Design and Analysis of Efficient MPSoC Simulation Techniques

Von der Fakultät für Elektrotechnik und Informationstechnik
der Rheinisch–Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften
genehmigte Dissertation

vorgelegt von
Diplom–Ingenieur Stefan Kraemer
aus Heidelberg/Baden-Württemberg

Berichter: Universitätsprofessor Dr. rer. nat. Rainer Leupers
Universitätsprofessor Dr.-Ing. Norbert Wehn

Tag der mündlichen Prüfung: 14.07.2011

Acknowledgments

This thesis is the result of more than five years of work. During that time I have been supported by many people. Without their help this thesis would not have been possible. Therefore I would like to take the opportunity to thank them.

First, I would like to thank my doctoral advisor Professor Rainer Leupers for giving me the opportunity to work in his group, and for his constant support and constructive feedback throughout the course of my research. I would like to thank him for sharing his experience with me and giving me the opportunity to learn the importance of details for successfully conducting an engineering project. His comments and insights have been a source for re-evaluating my ideas and unveiling new perspectives.

I am also thankful to Professor Gerd Ascheid and Professor Heinrich Meyr. Their discussions and experience have revealed interesting aspects and new ideas. Also I would like to thank Professor Norbert Wehn for his competent and helpful feedback during his review of my thesis.

During my time at the institute a number of people have enriched my professional life by giving me the opportunity to discuss my ideas and by giving valuable feedback. I am particular indebted to Dietmar Petras and Thomas Philipp for working with me on the Checkpoint/Restore framework. Without their contribution, their support and the good working atmosphere this work would have been impossible. I am also indebted to Lei Gao for good and inspiring discussions and cooperation on the hybrid simulation project. I also like to thank Christoph Schumacher and Jovana Jović for supporting me in the context of the NoC hybrid simulation. My time at the institute would have been less enjoyable without the funny moments and good discussions I had with my colleagues. I thank you all!

I was fortunate to have been supported by highly motivated students working towards their theses. I would like to thank Xiaowei Pan and Jan Weinstock for their excellent work which contributed to that thesis.

I am also very grateful to Christoph Schumacher, Felix Engel, Jovana Jović and Stefan Schürmans who sacrificed a lot of their time for carefully proof-reading this work. Their feedback throughout the different stages has been a valuable source for shaping this thesis upon completion.

Finally, I would like to thank those people I care most, my parents and my friends. Thank you Bernd, Felix and Franziska for supporting and encouraging me all the time and for being patient with me during the last months of my work. It is good to have friends that remind you from time to time that there is a world outside the university.

My biggest thanks go to my parents, who nurtured my interests in all technical problems and encouraged me to become an engineer.

Contents

Chapter 1

Introduction

1.1 Motivation

During the last decade embedded systems have pervaded nearly all aspects of our daily lives. The advancements in digital information technologies and especially in mobile communications [166] have been a main driver for this development. In fact, 98% of the processors manufactured today are used inside embedded systems [44]. The evolution of the embedded systems has been driven by the improvements in deep submicron technology as predicted by Moore's law [103]. These technology improvements have given the system designers the possibility to integrate more and more functionality onto a single chip and thus satisfy the often conflicting requirements, e.g. low power consumption and high throughput. The current level of integration allows to create a complete system-on-chip (SoC) including processors and peripherals. The downside of this development is the huge complexity of modern embedded systems that leads to a drastic increase in *non recurring engineering* (NRE) costs [65]. This phenomenon is also referred to as *crisis of complexity* [55] and boosted a lot of research in the field of electronic system level (ESL) design in order to efficiently manage and design such complex systems.

It can be observed that the way how the desired functionality of an embedded system is realized has changed over the different generations. There has been a transition from mostly hardwired functionality with limited usage of software to fully programmable systems, realizing most of its functionality purely in software. On the

