossover and mutation. This sequence of actions is in fact the only common denominator between GA implementations. There are many different implementation options for each step in the GA and choosing the right implementation is "the art of evolutionary algorithms", since (unfortunately) it is mostly a matter of designer's intuition and trial and error. A fundamental design choice is the genetic representation of the solution space, because each of the selection, crossover and mutation steps depends on it. Furthermore, it is known from literature that for some search problems the representation can influence GA performance. However, from a practical perspective, one typically chooses a representation that naturally fits the problem at hand, as this simplifies the implementation of the crossover and mutation operators.

Design point representation

In our case the problem is finding an optimal design candidate in a large space of possible design candidates that can be evaluated within Sesame. In Chapter 3 we saw that a design space in Sesame often consists of parameters that are related to the application-to-architecture mapping. For example, the number processors of a design can be completely determined by the mapping: processors to which no processes have been mapped can simply

be discarded. Similarly, if there is a choice between N different types of processors and a maximum of M processors, then the meta-platform (as introduced in Section 3.4 simply contains all MxN processors and the mapping determines the final configuration. If a suitable meta-platform can be found for a given DSE experiment, then the mapping specification provides a complete representation of a design point. However, creating a meta-platform can be hard for some architectural parameters, especially if the architectural topology is one of the design parameters. Furthermore, it doesn't work with "active" components (which are not dependent on input) or components that start simulation with an initialization routine, since these components may affect the simulation even if nothing is mapped onto it. Finally, one has to consider that the meta-platform specification grows combinatorially with the number of parameters. This can cause practical problems, considering that even unused model components have to be stored on disk (before simulation) or in memory (at runtime). However, as we will see below, a mapping-based representation using a meta-platform enables a very straightforward GA representation. We will use this representation for the case-studies throughout this chapter, since they are designed such that they do not suffer from the previously mentioned disadvantages.

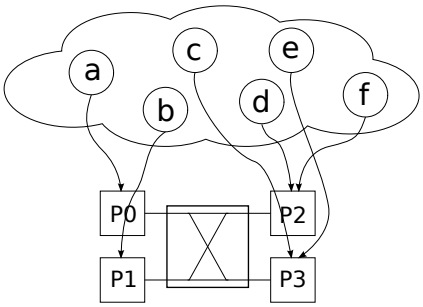


Figure 7.1: A simple homogeneous crossbar model

For the experiments in this chapter we will use a Sesame model that maps an application onto a crossbar-based meta-platform (Figure 7.1). A GA search algorithm will be used to find the best application-to-architecture mapping. Here we discuss the representation of the mapping as a GA chromosome. Firstly, we assume that there are no functional restrictions on the processors: all processors can execute all of the tasks (this is generally true for programmable processors). Secondly, we assume that each pair of processors can communicate so that there are no topological restrictions. Indeed, the crossbar in the proposed platform fully connects all processors, so processes can communicate regardless on which processor they are mapped. The result is that any task can be mapped onto any processor so that we do not have to make special provisions for infeasible mappings. As we saw in Chapter 3, KPN communication channels in Sesame are mapped explicitly onto communication structures in the architecture. However, the crossbar uses a local-write, remote-read paradigm, so that outgoing application channels are always mapped to the FIFO memory that is associated with the processor. This means that the channel mapping does not need to be specified, since it can be trivially derived from the task-mapping. So finally, we can conclude that the

platform can be completely specified by simply associating application tasks to processors:

$$\begin{array}{lcl} \text{Task } 1 & \Rightarrow & \text{Processor } 1 \\ \text{Task } 2 & \Rightarrow & \text{Processor } 2 \\ \text{Task } 3 & \Rightarrow & \text{Processor } 1 \\ \ldots & & \ldots \\ \text{Task } n & \Rightarrow & \text{Processor } k \end{array}$$

As a more convenient mapping description, we can use a vector of n processor identifiers, where the i-th index indicates the mapping target of task i:

$$\{p_0, \ldots, p_i, \ldots, p_{n-1}\}$$

Note that this commonly used description is very suitable to serve as the chromosome representation (or genotype) for a genetic algorithm. All possible combinations of integers will result in valid mappings, as long as $\forall i : p_i < k$. This means that implementing crossover and mutation operators will be relatively easy, since any recombination of such chromosomes results in a valid specification and the GA can be implemented without special consideration for "broken" chromosomes or repair mechanisms.

As a subsequent simplification, we will assume a homogeneous architecture where all processors are the same and a task incurs the same delay on each processor (except for additional delays caused by interfering tasks mapped onto the same processor). This condition also includes that the architecture is symmetrical: other delays from the system that affect the processor (e.g., network delays due to communication, memory delays, task scheduling overhead, etc.) should also be the same for each processor. This condition holds for many homogeneous architectures, for example, a multi-processor system where all processors are connected to a central bus with a non-prioritizing arbiter or indeed the crossbar system used in our model. A valid mapping specification is a partitioning of all n processes, where partitions may be empty or contain all n processes. Note that partitions may be empty (processor not in use) or contain all n processes (a single processor system). A processor that is not assigned any tasks (having an empty task partition) can be considered idle or non-existent. Note that many of the techniques presented in this chapter can be quite easily extended to more heterogeneous systems (this will be discussed as future work in Section 7.10).

Size of the design space

For a maximum of p processors in the platform, there are a total of p^n possible mapping descriptions. We need to be aware however, that multiple mapping specifications can describe the same mapping. For example, mapping $A : \{0,1,0,0,2,3\}$ and $B : \{2,0,2,2,3,1\}$ denote the same (duplicate) partitioning, and therefore an equivalent mapping on a homogeneous platforms. This can be solved by using a *normal form* notation of mappings, but here we introduce the *mapping distance* metric which can distinguish duplicate mappings and which will prove a useful concept later on. This metric is similar to the Hamming distance, or *edit* distance, except that it is independent of the symbols that make up the word.

We define the mapping distance $\delta(p,q)$ as the minimum number of task re-mappings that is required to transform mapping p in mapping q. For example, $\delta(A,B) = 0$ and $\delta(A,C) = 4$ if $C : \{0,1,2,3,1,1\}$. In general we have that for any p and q: $\delta(p,q) = \delta(q,p)$ and for duplicate mappings : $\delta(p,q) = 0$. Note that with minor modifications, the distance metric can also be defined for heterogeneous platforms (where duplicate mappings can still occur on groups of processors with the same type). A more detailed explanation of the distance metric is given in Section 7.6.2.

We can quantify the size of the design space as the number of partitionings (see Table 7.2). We can see that the design space is orders of magnitude smaller when we discard duplicate mappings, which is therefore an important consideration for the search algorithm. In the following we will show results of experiments where a genetic algorithm has been applied to our mapping design space using the vector mapping representation. Then we will give some suggestions on how the mapping distance technique can be used to improve the search algorithm. From the table we immediately see the exponential growth of the design

n	k	all descriptors (k^n)	unique points ($\sum_{i=1}^{k} S(n,k)$)
5	3	243	41
7	3	$> 2K$	365
9	4	$> 262K$	$> 11K$
11	4	$> 4M$	$> 175K$
13	4	$> 67M$	$> 2M$
15	4	$> 1G$	$> 44M$
15	9	$> 205T$	$> 1G$

Figure 7.2: Number of (unique) design points for given n, k

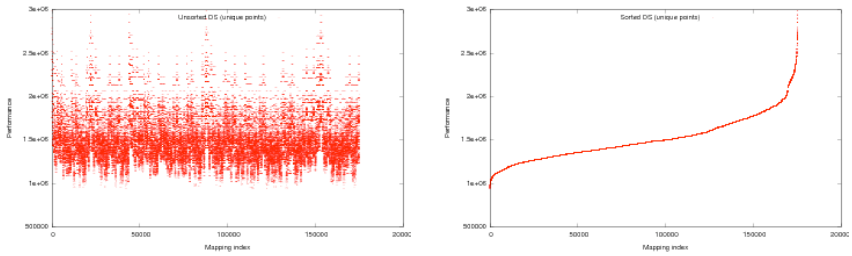
space: slight increases in the problem size (n or k) show orders of magnitude increase of the design space. We also observe that the number of unique mappings grows much slower than the number of chromosome representations (though it still grows exponentially). The difference between the number of all and unique representations is approximately: $k!$. In the next Section we investigate the GA behavior for one of these problems sizes.

7.4 Initial case study

Here we present the results of an initial, small-scale case study where the design space consists of an 11-process application mapped onto a 4-processor crossbar architecture. We also introduce a graphical method to quickly and concisely summarize and compare the results of different GA experiments, followed by a short note on implementation issues. But first we analyze the design space by exhaustive search.

Size and shape of the design space

As we can see from Table 7.2, a design space with $n = 11$ and $k = 4$ contains more than 4 million (175K unique) design points. We use an algorithm based on Nijenhuis en Wilf ([73], Chapter 11) to generate all unique design points and iteratively evaluate the Sesame model for each point. The result of this exhaustive search is that we have a complete view of the relationship between parameter space and objective space. We can pin-point the global optimum and use it in subsequent experiments to rate the relative quality of the results found by the GA. The unique design points are shown in Figure 7.3a in the order of Nijenhuis-generation and in Figure 7.3b as sorted by performance. Note that the same $275K$ of points are plotted in both figures, although more of them overlap in the sorted figure. The latter



(a) Order of generation (b) Sorted order

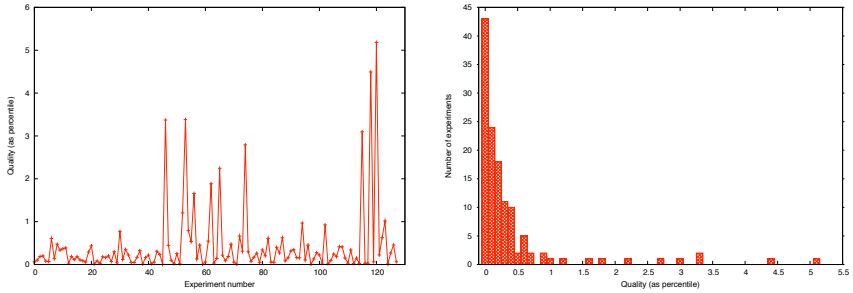
Figure 7.3: Unique mappings of 11-to-4 mapping design space

gives us an idea about the distribution of points in the objective space: most mappings perform in the middle performance segment, and a minority of mappings has either very good, or very bad performance. It is the goal of the DSE search algorithm for any given instance to identify the optimal mapping, or (if it proves impossible to find the optimum), to find a design point as close as possible to the optimum. In this case the optimal design point is $[0,1,2,1,1,3,0,0,0,3,0]$, which evaluates to 938460 clock cycles. Note that we can not make general assumptions about the shape of the design space and the distribution of high-quality results, which is likely to be different for other architectural platforms and application workloads.

Evaluation method

When analyzing results of heuristic search algorithms that are based on a random initialization (eg. the initial population in a Genetic Algorithm), we have to be aware that the result may be different each time the algorithm is run. In figure 7.4a we show the quality of the best design point found by 128 runs of our GA (we discuss the specific GA parameters later). Each result is shown as a percentile of its location in the sorted design space: this gives us an idea how “close” a design point is relative to the global optimum. In figure 7.4b the same data is shown as a histogram: although many results are very close to the global optimum (0-0.1 percentile), closer inspection of the data shows that the global optimum is not found in any of the experiments. From this we conclude that our sample design space is already quite

challenging for our GA. One can imagine that for bigger problem sizes, or more elaborate design spaces with more parameters, the design space search problem becomes even harder. In any case, it is critical to find search algorithms that show good convergence towards the optimum (whether they find the global optimum or not).



(a) Result for each experiment (b) As a histogram

Figure 7.4: Results of repeating a GA 128 times

In order to compare different search methods, we propose an evaluation method that is not based exclusively on the use of metrics in the objective space (in our case the simulated system runtime). Although objective space metrics are commonly used for this purpose, they can give a distorted view of the results, since the "landscape" of the design space can be unpredictable. For example, if we compare two search methods A and B for their performance on the design space in Figure 7.3b, then a difference between the search results of 250000 units in metric space equals a difference of 20000 design points if measured in the beginning segment of the sorted design space (closest to the optimum), but up to 110000 points in the middle segment of the design space. In this research we are more interested in a search method's convergence behavior, instead of the absolute values of the result. Therefore, we use a new metric that performs a fair comparison of different search methods (or of the same search method in different design problems). The metric is most easily represented as a graph, such as in Figure 7.5 where each line summarizes the results of all repetitions of one specific search method. The horizontal axis represents the quality of the result as a percentile towards the optimum (a lower percentile indicates a result closer to the optimum) and the vertical axis represents the probability of achieving a result with that quality. If one method is repeated r times, then the graph consists of r discrete dots, but a line is plotted through for visibility. In the following we will refer to this kind of plot as a Probability-Quality plot (PQ-plot). Since the x-axis requires the calculation of the result's percentile towards the optimum, PQ-pots like this are only possible for design spaces that have previously been exhaustively evaluated. In Section 7.8 we show a solution to represent larger design spaces in a similar way.

As an example consider the PQ-plot in Figure 7.5: the middle line shows the results of a GA with the following parameters: population size 15, 15 iterations, crossover probability 0.9, mutation probability 0.01. It shows that a result within the 0.1 percentile (belonging to the best $\frac{175K}{1000} \approx 175$) results can be achieved approximately 28% of the time (which has

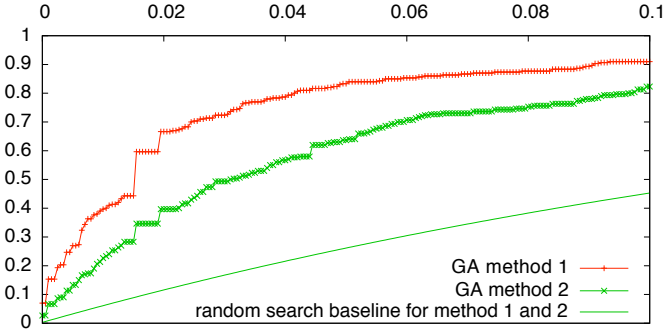


Figure 7.5: PQ-plot for repeated experiments of 2 different types of GA and one random search

been derived from repeating the method r times). The lowest, smooth line represents the theoretically derived probabilities of finding results using uniform random search (as will be discussed below). The top line represents another search method which dominates the middle line, indicating a better performing search method: it has been found that it has a higher probability of reaching search results with the indicated quality (or higher). In general it is desirable that a PQ-line that climbs rapidly to Probability=1, therefore PQ plots show only the interesting part of the x-axis (typically within 0.1-0.2 percentile). By comparing the lines for different search algorithms we can get an insight the relative (average) performance behavior.

An additional advantage of expressing the quality of an experimental result relative to the optimum is that we can make an easy performance comparison to random search. We define random search as the method that takes r random samples from the design space and returns the best result. It serves as a comparative baseline for experiments which can be plotted in a PQ-plot: each proposed algorithm should at least perform better than random search. We choose r to be equal to the number of evaluations of the other algorithm, so for a GA this would be population size $\times$ number of iterations When comparing different population sizes, multiple random-baselines will be shown: the plot key then lists the number of evaluations in braces to identify which baseline belongs to a particular GA experiment. Instead of performing an actual random search, we can theoretically derive the random baseline (the probability of finding a result within C percentile of the optimum with r random samples):

$$P_{\text{find}} = 1 - \left(1 - \frac{C}{100}\right)^r$$

Although the PQ-plots give a detailed overview of a search method's behavior, it does not specify the statistical significance of the difference between two or more search methods. In other words: when lines are close together, does the one on top represent a truly better method, or is it just a side-effect of the non-determinism of the initial random population? In order to discriminate these cases we compute for each experiment an approximate

confidence interval of the mean value of the r repetitions of a single search method. The confidence interval indicates how certain (as specified by the confidence level) we are that the real mean (mean and variance of the distribution represented by the r samples are unknown) lies within the confidence interval. The more the confidence intervals for different experiments are non-overlapping, the more significant the difference of the mean behavior. The confidence intervals for the results in Figure 7.5 is given in Figure 7.6 for different confidence levels (the mark in the center of a confidence interval represents the sample mean).

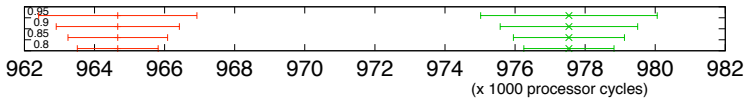


Figure 7.6: Confidence intervals for the experiments in Figure 7.5

The boundaries of the interval are computed as follows: $\text{mean}(x) \pm t(r-1)\frac{\text{sdev}(x)}{\sqrt{r}}$, where $t(r-1)$ is the upper $(1-C)/2$ critical value for the t distribution with $r-1$ degrees of freedom. Note that the confidence interval is only *approximate*, since there is no reason inherent to the model to assume that the samples will be normally distributed. However, we have performed several normality-tests on results of repeated GA experiments, showing they indeed follow the normal distribution quite well [2].

Implementation

The results in this chapter are obtained using a standard genetic algorithm library.In the next section we look at the DSE results using the unmodified (standard) GA, but in Sections 7.7 and 7.8 we introduce certain additional features and GA operators. Experiments can be done using "simulator-in-the-loop" (where the fitness function consists of a call to the Sesame simulation model), or using a database of cached simulation results. A separate program is available to exhaustively search the design space by generating every (unique) design point, run it through the simulator and store the result in a local database. In that case the GA fitness function is simply a lookup in the database of cached simulation results, which is of course only possible for relatively small design spaces. Considering the large number of GA-based experiments that has been done for this research, we exploit easily available parallelism in order to speed up the runtime of experiments. We can use parallelism at two levels: for evaluating single design points (e.g., in order to create the database exhaustive results) or for repeating experiments to measure average search-performance results (the experiments will run in parallel). The program to start and evaluate these DSE experiments

[2] The various distributions that are inherent to the simulation model (modeling different latencies in all parts of the model) may indeed accumulate to a normally distributed performance number, perhaps as an effect of the *central limit theorem* (CLT). However, we have no way to motivate the occurrence of normality or to prove applicability of the CLT. Therefore we emphasize that we claim (approximate) normality on the basis of the observations only.

will spawn a parameterizable maximum number of parallel processes. Our experiments showed that maximum speedup can be achieved by setting this parameter to the number of available cores in the shared memory multi-core system. For example, since most of our experiments have been run on a 48-core Magny cours AMD machine, we typically repeat experiments in multiples of 48. We note that the particular type of parallelism exploited here is "embarrassingly parallel": linear speedup can be achieved with more resources, such as for example by using a large compute cluster. In this chapter we do not report on the (wall-clock) runtime of experiments (for this we refer to Chapter 4), instead we use the required number or design point evaluations (database lookups or calls to the simulator), which take a constant amount of time on any given platform. This allows for a more useful metric of design space search performance by considering separately the issue evaluating a single design point and the issue of searching through the design space. For example, other DSE approaches may be better at searching through a design space, but require a longer evaluation time for a single design point, or vice versa.

7.5 Initial case study - parameters

In this section we perform an initial exploration of the effect of different parameters on the performance of the standard GA. We compare the results with the theoretical random-search baseline and with each other. The parameters available in typical GA implementations are: population size, crossover and mutation rate, selection strategy and selective pressure. Furthermore, we investigate the population diversity throughout the run of the GA. Although the experiments in this section give us an indication of good GA parameters, we can't assume that these will be suitable for all case-studies in general (where design spaces and problem sizes may be different). When we consider a different design problem, we always perform sanity-checks to see that (for the given problem) we are still using a good set of GA parameters. Here we report our findings for the case study from the previous section. We discuss the issue of the generic applicability of these findings later in this chapter.