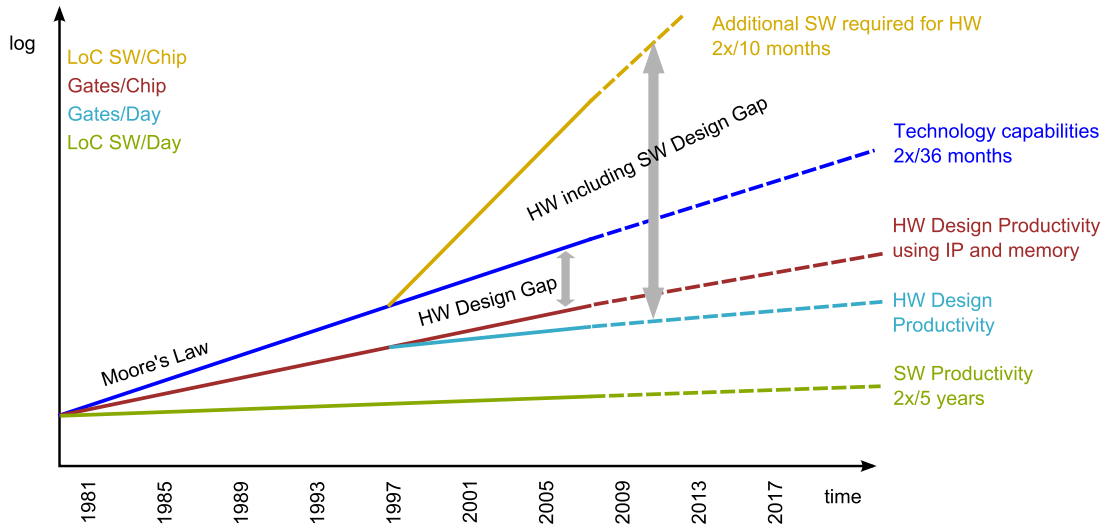


Figure 1.1: Hardware and software design gaps, based on [65]

one hand, this is due to the increased amount of available computing power in embedded systems. On the other hand, realizing the functionality in software increases the flexibility of an embedded system. Hence, it is possible to reuse the same hardware platform for different applications reducing the NRE costs. Actually, it can be observed that most of the added value in the consumer market stems from the software. Furthermore, embedded software offers companies an easy way to differentiate themselves from each other by adding new features and extending the functionality of their embedded systems. As a consequence of this development, the complexity of embedded software has been constantly growing over the years as pointed out by the international technology roadmap for semiconductors (ITRS) [65]. From Figure 1.1 it can be seen that embedded software complexity has already outpaced the HW complexity when comparing lines of code (LoC) written in C and HDL. Even worse, the software (SW) productivity is increasing much slower than the hardware (HW) productivity. This leads to the *embedded software gap*. This gap is reflected by the fact that today the number of embedded software developers is much higher than the number of hardware developers [98]. In order to mitigate the software design gap, improved software development tools are required.

Traditionally, the development of the embedded software can only start once a first hardware prototype of the target architecture is ready, leading to a purely sequential design flow. However, the shrinking time-to-market that can be observed especially in the consumer market, together with the increasing complexity of embedded systems requires a hardware/software co-design methodology in order to cope with the system complexity and the stringent time requirements of the projects.

Virtual platforms (VP) are executable software models of hardware systems, allowing to simulate the entire system before it is built in real hardware. Thus, they can be utilized to develop and test embedded software efficiently, long before the complete system is ready in hardware. Using the VP concept it is possible to practically perform hardware/software co-design. Moreover, since VPs are pure software models of the hardware, they have the advantage that their internal states can be more easily observed and debugged, compared to a real hardware implementation. Like for all other types of simulation, the simulation speed of VPs is crucial to use them efficiently. The key components of a VP are the processor simulators. In today's VPs mostly instruction set simulators (ISS) are employed to simulate the different processor cores of an embedded system. Thus, the overall simulation speed is mainly determined by the simulation speed of the processor simulators. With state-of-the-art ISS based simulation techniques a simulation speed of up to 100–500 MIPS is possible, depending on the complexity of the simulated processor. However, the current simulation techniques have been pushed to their limits. Going beyond this limit will require a higher abstraction level, which comes at the price of lower timing accuracy of the simulation. In general, the simulation speed of a VP is inversely related to the accuracy of the simulated platform. Especially for hardware designers and timing verification purposes a completely accurate simulation environment is of utmost importance. However, for the development of embedded software the accuracy requirements are generally less stringent. For efficient software development it is more important to find a proper balance between simulation accuracy and simulation speed. In particular, considering the growing complexity of the software stack, a sufficiently fast simulation environ-