First we consider the population size parameter, which can be regarded as one of the most influential parameters. Larger populations will typically result in better GA performance, since the initial (random) population is more diverse and because the GA has more internal parallelism, so that more recombinations can occur in the same iteration (thus possibly converging faster to the optimum). However, larger populations come at the cost of more evaluations. For the current design space we can simply use pre-computed results from a complete store in memory, but when using the simulator to evaluate chromosomes, the GA execution time will increase with larger populations. The extra execution time can sometimes be mitigated by the fact that the GA converges faster and fewer GA iterations are required or because a much better optimum is found. Therefore, the best GA is a trade-off between number of evaluations (depending on population size, number of iterations, population diversity) and resulting quality. In Figure 7.7 the P-Q plot is given of 300 repeats of a GA with different population sizes and constant mutation rate (0.1), crossover probability (0.9), tournament selection and a uniform crossover strategy. We can see that the largest pop-

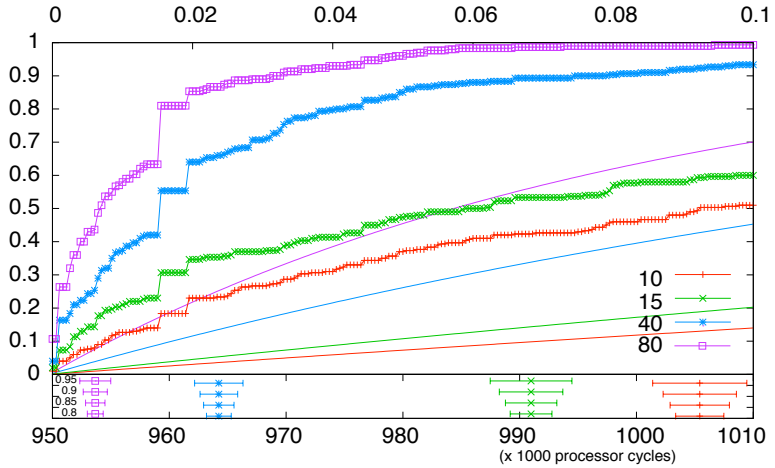
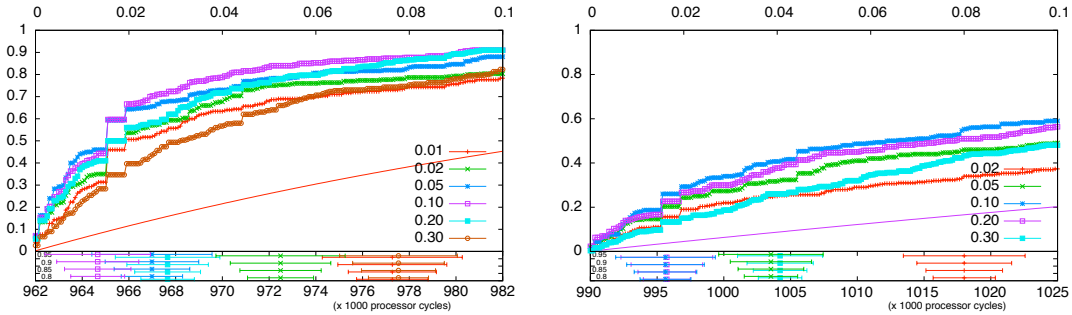


Figure 7.7: P-Q plot for different population sizes

ulation size (80 chromosomes) has a better chance to find results of certain quality. Note that the figure shows multiple random-search baselines for fair comparison with each of the GAs, allowing random-search the same number of evaluations (as indicated by the last number in the plot legend). From the results we conclude that although population size 80 obtains the best result, a population of 40 is a reasonable performance-execution time trade-off. We take population size 40 as a basis for comparison in the remainder of this section.

Next we look at the influence of the mutation parameter, the left side of Figure 7.8a shows the result for population size 40. In both figures we see a trend that a very small mutation rate does not yield a good performance (mutation 0.01, 0.02), nor does a very high mutation rate (0.3). The most effective mutation rate seems to be between 0.05 and 0.2. The existence of an optimal mutation range can be explained by the fact that mutation enables diversification in the population. This can result in new optimal solutions to be found and to avoid sticking to local optima in the design space. However, when there's too much mutation, the GA becomes more and more like a random search and is unable to converge towards any optima (local or global). It seems that the mutation rate has an even greater influence on a GA with a smaller population (compare e.g. the difference between mutation 0.05 and 0.20 in both figures). A reasonable explanation seems that the larger population has more randomness from its initial population, whereas a smaller population does not. For now we select mutation rate 0.10 as the basis for our future experiments.

In Figure 7.9 we see performance for different crossover probabilities. A higher crossover probability indicates that two genes exchange more genetic material once they have been selected to create offspring. We can see that the results are close together for significantly different crossover rates, suggesting that the crossover rate has little influence on the GA. Indeed, in the wider GA research field it is a hot topic of debate how important actually is the crossover step in a GA. There is no general consensus on this topic, but including some form of crossover seems to be generally beneficial. In the following, we will continue to use



(a) Population size 40 (b) Population size 15

Figure 7.8: P-Q plot for different mutation rate

the commonly applied 0.9 crossover rate.

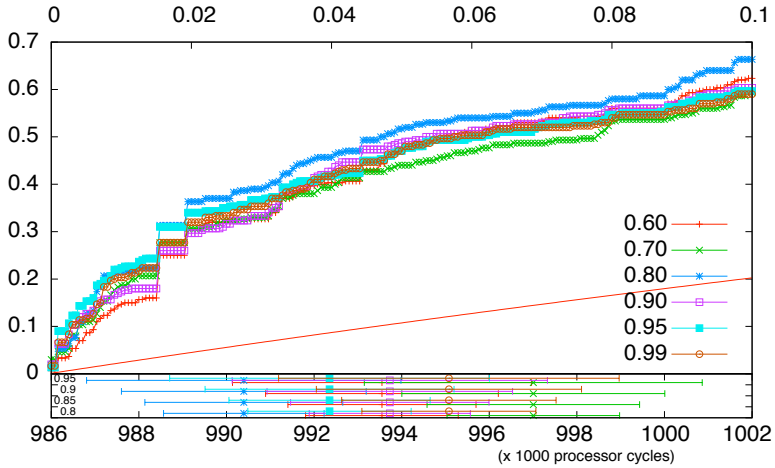
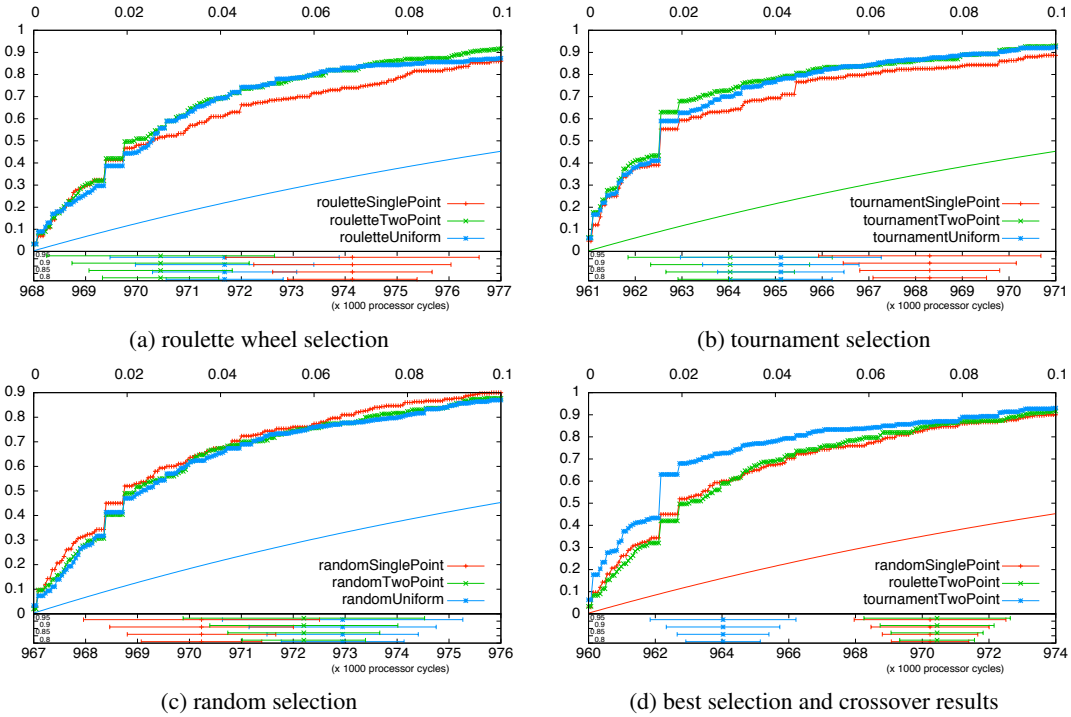


Figure 7.9: P-Q plot for different crossover probabilities

Furthermore, the method that is used to select which chromosomes are candidates for crossover may affect GA performance: the selection strategy. A selection strategy typically selects two chromosomes from the population, which the crossover method will then combine into two new chromosomes. Here we consider some of the commonly used selection strategies: roulette-wheel (selection proportional to the chromosome fitness), tournament (select fittest chromosome from a small random subset of the population) and random (select random chromosomes from population). For each selection strategy we consider a few types of crossover mechanisms: single-point crossover (breaking a chromosome at a random point and exchange the end-pieces), two-point crossover (breaking a chromosome at two points and exchange the middle piece) and uniform crossover (a child chromosome is built-up as a sequence of genes that may come from any one of the parents). The results are shown in Figure 7.5. Each graph (7.10a,7.10b,7.10c) shows the result of a particular



(a) roulette wheel selection

(b) tournament selection

(c) random selection

(d) best selection and crossover results

Figure 7.10: P-Q plots for different selection strategies and crossover methods

selection strategy in combination with the three types of crossover. The last graph (7.10d) compares the best result of each selection type. First of all we observe that the PQ-lines in each case lie very close together: apparently the choice of crossover method does not influence GA performance in a significant way. The overlapping confidence intervals confirm this: only at a low significance level can we distinguish the use of different crossover operators. For example, in Figure 7.10b, only the single-point crossover has a non-overlapping confidence interval, but only for confidence levels 0.85 or lower. In Figure 7.10d the best result for each selection type is plotted, and we see that tournament selection (with two-point crossover) shows a measurable improvement over roulette and random selection.

One final GA parameter is selective pressure, which can be defined as the preference of the selection method to choose chromosomes with a better fitness over chromosomes with worse fitness. A high selective pressure can allow a GA to converge quicker, but can also mean that the GA is likely to get stuck in local optima. With selective pressure too high, only successful chromosomes make it into the next population and successful combinations with lesser chromosomes may be missed. This can also be translated into a diversity problem: only genetic material from chromosomes with high fitness make it into the new population, so the diversity of the new population may be low. On the other hand, when selective pressure is too low, a GA will have problems converging to any optimum (thus resembling a random walk). Finding a good balance between convergence and diversity is considered an

important issue in the field of evolutionary algorithms. We will revisit this topic later in this section and throughout this chapter.

In our current experiment we can most easily try out different selective pressures for the tournament selection method. Selective pressure can be increased by increasing the size of the random subset of the population (from which the chromosome with highest fitness is selected) as weak individuals have to compete with more other chromosomes. Vice versa, selective pressure goes down for a smaller set (note that set-size 1 is equal to random selection). The results in Figure 7.11 show that varying subset sizes between 2 and 16 has very little influence on GA performance for the current problem.

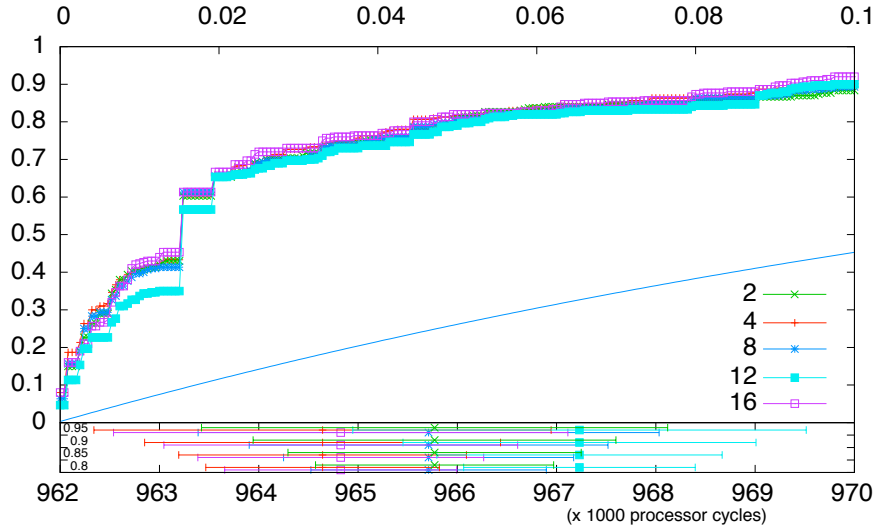


Figure 7.11: P-Q plot for varying selective pressure

Next we take a look at the diversity of the populations during the run of the GA. For this purpose we measure in every generation of the GA how many of the population individuals are new (have not appeared in any previous generation). Since the results show a similar trend in the different experiments discussed so far, we only show the diversity for the experiment with different mutation rates. The left side of Figure 7.12 shows the number of new individuals per generation, whereas the right shows the same data cumulatively. The points shown are averages for 300 repetitions of each GA; the initial population is shown as generation 0. Not surprisingly, the highest mutation rate (0.20) introduces the most new individuals per generation. In all cases, the number of new individuals drops quickly even after the second generation, and the last generations barely introduce more than 1 new individual. In the right figure the last data point on each graph shows the total number of unique values throughout the run of the GA. We observe that increasing the mutation rate four-fold (from 0.05 to 0.20), the diversity increases by only 23%. Moreover, we know from our previous experiments that increasing mutation rate beyond 0.20, has an adverse affect on the quality of the found optima. Indeed, it is generally known that there is a delicate balance between

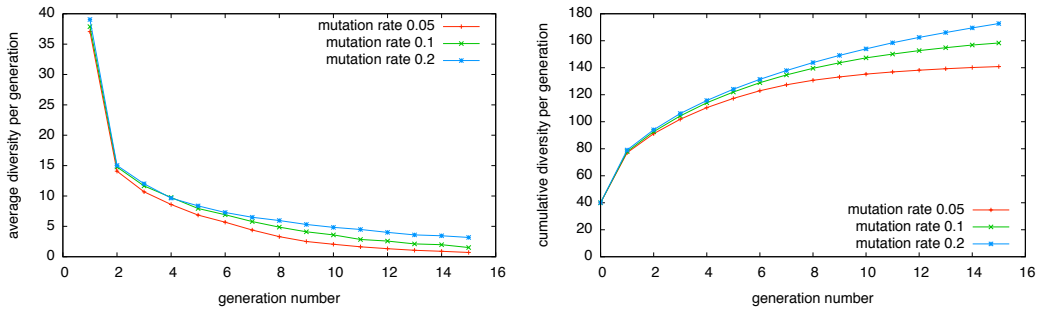


Figure 7.12: Diversity per generation (left) and cumulative (right)

diversity (which can be improved by higher mutation rate) and convergence (which may be disrupted by a too high mutation rate): a higher mutation rate may prevent the GA from getting stuck in local optima, but may also prevent convergence towards the best solution. We revisit this topic in Section 7.7, where we propose a new method of managing population diversity based on the distance metric.

The diversity analysis of Figure 7.12 suggests a particular performance optimization of the GA: individuals that occur multiple times do not really need to be evaluated again. By storing the evaluation result of each individual, the number of calls to the simulator can be reduced. We have not applied this optimization in our case, but the performance of the GA could be increased significantly this way. This is particularly true for DSE case studies that perhaps use a simulator that takes orders of magnitude more time to evaluate a single design point (e.g., instruction set simulators). As an example, the GA with mutation rate 0.2 calls the simulator $40 \times 16 = 640$ times (including the initial population), out of which only 173 on average are unique individuals. By using the proposed optimization, a speedup of approximately 3.6 can be obtained over a non-distributed GA implementation (where evaluations are performed sequentially).

Finally, we note that the baseline in the PQ plots is based on uniform random search using the number of design point evaluations as given by the non-optimized number of evaluations:

$$(\text{number of generations} + 1) \times \text{population size}$$

If we would allow random search to evaluate a number of individuals that is equal to the accumulated number of unique individuals in each of a GA's populations, then the PQ-plot of the GA would show even better performance relative to the random search baseline.

7.6 Distance metric

In this section we expand on the topic of the distance metric, which was shortly mentioned before in Section 7.3. First we discuss some properties of the design space that motivates the possible usefulness of such a metric. Subsequently, the distance metric will be described in

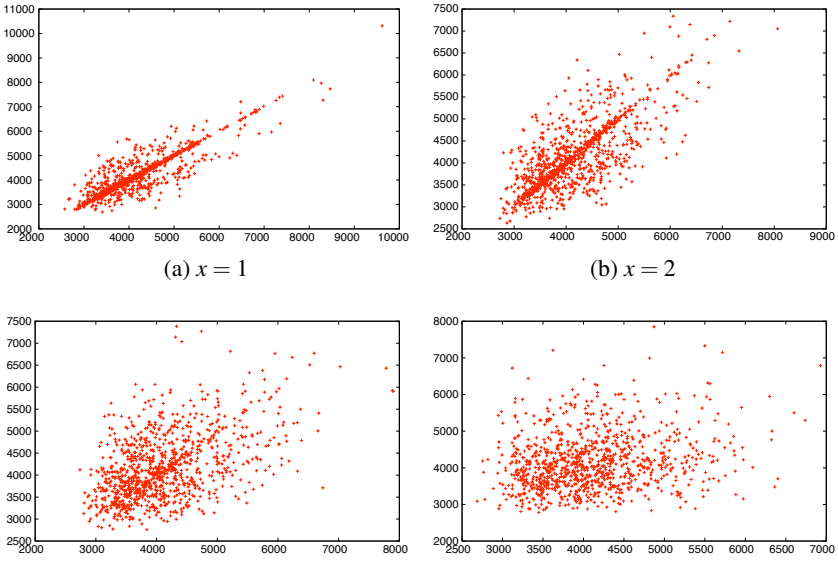
detail and possible implementations are discussed. In the remainder of this chapter we will consider how the standard GA methods can be improved using the distance metric.

7.6.1 Observations on the design space

As we have described previously, our chosen design space focuses on the mapping between application tasks and architectural resources. The performance of a single design point is heavily influenced by the (communication) dependencies between nodes in the task-graph and the dependencies that are introduced by sharing of architectural resources. The Sesame simulator captures these dependencies in its models and the resulting performance evaluation. A small change in the dependencies can, in theory, result in a completely different performance result. For example this is the case when we change the mapping of a single task such that it is added to or removed from the dependency chain that is part of a performance bottleneck in that design point. However, intuitively, we think that in general most small changes will not result in a hugely different performance result. We confirm this hypothesis by checking the correlation of performance results for pairs of design points. First a random set of design points is created, then we mutate each chromosome in x positions by randomly choosing the positions *and* its mutation value. The results are shown in Figure 7.13 for a design problem where an application with 20 nodes is mapped onto a homogeneous, symmetrical architecture with 8 processors. Each dot represents a pair of design points with the performance of the first design point along the x-axis and the performance of the second along the y-axis. We can see a clear correlation between points that have mutated in 1 or 2 positions (top graphs), but this relation fades and disappears with changes in 4 or 8 positions (bottom graphs). Apparently, it is indeed true that (on average) small changes in the design point specification lead to small performance differences. We propose methods to exploit this correlation by integrating this piece of domain knowledge into our GA methods in Section 7.8.

7.6.2 Distance metric details

Here we present an implementation of the distance metric as defined in Section 7.3. For any pair of mappings (A,B), the algorithm will perform stepwise reassignment of tasks in B such that the result is equivalent to A and that the number of required reassignments is minimal. The algorithm considers the mapping as a partitioning of a set of *task groups*: processes mapped onto the same processor are in the same task group. In each recursive iteration of the algorithm, a pair of task groups $(\mathrm{tg}_A, \mathrm{tg}_B)$ will be selected where tg_A is a task group from mapping A and tg_B is a task group from mapping B. Next, certain tasks will be reassigned such that group tg_B becomes the equivalent of task group tg_A. In each level of recursion another pair of task groups will be selected and again reassignment of tasks takes place. This continues until the task groups in each mapping are equivalent: this means that mappings A and B are now equivalent. The essence of the algorithm is to find a sequence of task group pairs such that the accumulated number of reassignments to turn A into B is minimal.



(a) $x = 1$ (b) $x = 2$

(c) $x = 4$ (d) $x = 8$

Figure 7.13: Correlation between results with mutation in x gene positions

Let **map** be a function that maps n tasks onto a k-processor system.

$$\mathbf{map} : \{0 \ldots (n-1)\} \longmapsto \{0 \ldots (k-1)\}$$

Let A and B be two mappings where the i-th element in A is denoted as $A[i]$:

$$A = [A[0], \ldots, A[n-1]] = [\mathbf{map}_A(0), \ldots \mathbf{map}_A(n-1)]$$
$$B = [B[0], \ldots, B[n-1]] = [\mathbf{map}_B(0), \ldots \mathbf{map}_B(n-1)]$$

Then a task group of a mapping A is defined as a set

$$\mathrm{tg}_{A,x} = \{t \in \{0 \ldots n-1\} \mid \mathbf{map}_A(t) = x\}$$

Note that when mappings A and B do not map to the same number of processors, then one of the mappings has some empty task groups (which does not influence the working of the mapping distance algorithm).

Next we list the three important stages from the recursive algorithm:

Step 1: group selection

Find a task group from each mapping to form a pair:

$$(\mathrm{tg}_{A,i}, \mathrm{tg}_{B,j}) \quad (i, j \in (0 \ldots k-1))$$

such that:

1. they share the maximum number of tasks: i, j with $\max(|\,\mathrm{tg}_{A,i} \cap \mathrm{tg}_{B_j}\,|)$, *and*
2. i and j have not been part of a task group pair in a previous iteration of the algorithm

If there are no more task groups that meet the requirements, then B is now equivalent to A and the algorithm has finished. The accumulated value of the distance counter is returned as well as the sequence of task group pairs that was used to rewrite B to A (the latter will be used later to generate a "minimum path" from A to B, see Section 7.8.2. Note that there may be more than one sequence that transforms A to B with the same number of reassignments, but finding one such sequence is sufficient for our purpose.

Step 2: recursion

The pair found by the previous step will be used for task reassignment. However, there may be multiple pairs that have intersections of the same (maximum) size. In this case it is unknown which pair should be used for reassignment, so there's no other option than to try all of those pairs. To this end, a copy B' of B is created of every pair and the task reassignment function (see below) is applied. Next, the distance algorithm will be called recursively to calculate the distance between A and every B'. When the recursion has finished, we select and return:

- the minimum found distance between A and B'
- the corresponding sequence of task pairs

Reassignment Function

This function takes as input a task group pair $(\mathrm{tg}_{A,i}, \mathrm{tg}_{B',j})$ from mappings A and B' respectively. The pair will now be used to modify mapping B' such that $\mathrm{tg}_{B',j}$ includes at least those tasks that are in tg_{A_i}:

$$\forall y \in (0 \ldots k-1): \quad B'[y] \overset{reassign}{:=} j \quad \text{if} \quad A[y] = i \text{ and } B'[y] \neq j$$

This results in: $\mathrm{tg}_{A,i} \subseteq \mathrm{tg}_{B',j}$. Note that the additional tasks $\{\mathrm{tg}_{B',j} - \mathrm{tg}_{A,i}\}$ (if any) will be reassigned in a later iteration such that finally $\mathrm{tg}_{A,i} = \mathrm{tg}_{B',j}$. Note that the number of reassignments may be 0, in which case the distance counter is not increased.

7.6.3 Example

To illustrate the above algorithm, consider the following two mappings A and B for a 6 process application and any 4 processor (homogeneous and symmetrical) architecture.

$$A : [0,1,2,2,2,3] \quad B : [0,1,1,0,0,0]$$

In the **first iteration** of the algorithm we have the following task groups:

$$\begin{array}{ll} \mathrm{tg}_{A,0} = \{0\} & \mathrm{tg}_{B,0} = \{0,3,4,5\} \\ \mathrm{tg}_{A,1} = \{1\} & \mathrm{tg}_{B,1} = \{1,2\} \\ \mathrm{tg}_{A,2} = \{2,3,4\} & \mathrm{tg}_{B,2} = \{\} \\ \mathrm{tg}_{A,3} = \{5\} & \mathrm{tg}_{B,3} = \{\} \end{array}$$

We find that the pair $(\mathrm{tg}_{A,2}, \mathrm{tg}_{B,0})$ has the largest overlap (tasks 3 and 4). Therefore: $B'[y] = 0$ if $A[y] == 2$, results in: $B' = [0,1,0,0,0,0]$. The distance counter is incremented by 1, because task 2 was reassigned. In the **second iteration** the relevant remaining taskgroups are:

$$\begin{array}{ll} \mathrm{tg}_{A,0} = \{0\} & \mathrm{tg}_{B',1} = \{1,2\} \\ \mathrm{tg}_{A,1} = \{1\} & \mathrm{tg}_{B',2} = \{\} \\ \mathrm{tg}_{A,3} = \{5\} & \mathrm{tg}_{B',3} = \{\} \end{array}$$

The largest overlap is between the pair $(\mathrm{tg}_{A,1}, \mathrm{tg}_{B',1})$, consisting of task 1 only. No tasks are reassigned, since $B'[1]$ is already set to 1. Therefore the distance counter is not incremented. In the **third iteration** the relevant remaining task groups are:

$$\begin{array}{ll} \mathrm{tg}_{A,0} = \{0\} & \mathrm{tg}_{B'',2} = \{\} \\ \mathrm{tg}_{A,3} = \{5\} & \mathrm{tg}_{B'',3} = \{\} \end{array}$$

All combinations of task groups from A and B'' now have the same intersection (the empty set). At this point we do not know with which pair to proceed, therefore the algorithm is run recursively for each pair: $(\mathrm{tg}_{A,0}, \mathrm{tg}_{B'',2})$, $(\mathrm{tg}_{A,0}, \mathrm{tg}_{B'',3})$, $(\mathrm{tg}_{A,3}, \mathrm{tg}_{B'',2})$, $(\mathrm{tg}_{A,3}, \mathrm{tg}_{B'',3})$. In this case all 4 recursive branches will find a minimum of 2 additional reassignments. In general one of the shortest recursive branches is selected. In the following we show only the first recursive branch, so we reassign according to pair $(\mathrm{tg}_{A,0}, \mathrm{tg}_{B'',2})$: $B''' = [2,1,0,0,0,0]$ and the distance counter is incremented with one, making the total recorded number of reassignments 2. In the **fourth iteration** the relevant remaining task groups are:

$$\mathrm{tg}_{A,3} = 5 \qquad \mathrm{tg}_{B''',3} = \{\}$$

The only possible pair is $(\mathrm{tg}_{A,3}, \mathrm{tg}_{B''',3})$. We apply the reassignment rule and with one reassignment we get: $B'''' = [2,1,0,0,0,3]$. The algorithm is finished and B'''' is now an equivalent mapping to A. The distance counter has reached its final value of 3, which is the guaranteed minimum number of reassignments required to change mapping B into mapping A. The sequence of task group pairs used was:

$$(\mathrm{tg}_{A,2}, \mathrm{tg}_{B,0}),\ (\mathrm{tg}_{A,1}, \mathrm{tg}_{B',1}),\ (\mathrm{tg}_{A,0}, \mathrm{tg}_{B'',2}),\ (\mathrm{tg}_{A,3}, \mathrm{tg}_{B''',3})$$

In this example, the following sequence would also have found the minimum:

$$(\mathrm{tg}_{A,2}, \mathrm{tg}_{B,0}),\ (\mathrm{tg}_{A,1}, \mathrm{tg}_{B',1}),\ (\mathrm{tg}_{A,3}, \mathrm{tg}_{B'',3}),\ (\mathrm{tg}_{A,0}, \mathrm{tg}_{B''',2})$$

7.6.4 Performance improvement

As mentioned before, the essence of the distance algorithm from the previous section is to find the sequence of task groups pairs that leads to the minimum number of reassignments. It exhaustively tries all all permutations of pairs, resulting in an algorithmic worst-case complexity of $O(k!n)$. Using a built-in optimization, it can perform better only when the two input mappings have overlapping task groups that *do not* overlap by the same number of tasks. Although the algorithmic complexity is fine for small k and n, it does not scale well to large problem sizes. Fortunately, it is possible [100] to reduce the distance metric problem to an assignment problem, which can be solved in only $O(n^3)$ using the Munkres assignment algorithm [57].

Munkres works on an assignment-cost matrix, e.g., workers (in columns) perform a job (in rows) for a cost specified in the matrix. The Munkres algorithm then finds the minimal assignment of jobs to workers such that the total cost is minimized. We can define the matrix for two mappings A and B as $m_{ij} = n - |\mathrm{tg}_{A,i} \cap \mathrm{tg}_{B,j}|$ for each task group i from A and j from B. In this way, m_{ij} represents the cost of transforming task group i to task group j: the cost is lower when the groups overlap more. Note that m_{ij} is in general not equal to the required number of reassignments, but rather we choose n (number of tasks) so that all $m_{ij} \geq 0$. The outcome of the Munkres algorithm is a list of pairs of task groups such that the cost is minimal. By applying the list of pair groups as reassignments to B, we can transform B into A and obtain the distance value. An example of reassignment according to a list of task groups is given in Section 7.8.2.

7.7 Initial case study with distance metric

In Section 7.4 we mentioned that the balance between diversity and convergence in a GA is delicate and that both properties are important for a successful GA. In the previous section the distance metric was proposed as a way to discriminate equivalent and similar designs. Here, we propose a few simple methods to incorporate the distance metric into the GA in order to manage (increase) population diversity in the hope that this will help the GA to sweep a broader range in the design space. We perform the same evaluation as in Section 7.4 using the same design problem.

We have previously seen that our GA tends to converge towards a very homogeneous population: in some runs we saw that the final generations hardly introduce any new chromosomes. This means that at that point, the GA has converged prematurely to a local optimum, or it has simply already achieved its best possible result. Here we test what happens when we artificially introduce more diversity in the population of each GA iteration. For this purpose we analyze in each GA generation, the clustering of elements in terms of the distance metric. For each individual we calculate the average distance towards the other individuals; if the average distance is low, then we consider the individual to be very clustered. In the first of the two methods introduced here, we select a few of the most clustered individuals in the GA population for replacement by new, randomly initialized individuals. The rationale

is that the insertion of random genetic material will help the GA to continuously consider alternatives other than the solutions on which it is converging (thus increasing diversity). This is in many ways similar to the function of the GA mutation operator, except that we target specific individuals that will be completely replaced, whereas mutation randomly modifies only small parts of the chromosome (genes). In order not to override completely the natural convergence of the GA, we first replace only a percentage of population individuals. The result is shown in the left part of Figure 7.14, the line indicated with replacement 0 is the normal GA (without any replacements) as a reference. The other lines represent GAs where 5, 10, 20 or 40 percent of most clustered part of the population gets replaced. We see that none of these GAs is able to perform better than the reference GA, although, up-to 20% can be replaced without much harm to the GA (40% however clearly reduces GA performance). The first of the three numbers indicates the cumulative diversity of the population (excluding initial population). It shows that a higher replacement rate does increase diversity, but clearly this does not help to improve overall GA performance. In a follow-up experiment we replace a dynamically changing proportion of the population instead of a statically fixed percentage. Here we increase the rate of replacement with 1 per generation, e.g., 1 in the first generation, 2 in the second generation, etc. However, we vary the parameter I, which indicates how many generations proceed normally before the first replacement starts. So one replacement occurs in generation I, two replacements in generation $I+1$, etc. In our experiment we have a maximum of 15 generations, so we choose $I = 1, 4, 9$ and inf, where inf indicates the normal GA (no replacements). The right side of Figure 7.14 shows clearly that $I = 1$ gives the worst results . This is perhaps not surprising since $I = 1$ replaces a number of individuals in the population equal to the generation number: in the last population of the GA, all elements will be replaced by random values. Apparently, this interferes too much with the normal working of the GA. Different values for I seem to do slightly better: the best result being obtained by $I = 9$, which starts replacing individuals in the ninth generation. However the confidence intervals show that the result of $I = 9$ is not significantly better than the normal GA ($I =$ inf).

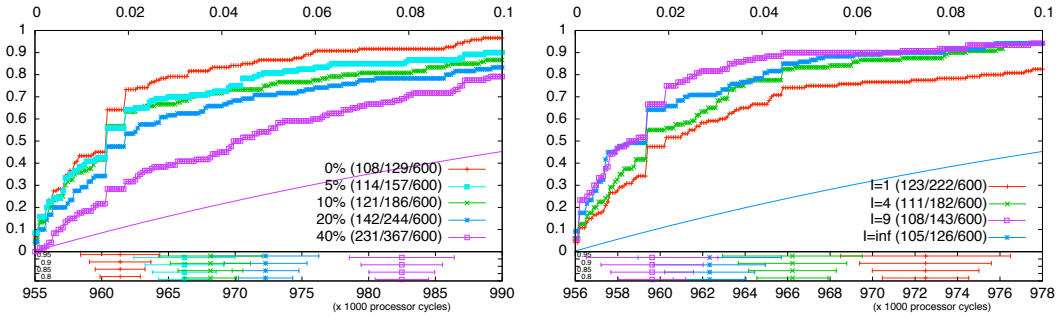


Figure 7.14: Replacing most clustered elements (static (l) and dynamic (r) proportion)

In a follow-up experiment we replace a dynamically changing proportion of the population instead of a statically fixed percentage. Here we increase the rate of replacement with

1 per generation, e.g., 1 in the first generation, 2 in the second generation, etc. However, we vary the parameter I, which indicates how many generations proceed normally before the first replacement starts. So one replacement occurs in generation I, two replacements in generation $I+1$, etc. In our experiment we have a maximum of 15 generations, so we choose $I = 1, 4, 9$ and inf, where inf indicates the normal GA (no replacements). The right side of Figure 7.14 shows clearly that $I = 1$ gives the worst results . This is perhaps not surprising since $I = 1$ replaces a number of individuals in the population equal to the generation number: in the last population of the GA, all elements will be replaced by random values. Apparently, this interferes too much with the normal working of the GA. Different values for I seem to do slightly better: the best result being obtained by $I = 9$, which starts replacing individuals in the ninth generation. However the confidence intervals show that the result of $I = 9$ is not significantly better than the normal GA ($I =$ inf).

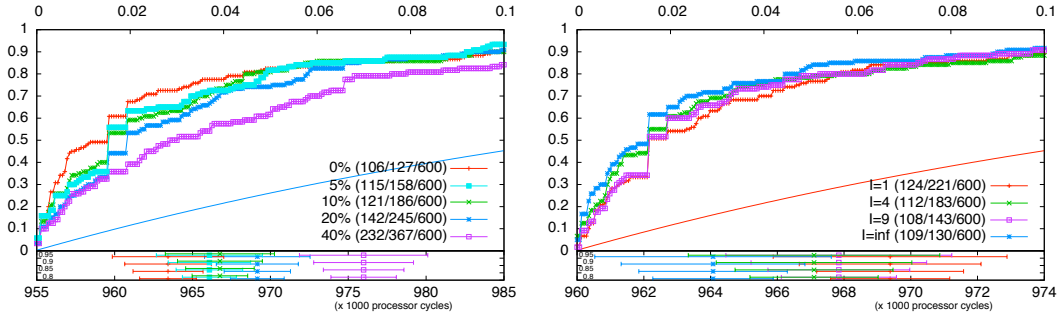


Figure 7.15: Replacing least clustered elements (static (l) and dynamic (r) proportion)

Perhaps we have overlooked a possible risk in the proposed strategy: it can be expected that the most clustered elements are often also the best (fittest) elements in a given population, since the cluster is the convergence "focal point" of the GA. Moreover, these fit elements are more likely to contain fragments of genetic material that contribute to the final optimal solution. So by replacing the most clustered elements, we may well be disrupting the natural GA convergence. Therefore, we now test what happens if we replace the *least* clustered elements: the results are shown in Figure 7.15 for both the static and dynamic replacement schemes. We see that the results for various I are very close to, but generally worse than the standard GA and that the confidence intervals are mostly overlapping, so we have to conclude again that the new strategy has no benefit.

Finally, we analyze how much diversity is actually introduced by the different methods presented in this section. The legend in each of the four plots shows three numbers measuring diversity. The first two numbers measure the average cumulative number of unique chromosomes (according to the distance metric) in the final generation. This is equivalent to the total number of unique individuals during the run of the GA. The first number measures diversity after application of the crossover and mutation operators, whereas the second number measures before. The third number is the product of number of generations and population size (the random-search baseline is calculated using this number, as we

saw in Section 7.5). Theoretically, the static method at 20% replacement introduces up-to $15 * (40 * 0.2) = 120$ new individuals which is the same as the dynamic method for $I = 1$: $1 + 2 + \ldots + 15 = 120$. For example, in Figure 7.15, comparing the second diversity number between 0% and 20%, we indeed see (approximately) the expected diversity increase (127 vs. 245). If we compare the first diversity number for the static and dynamic experiments, then we see that the static method has a higher diversity (142) than the static method (124). A similar observation can be made for the experiments in Figure 7.14. Apparently the dynamically increasing rate of replacement has a smaller effect on diversity than the dynamic method. Moreover, we observe the difference between the first and the second diversity number. Many of the new individuals that are inserted at the start of a generation (the second number) have disappeared by the end of the generation (the first number). The difference can be explained by the fact that randomly inserted individuals are lost from the population after selection, crossover and mutation. This suggests that the randomly inserted individuals are –on average– of insufficient quality to survive to the next generation.

We conclude that the methods presented in this section are able to increase diversity in the GA populations. However this is clearly shown to have no positive impact on the overall performance of the GA. In the next section, a different approach proves to be more successful.

7.8 GA with integrated distance metric

As we have seen in Section 7.7, the methods to control population diversity based on the distance metric did not contribute to improving GA performance. It is possible that the problem with those methods was cause by the fact that they were artificially layered on top of the normal GA operation. After each generation (or similarly: at the beginning of the next GA generation), the population would be artificially changed. Although this increased overall diversity in the populations, there was no measurable and predictable improvement in the GA performance. Another disadvantage of the proposed GA-extensions was the computational overhead of computing the average cluster distance in each generation, which required n^2 distance calculations for population size n.Therefore, scalability would be a problem for larger population sizes. To avoid such problems, we continue the search for generic, scalable improvements of the standard GA solutions in this section. However, we now propose more integrated approaches based on our previous observations about the design space in the context of the distance metric. This has resulted in the definition of new GA-operators for crossover and mutation. These new GA-operators attempt to optimize GA performance either by

- reducing the redundancy present in the chromosome representation
- using a distance metric-based crossover

Combinations of these are also possible and we will show and analyze the experimental results in a way similar to the previous sections. In this case, however, we scale up the

reference design problem to a size that is no longer practical for exhaustive evaluation, but which is more realistic for practical design problems.

7.8.1 Reducing representation redundancy

A chromosome representation of a design point with k processors can be represented in $k!$ different ways by permutating the k processor labels. This is sometimes referred to as the "symmetry" of the search space ([100]). Search algorithms may be affected by symmetry in the search space, and is known to have a negative impact on the performance of genetic algorithms in some problem domains, such as for example in the case of the graph coloring problem ([106]). We investigate whether the same is true for the search spaces of our design problem. For this purpose we propose a set of genetic operators that enable the GA to traverse the design space without symmetry. Intuitively, removing the symmetry from the design space should result in a more efficient search, as it effectively makes the design space smaller. But it may equally be the case that the GA, which is optimized for combinatorial problems, searches through the symmetrical subspaces with ease. Therefore, whether symmetry is a limiting factor on performance is yet to be determined.

We observe that for our chosen chromosome representation, it is easy to convert a set of chromosomes to equivalent chromosomes with a representation from the same, symmetry subspace. Note that the Nijhuis-Wilf algorithm (that we used in Section 7.4) to exhaustively generate all design points) produces all design points in base-symmetry form. Following the naming convention from Section 7.6.2, it holds for each base-symmetry chromosome representation A:

$$\begin{aligned} A[0] &= 0 \\ A[i+1] &\leq max(A[0],\ldots,A[i])+1 \end{aligned}$$

In the work of [106] the assignment function represented by A is called a Restricted Growth Function (RGF): from left-to-right (starting with 0), the mapping target is identified using the lowest possible number. This leads to a simple (order $O(N)$) re-assignment function to change a mapping to its unique equivalent in the base-symmetry space. This is shown in Algorithm 1; *base* is an array initialized to -1 for all elements, A is the design point to be re-written.

We now use the baseform function to enforce that subspace boundaries are not crossed during the normal operation of the GA. The simplest way to implement this is to append the baseform function to each normal GA operator. So, for example, a normal 2-point crossover could transform parent chromosomes A and B in child chromosomes A' and B'. After a subsequent application of the baseform function to both chromosomes, the result is A'' and B''.

Algorithm 1 The *baseform* function

```
cnt ← 0
base ← [−1,...,−1]
for i = 0 → (n − 1) do
  idx ← A[i]
  if base[idx] < 0 then
    base[idx] ← cnt
    cnt ← cnt + 1
  end if
  A[i] ← base[idx]
  i ← i + 1
end for
```

before crossover:	[0,0,0,0,0,0,1,2]=A
	[0,1,1,1,2,2,2,2]=B
after crossover:	[0,0,0,0,2,2,2,2]=A'
	[0,1,1,1,0,0,1,2]=B'
after baseline:	[0,0,0,0,1,1,1,1]=B''
	[0,1,1,1,0,0,1,2]=B''

The mutation function is similarly appended with the baseform function. With the extended crossover and mutation operators, all chromosomes in each generated population are guaranteed to remain in the base-symmetry space. Note that it is possible, but not necessary to apply the baseform function to the initial random population, since all individuals would automatically be transformed to baseform notation after the first GA iteration.