ment is of great importance to completely execute and test the software running on the embedded system. Compared to the traditional and more hardware centric simulations, the simulation requirements for software developers are quite different. Besides the simulation speed also the possibility to efficiently debug the simulated software plays an important role. Hence, all kinds of simulation techniques requiring extensive pre- or post-processing every time the software has changed are not well suited for debugging. These techniques have been developed for estimating the performance of a hardware platform for a given and fixed set of software. In the context of software debugging it is also important to note that the accuracy requirements change during simulation. For example, in case that the software developer wants to investigate the reason why a software produces a wrong result at a certain point in time, he first needs to drive the software into the desired state. During this process the simulation speed is the primary concern since the problematic software state might only be reached after a long simulation time. The timing accuracy of the simulation is only of secondary concern as long as the simulation is functionally correct. Once the simulation has reached the desired state, the developer can start the debugging of the application. During the debugging process, the simulation accuracy is of primary concern in order to pinpoint the exact location and the precise reason for the software failure.

This thesis presents a solution to the aforementioned problem of efficient software simulation. Three different techniques are presented to efficiently simulate VPs exploiting the particular requirements for embedded software simulation.

A checkpointing technique based on *process checkpointing* has been developed taking the specific requirements and characteristics of VPs into account. In particular, the dependencies between the simulation and the operating system need to be considered during the checkpointing process. This technique does not improve the simulation speed itself, but it can be utilized for reducing the time spent simulating by restoring previously saved states of the entire simulation environment.

To speed up the simulation itself a novel hybrid simulation technique called *HySim* has been developed. This hybrid simulation technique gives the user the possibility to switch between a fast but inaccurate and a slow but detailed simulation. Hence, the software developer has the possibility to decide which parts of the application should be simulated in detail and which parts of the application should be executed at a high simulation speed. In order to demonstrate the capabilities of this approach the hybrid simulation framework has been integrated into a commercially available SystemC based simulation framework. A complex and scalable hardware platform and its corresponding software stack serve as a case study. Both techniques have been developed in the context of the European SHAPES project [142].

With the growing complexity of the MPSoCs also the complexity of the interconnect networks is growing and hence their impact on the simulation speed. Therefore a hybrid network-on-chip (NoC) simulation approach has been developed that offers the software developer a slow and accurate NoC simulation mode or a fast and less accurate simulation mode. During the fast simulation mode the complete NoC is bypassed and the timing of the data transfers is estimated by an analytical NoC model based on queuing theory. A scalable NoC simulation framework has been used in order to

study the impact on hybrid NoC simulation on the timing accuracy and the simulation speed.

1.2 Outline of the Thesis

This thesis is organized as follows. Chapter 2 provides the background necessary to understand the concept of VPs and its impact on the development process of embedded systems. Furthermore, a case study of a VP implementation is presented to evaluate the presented concepts. Using the example of a virtualized USB connection, the importance of virtual I/O for VPs is discussed. Afterwards, Chapter 3 presents the related work in the field of processor simulation as well as a short overview of the academically and commercially available simulation frameworks. A taxonomy of the different simulation techniques is given and the advantages and disadvantages of the different simulation techniques are clearly pointed out. Surveys of publications specifically related to individual chapters are presented in the corresponding chapters. The VP developed in the context of the European SHAPES [142] project serves as a primary driver for the work presented in this thesis. Therefore, Chapter 4 gives an overview of the hardware platform and the corresponding software toolchain developed in the course of the SHAPES project. Moreover, the main driver applications of this project are presented since they are used as primary benchmarks for the simulation techniques presented in this thesis. Chapter 5 presents a checkpointing technique specifically tailored towards the needs of SystemC based VPs. This technique allows storing and restoring complete simulations and hence leads to reduction of the time spent simulating. Chapter 6 presents a novel hybrid processor simulation technique called *HySim*, that allows switching at simulation time between an accurate but slow simulation mode and a fast but inaccurate one. The complete software tool flow as well as the details about the required instrumentation process are discussed. Furthermore, the integration of this hybrid simulation technique into a commercially available simulation environment is presented. Chapter 7 presents a hybrid simulation technique that is specifically tailored to speed up the simulation of interconnects such as networks-on-chip (NoC). An analytical NoC model based on queuing theory is used to estimate the contention in the network while bypassing the detailed network simulation. Chapter 8 summarizes the results of this thesis and gives an outlook for future research.

Chapter 2

Introduction to Virtual Platforms

As described in Chapter 1, the amount of software running on SoCs is doubling every 10 month. Hence, the software development plays a central role in the design process of SoCs. The possibility to verify functionality as well as to analyze and to optimize the performance of the applications intended to run on the SoC is very important. However, the traditional design flow, as shown in Figure 2.1, does not scale well with the increased software complexity. Due to the purely sequential design flow, the software development can only start once a hardware prototype (either FPGA or silicon) is ready. To a certain extent, it is possible to develop the application directly on a standard PC. However, this approach neglects any information about the target system. Therefore, this approach is not suitable for predicting the timing behavior and the resource consumption of the application on the real hardware. Simulation based approaches can mitigate this problem.

In general, the term *virtual platform* (VP) refers to an executable specification of a hardware system. The software running on the simulated hardware cannot tell the difference between the VP and the real hardware. Thus, VPs are very well suited for software development, especially during early design stages when no hardware prototype is at hand. The use of VPs mitigates the dependency between the hardware development and the software development by simulating the hardware of the SoC in software. Moreover, the concurrent development of HW and SW allows joint optimization of both aspects in order to reach the desired system performance. Hence, the availability of VPs enable HW/SW co-design. Not only the software development benefits from the utilization of VPs, but also the overall system design by getting early feedback on the execution characteristics of the application. This kind of information allows to improve the design and to minimize the risk of an expensive re-spin during late design stages.

In the following section the characteristics and advantages of VPs are described in detail. Afterwards a case study describing a VP for a wireless Internet device is presented.

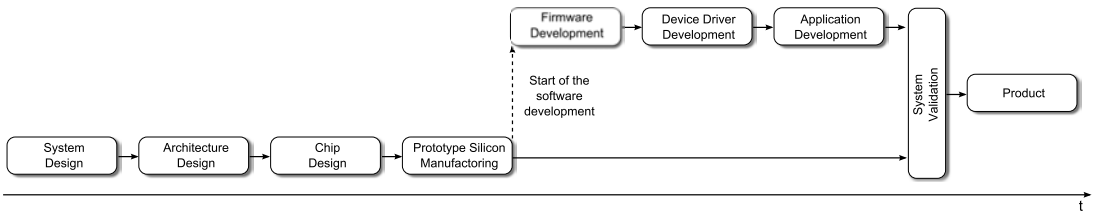


Figure 2.1: Traditional SoC design flow

2.1 Characteristics of Virtual Platforms

It is important to first understand the major use cases for VPs, before describing the requirements and special characteristics of VPs.

All application areas of VPs have in common that they require a joint design of the hardware aspects and the software aspects in order to meet the tight design requirements. The VP concept is mainly adopted by engineers responsible for system design, SoC design and low level software development. The different use cases of VPs can be classified into four principal cases:

- *Design space exploration*: At the system level VPs can be utilized for fast design space exploration, system architecture analysis and system optimization.
- *Early SW development*: VPs enable early software development including performance evaluation, validation and application debugging long before a first hardware prototype is ready.
- *HW development*: VPs can simplify the hardware development. Especially, verification and debugging of the low-level hardware-software interaction.
- *HW/SW partitioning*: The high complexity of today's embedded systems requires a well balanced HW/SW partitioning of the system functionality in order to meet all design constraints. In particular VPs enable the system designer to sweep through the large design space to identify a good partitioning solution.

2.1.1 Advantages

As shown in Figure 2.1 the traditional design flow is purely sequential. Thus, the software development is only possible after the hardware prototype or the development board is available. Compared to this flow a VP based approach allows the creation of a software development environment at a much earlier point in time. Figure 2.2 illustrates the SoC design approach using a VP. The development of the VP itself takes place during an early design stage in parallel with the hardware design in order to ensure the early availability of a software development environment. Making use of the VP the software developers can start to develop, optimize and integrate the software intended to run on the SoC. The run-time characteristics obtained by the simulation can be utilized not only to improve the software, but also to improve the hardware architecture. This leads to a joint, iterative optimization of software and hardware until all system requirements are met. Furthermore, parts of the system validation can also be moved to earlier design stages.

The different advantages of utilizing VPs for SoC design can be grouped with respect to three different categories: *hardware design*, *software design* and *system validation*.