7.8.2 A distance-metric based cross-over operator

Our initial motivation to introduce the distance metric was to provide a measure of similarity between design points. We saw in Section 7.6.2 that the distance metric can relate any two design points by finding a minimal set of atomic changes transforming design point A into design point B. While the number of changes can be used to measure similarity, the resulting set of changes can also be used to provide some much-needed structure in the complex mapping design space. For this purpose we define a sequence of intermediate design points that is the result of applying one such minimal set of atomic changes to A (in unspecified order). The sequence of intermediate design points represents a "path" from A to B:

$$A, A^1, A^2, \ldots, A^{n-1}, B \quad \text{where } n = \text{distance}(A, B)$$

The intermediate design points A^i will share a varying amount of characteristics from both A and B, since every application of an atomic change helps to transform A into B. We note that exchanging properties from parent chromosomes is the main purpose of the GA crossover operator. Therefore, we propose to use the constructed path as the basis for a new type of crossover operator: the distance-path crossover.

Next we show how such a path can be constructed and how a crossover operator can be implemented. We already determined that the distance metric problem is equivalent to a group assignment problem: and indeed both distance algorithms from Sections 7.6.2 and 7.6.4 can provide us with both the resulting distance as well as (one of) the minimal group assignments. The minimal group assignments can be used to build the path between A and B by iteratively applying the re-assignment function (Section 7.6.2) for each group.

For example, the mappings A and B:

$$A = [0,1,1,0,2,3]$$
$$B = [0,1,2,2,1,2]$$

have a distance 3 with matching groups: $(1^A,1^B)$, $(2^A,4^B)$ and $(3^A,2^B)$ (using the notation from Section 7.6.3). We rewrite A into B by applying the reassignment function to each of the group pairs in order:

	[0,1,1,0,2,3]=A
$(1^A,1^B)$: on every i where $B[i]=1$ assign 1 to $A[i]$	
$i=4$, result:	[0,1,1,0,<u>1</u>,3]=A'
$(2^A,4^B)$: on every i where $B[i]=4$ assign 2 to $A[i]$	
no such i	
$(3^A,2^B)$: on every i where $B[i]=2$ assign 3 to $A[i]$	
$i=2$, result:	[0,1,<u>3</u>,0,1,3]=A''
$i=3$, result:	[0,1,3,<u>3</u>,1,3]=A'''
	[0,1,2,2,1,2]=B

The path from A to B is given in the right column, note that A''' is symmetrical to B. Paths will be longer when A and B are less similar and when the problem space (and thus the chromosomes) are longer.

We propose that the new crossover operator creates offspring by simply selecting two random elements from the path between parents A and B. Special provisions can easily be made if one objects against the fact that the children may be the same as each other or as one of their parents. Finally we note that the offspring created by this crossover mechanism *only* mixes genetic material from the parents, and that no new or random material is inserted. In other words, properties from an element C that is not on the path will not occur in the offspring, since the distance metric is guaranteed to find the *minimal* path between A and B.

7.8.3 Combination of approaches

The baseform and crossover techniques that were presented in the previous sections can also be used in combination. In that case we first perform the baseform method after the modified crossover operator. In the example of the previous section this would mean that if A'' was the result of the crossover, then application of the baseform function would convert it to $[0,1,2,0,1,2]$. We note that it is possible to delay the baseform function until after the mutation operator, so that the baseform conversion needs to be performed only once.

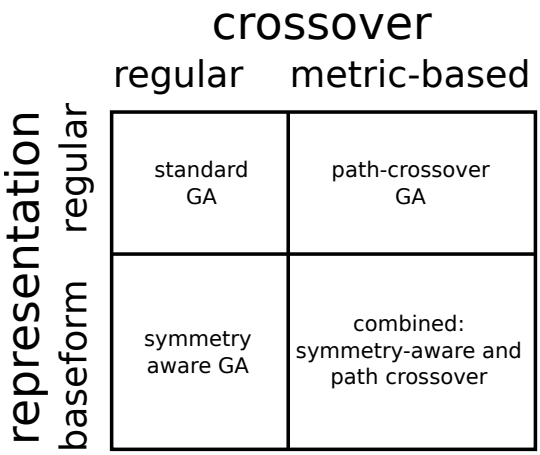


Figure 7.16: Summary of different GA methods

Finally we summarize in Figure 7.16 that the proposed new approaches give rise to 3 new GA methods. These will be evaluated in the next section.

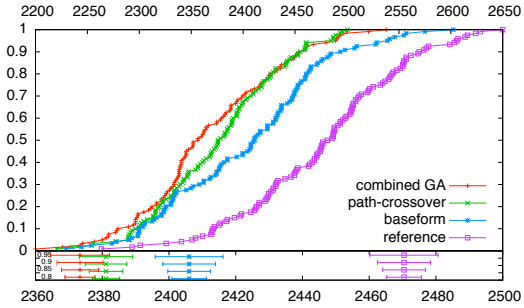
7.8.4 Experiments

A set of experiments has been performed to determine the impact of the two approaches that were described previously. We look both at the impact of the crossover and baseform extensions separately as well in the combined approach. As the basis for comparison we use a GA with tournament selection and uniform crossover. To challenge the search algorithm, we use a different, and (compared to the previous experiments) much larger stochastic application model consisting of 20 processes. They are mapped onto a homogeneous 8-processor architecture using a single shared bus for communication. Note that this significantly increases the design space to approximately $3 \cdot 10^{13}$ unique design points.

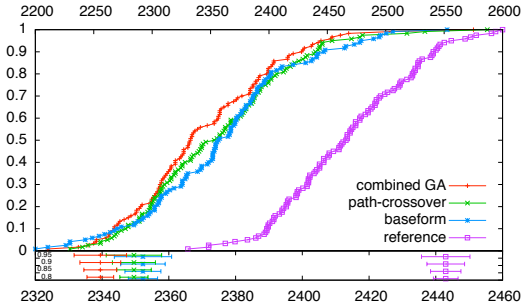
Part 1

This results in a series of P-Q plots (Figure 7.17 and Figure 7.18) that show the results of the four types of GA: regular (reference) GA, GA with baseform, GA with new crossover and finally a GA with baseform and new crossover. In the following we will refer to these simply as: *reference*, *baseform*, *crossover* and *combined* GA. In the first experiment (Figure 7.17), we test three different mutation rates (top-to-bottom: 0.15, 0.1 and 0.05) as well as for two different population sizes (left: 40, right: 80). In all other aspects, the GAs use the same parameters (e.g., crossover rate 0.9) and they all run for 30 generations.

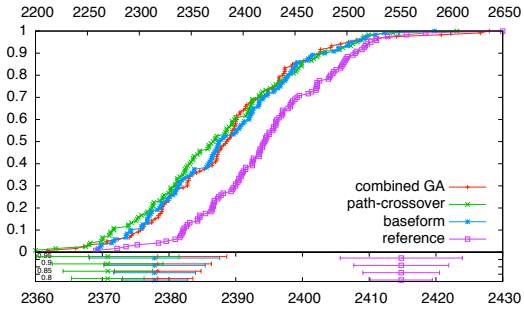
Note that in contrast to previous P-Q plots, the quality-axis of the top figure is given not as a relative percentile, but as absolute values in the objective domain (simulated cycles). This change is necessary as we can not determine the true optimum anymore. The design space (20 processes onto 8 processors) is now sufficiently large that exhaustive search becomes impractical. The P-Q graphs are now effectively cumulative distribution functions of



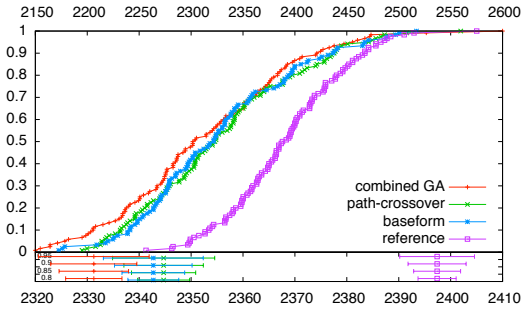
(a) mutation rate=0.15, population size 40



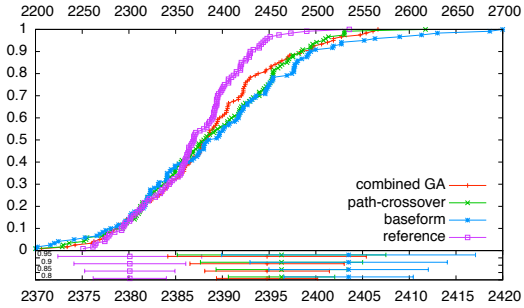
(b) mutation rate=0.15, population size 80



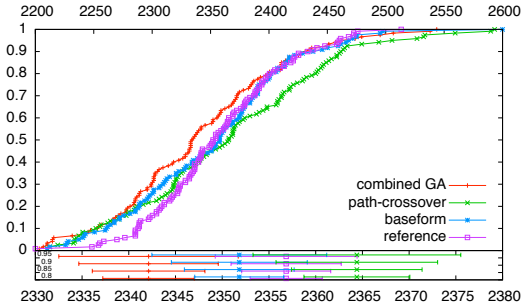
(c) mutation rate=0.10, population size 40



(d) mutation rate=0.10, population size 80



(e) mutation rate=0.05, population size 40



(f) mutation rate=0.05, population size 80

Figure 7.17: P-Q plots for GAs with combined extensions

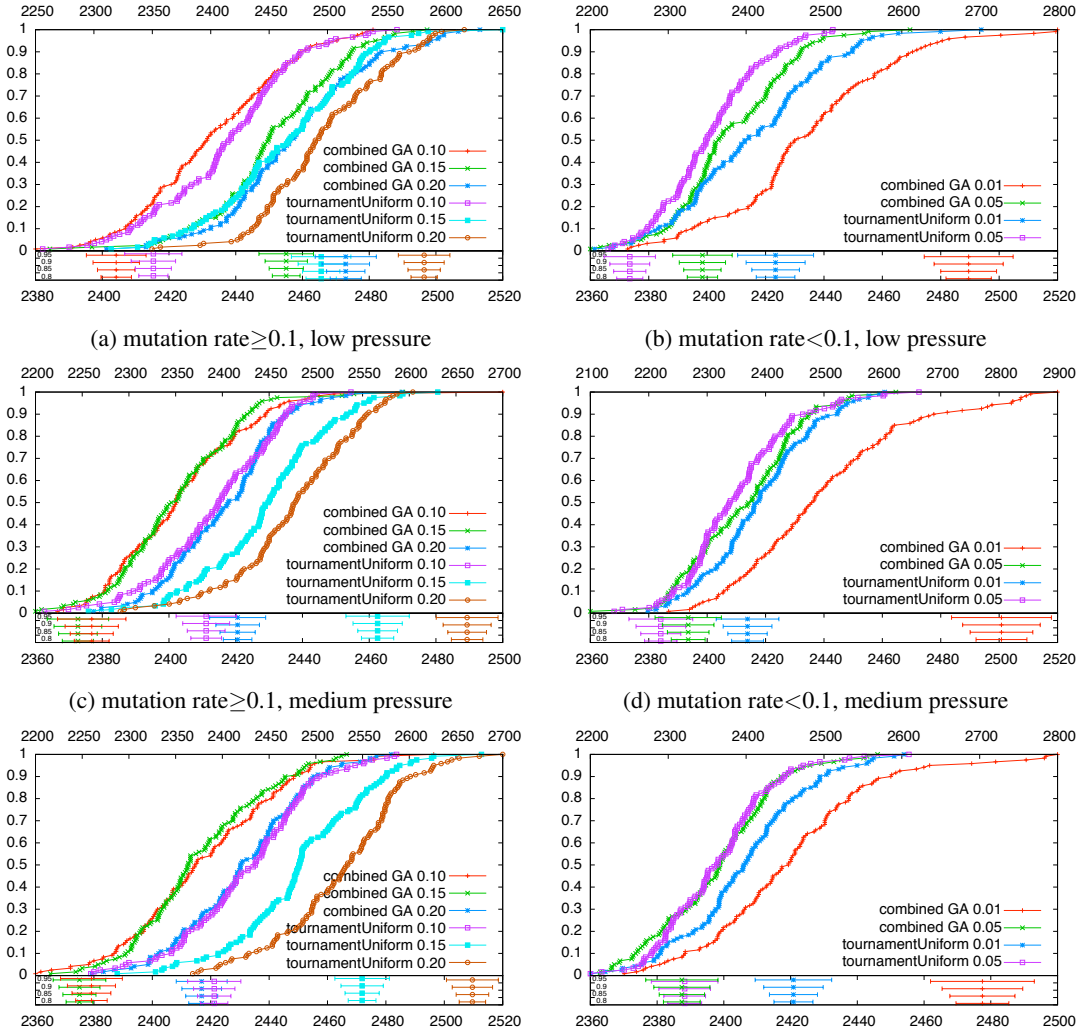
the experimental data (known as the empirical CDF). They can be interpreted the same as the previous P-Q plots: dominating lines (towards the top-left) show a better performing GA. The confidence intervals at the bottom of each graph are the same as before. Also note that an initial exploration (not reported here) showed that for the new design problem, tournament selection with uniform crossover works best. Therefore, in the following experiments we use this as the reference GA (denoted with the label `tournamentUniform` or simply `reference`).

The first subplot (Figure 7.17a) shows that the extended GAs perform better than the reference GA. The difference between the average result (as indicated by the center of the confidence interval) of reference GA and the best performing GA (combined method) is quite large: 2470 vs. 2373 in terms of absolute performance. Moreover, the confidence intervals between the reference and combined GA are clearly disjoint, confirming the reliability of our observation. As we do not have access to the global optimum of this design space, it is not possible to quantify exactly the meaning of a reduction of approximately 100 units (cycles) in objective space. However, we can see that the best result that we found in all experiments is 2331 when using population size 80 (Figure 7.17d), which is approximately 40 units lower than our best average result in Figure 7.17a. So, as a very rough, intuitive comparison we say that a reduction of 100 is 2.5 times the improvement of doubling the population size. This is significant, considering that increasing the population size increases the number of evaluations and thereby the search cost. In relative terms we can compare the reference and the combined GA in Figure 7.17a: the probability of finding results within the range 2300-2475 differs around a factor 2 or 3.

In the same figure (7.17a) we see that the results of using only the baseform or only the path-crossover are also much better than the reference GA. The baseform-only GA seems to perform slightly worse than using only the path-crossover for mutation rate 0.15 (Figures 7.17a and 7.17b). The confidence interval plot of 7.17a clearly shows the order in performance (from high to low): the combined method, path-crossover, baseform and lastly the reference GA. The experiment with a larger population (Figure 7.17b) shows the same ordering, although the difference between path-crossover and baseform are less pronounced.

In the experiments with lower mutation rate (7.17c to 7.17f), the difference in performance between all GA methods becomes smaller. For mutation rate 0.10 (Figure 7.17c and 7.17d), the performance of the extended GA methods start to become very similar (see the confidence plots), but there is still a significant difference with the reference GA. Only for the lowest mutation rate (Figure 7.17e and 7.17f) does the reference GA catch up. For population size 40 the reference GA even performs a little better than the other methods, though not so for the larger population size of 80 (Figure 7.17f).

So we tentatively conclude that in almost all cases the extended GAs perform better, or at worst similar, to the reference GA. However, the impact seems to be inversely proportional to the mutation rate: a higher mutation rate means a larger performance difference. Therefore, a designer can use the baseform and crossover extensions with confidence, as long as he chooses a sufficiently high mutation rate (≥ 0.10). Moreover, the baseform, path-crossover and combined methods introduce only a small bit of computational overhead to



(a) mutation rate≥0.1, low pressure

(b) mutation rate<0.1, low pressure

(c) mutation rate≥0.1, medium pressure

(d) mutation rate<0.1, medium pressure

(e) mutation rate≥0.1, high pressure

(f) mutation rate<0.1, high pressure

Figure 7.18: P-Q plots for metric-based GA vs. regular GA

the standard GA: specifically they are much better scalable than the methods proposed in Section 7.7. From these experiments we conclude that using a sufficiently high mutation rate, the proposed GA extensions perform the same, if not better in all cases. For higher mutation rates or bigger populations, the combined method that combines the baseform and path-crossover, performs best.

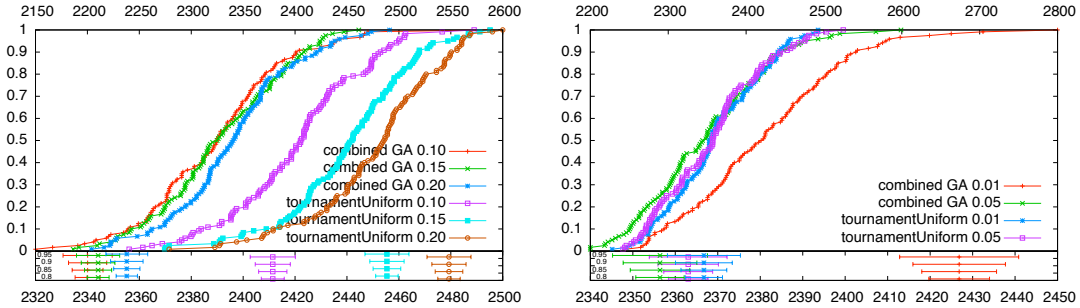
Part 2

In a follow-up experiment we check the impact of selective pressure on the performance of the GAs. In Figure 7.18 the PQ-plots are shown for a series of experiments where we vary the pressure (top-to-bottom: low, medium and high) and different mutation rates (left:0.10, 0.15 and 0.20, right:0.01 and 0.05). Population size is 40 in all experiments. Since the combined method performed consistently well in the last experiment, each plot now only compares the combined method with the reference GA. If we compare the graphs on the left with those on the right, we can immediately see that for all different selective pressure rates the combined GA methods perform significantly worse with a low mutation rate: mutation rate 0.01 has the worst performance. Mutation rate 0.05 performs better, but still worse for low pressure (Fig. 7.18b). In the case of medium and high pressure and mutation rate 0.05 (Fig. 7.18d and 7.18f), the combined and reference methods perform practically the same.

The situation in the left figures (with higher mutation rates) is very different; now the reference GA with mutation rate 0.10 is always one of the best performing methods. There is a noticeable trend related to the pressure indicating that a higher pressure increases the difference between the combined method and the reference GA. In all of the three left-hand graphs with mutation rates ≥ 0.10, the best performing *reference* GA is the one with mutation rate 0.10. In case of low pressure (Figure 7.18a), the combined method performs only slightly better than the reference GA with mutation rate 0.10. However, when we increase the pressure, then the distance between the two becomes much larger. For example, in case of medium pressure (Figure 7.18c), the combined method GA with mutation rate 0.15 has an average result of 2376 and the reference GA 2410. And with high pressure (Figure 7.18e), the combined method performs approximately the same, but the reference GA performs even worse, thus increasing the difference. A further observation is that when we increase pressure, the difference between the three extended GAs becomes smaller: for medium and high pressure, the results for mutation rate 0.10 and 0.15 are overlapping and the result for mutation rate 0.20 is closing in. The difference between the three reference GAs, however, seems to be constant from low to high pressure.

We conclude from these experiments that the performance of the combined GA works best for higher pressure and a mutation rate of 0.10 or 0.15. Where the combined GA method seems to benefit from higher pressure and mutation rates, the opposite is true for the reference GA. In fact, the best result with the reference GA is obtained with mutation rate 0.05 and low pressure (Figure 7.18b). This is in fact the only time that the standard GA is able to obtain a better average result (for the same pressure) than the combined GA: an average value of 2372 (Fig. 7.18b versus 2404 for the combined GA (Fig. 7.18a). However, for the medium and high pressure cases, the combined GA method always results in a better

average result, for both mutation rate 0.10 and 0.15. These mutation rates are consistent with the previous set of experiments that also identified 0.10 and 0.15 to be much better mutation rates for the combined GA than a mutation rate < 0.10. In the experiments of Part 1, we also saw that the extended GA types performed better compared to the reference GA when the population size was bigger. To verify that this is a consistent trend, we re-do a sample of the experiments (the ones shown in Figure 7.18c and 7.18d) for a population size of 80. The results in Figure 7.19 immediately show the increasing difference between the combined GAs and the reference GAs. In Figure 7.19a, the combined GAs are clustered to the left, and the confidence intervals show a large gap between the results of even the worst combined GA (with mutation rate 0.20) and the best reference GA (mutation rate 0.10). In Figure 7.19b we see the results for lower mutation rates and as we saw before, only the very low mutation rate of 0.01 performs badly for the combined GA. The combined method with mutation rate 0.05 (7.19b) already performs similarly to the reference GA (their confidence intervals overlapping). Moreover, even the best performing reference GA (mutation rate 0.05) only gets an average result of 2363 (7.19b), whereas all of the combined GAs in Fig. 7.19a obtain a better average result. This shows that the combined GA performs the same or better than the reference GA for a range of mutation rates 0.05-0.20. In comparison, the reference GA is only working well in the range 0.01-0.05 and even then, its results do not reach the optima that the combined method GA finds.