Hardware design By using simulations at different levels of detail (mixed-level simulation) it is possible to iteratively refine the VP and verify the different building blocks. For example, a VP can be used to generate stimuli for a detailed RTL-based simulation of a given hardware block. Hence, it is possible to stepwise refine the accuracy of a VP. Furthermore, the hardware designer can easily test different hardware/software partitions in order to find the best match. In other words, VPs enable the hardware designer to perform design space exploration and optimization of the platform architecture.

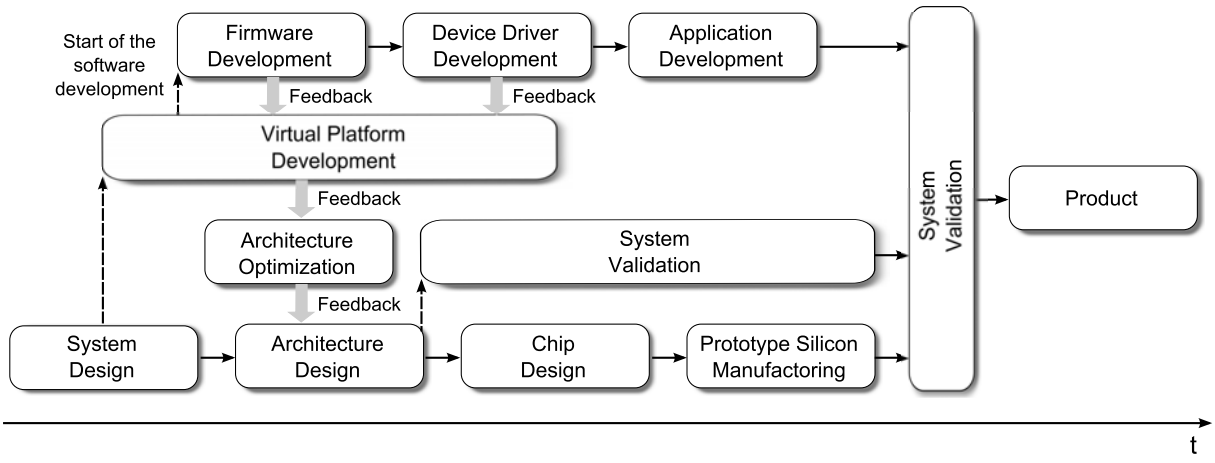


Figure 2.2: SoC design flow using a virtual platform

Software design The software developers benefit most from the utilization of VPs. First of all, the fact that it is possible to run software on the target system without the need of a hardware prototype is a big advantage. Second, VPs provide a superior visibility and controllability compared to the real hardware systems. For instance, it is possible to set breakpoints on every memory element and signal of the entire VP, including the buses and the peripheral blocks. In real hardware, the visibility of the internal states is limited. Moreover, it is possible to halt the simulation at any point in time which is – especially for multi-core systems – not possible is hardware.

System validation The early availability of VPs gives the possibility to start validating the system much earlier than in the traditional flow. The VP can be used to validate hardware as well as the software aspects of the design.

For the overall SoC design the use of VPs ensures that the entire design team shares the same development environment and specification. Hence, all members of the system development, the software development, the hardware development and the verification team see the identical functionality and timing. Due to the fact that VPs are just software, they can be more easily distributed to design teams or customers rather than sending hardware prototypes all over the world. Furthermore, fixing bugs or adding new features can be easily done using VPs without the need for an expensive development of a new hardware prototype.

In general, the SoC design flow using VP mitigates the dependency of software development on hardware which leads to a considerably shortened design cycle. However, this is achieved by investing additional effort during the early design stages in creating and maintaining a VP of the system.