(a) mutation rate$\geq$0.1, medium pressure

(b) mutation rate$<$0.1, medium pressure

Figure 7.19: P-Q plots for metric-based GA vs. regular GA: population size 80

7.9 Conclusion

In this chapter we have investigated a DSE approach using genetic algorithms as the basis for automatically traversing the design space. A metric was introduced that can quantify the similarity between design points. Originally the metric was intended as a method to control population diversity and to apply it to the GA in order to improve GA performance. In Section 7.7 we have shown that it is indeed possible to increase population diversity by

applying a few simple, distance metric-based methods. However, these methods proved unable to actually increase GA performance. We speculate that they do not work for the reason that by replacing individuals in the GA population, they interfere too abruptly with the natural convergence process of the GA.

Therefore, in Section 7.8, we introduced two new metric-based extensions to the GA that are scalable and fit more naturally to the GA process. The combination of the two new methods results in a third, *combined* method, which in the experiments often showed the best performance. In the presented experimental results, the GAs with the proposed extensions perform at least as well, but often better than the reference GA. Important is the finding that we could identify a clear trend to show for which parameters the extended GA methods performed better. Once more of such trends are identified and verified, a system designer can more accurately choose a search method to fit his design problem. In particular we showed that the extended GA methods benefit from high mutation and high selective pressure. We hypothesize that the higher mutation rate keeps population diversity high (as we saw in the experiments earlier in this chapter), while the high selective pressure improves convergence to the optimum result, but more research is required to verify this. Also, we observed that the extended GA methods work better for larger population sizes. We consider this to be a desirable property, since for more complex design spaces, population sizes are usually increased to exploit more parallel search within the GA. Finally, we observed that the extended GA methods seem to be effective for a wider range of GA parameters than the regular, reference GA. The former only seemed to perform well for low mutation rates and low pressure, whereas the extended methods performed better in all other situations. We are hopeful that such knowledge can reduce the effort that is required to find the right combination of GA parameters, but more research in this area is required.

7.10 Future work

There are many interesting directions that could be explored as a logical follow-up to the contributions and case studies presented in this chapter and in general for the field of automated DSE for system-level design. First of all, it would be worth to see whether the proposed distance metric can be useful towards exploration of a wider range of system models and parameters. For example, the distance metric could be extended to heterogeneous systems by adding extra weight to the metric if a task is mapped onto a different type of processor. This metric extension would work the same for homogeneous processor sub-groups, but increases the distance value for each task mapped onto different types of resources (even if –according to the task partitioning– it is in the same task-group). Also, we intend to investigate the performance of the distance metric approach when multiple objectives are taken into account, for example considering power and cost in addition to system performance. Future experiments with the mapping distance metric could scale to even larger design spaces, e.g., mapping application workloads with hundreds of processors. In order to reduce the overhead of the simulator as the fitness function in the GA, we could also look for ways to estimate the performance based on previously evaluated design points and then in the GA we

inter leaf simulation-based evaluation with estimation. This would introduce the possibility to trade-off a lower DSE accuracy for higher DSE method execution times.

We could also explore whether a distance-like metric could be useful for design space dimensions that are not directly mapping-related. Consider for example memory or buffer size of a particular system component for which a range of (discrete integer) values need to be explored. The memory size parameter can easily be encoded in the gene, but then we can ask how to apply the distance metric to it. It may even be possible to apply the distance metric when the design point is not based on a meta-platform mapping at all, but rather for example the generator-approach as discussed in Section 3.4. In any case we recommend to keep in mind that new approaches may work in unexpected ways: in our case the distance metric was originally developed to manage diversity, but turned out to perform better as a construct to build a new type of crossover operator.

Also, recent advances in the general field of evolutionary computation may be very suitable for application in our DSE domain. We specifically note the class of niching genetic algorithms that recognizes the problem of having multiple optima of equal interest that reside in different segments of the design space. Niching GAs attempt to converge the GA population simultaneously towards multiple optima, instead of just one as with regular GAs. Interestingly, niching GAs require a distance metric to be defined on the parameter search space. The distance metric that was defined in this chapter for measuring similarity between mapping specifications could be suitable for implementing niching GAs, but further research is required.

As we have seen in our work, the GA method (although generally efficient) suffers from the problem of having itself quite a lot of parameters that effect its efficiency in complex ways. The selection of parameters is often so complex that it sometimes seems that GAs only replace the original problem by the problem of finding the right GA parameters. An interesting experiment would be to optimize the choice of parameters for a GA (which is solving the original problem) by a means of another GA. Such a recursive use of GA optimization would however be highly inefficient for DSE case studies where the computational bottleneck is by caused by the fitness function (for example if fitness is determined by a (system-level) simulator). However, it could result in new knowledge about GA parameter influence. Finally, it could be worth to study methods for automatically selecting the best GA parameters (such as was done for simulated annealing methods in [77]).

As a final point we emphasize the problem of the large diversity of evaluation and comparison methods that are currently being used in the context of research in the area of DSE for system-level design. There are likely many valuable contributions that improve DSE in one way or another, but without a common framework or methodology, their individual merits are only applied to a limited sub-domain of DSE problems. Moreover, this fragmentation in the research field may well hinder future advances, as it is not a big stretch of the imagination to expect that break-through approaches will come from the combination and evolution of existing methods. A more collectively orchestrated approach in the research towards DSE search problems may also provide a solution here. At the core could be a standardized benchmark of industrially-relevant real and synthetic design problems as an analog

to application benchmarks that have a longstanding tradition to provide a common performance evaluation platform for various computer architectures. It could be worth investing in the generation of large, pre-computed design spaces (relating parameter to objective values), so that DSE methods can be tested without the computational overhead of evaluating single design points (by simulation or otherwise).

Chapter 8

Case studies

8.1 Introduction

In this chapter we present two case studies that show Sesame and Daedalus in action. The first case study considers a crossbar-based MPSoC platform running a Motion-JPEG application. Sesame is deployed to evaluate a wide range of design candidates, some of which are selected for synthesis resulting in a working prototype on FPGA. The implemented prototype platform instances are subsequently used to validate Sesame's model accuracy. In the second case study, an industrially relevant tiled architecture is presented that performs JPEG compression. It illustrates how the Daedalus/Sesame combination can be effectively used to design large MPSoC platforms.

8.2 Case study 1: Exploration and validation

Here we present a case study in which we applied Daedalus to explore different implementation options for a Motion-JPEG (M-JPEG) encoder application mapped onto a heterogeneous MP-SoC architecture. The case study illustrates Daedalus' design steps and demonstrates its potentials to quickly experiment with different MP-SoC architecture designs during the very early stages of design.

The KPN specification of the M-JPEG application was derived from sequential C code using the KPNgen tool as described in Section 2.3. A small manual modification (taking no longer than 30 minutes) to the original M-JPEG code was necessary to meet the KPNgen input requirements. The resulting Kahn application specification consists of 6 processes, as shown in the top part of Figure 8.1. Generating the KPN specification is a one-time effort since the same specification is used for all subsequent implementation and exploration steps.

To study target MP-SoC architecture instances for the M-JPEG application, we selected a crossbar-based MP-SoC platform with up to 4 processors (MicroBlaze or PowerPC) and distributed memory. At the bottom part of Figure 8.1, a 4-processor instance of this platform is depicted. We modeled this platform architecture with the Sesame framework. The pro-

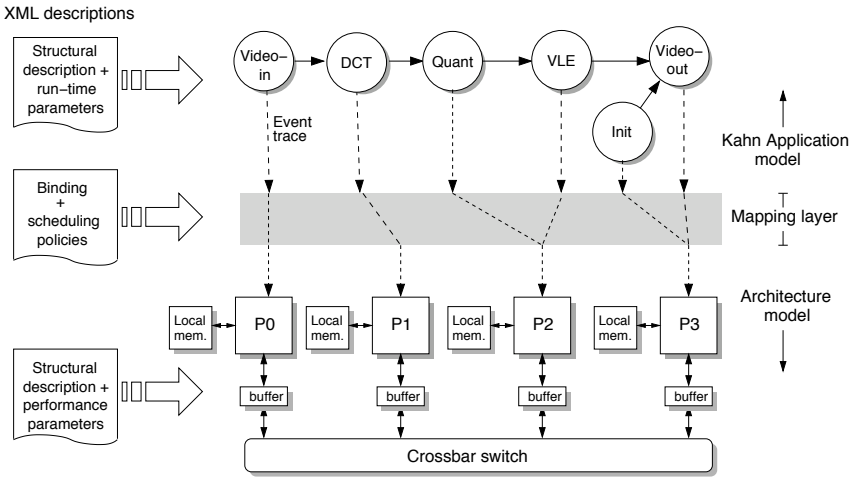


Figure 8.1: Sesame's layered infrastructure.

cessor, memory and interconnect components in our architecture model were taken directly from Daedalus' high-level model component library. Only the performance parameters specific to the selected platform architecture needed to be specified, such as the latencies for computational actions, the latencies for setting up and communicating over the crossbar, and so on. We determined the values of these performance parameters by a combination of measurements on an ISS simulator (for the computational latencies on the MicroBlaze and PowerPC processors) and on the actual hardware itself. Note that this needs to be done only once for each application, since the values can be reused throughout the exploration process. More information about the calibration of our architectural performance models can be found in Chapter 4. Moreover, the mapping layer in our system-level model is configured such that it models the static scheduling scheme as facilitated by the ESPAM framework (see Section 2.5). To this end, for shared architecture components, the mapping layer dynamically groups trace events that originate from the same Kahn process and interleaves these event groups in the same manner as would be the result of ESPAM's static scheduling.

In our design space exploration experiments, we selected three degrees of freedom, namely the number of processors in the platform (1 to 4), the type of processors (MicroBlaze or PowerPC) and the mapping of application processes onto the processors. For the sake of simplicity, the network configuration (crossbar switch) as well as the buffer/memory sizes remained unaltered (although these could also have been included in the exploration). For this particular case study, we were able to exhaustively explore the resulting design space – consisting of 10,148 design points – using system-level simulation, where the M-JPEG application was executed on 8 consecutive 128x128 resolution frames for each design point. As can be seen in Table 8.2, this design space sweep took 2.5 hours, demonstrating Sesame's efficiency. Figure 8.2 shows for three platform instances the relation between mappings and system performance, where we sorted the different mapping instances on performance. It clearly illustrates the importance of finding a good mapping since non-optimal mappings on

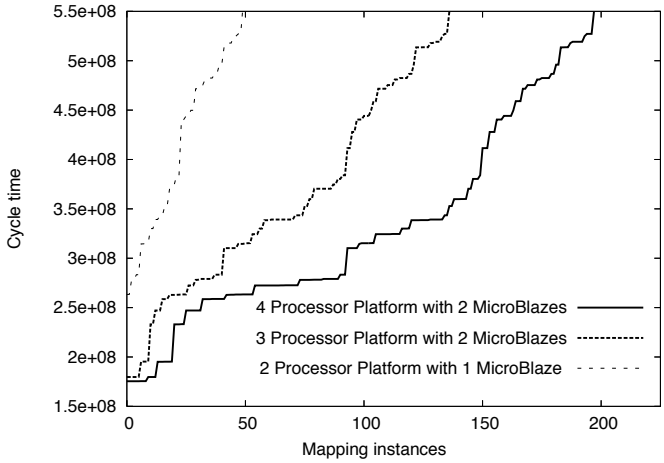


Figure 8.2: Performance of mappings on different platforms.

larger MP-SoC platforms may perform worse than a good mapping on smaller MP-SoCs.

To validate our DSE experiments, we selected a number of design points with random application-to-architecture mappings and synthesized and prototyped them using ESPAM. The results of these validation experiments are shown in Figure 8.3. Note that a synthesized platform can only contain up to two PowerPCs due to the Xilinx Virtex-II-Pro FPGA chip (xc2vp20) that is used for prototyping. For the chosen design points, our abstract system-level simulations adequately show the correct performance trends, with an average error of 12% and worst-case error of 19%. The inaccuracies in terms of absolute cycle numbers are mainly caused by the modeling of the PowerPC processors. This because these processors are connected to the crossbar using a bus that is also used for access to the processor's local data and instruction memory. Since we do not explicitly model (contention on) this bus, our abstract PowerPC performance model is too optimistic.

Naturally, we also used our exploration results to find the best mapping for each platform instance. The graph on the left-hand side in Figure 8.4 shows the best design points found by our DSE for purely MicroBlaze based platforms, together with the real measurements from the prototypes of these design points. Clearly, our abstract performance models quite accurately reflect the performance behavior of the actual systems. When introducing one PowerPC in the platform, as depicted on the right-hand side in Figure 8.4, the absolute errors become larger (due to the inaccuracy of our current high-level PowerPC model, as explained above) but the correct performance trend is still shown. For MP-SoCs with more than two processors, this inaccuracy seems to be amortized again.

To give an impression of overall resource utilization of the multiprocessor systems that are generated by Daedalus' ESPAM tool, Table 8.1 shows the utilization of FPGA resources for an MP-SoC containing 4 MicroBlazes. Here, we recognize FPGA resource utilization for the entire MP-SoC, as well as specific utilization results for the Communication Controllers (CCs) that glue together the processors with the interconnect and the crossbar interconnect

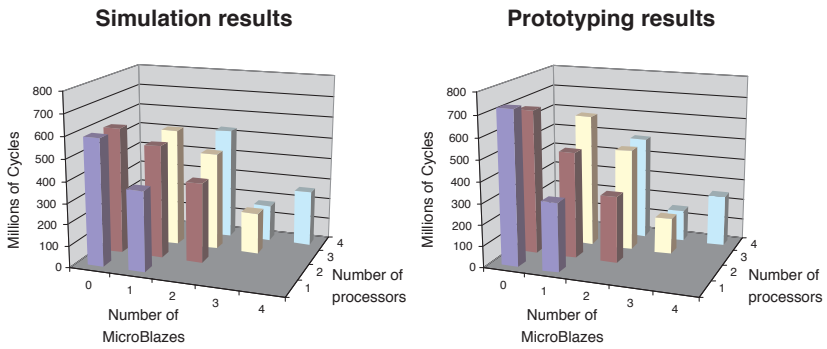


Figure 8.3: Validation experiment: simulation results (left) and actual measurements (right).

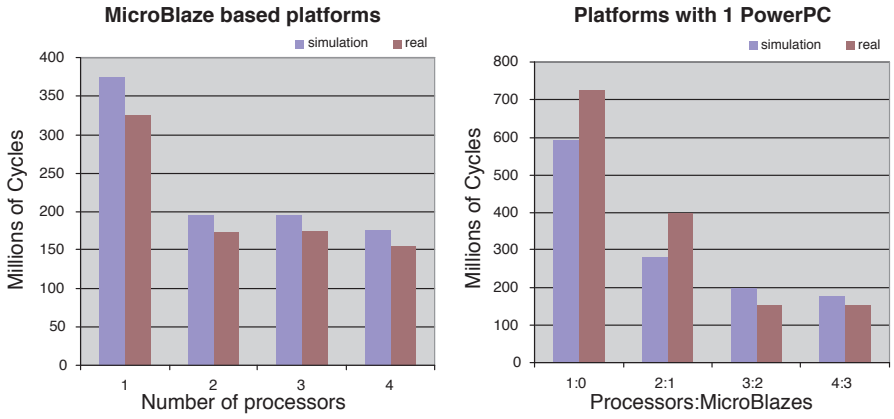


Figure 8.4: Best mapping results for MicroBlaze based platforms (left) and platforms with one PowerPC (right).

itself. As can be seen, the MP-SoC only takes about 40% of the FPGA slices, of which about 5% is used for the communication components. We note that the high BRAM usage reported in the last column is due to the complexity of the M-JPEG application, which causes each processor's local program and data memory to be quite large. We emphasize that the high BRAM usage is not caused by the implementation of the communication memories (the FIFO memories to which the Kahn channels are mapped), since they only use a maximum of 9 BRAMs.

Table 8.1: Resource Utilization for MicroBlaze based system.

	#Slices	#4-Input LUT	#Flip-Flops	#BRAMs
4 proc. system	3653 (39%)	4748 (25%)	2357 (12%)	85 (60%)
4 CCs	288 (2%)	468 (2%)	116 (1%)	—
4-Port crossbar	397 (3%)	587 (3%)	56 (1%)	—

Table 8.2 shows a breakdown of the execution time for each step in Daedalus' design flow in the case that one selected MP-SoC platform instance (a 4-processor MicroBlaze based architecture) is synthesized and implemented. The processing times were measured on a 1.8 GHz Pentium 4. Note that some of the steps only need to be performed once (such as the KPN derivation), after which, for example, the synthesis and physical implementation stages can be iterated several times to prototype different platform instances. The results from Table 8.2 demonstrate that the entire design trajectory, from sequential application specification to MP-SoC prototype executing the parallelized application on top of it, takes only a matter of hours. Evidently, this allows designers to quickly *prototype and assess* different platform instances and implementation choices during the very early design stages. Also noticeable is the fact that the system-level DSE component (Sesame) still requires a relatively high amount of manual effort. The manual effort listed in Table 8.2 is mainly due to the construction of the platform architecture model and the adaptation/construction of scripts that perform the automatic design space exploration. Not taken into account is the calibration of model components, which is a one-time effort for every application that is studied.

Table 8.2: Processing Times (hh:mm:ss).

Tool	KPN Derivation	Syst.level DSE	Syst.level to RTL conv.	Physical Impl.	Manual effort
KPNgen	00:00:22	–	–	–	00:30:00
Sesame	–	02:30:00	–	–	01:30:00
ESPAM	–	–	00:00:24	–	00:10:00
XPS tool	–	–	–	02:09:00	–

8.3 Case study 2: a tiled MPSoC architecture

We have initiated a project together with a Dutch SME called Chess B.V. (www.chess.nl), which involves the design of a still image compression system for very high resolution images. Chess B.V. is a company that provides image processing solutions for customers that build industrial process monitoring and medical appliances. With respect to this, the still image compression systems for different customers have to meet different performance and cost requirements. Chess needs tool support for very fast exploration and implementation of alternative systems (e.g., MP-SoCs realizing JPEG or JPEG2000 encoders) whereby trade-offs can be made between cost, design time, space, performance, etc. in order to offer its customers several solutions at different prices and let the customers select the most suitable ones. The Daedalus framework provides such tool support for MP-SoC design. Therefore, it is used in a project with Chess for design space exploration (DSE) at a high-level of abstraction by running system simulations and at implementation level by evaluating real system prototypes. In this section, we report on the project's initial findings and results obtained by deploying the Daedalus framework in the early design stage of JPEG-based image compression MP-SoCs.

Before describing our DSE experiments, we first would like to point out that the design space targeted by our implementation-level DSE is currently constrained by:

1) *The amount of the available memory.* In order to achieve high performance, in our MP-SoCs 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 MP-SoC 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 MP-SoC memory to be up to 288KB, being the amount of on-chip memory of this FPGA.

2) *The type of the processing components.* The MP-SoCs are built of components from our library. Our library is under development and currently contains two programmable processors: *PowerPC* 405 (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 MP-SoCs are limited to *MicroBlaze* processors and *DCT* HW IP cores only.

8.3.1 Simulation-level DSE

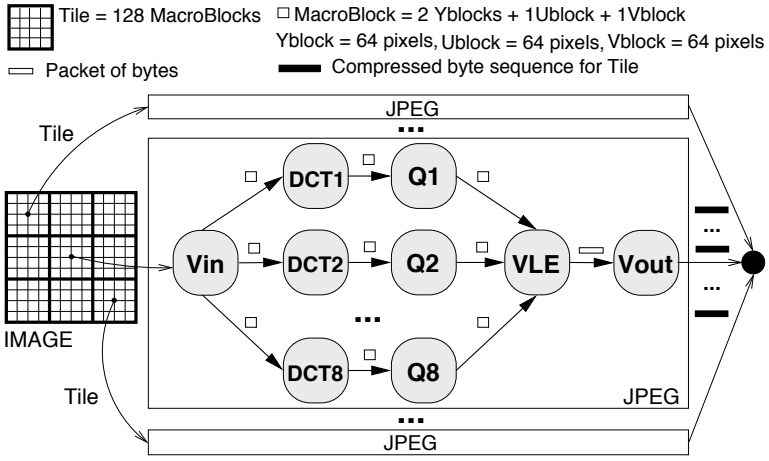


Figure 8.5: The JPEG application KPN

In our experiments, 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 8.5, which also shows the applied KPN application specification for a JPEG encoder.

The JPEG KPN for a single tile can again exploit task-level parallelism by pipelining tasks as well as data-parallelism by performing multiple *DCT*s and *Quantizations* in parallel. By deploying Sesame's efficient system-level simulations, we explored a substantial number of different implementations of a single JPEG encoder in the system, as represented by the KPN in Figure 8.5. To this end, we mapped the KPN to a variety of MP-SoC architectures,

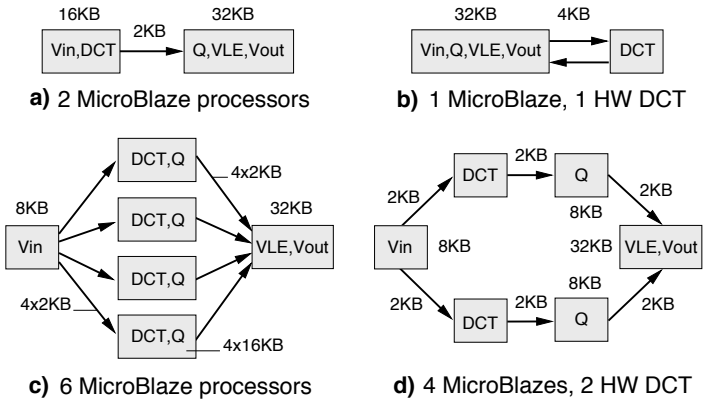


Figure 8.6: Alternative design instances to process one tile

ranging from a 1-processor system (all KPN processes mapped to a single processor) to a 19-processor system (every process in a KPN with 8 data-parallel streams mapped to a different processor). Moreover, in our Sesame-based exploration we also varied the type of processors in these MP-SoC platforms: KPN tasks can be executed on a *MicroBlaze*, while for the *DCT*, *Q*(uantization) and *VLE* tasks we also assessed dedicated HW IP implementations. Evidently, our simulation-level DSE also explores *'non-implementable' design instances*. That is, design instances that cannot be further explored at implementation level since it uses HW IP components that are not (yet) available in our library of RTL-level IP components. In Figure 8.6, four (implementable) example design instances are depicted.

Figure 8.7 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 the next section). Since the memory utilization of all design points in Figure 8.7 is below 288KB, they will all fit on the targeted FPGA memory-wise. But, as will be shown further on, the real system 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 8.7 can be classified as three types of design instances: 1) design instances that are 'implementable' (i.e., do not use the non-implemented HW IP components for the *Q* and *VLE* tasks) but are not part of the Pareto front, 2) implementable design instances that are part of the Pareto front, and 3) design instances that are non-implementable (i.e., contain HW IP components for the *Q* or *VLE* tasks). Moreover, the homogeneous design instances (i.e., the platforms only using

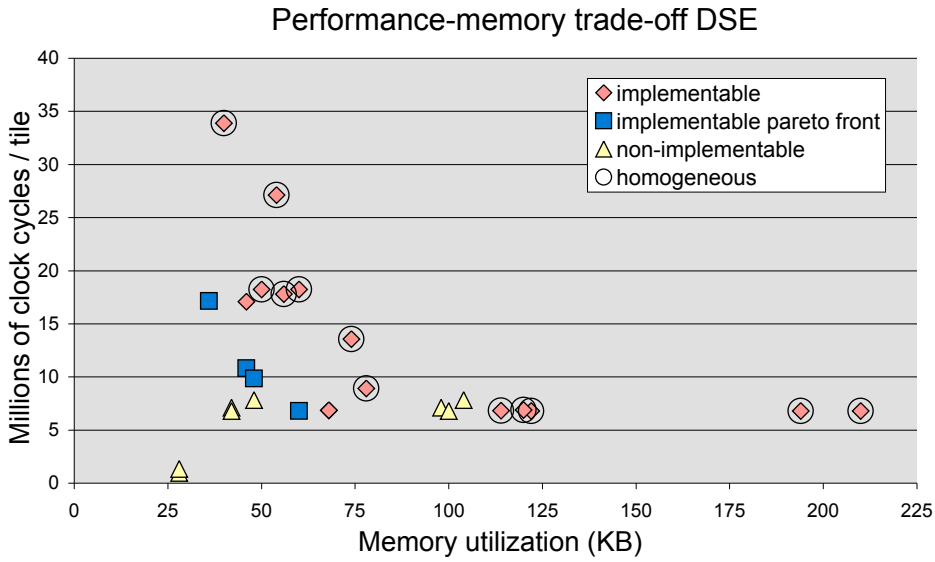


Figure 8.7: DSE for performance/memory utilization trade-offs

MicroBlaze processors) are tagged with circles.

A number of observations can be made from Figure 8.7. 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 8.6(b) and 8.6(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 8.7, 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 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. Here, we found that increasing data-parallelism beyond 4 parallel *DCT*-*Q* streams (see Figure 8.5) does not improve performance anymore as the *VLE* becomes the bottleneck.

As mentioned and as will be illustrated in the next section, the design points in Figure 8.7 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 Giga cycles to encode an image with a 1 Gigapixel resolution. For implementable solutions, the Pareto optimal design instances from Figure 8.7 are obvious candidate building blocks for the MP-SoC.

Figure 8.8 shows the estimated maximum performance – in terms of speed-up over a single JPEG encoder executed on one *MicroBlaze* – for different JPEG compression MP-SoCs realized with a combination of implementable design instances from Figure 8.7. The

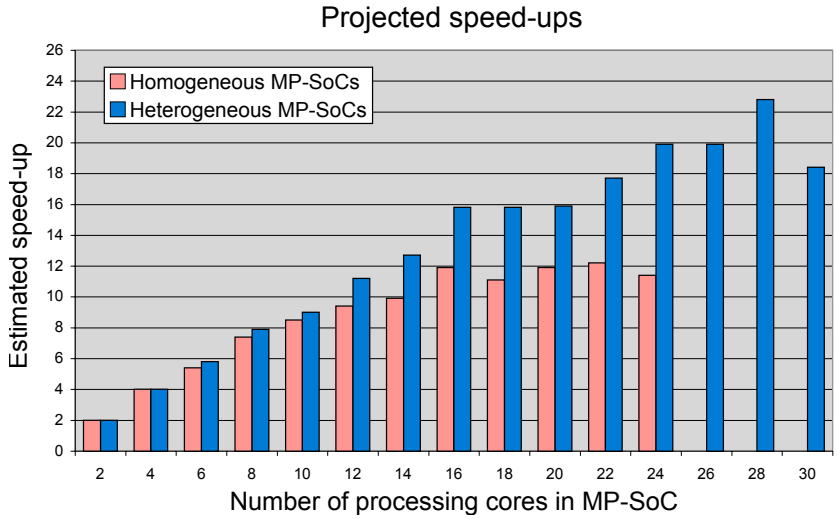


Figure 8.8: Estimated speed-ups.

x-axis indicates the number of processing cores (either *MicroBlaze* or HW IP) in the MP-SoC, and the y-axis shows the estimated speed-up for the optimal combination of design instances for a specific number of cores in the MP-SoC 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 *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 the next section, more examples of, sometimes hybrid, combinations of design instances will be discussed.

Essentially, Figure 8.8 provides a projection of the feasible system performance, given the constraints of the targeted FPGA. For homogeneous solutions, our simulations predict that a speed-up of around 10 to 12 is attainable. Our memory utilization model indicates that scaling the homogeneous system beyond 24 cores is not possible because of the memory constraints. For heterogeneous systems, on the other hand, our memory model indicates that the system can be scaled to 30 cores since the HW IP components only use a fraction of the memory used by a *MicroBlaze*. Here, our predictions show that a speed-up of around 20 to 22 is feasible. The results from our simulation-level DSE, as displayed in Figures 8.7 and 8.8, are used in the next section for steering the implementation-level DSE. These implementation-level experiments will also provide a validation of our simulation-based predictions.

8.3.2 Implementation-level DSE

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 promis-

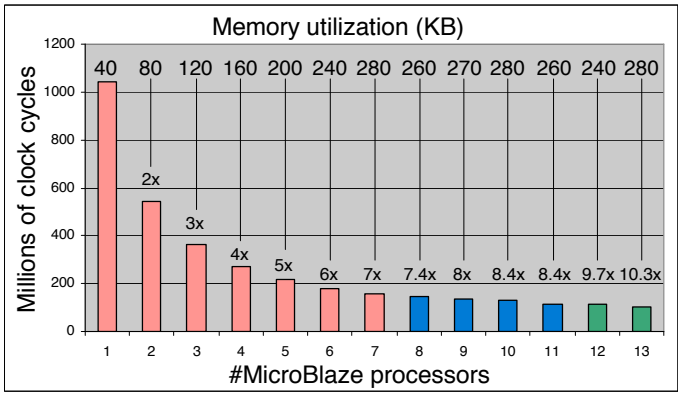


Figure 8.9: Performance results: homogeneous MP-SoCs.

ing design points. Thus, we perform 100% accurate exploration in the narrowed design space by generating real MP-SoC prototypes and we measure the actual performance/cost in order to select the optimal MP-SoC designs given a set of physical implementation constraints. Below, we present our initial implementation-level DSE results for MP-SoCs implemented on Xilinx FPGAs.

Due to the aforementioned implementation-level constraints, some of the best design points found by the simulation-level DSE (see Figure 8.7) 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 8.8. From them, we selected only the instances that have *efficiency* above 0.8, where

$$efficiency = \frac{speed - up}{number\ of\ cores}.$$

This selection resulted in implementations of homogeneous MP-SoCs with up to 13 *MicroBlaze* processors and heterogeneous MP-SoCs with up to 24 cores. Evidently, better performance is delivered by the heterogeneous systems, however, the homogeneous systems add more flexibility in choosing the right solution, e.g., performance/cost, for a particular customer.

The implementation results for the homogeneous MP-SoCs are depicted in Figure 8.9. The x-axis represents the number of *MicroBlaze* processors in an MP-SoC and the y-axis depicts the number of clock cycles (in millions) to compress one image consisting of 32 tiles. Above each bar, we indicated the achieved speed-up of the particular MP-SoC compared to a single *MicroBlaze* system (the leftmost bar). At the top of the figure, we present the amount of memory utilized by each MP-SoC.

As mentioned before, our JPEG encoding MP-SoCs 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

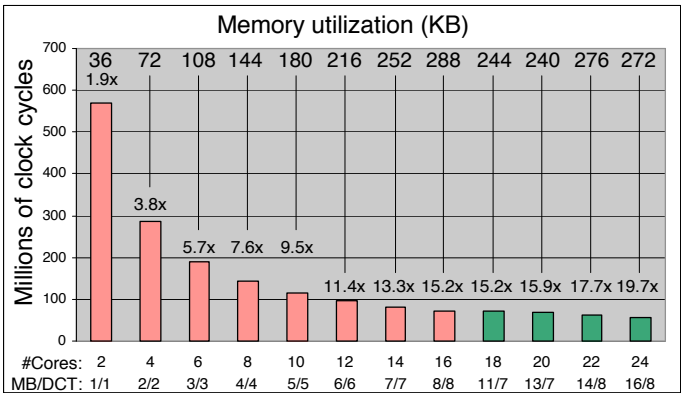


Figure 8.10: Performance results: heterogeneous MP-SoCs.

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 8.9) 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 8.6(a), where the *Vin* and all *DCT* processes (see Figure 8.5) 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 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 MP-SoCs 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 MP-SoC and 8.4x for an 11-processor MP-SoC, 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 8.6(c). This architecture requires 120KB of memory. The 13-processor MP-SoC 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 8.9. The achieved speed-up of 12- and 13-*MicroBlaze* systems is 9.7x and 10.3x respectively, compared to a 1-*MicroBlaze* system.

The implementation results for the heterogeneous MP-SoCs are depicted in Figure 8.10. The notation is the same as in Figure 8.9 with the only difference that the x-axis of Figure 8.10 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 MP-SoCs consisting of up to 24 cores. As a reference number to estimate the speed-up of each MP-SoC, we again used the number of clock cycles of the 1-*MicroBlaze* system (see the leftmost bar in Figure 8.9). We started with a 2-core system consisting of 1 *MicroBlaze* and 1 *DCT* IP. Its architecture is depicted in Figure 8.6(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 8.10), which actually is in line with Amdahl's law. Similarly to the experiments with the homogeneous systems, we continued with points that exploit data-level parallelism between the tiles, increasing the number of tiles processed in parallel. The 2-core system 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 8.6(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 8.10.

Like in the previous experiment, with the given constraints larger MP-SoCs can be implemented (and higher speed-ups can be achieved respectively) by exploiting data-level parallelism between the tiles and data- and task-level parallelism within the tiles. The most efficient heterogeneous MP-SoC architecture found by the simulation-level DSE to exploit data- and task-level parallelism within a tile is depicted in Figure 8.6(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 8.8 which actually are combinations of 2-cores per tile (2-*CPT*) and 6-cores per tile (6-*CPT*) architectures (see Figure 8.6(b) and Figure 8.6(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 3 2-*CPT* architectures and 2 tiles are processed by 2 6-*CPT* architectures. The speed-up of this MP-SoC is 15.2x. The 20-core system processes 4 tiles in parallel: 1 tile is processed by 1 2-*CPT* architecture and 3 tiles are processed by 3 6-*CPT* architectures. 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 MP-SoC consists of 14 *MicroBlazes* and 8 *DCT* IPs that process 5 tiles

in parallel: 2 tiles are processed by 2 2-*CPT* architectures and 3 tiles are processed by 3 6-*CPT* architectures. The 24-core MP-SoC, consisting of 16 *Microblazes* and 8 *DCT* IPs, processes 4 tiles in parallel utilizing 4 6-*CPT* architectures. The achieved speed-up by this system is 19.7x compared to a 1-*MicroBlaze* system.

8.3.3 Evaluation

All presented DSE experiments and the real implementation of 25 MP-SoCs on FPGA were performed in a short amount of time, 5 days in total, due to the highly automated Daedalus design flow. Around 70% of this time was taken by the low-level commercial synthesis and place-and-route FPGA tools. The obtained results show that the Daedalus high-level MP-SoC models are capable of accurately predicting the overall system performance, i.e., the performance error is around 5%. By exploiting the data- and task-level parallelism in the JPEG application, Daedalus can deliver scalable MP-SoC 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 MP-SoC system 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.

8.4 Conclusions

In this chapter we illustrated Daedalus' design steps and demonstrated its efficiency by means of two different case-studies. It was shown how the Daedalus framework bridges the so-called implementation gap between system-level platform specifications and the actual physical implementations of these platforms. Daedalus' integrated and highly-automated environment for system-level architectural exploration, system-level synthesis, programming and prototyping proves to be highly effective for the design of multimedia MPSoC platforms.

Chapter 9

Conclusions

In the introduction of this thesis, we sketched some of the problems that currently play a major role in the design, development and implementation process of modern embedded systems. Such systems are used for more and more complex tasks, supported by increasing on-chip resource availability and implementation technologies. While the design of these systems becomes increasingly complex, there is also increasing demand and pressure to design such systems in shorter time frames. This in turn requires new design methodologies and tools to support the design process and offer automatic transitions from high level system specifications down to prototype system implementations. The research presented in this thesis was done in the context of the Daedalus design flow presented in Chapter 2. Daedalus is a good example of a next-generation design methodology supported by the necessary tools for each step in the design process. It takes the designer from a sequential application to a fully implemented FPGA prototype MPSoC system in a matter of hours. The three major tools in Daedalus take care of application parallelization, design space exploration and system implementation. The underlying methodology relies heavily on a few important methodology principles that favor the use of high-level system specifications, component-based design and component reuse, separation of concerns and a high degree of design process automization.

The contributions of this thesis focus on the design space exploration phase of the design process. This is a crucial phase in the overall design process, since the initial models that are created in the early stages of the design process can support the designer at the later stages, where the cost of recovering from wrong design decisions is typically higher due to the more detailed level of specification at that stage. In order to provide this kind of early feedback to the designer, the abstract system-level models have to meet a few criteria: in particular, they have to be sufficiently accurate, fast and relatively easy to construct. These criteria are often non-orthogonal and trade-offs have to be found in order to support an efficient and successful design process. In the introduction chapter (Section 1.6), we made a parallel between these trade-offs and the design criteria of the embedded MPSoC systems themselves. Indeed, we sketched the criteria for system-level modeling in a system of perpendicular axis (repeated here in Figure 9.1) as an analogy of the more common depiction of multi-dimensional parameter or objective spaces for the properties of the system under

study (an abstract representation of the latter is shown in Figure 1.2). As was discussed in the introduction, we identified specific criteria for evaluating single design points (indicated by *accuracy*, *speed* and *effort*) and for the traversal of the design space (indicated by *convergence*, *confidence* and *effort*). Although this list is perhaps not complete, we feel it captures the most pressing requirements to help to achieve the goal of successfully traversing the design process. In the following we will summarize the various topics addressed in this thesis according to these criteria and consider how they relate and contribute towards a design space exploration methodology that is ideally suited to support the process of modern embedded system design.

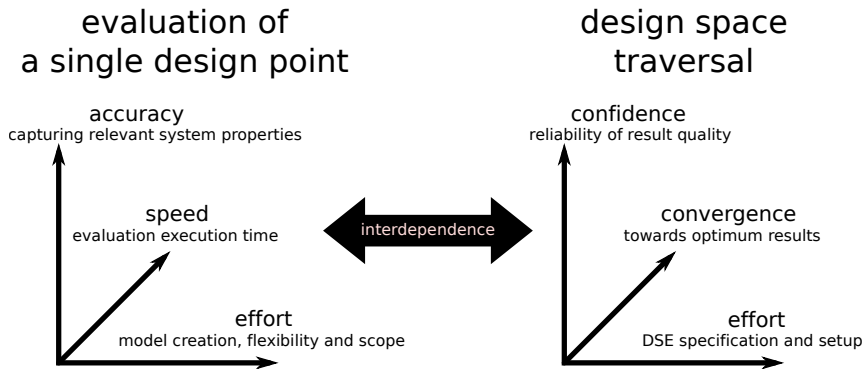


Figure 9.1: The two-part taxonomy of concerns for efficient DSE

9.1 Evaluation of a single design point

The ability to quickly and accurately model a design point are two of the basic requirements for any system-level modeling and simulation tool. Much work has already been done in this area in the past few decades. As we have shown, our Sesame tool (using non-refined model components) adheres to a higher level of modeling abstraction than most other tools. High abstraction levels can offer fast execution times for simulation, but at the same time makes it more challenging to maintain sufficient modeling accuracy. Sesame's models also allow for quickly setting up models that are sufficiently flexible to model a wide range of systems. These topics were addressed in Chapter 3, 4, 5, and 6. Below, we summarize their specific contributions in a bit more detail.

Accuracy

Model accuracy refers to the ability of the simulation model to capture in a realistic way the properties of a system in one or more evaluation criteria in objective space: e.g., performance, power usage or cost. As was explained in detail in Chapter 3, a Sesame model associates latencies to application events using a pre-defined (application dependent) latency table. In Chapter 4 we proposed two techniques that allow a designer to calibrate the model

by deriving values for the latency table using an accurate, lower level Instruction Set Simulator (ISS). We proposed both an on-line and an off-line technique where the ISS is used to accurately measure latencies of the various application events. The off-line technique measures (average) latencies of application events at design time, which are then used in the model's latency tables and can be reused for different architectural design options. Off-line calibrated models have fast execution times, but recalibration is needed when dynamic application behavior changes the number or type of trace instructions. On-line calibration combines Sesame's high-level models with one or more instances of the ISS at runtime: measurements on event latencies are now done during model execution. The resulting co-simulation (Sesame application model + ISS) now measures a latency for each individual application event, instead of using an average value. This improves modeling accuracy in the architecture model and has the additional benefit that no recalibration is required for dynamic behavior in the application.

Moreover, the signature-based method for model calibration is presented (Section 4.6), which has the additional benefit of doing calibration in an application independent way. On the basis of a training set of benchmark applications, the performance of a specific type of processor is captured by a *processor signature*. On the basis of this processor signature, an application event (represented by an *application signature*) can then be assigned a latency value. Possible sources of inaccuracy for this method lie in the regression model for deriving the processor signature and the selection of training benchmarks. However, a design space exploration case study showed that the calibrated model performs good relative comparison of design points (although the absolute estimation was overestimated).

Finally, the case study in Chapter 8 showed that our simulation models are accurate and useful for design space exploration by validating our simulation results with measurements on actual prototype platforms as generated by the Daedalus design flow.