2.1.2 Requirements

As described in the previous section VPs offer various advantages during the SoC development, e.g. they increase the productivity of the developers by offering an early SW development environment. Furthermore, they are providing a development plat-

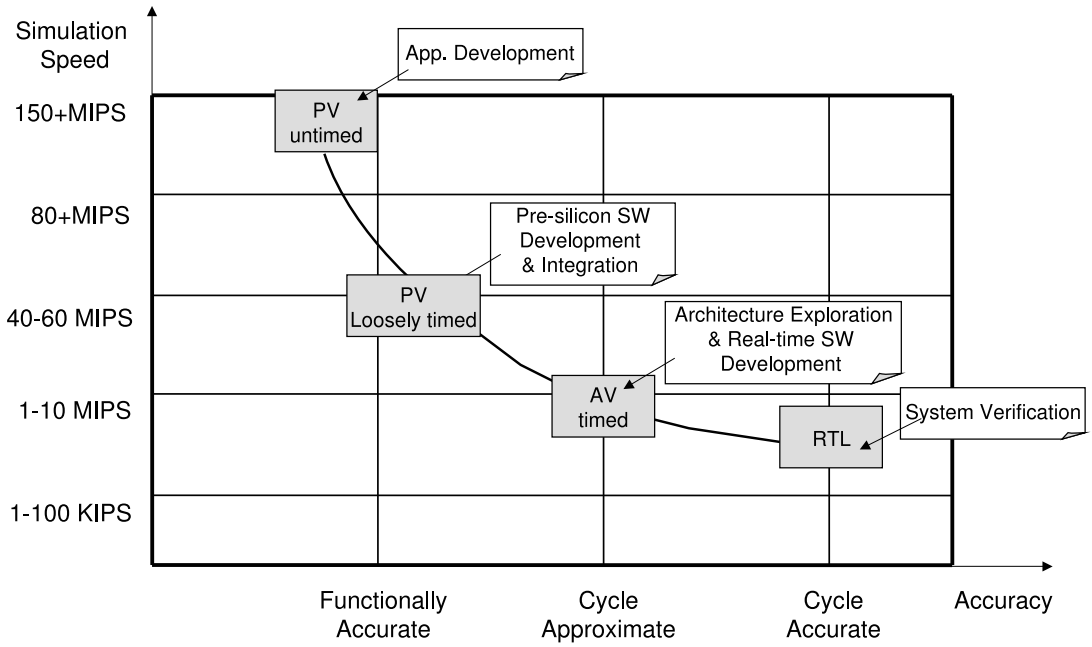


Figure 2.3: Relationship between simulation speed and accuracy

form for architectural analysis, joint hardware/software development, and system validation. In order to asses the impact of VPs on the SoC design, it is important to understand what kind of criteria and requirements need to be satisfied in order to create a VP with the desired functionality. In this sub-section the requirements to create a VP are addressed.

2.1.2.1 Performance & Accuracy

The most important aspects of a VP are the performance and accuracy. The performance refers to the simulation speed and the accuracy indicates the fidelity of the results obtained from a VP compared to the hardware implementation. In general, the user aims at VPs that offer high simulation speed and have a high accuracy at the same time. However, due to the inverse relation between simulation speed and simulation accuracy it is not possible to create a VP that fullfills both requirements at the same time. Depending on the targeted use-case of the VP the developers need to carefully balance simulation speed and accuracy.

Figure 2.3 depicts the relationship between simulation speed and accuracy for different levels of abstraction. In addition for each abstraction level an example use case is given. It can be seen that the register transfer level (RTL) provides the highest simulation accuracy but also the slowest simulation speed. Apart from that, an RTL model becomes only available late in the design process since all hardware details need to be fixed before. For software developers the simulation speed that can be achieved by RTL simulators is too low in order to simulate long running applications efficiently. Usually RTL based simulation is used for system verification during late design stages.

By rising the abstraction level to *architecture view* (AV) and further to *programmer's view* (PV), the simulation speed increases. Depending on the intended use case, the

developers can decide which abstraction layer is appropriate. For instance, if the VP is mainly intended for pre-silicon software development and integration of embedded software, then the loosely timed programmer's view offers a good balance between simulation speed and accuracy. For early application development even an untimed PV model can be used in order to achieve high simulation speed.

2.1.2.2 Analysis & Debug

Besides the requirements of simulation speed and accuracy of a VP, there are two further aspects which are very important. First, the capability of efficiently debugging and second the possibility of analyzing the application. Compared to the real hardware, debugging and software analysis can be performed much easier on a VP due to the full visibility of the internal states. The main advantages of debugging and software analysis on VPs are presented in the following.