Speed

The *speed* concern refers to the performance of the evaluation mechanism of a design point. As various experiments have shown, Sesame's abstract executable simulation models are fast: running a typical system-level simulation of a complete MPSoC system in a fraction of a second. We exploited this performance in Chapter 7 where we integrated the simulator directly as the fitness function of various automatic design space exploration methods based on genetic algorithms. Faster design point evaluation can benefit design space exploration, since a larger part of the design space can be explored in the same time. However, higher abstraction levels of modeling can reduce accuracy as was discussed before. What is interesting in this respect is that Sesame supports the designer in this trade-off by enabling gradual model refinement (Section 3.2.3). Moreover, the calibration techniques from Chapter 4 also support this trade-off. For example, compared to the on-line trace calibration technique, the off-line technique offers higher simulation speeds but possibly at the cost of lower accuracy. Performance of the on-line trace calibration method is constrained by the performance of multiple instances of the lower-level simulators, although we have shown that there are opportunities to mitigate this problem by parallelization. The signature-based

calibration technique resides in the middle of the on-line and off-line trace calibration methods trade-off and thus provides a viable alternative.

Effort

We consider design *effort* a highly important concern in the system design process as overall design time is sure to increase when large amounts of effort are needed to create a suitable model. Sesame's high level of modeling abstraction is one of the main reasons why models are easy create: Sesame's library of standard components is often sufficient to build an initial model in a few hours (which may subsequently be refined). Also contributing to effort reduction in Sesame (Chapter 3) is the separation of application and architecture models, the separation of model specification (in YML) and model behavior (component implementations), as well as its focus on modularity and component reuse. Moreover, effort is reduced by providing tool support as much as possible for routine modeling activities: an example of this is the automatically generated virtual layer, which also proved particularly useful for the stochastic multi-applications (Section 5.2) and reconfigurable models (Section 6). Additional tool support was provided for storing, querying and visualizing the results of design space explorations (6.2.3). The trace calibration and signature-based calibration methods can also be seen in light of effort-reduction, since they provide a standard way towards a more accurate or more refined (mixed-level) simulation model.

We explicitly include modeling scope of the tools in the *effort* concern, since a modeling tool that can not be used for wide range of different systems, will always result in an increased effort. This effort could consist of, for example, structural or superficial modification of the tool to make it fit the design problem, or an interoperability effort needed to make a combination of tools solve the single problem. To avoid such effort to become part of the design process, a modeling and simulation tool should ideally be capable to model a large variety of different types of systems. Extending the scope of Sesame's modeling capabilities is one of the main contributions of this thesis. In particular, Chapter 5 extended Sesame with components and mechanisms for multi-application modeling and Chapter 6 for modeling partially dynamic reconfigurable systems. Together with multi-application modeling, we also introduced stochastic workload modeling (Section 5.2), which can provide feedback and testing capabilities even before the final target application workload has been fully defined (thus possibly reducing design times). Moreover, Sections 5.4.1 and 5.4.2 discussed some possibilities to capture the effects of inter and intra-task level dynamic behavior in a Sesame model. Chapter 6, provided modeling components and techniques to allow Sesame to model partially dynamic reconfigurable systems. A clear separation was made between the components and techniques that are generic and thus reusable in other types of reconfigurable systems, and those more geared towards the Molen platform used for the case-study.

Finally, we would like to emphasize that many of the modeling methods and techniques in Chapters 4,5 and 6 are generically applicable, even outside the context of Sesame.

9.2 Traversing the design space

In this section we look at the different criteria (according to the right part of the taxonomy represented in Figure 9.1): the efficient traversal of the design space. During the traversal, of course, we benefit from an optimal trade-off between *speed*, *accuracy* and *effort* for evaluating a single design point (the left part of the taxonomy). For example, the ability to quickly, easily and accurately evaluate a single design point helps to accurately evaluate a larger set of design candidates, thus increasing our chance to find optimal results. For smaller design problems this may indeed be sufficient, but for large design spaces, the total number of design points that can be feasibly evaluated may still be a very small fraction of the entire design space. Therefore, even if the *speed*, *accuracy* and *effort* trade-off is ideal, we still have to make sure that each evaluation of a design point contributes as much as possible to the an optimal traversal of the design space. A crucial component towards this goal is the search algorithm that navigates the design space towards areas of interest by proposing which design points to evaluate next. Regardless of the specific type of search method that is used for traversal, we proposed in Chapter 1 that its success (measured as its contribution towards the ideal DSE scenario) depends on three major concerns: *confidence*, *convergence* and *effort*. We note that once again these concerns typically can not be considered in isolation, since they are highly interdependent, contradictory and sometimes overlapping. The state-of-the-art in design space exploration research can be summarized as finding a good trade-off between these concerns. We will shortly discuss each of these concerns below and show how they relate to this thesis (mainly Chapter 7).

Confidence

The *confidence* concern refers to a measure of how certain we are that the results from the design space traversal are indeed the optimal results that we are looking for. For example, if the traversal consists of evaluating every design point (exhaustive search), then we can claim with 100% confidence that we can find the optimal result. However, when exhaustive search is not possible, then we typically have to resort to heuristic search algorithms that can not guarantee finding an optimum (but return a best-effort result instead). In some cases we can relate *confidence* to *coverage*: the fraction of the design space that was evaluated. This was for example the case with the uniform random search that was used as a baseline comparison in the PQ-plots introduced in Section 7.4. However, more advanced search algorithms (such as the various GA variants in Chapter 7), not only showed much better performance than random search, but also clearly showed that higher coverage did not necessarily improve the results (Section 7.7).

There are no generic and absolute metrics available to quantify *confidence*, rather we postulate that it will remain (for the foreseeable future) one of the intangible factors that constitute the “the art of system design”. In practice, a designer will typically be satisfied when he can use (with minimal effort, see below) a search algorithm that has previously shown good results for similar design problems. *Confidence* is then replaced by an approximate, experience-based assumption that the algorithm will perform well in general.

However, complete surveys that match search algorithms to specific categories of design problems have not been established in the system-design field. Moreover, literature shows that improvement of search algorithms is always possible for specific problems, thus reducing confidence that a general solution can provide the best results. For example, it turns out that simple applications of domain-specific knowledge can quite easily improve search performance. An example in this thesis were the distance-based GA extensions which improved upon the standard GA (Section 7.8).

Finally, a note on some other facets of *confidence* that are more concrete, but on a smaller scale. Firstly, it is generally preferable that a particular search algorithm is stable: repetitions of the algorithm give results within a small, consistent interval (we checked this using confidence intervals in Chapter 7). In this way, the designer can be assured that even a single design space traversal with the algorithm gives the best possible result, thus shortening exploration time. Secondly, confidence can be improved if a search algorithm has few parameters that influence its performance, since these would have to be tried by the designer. In our case studies we saw that this was quite a problem with GAs, but we also saw that parameter sensitivity seemed to be reduced with the proposed GA extensions. More research in this area is required. Finally, confidence improves by reducing as much as possible any discrepancy between the evaluation mechanism (e.g. simulator) and the real system. Within the Daedalus design flow, there are sufficient calibration and validation opportunities, as we have shown in Chapter 8. Equally relevant is the model-to-model validation in Chapter 4 where we showed that signature-based models show largely the same relative performance trend (Figure 4.12) required for design space exploration.

Convergence

Convergence refers to the desirable property of a design space traversal method to locate the optimum with as few iterations or evaluations as possible. It is an analogous concern to *speed* in the left side of the taxonomy, but now referring to the performance of the entire design space traversal. We note that *convergence* needs to be optimized as a trade-off between two extremes. Convergence that is too low indicates that the search algorithm is unable to extract the relevant information from the evaluated design points that could guide the search to an optimum. Too high convergence typically occurs when the search algorithm finds such information, but it leads to a local, instead of a global optimum. In both cases no good quality results can be found and therefore, finding the right balance for convergence is critical. Depending on the specific search algorithm, there may be parameters that stimulate a particular convergent behavior. For example, we experimented with the selective pressure parameter in Chapter 2 and attempted to control convergence by means of population diversity in Section 7.7. *Convergence* captures one of the essential components of design space exploration mechanisms and more research in the DSE area will be needed to understand it better.

Effort

This concern refers to the engineering effort that goes into a design space exploration case study. We note that it is similar to the *effort* concern as was described previously for the evaluation of a single design point. Indeed, the motivation for this concern is the same: it is one of the main time consuming activities in the entire DSE process. Yet, even if we consider the *effort* concern solved for a single design point, then there are other, related concerns specific to the process of design space traversal. Although this was not one of the main topics in this thesis, we did frequently touch upon it, so we provide some conclusions here. Steps in the design space exploration phase that typically require (manual) effort include: 1) the definition of a design space exploration experiment (e.g., parameter space boundaries, and search algorithm parameters), 2) automatic transformation of design points to simulator input specifications, and 3) methods to run and evaluate the results of the DSE. Points (1) and (2) were shortly discussed in Section 3.4 and (3) in 3.5. We identified two methods that are frequently used: the meta-platform specification and the generator-approach. We use the former extensively in Chapter 7 where we also propose a solution to the negative effects of an evolutionary algorithm that is inherent in the the symmetry of the mapping-based design point specification.

Most importantly, we emphasize that much of the engineering effort could (theoretically) be non-recurring. Instead, we see that creation of the tools required by (1), (2) and (3) is typically repeated for specific tools or even users, although this burden could be shared by the whole research community (thus shortening design cycles even further). This would, however, require a large amount of coordination work to firstly standardize specifications and methods, for example to define interchangeable, high-level specifications of a design space (or design problem) and secondly, provide a framework in which exploration methods and tools can be compared on a set of standardized benchmarks. Such a framework could be based on a software design where tool-specific back-ends are provided as plug-ins, while the core functionality is shared. Research and engineering in this particular subdomain of (embedded) system design is however still in its infancy and the required coordination for shared frameworks and benchmarks will remain future work.

On the topic of (3) we have seen in Chapters 2 and 3 that within the context of Daedalus, the results of a design space exploration can be stored in an interchangeable format (XML), as well as in a database. The database allows for evaluation of a previously evaluated design space by means of queries expressed in a standardized query language, while the XML format promotes tool interoperability. The latter has been exploited by the various design space visualization tools that have been developed for Sesame (see Section 3.5).

Bibliography

[1] ConvergenSC, CoWare, http://www.coware.com/.

[2] Seamless, Mentor Graphics, http://www.mentor.com/.

[3] System Studio, Synopsys, http://www.synsopsys.com/.

[4] Xilinx: http://www.xilinx.com.

[5] T.M. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59 – 67, Feb. 2002.

[6] T. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, 1996.

[7] A. Baghdadi, N-E. Zergainoh, W. Cesario, T. Roudier, and A.A. Jerraya. Design space exploration for hardware/software codesign of multiprocessor systems. *Rapid System Prototyping, IEEE International Workshop on*, 0:8, 2000.

[8] J. R. Bammi, E. Harcoun, W. Kruijtzer, L. Lavagno, and M. Lazarescu. Software performance estimation strategies in a system level design tool. In *International Conference on Hardware Software Codesign (CODES)*, pages 82–87, 2000.

[9] G. Beltrame, C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, and V. Trianni. An assembly-level execution-time model for pipelined architectures. In *Proc. of Int. Conference on Computer Aided Design (ICCAD)*, pages 195–200, 2001.

[10] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. SystemC cosimulation and emulation of multiprocessor SoC designs. *IEEE Computer*, 36(4):53–59, 2003.

[11] I. Beretta, V. Rana, D. Atienza, and D. Sciuto. A mapping flow for dynamically reconfigurable multi-core system-on-chip design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(8):1211 –1224, aug. 2011.

[12] David Bernstein, Michael Rodeh, and Izidor Gertner. On the complexity of scheduling problems for parallel/pipelined machines. *IEEE Trans. Comput.*, 38:1308–1313, September 1989.

[13] Bryan Black and John Paul Shen. Calibration of microprocessor performance models. *Computer*, 31:59–65, May 1998.

[14] J.-Y. Brunel, W. M. Kruijtzer, H. J. H. N. Kenter, F. Pétrot, L. Pasquier, E. A. de Kock, and W. J. M. Smits. Cosy communication ip's. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 406–409, New York, NY, USA, 2000. ACM.

[15] L. Cai and D. Gajski. Transaction level modeling: An overview. In *Proc. of the Int. Conference on HW/SW Codesign and System Synthesis (CODES-ISSS)*, pages 19–24, Oct. 2003.

[16] Jean Paul Calvez, Dominique Heller, and Olivier Pasquier. Uninterpreted co-simulation for performance evaluation of hw/sw systems. In *Proceedings of the 4th International Workshop on Hardware/Software Co-Design*, CODES '96, pages 132–, Washington, DC, USA, 1996. IEEE Computer Society.

[17] Vincenzo Catania and Maurizio Palesi. A multi-objective genetic approach to mapping problem on network-on-chip. *JUCS*, 22, 2006.

[18] Karam S. Chatha and Ranga Vemuri. An iterative algorithm for partitioning and scheduling of area constrained HW-SW systems. In *IEEE International Workshop on Rapid System Prototyping RSP'99*, pages 134–139, 1999.

[19] K. Ben Chehida and Michel Auguin. HW/SW partitioning approach for reconfigurable system design. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems CASES'02*, pages 247–251. ACM, 2002.

[20] D. Lyonnard et al. Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip. In *Proc. of the Design Automation Conference (DAC'2001)*, June 18-22 2001.

[21] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Trans. Evolutionary Computation*, 6(2):182–197, 2002.

[22] L. Eeckhout, K. de Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *ISPASS '00: Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 1–6, Washington, DC, USA, 2000. IEEE Computer Society.

[23] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, 2003.

[24] L. Eeckhout, S. Nussbaum, JE Smith, and K. De Bosschere. Statistical simulation: adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, 2003.

[25] C. Erbas, S. Cerav-Erbas, and A.D. Pimentel. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *Evolutionary Computation, IEEE Transactions on*, 10(3):358 – 374, June 2006.

[26] Joachim Falk, Joachim Keinert, Christian Haubelt, Jürgen Teich, and Christian Zebelein. Integrated modeling using finite state machines and dataflow graphs. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 1041–1075. Springer US, 2010.

[27] Emanuel Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2:5–30, 1996. 10.1007/BF00226291.

[28] D. Feitelson. Workload modeling for performance evaluation. In M. C. Calzarossa and S. Tucci, editors, *Performance Evaluation of Complex Systems: Techniques and Tools*, pages 114–141. LNCS, Springer, 2002.

[29] F. Ferrandi, P.L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo. Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(6):911 –924, june 2010.

[30] Wenyin Fu and Katherine Compton. An execution environment for reconfigurable computing. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines FCCM'05*, pages 149–158, 2005.

[31] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

[32] P. Gerin, S. Yoo, G. Nicolescu, and A. A. Jerraya. Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures. In *Proc. of the Int. Conference on Asia South Pacific Design Automation*, pages 63–68, 2001.

[33] A. Gerstlauer and D. Gajski. System-level abstraction semantics. In *Proc. 15th Int. Symposium on System Synthesis (ISSS'02)*, pages 231–236, October 2-4 2002.

[34] A. Gerstlauer, C. Haubelt, A.D. Pimentel, T.P. Stefanov, D.D. Gajski, and J. Teich. Electronic system-level synthesis methodologies. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(10):1517 –1530, oct. 2009.

[35] A. Gerstlauer, H. Yu, and D. D. Gajski. RTOS modeling for system level design. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, page 10130, 2003.

[36] S.V. Gheorghita, T. Basten, and H. Corporaal. Application scenarios in streaming-oriented embedded system design. In *Proc. of the Int. Symposium in System-on-Chip*, 2006.

[37] Beltra Giovanni, Dario Bruschi, Donatella Sciuto, and Cristina Silvano. Decision-theoretic exploration of multiprocessor platforms. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, pages 205–210, New York, NY, USA, 2006. ACM.

[38] P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *Proc. of the Design, Automation, and Test in Europe (DATE) Conference*, pages 580–588, 2001.

[39] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[40] 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.

[41] F. Hessel, V. M. da Rosa, I. M. Reis, R. Planner, C. A. M. Marcon, and A. A. Susin. Abstract RTOS modeling for embedded systems. In *Proc. of the 15th IEEE International Workshop on Rapid System Prototyping (RSP'04)*, pages 210–216, 2004.

[42] K. Hines and G. Borriello. Dynamic communication models in embedded system co-simulation. In *Proc. of the Design Automation Conference*, pages 395–400, June 1997.

[43] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), August 1978.

[44] Pao-Ann Hsiung, Chao-Sheng Lin, and Chih-Feng Liao. Perfecto: A systemc-based design-space exploration framework for dynamically reconfigurable architectures. *ACM Transactions on Reconfigurable Technology and Systems*, 1(3):1–30, 2008.

[45] Chen Huang and Frank Vahid. Dynamic coprocessor management for fpga-enhanced compute platforms. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems CASES'08*, pages 71–78, 2008.

[46] Stanley Jaddoe and Andy D. Pimentel. Signature-based calibration of analytical system-level performance models. In *Proceedings of the 8th international workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '08, pages 268–278, Berlin, Heidelberg, 2008. Springer-Verlag.

[47] Stanley Jaddoe, Mark Thompson, and Andy D. Pimentel. Signature-based calibration of analytical performance models for system-level design space exploration. In *Transactions on High-Performance Embedded Architectures and Compilers (Trans. on HiPEAC)*, volume 4, 2009.

[48] Zai Jian Jia, A.D. Pimentel, M. Thompson, T. Bautista, and A. Nunez. Nasa: A generic infrastructure for system-level mp-soc design space exploration. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, pages 41 –50, oct. 2010.

[49] Yung-Chuan Jiang and Jhing-Fa Wang. Temporal partitioning data flow graphs for dynamically reconfigurable computers. *IEEE Trans. on VLSI Systems*, 15(12):1351–1361, 2007.

[50] PJ Joseph, K. Vaswani, and MJ Thazhuthaveetil. Construction and Use of Linear Regression Models for Processor Performance Analysis. In *Proc. of the Int. Symposium on High-Performance Computer Architecture*, pages 99–108, 2006.

[51] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, 1974.

[52] Kangas, T. et al. UML-based multi-processor SoC design framework. *ACM Trans. on Embedded Computing Systems*, 5(2):281–320, May 2006.

[53] Kurt Keutzer, Sharad Malik, Senior Member, A. Richard Newton, Jan M. Rabaey, and A. Sangiovanni-vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19:1523–1543, 2000.

[54] B. Kienhuis, E. F. Deprettere, P. van der Wolf, and K. Vissers. A methodology to design programmable embedded systems: The Y-chart approach. In E.F. Deprettere, J. Teich, and S. Vassiliadis, editors, *Embedded Processor Design Challenges*, pages 18–37. Springer, LNCS 2268, 2002.

[55] D. Kim, Y. Yi, and S. Ha. Trace-driven hw/sw cosimulation using virtual synchronization technique. In *Proc. of the Design Automation Conference*, June 2005.

[56] G. Kotsis. A systematic approach for workload modeling for parallel processing systems. *Parallel Computing*, 22(13):1771–1787, 1997.

[57] Harold W Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[58] Akash Kumar, Bart Mesman, Bart Theelen, Henk Corporaal, and Ha Yajun. Resource manager for non-preemptive heterogeneous multiprocessor system-on-chip. In *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia ESTMED'06*, pages 33–38, Washington, DC, USA, 2006. IEEE Computer Society.

[59] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):203 –215, feb. 2007.

[60] L. Lavagno, C. Passerone, V. Shah, and Y. Watanabe. A time slice based scheduler model for system level design. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 378–383, 2005.

[61] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proc. of the ACM/IEEE international symposium on Microarchitecture (Micro)*, pages 330–335, 1997.

[62] Y. Liu, S. Chakraborty, and W. T. Ooi. Approximate VCCs: a new characterization of multimedia workloads for system-level MpSoC design. In *Proc. of the conference on Design Automation (DAC)*, pages 248–253, 2005.

[63] Karthikeya M., Gajjala Purna, and Dinesh Bhatia. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Transactions on Computer Aided Design Integrated Circuits Systems*, 48(6):579–590, 1999.

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

[65] S. Mahadevan, F. Angiolini, M. Storgaard, R. G. Olsen, J. Sparso, and J. Madsen. A network traffic generator model for fast network-on-chip simulation. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 780–785, 2005.

[66] Grant Martin. Overview of the MPSoC Design Challenge. In *Proc. Design Automation Conference (DAC)*, San Francisco, USA, July 24-28 2006.

[67] V. Mathur and V.K. Prasanna. A hierarchical simulation framework for application development on system-on-chip architectures. In *ASIC/SOC Conference, 2001. Proceedings. 14th Annual IEEE International*, pages 428 –434, 2001.

[68] A. Maxiaguine, Y. Zhu, S. Chakraborty, and W-F. Wong. Tuning SoC platforms for multimedia processing: identifying limits and tradeoffs. In *Proc. of the Int. conference on Hardware/software codesign and system synthesis (CODES-ISSS)*, pages 128–133, 2004.

[69] A. Mihal and K. Keutzer. Mapping concurrent applications onto architectural platforms. In A. Jantsch and H. Tenhunen, editors, *Networks on Chips*, pages 39–59. Kluwer Academic Publishers, 2003.

[70] S. Mohanty and V. K. Prasanna. Rapid system-level performance evaluation and optimization for application mapping onto SoC architectures. In *Proc. of the IEEE International ASIC/SOC Conference*, 2002.

[71] Orlando Moreira, jacob Jan-David Mol, and Marco Bekooij. Online resource management in a multiprocessor with a network-on-chip. In *Proceedings of the ACM symposium on Applied computing SAC'07*, pages 1557–1564, New York, NY, USA, 2007. ACM.

[72] Mayan Moudgill, John-David Wellman, and Jaime H. Moreno. Environment for powerpc microarchitecture exploration. *IEEE Micro*, 19:15–25, May 1999.

[73] A. Nijenhuis and H.S. Wilf. *Combinatorial algorithms for computers and calculators*. Computer science and applied mathematics. Academic Press, 1978.

[74] H. Nikolov, T. Stefanov, and E.F. Deprettere. Multi-processor system design with ESPAM. In *Proc. of the Int. Conf. on HW/SW Codesign and System Synthesis (CODES-ISSS'06)*, pages 211–216, Oct. 2006.

[75] Juanjo Noguera and Rosa M. Badia. System-level power-performance trade-offs in task scheduling for dynamically reconfigurable architectures. In *Proceedings of the international conference on Compilers, Architecture and Synthesis for Embedded Systems CASES'03*, pages 73–83, 2003.

[76] Vincent Nollet, Prabhat Avasare, Hendrik Eeckhaut, Diederik Verkest, and Henk Corporaal. Run-time management of a mpsoc containing fpga fabric tiles. *IEEE Trans. Very Large Scale Integr. Syst.*, 16(1):24–33, 2008.

[77] Heikki Orsila, Emo Salminen, and Timo D. Hämäläinen. Parameterizing simulated annealing for distributing kahn process networks on multiprocessor socs. In *SOC'09: Proceedings of the 11th international conference on System-on-chip*, pages 19–26, Piscataway, NJ, USA, 2009. IEEE Press.

[78] A. D. Pimentel and C. Erbas. An IDF-based trace transformation method for communication refinement. In *Proc. of the Design Automation Conference*, pages 402–407, June 2003.

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

[80] A. D. Pimentel, M. Thompson, S. Polstra, and C. Erbas. Calibration of abstract performance models for system-level design space exploration. *Journal of Signal Processing Systems for Signal, Image, and Video Technology*, 50(2), 2008.

[81] Roberta Piscitelli and Andy D. Pimentel. A high-level power model for mpsoc on fpga. In *IPDPS Workshops*, pages 128–135, 2011.

[82] W. Qin and S. Malik. Flexible and formal modeling of microprocessors with application to retargetable simulation. *Design, Automation and Test in Europe (DATE) Conference*, pages 556–561, 2003.

[83] Yang Qu and Juha-Pekka Soininen. SystemC-based design methodology for reconfigurable system-on-chip. In *Proceedings of the Euromicro Conference on Digital System Design DSD'05*, pages 364–371, 2005.

[84] Daler N. Rakhmatov and Sarma B. K. Vrudhula. Hardware-software bipartitioning for dynamically reconfigurable systems. In *Proceedings of the tenth international symposium on Hardware/software codesign CODES'02*, pages 145–150, New York, NY, USA, 2002. ACM.

[85] Tero Rissa, Milan Vasilko, and Jarkko Niittylahti. System-level modelling and implementation technique for run-time reconfigurable systems. In *Proceedings of the international Symposium on Field-Programmable Custom Computing Machines FCCM'02*, page 295, 2002.

[86] K. Sigdel, M. Thompson, C. Galuzzi, A.D. Pimentel, and K. Bertels. rsesame - a generic system-level runtime simulation framework for reconfigurable architectures. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 460 –464, dec. 2009.

[87] Kamana Sigdel, Carlo Galuzzi, Koen Bertels, Mark Thompson, and Andy D. Pimentel. Runtime task mapping based on hardware configuration reuse. In *ReConFig*, pages 25–30, 2010.

[88] Kamana Sigdel, Mark Thompson, Carlo Galuzzi, Andy D. Pimentel, and Koen Bertels. Evaluation of runtime task mapping heuristics with rsesame - a case study. In *DATE*, pages 831–836, 2010.

[89] Kamana Sigdel, Mark Thompson, Andy D. Pimentel, Carlo Galuzzi, and Koen Bertels. System-level runtime mapping exploration of reconfigurable architectures. In *IPDPS*, pages 1–8, 2009.

[90] Kamana Sigdel, Mark Thompson, Andy D. Pimentel, Todor Stefanov, and Koen Bertels. System-level design space exploration of dynamic reconfigurable architectures. In *SAMOS*, pages 279–288, 2008.

[91] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai. Challenges in computer architecture evaluation. *Computer*, 36(8):30–36, 2003.

[92] A. Snavely, L. Carrington, and N. Wolter. Modeling application performance by convolving machine signatures with application profiles. In *Proc. of the IEEE Workshop on Workload Characterization*, pages 149–156, 2001.

[93] R. Srinivasan, J. Cook, and O. Lubeck. Performance Modeling Using Monte Carlo Simulation. *IEEE Computer Architecture Letters*, 5(1), 2006.

[94] T. Stefanov, B. Kienhuis, and E. F. Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *Proc. of the Int. Symposium on Hardware/Software Codesign (CODES)*, pages 7–12, May 2002.

[95] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. F. Deprettere. System design using Kahn process networks: The Compaan/Laura approach. In *Proc. of the Int. Conference on Design, Automation and Test in Europe (DATE)*, pages 340–345, February 2004.

[96] Greg Still, Roman Lysecky, and Frank Vahid. Dynamic hardware/software partitioning: A first approach. In *Proceedings of the Design Automation Conference DAC'03*. ACM, 2003.

[97] Toktam Taghavi and Andy D. Pimentel. Visualization of multi-objective design space exploration for embedded systems. In *Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, DSD '10, pages 11–20, Washington, DC, USA, 2010. IEEE Computer Society.

[98] Toktam Taghavi and Andy D. Pimentel. Design metrics and visualization techniques for analyzing the performance of moeas in dse. In Luigi Carro and Andy D. Pimentel, editors, *ICSAMOS*, pages 67–76. IEEE, 2011.

[99] Toktam Taghavi, Mark Thompson, and Andy D. Pimentel. Visualization of computer architecture simulation data for system-level design space exploration. In Koen Bertels, Nikitas J. Dimopoulos, Cristina Silvano, and Stephan Wong, editors, *SAMOS*, volume 5657 of *Lecture Notes in Computer Science*, pages 149–160. Springer, 2009.

[100] El-Ghazali Talbi and Benjamin Weinberg. Breaking the search space symmetry in partitioning problems: An application to the graph coloring problem. *Theoretical Computer Science*, 378(1):78 – 86, 2007.

[101] J. Teich, T. Blickle, and L. Thiele. An evolutionary approach to system-level synthesis. In *Proceedings of the 5th International Workshop on Hardware/Software Co-Design*, CODES '97, pages 167–, Washington, DC, USA, 1997. IEEE Computer Society.

[102] Bart D. Theelen, Marc Geilen, Twan Basten, Jeroen Voeten, Stefan Valentin Gheorghita, and Sander Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *MEMOCODE*, pages 185–194, 2006.

[103] R. Thid, I. Sander, and A. Jantsch. Flexible bus and NoC performance analysis with configurable synthetic workloads. In *Proc. of the Conference on Digital System Design*, pages 681–688, 2006.

[104] M. Thompson and A. D. Pimentel. A high-level programming paradigm for SystemC. In *Proc. of the Int. Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS)*, pages 530–539, July 2004.

[105] M. Thompson, A. D. Pimentel, S. Polstra, and C. Erbas. A mixed-level co-simulation method for system-level design space exploration. In *Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, ESTMED '06, pages 27–32, Washington, DC, USA, 2006. IEEE Computer Society.

[106] Allan Tucker, Jason Crampton, and Stephen Swift. Rgfga: An efficient representation and crossover for grouping genetic algorithms. *Evol. Comput.*, 13:477–499, December 2005.

[107] Nico L. J. Ulder, Emile H. L. Aarts, Hans-Jürgen Bandelt, Peter J. M. van Laarhoven, and Erwin Pesch. Genetic local search algorithms for the travelling salesman problem. In *PPSN*, pages 109–116, 1990.

[108] Keith Underwood. Fpgas vs. cpus: trends in peak floating-point performance. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 171–180, New York, NY, USA, 2004. ACM.

[109] Christine L. Valenzuela and Antonia J. Jones. Evolutionary divide and conquer (i): A novel genetic approach to the tsp. *Evolutionary Computation*, 1(4):313–333, 2011/10/28 1993.

[110] P. van Stralen and A. Pimentel. Scenario-based design space exploration of mpsocs. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 305 – 312, oct. 2010.

[111] Peter van Stralen. Scenario based design space exploration. Master's thesis, Universiteit van Amsterdam, September 2009.

[112] Peter van Stralen and Andy D. Pimentel. A trace-based scenario database for high-level simulation of multimedia mp-socs. In Fadi J. Kurdahi and Jarmo Takala, editors, *ICSAMOS*, pages 11–19. IEEE, 2010.

[113] G.V. Varatkar and R. Marculescu. On-chip traffic modeling and synthesis for MPEG-2 video applications. *IEEE Trans. on Very Large Scale Integration Systems*, 12(1):108–119, January 2004.

[114] S. Vassiliadis, G. N. Gaydadjiev, K.L.M. Bertels, and E. Moscu Panainte. The molen programming paradigm. In *Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 1–10, July 2003.

[115] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K.L.M. Bertels, G.K. Kuzmanov, and E. Moscu Panainte. The molen polymorphic processor. *IEEE Transactions on Computers*, pages 1363– 1375, November 2004.

[116] S. Verdoolaege, H. Nikolov, and T. Stefanov. PN: a tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems*, vol. 2007, Article ID 75947, 2007.

[117] L. Darrell Whitley, Timothy Starkweather, and D'Ann Fuquay. Scheduling problems and traveling salesmen: The genetic edge recombination operator. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 133–140, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[118] Ch. Ykman-Couvreur, E. Brockmeyer, V. Nollet, T. Marescaux, F. Catthoor, and H. Corporaal. Design-time application exploration for mp-soc customized run-time management. In *Proceesings of International Symposium on System-on-Chip*, pages 66–69. IEEE Computer Society, 2005.

[119] Pavel G. Zaykov, Georgi K. Kuzmanov, and Georgi N. Gaydadjiev. Reconfigurable multithreading architectures: A survey. In *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, pages 263–274, Berlin, Heidelberg, 2009. Springer-Verlag.

[120] E. Zitzler, K. Giannakoglou, D. Tsahalis, J. Periaux, K. Papailiou, T. Fogarty (eds, Eckart Zit. Ler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm for multiobjective optimization, 2002.

Samenvatting

Moderne embedded systemen moeten vaak verschillende complexe taken uitvoeren, zijn onderhevig aan een keur van (soms tegenstrijdige) functionele en non-functionele eisen en moeten bovendien in relatief korte tijd ontworpen kunnen worden. Dit maakt het ontwerpproces van zulke systemen zeer ingewikkeld en daarom moet de ontwerper in zijn taak ondersteund worden door goede software-tools die efficiënte ontwerptechnieken implementeren en het ontwerpproces zoveel mogelijk automatiseren. Door vroeg in het ontwerpproces te beginnen met het modelleren en simuleren van het systeem, krijgt men eerder feedback over de gevolgen van bepaalde ontwerpbeslissingen. Hiermee is in het ontwerpproces veel winst te behalen, aangezien een juiste beslissing later in het ontwerpproces veel tijd en kosten kan besparen. Goede modelleringstools en en exploratie-technieken zijn hierbij wel van essentieel belang.

In de introductie van dit proefschrift stellen we een aantal voorwaarden op waaraan een modelleer- en exploratieomgeving zou moeten voldoen om zo goed mogelijk ondersteuning te bieden aan het ontwerpproces. We maken een onderscheid tussen het snel en accuraat modelleren van een enkel systeemontwerp en het vinden van de juiste ontwerp uit een zoekruimte van alle mogelijke ontwerpen (ook wel Design Space Exploration of DSE genoemd). Hoofdstuk 3 beschrijft de Sesame tool en laat zien dat een ontwerper in korte tijd een hoog-niveau simulatie model kan samenstellen door middel van een system-level model specificatie en herbruikbare simulatie componenten. In Hoofdstuk 2 wordt Sesame in de context geplaatst van Daedalus: een complete toolflow voor het ontwerpen van multiprocessor system-on-chip (MPSoC) systemen, van applicatie, via design space exploratie (Sesame) naar een volledig geïmplementeerd en functioneel prototype.

In Hoofdstuk 4 laten we zien hoe de hoog-niveau simulatiemodellen van Sesame kunnen worden gecalibreerd: drie verschillende methoden worden voorgesteld die de nauwkeurigheid van het Sesame model verhogen door op efficiënte wijze gedetailleerde performance informatie aan het model toe te voegen. Vervolgens kijken we naar het type systemen dat kan worden gemodelleerd in Sesame en breiden dat uit met multi-applicatie systemen (Hoofdstuk 5) en dynamisch herconfigureerbare systemen (Hoofdstuk 6). In het eerste geval introduceren we een hiërarchisch systeem van schedulers zodat meerdere applicaties volgens verschillende realistische schedules op het architectuur model kunnen worden afgebeeld. Ook definiëren we hier een model voor stochastische applicatie modellen die gebruikt kunnen worden voor speculatieve DSE. Vervolgens breiden we Sesame uit voor het modelleren

voor een opkomend type MPSoC systemen: partially dynamic reconfigurable systems. We laten zowel de generieke model componenten hiervoor zien, als ook de resultaten van een volledig model gebaseerd op een realistische case study.

Het voorlaatste deel (Hoofdstuk 7) richt zich op het efficiënt afzoeken van ontwerpruimten. We gebruiken hiervoor de bekende metaheuristisch methode van genetische algoritmen (GAs), analyseren de resultaten op basis van een stochastisch gegenereerde case-study en bediscussiëren de voor- en nadelen. Door het gebruik van domein-specifieke kennis ontwerpen we een nieuwe set van genetische operatoren die het GA efficiënter laten werken.

Tenslotte komen in Hoofdstuk 8 twee grotere case-studies aan bod die laten zien hoe verschillende van de besproken technieken kunnen worden gebruikt in industrie-relevante ontwerpproblemen. We doen dit in de context van de Daedalus design flow en in het eerste geval wordt het volledige ontwerp- en implementatieproces doorlopen voor verschillende ontwerp-varianten van een multiprocessor MPSoC media systeem. We vergelijken de simulatieresultaten met metingen aan de verschillende systeemprototypes en komen tot de conclusie dat de simulatiemodellen inderdaad goed bruikbaar zijn voor het efficiënt doorzoeken van ontwerpruimtes. In een volgende case-study laten we het ontwerp zien van een heterogeen MPSoC systeem met maximaal 24 cores voor een JPEG compressie applicatie. In de conclusies komen we terug op de in de introductie gestelde eisen voor het bevorderen van het ontwerpproces en maken we de balans op.

Scientific output

Sesame, Daedalus and Case-studies

- M. Thompson, T. Stefanov, H. Nikolov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere, *A Framework for Rapid System-level Exploration, Synthesis, and Programming of Multimedia MP-SoCs*, in the Proc. of the ACM/IEEE/IFIP Int. Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS '07), pp. 9-14, Salzburg, Austria, Oct., 2007

- C. Erbas, A. D. Pimentel, M. Thompson, and S. Polstra, *A Framework for System-level Modeling and Simulation of Embedded Systems Architectures*, in EURASIP Journal on Embedded Systems, 2007, available online: DOI 10.1155/2007/82123

- H. Nikolov, M. Thompson, T. Stefanov, A. D. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. F. Deprettere, *Daedalus: Toward Composable Multimedia MP-SoC Design*, invited paper, in the Proc. of the ACM/IEEE Int. Design Automation Conference (DAC '08), pp. 574-579, Anaheim, USA, June, 2008

- A. D. Pimentel, T. Stefanov, H. Nikolov, M. Thompson, S. Polstra and E. F. Deprettere, *Tool Integration and Interoperability Challenges of a System-level Design Flow: a Case Study*, invited paper, in the Proc. of Int. Symposium on Systems, Architectures, MOdeling and Simulation (SAMOS '08), pp. 167-176, LNCS, Samos, Greece, July, 2008

Model Calibration

- M. Thompson, A. D. Pimentel, S. Polstra and C. Erbas, *A Mixed-level Co-simulation Method for System-level Design Space Exploration*, in the Proc. of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '06), pp. 27-32, Seoul, Korea, Oct., 2006

- A. D. Pimentel, M. Thompson, S. Polstra and C. Erbas, *On the Calibration of Abstract Performance Models for System-level Design Space Exploration* , in the Proc. of the Int. Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation (IC-SAMOS '06), IEEE CAS, pp. 71-77, Samos, Greece, July, 2006

- A. D. Pimentel, M. Thompson, S. Polstra and C. Erbas, *Calibration of Abstract Performance Models for System-level Design Space Exploration*, in Journal of Signal Processing Systems for Signal, Image, and Video Technology, pp. 99-114, Vol. 50 (No. 2), Feb. 2008, Springer

- S. Jaddoe, M. Thompson, and A. D. Pimentel, *Signature-based Calibration of Analytical Performance Models for System-level Design Space Exploration*, in Transactions on High-Performance Embedded Architectures and Compilers (Trans. on HiPEAC), Vol. 4 (No. 4), 2009

Modeling Dynamically Reconfigurable Systems

- K. Sigdel, C. Galuzzi, M. Thompson, A.D. Pimentel, K. Bertels, *Runtime Task Mapping Based on Hardware Configuration Reuse*, in the Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '10), Cancun, Mexico, December 2010

- K. Sigdel, M. Thompson, A. D. Pimentel, K.L.M. Bertels, and C. Galuzzi, *Evaluation of Runtime Task Mapping Heuristics with rSesame - A Case Study*, in the Proceedings of the Int. Conference on Design, Automation, and Test in Europe (DATE'10), Dresden, Germany, March 2010

- K. Sigdel, M. Thompson, A. D. Pimentel, K.L.M. Bertels, and C. Galuzzi, *rSesame - A Generic System-level Runtime Simulation Framework for Reconfigurable Architectures*, in the Proceedings of the International Conference on Field-Programmable Technology (FPT '09), Sydney, Australia, December 2009

- K. Sigdel, M. Thompson, A. D. Pimentel, K.L.M. Bertels, and C. Galuzzi, *System Level Runtime Mapping Exploration of Reconfigurable Architectures*, in the Proceedings of the 16th Reconfigurable Architectures Workshop (RAW '09), Rome, Italy, May 2009

- K. Sigdel, M. Thompson, A. D. Pimentel, T. Stefanov, and K. Bertels, *System Level Design Space Exploration of Dynamic Reconfigurable Architectures*, in the Proc. of Int. Symposium on Systems, Architectures, MOdeling and Simulation (SAMOS '08), pp. 279-288, LNCS, Samos, Greece, July, 2008

Multi-application Modeling

- M. Thompson and A. D. Pimentel, *Towards Multi-application Workload Modeling in Sesame for System-level Design Space Exploration*, in the Proc. of the 7th Int. Workshop on Embedded Computer Systems: Architectures, MOdeling, and Simulation (SAMOS '07), pp. 222-232, LNCS, Samos, Greece, July, 2007

Other Tools and Techniques

- M. Thompson and A. D. Pimentel, *A High-level Programming Paradigm for SystemC*, in the Proc. of the 4th Int. Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS '04), pp. 530-539, LNCS, Samos, Greece, July, 2004

- Z. J. Jia, A. D. Pimentel, M. Thompson, T. Bautista, A. Nunez, *NASA: A Generic Infrastructure for System-level MP-SoC Design Space Exploration*, in the Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '10), Scottsdale, AZ, USA, Oct. 2010

- T. Taghavi, M. Thompson, and A. D. Pimentel, *Visualization of Computer Architecture Simulation Data for System-level Design Space Exploration*, in the Proc. of Int. Symposium on Systems, Architectures, MOdeling and Simulation (SAMOS '09), Samos, Greece, July 2009

- T. Taghavi, A.D. Pimentel, M. Thompson, *System-level MP-SoC Design Space Exploration using Tree Visualization*, in the Proc. of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '09), Grenoble, France, Oct. 2009