In a traditional design flow, software developers can only develop their software and analyze its performance after both software and hardware are ready. If it turns out that the performance of the underlying hardware is not sufficient, a costly and time consuming re-design of the hardware subsystem is required. The goal of VPs is to fill this gap and provide the system developer and the software developer with a framework for early performance estimation. For example, during the early design stages system architects can use VPs to explore different hardware/software partitionings. Furthermore, VPs offer very powerful software analysis features since virtually all aspects of a VP can be instrumented and recorded during simulation. Thus, the software developer is able to get a detailed analysis of the hardware software interaction that would be very hard to obtain using a hardware prototype.

Debugging embedded software – especially for low level drivers or firmware – on real hardware systems is not an easy job for software developers. This is mainly because of the limited debugging capabilities of real hardware. For example, during hardware design the developers need to decide which internal states will later be accessible via the debug interface. In order to keep the overhead for debugging logic small, not all internal states might be directly addressable. Moreover, it is a very complex task to stop a hardware platform at an arbitrary point in time. VPs can support the software developers by providing good *observability*, *controllability* and a *deterministic* execution of the simulated hardware. For example, the simulation of a VP can be halted at any point or at any time and it is possible to investigate all simulated hardware structures. Together with the fact that the execution is completely deterministic, it is possible to debug also rare events like an interrupt triggered event or even an interrupt triggered event interrupting another interrupt.

In the next section a case study is presented showing how VPs can be used for software development and debugging.

2.2 Case Study: Wireless Internet Device

A wireless Internet device called *Chumby* has been selected as driver to demonstrate the capabilities of VPs. The Chumby device is an ambient consumer electronics product made by Chumby Industries, Inc. [27]. According to their website, "*Chumby is a compact*

Component	Comment
350 MHz ARM9	Part of the Freescale iMX21 [45]
Crypto processor	STMicroelectronics STR711FR0/1 ARM7
64 MB SDRAM	
64 MB Flash ROM	
320x240 TFT	Display with touchscreen
Touchscreen controller	
2 Wstereo speakers	With headphone socket
Three USB 2.0 ports	One internal, two external
USB WiFi adapter	Supporting 802.11g
Built-in microphone	
3-axis accelerometer	
Serial port	

Table 2.1: Hardware overview of the Chumby device

wireless fidelity (WiFi) device that displays useful and entertaining information from the web". It can be used to display news, play music and show videos, and so forth.

The main reason for selecting this device for the case study is the fact, that Chumby is designed to be customizable by users. In order to attract users to do so, Chumby Industries has released specifications of Chumby's hardware and the source code of the operating system, the drivers and the applications. Hence, with all this information available, Chumby is an ideal platform to study the impact of the creation of a VP.

2.2.1 Chumby Device

Before describing the creation process of the VP for the Chumby device, it is important to understand the hardware platform as well as the software structure used in this device.

2.2.1.1 Hardware Architecture

The Chumby device is designed as open source hardware, with schematics, printed circuit board (PCB) layouts and packaging designs available. The key hardware components are summarized in Table 2.1.

The Freescale iMX21 SoC forms the basis of the Chumby device. The different components of the SoC are shown in Figure 2.4. This SoC contains an ARM926 processor, an enhanced multimedia accelerator, a standard system I/O, an external connectivity interface, e.g. a universal serial bus (USB), and so forth.

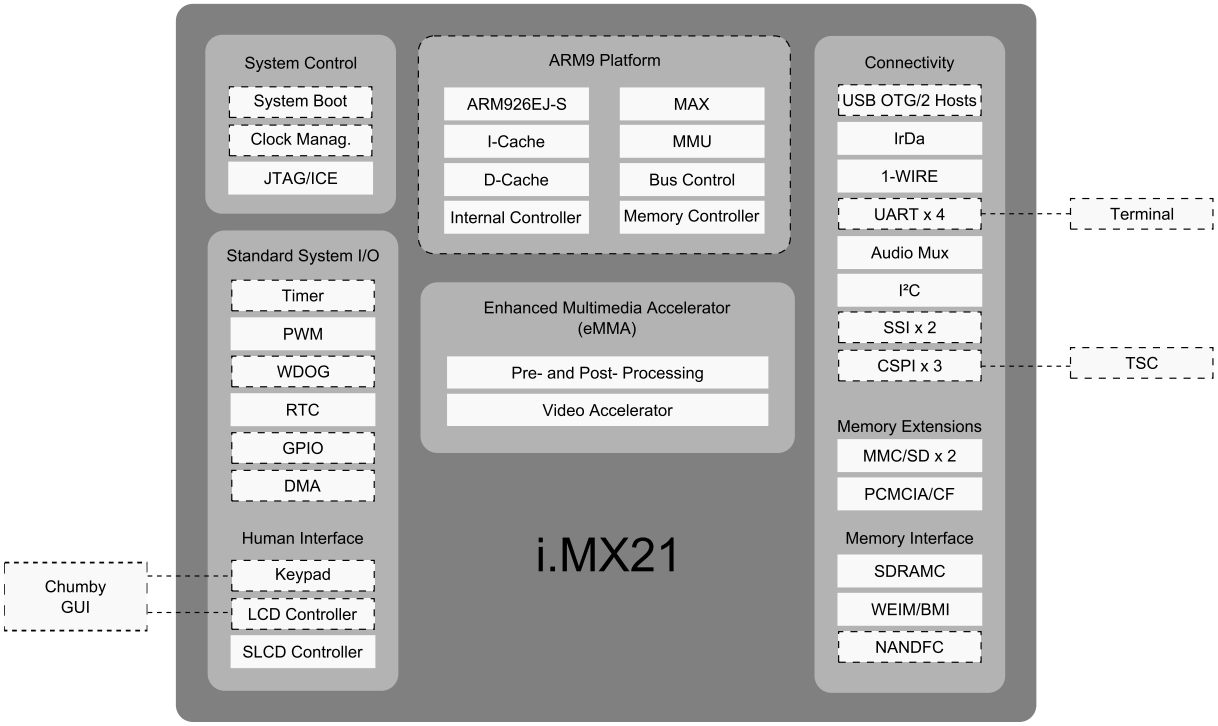


Figure 2.4: iMX21 diagram. Components with dashed lines have been modeled inside the VP

2.2.1.2 Software Architecture

The operating system (OS) running on Chumby devices is a modified Linux OS for embedded systems. In order to guarantee short booting times, a lightweight Linux kernel is used. As file system the *compressed ROM file system* (CRAMFS) is utilized, which is a read-only Linux file system designed for simplicity and space efficiency. The complete software stack is publicly available as open source with the exception of the Adobe Flash player, which is only available as binary. Because Flash is closed source and makes use of low level HW, debugging this part of the software stack is much more difficult than the rest.

2.2.2 Objectives of the Virtual Platform

The primary objective is to create a VP for this WiFi enabled Internet device Chumby. This VP should run sufficiently fast to be suitable for interactive software development and testing of the hardware dependent software. The detailed objectives will be described in following.

2.2.2.1 Simulation Performance & Accuracy

The heart of the Chumby device is the iMX21 [45] application processor which offers a rich set of features. In order to limit the modeling effort and the complexity of the VP only the parts of the iMX21 that are actually used by the Chumby device are modeled.

Besides the SoC also some external peripherals, e.g. the touch screen controller (TSC) and the LCD display, have been modeled.

As discussed in Section 2.1.2 it is important to find a good balance between simulation speed and accuracy for each VP. On the one hand, a sufficiently high simulation speed of the VP is required for the SW developers to test their applications. Especially for real time applications, e.g. a video player, the simulation speed is of great importance. On the other hand, the VP should be accurate enough, so that the simulated software shows the same behavior as on the real hardware. As a compromise between accuracy and simulation speed an *instruction accurate (IA)* processor simulator for the ARM926 processor has been selected. All the peripheral components are modeled at a very high abstraction level called programmers view (PV) for the sake of simulation speed.

2.2.2.2 Running Software on the Virtual Platform

In order to demonstrate that VPs are applicable and reliable for software development, the device is modeled in such a way that it can run the unmodified software. To interact with the software running on the Chumby device the touch screen needs to be modeled. In the simulated touch screen, the touch events are mimicked by mouse click events on the simulated screen. Since Chumby is an Internet device, it is important that the VP is capable of accessing the Internet. Details about the realization of the Internet access can be found in Section 2.2.3.

2.2.2.3 Connectivity

The Chumby device provides several routes to access the external world, e.g. the universal asynchronous receiver/transmitter (UART) and the USB port. One of the UARTs is configured in a way that a terminal device can be connected to it to monitor the kernel messages. Moreover, this terminal allows controlling the Chumby device after the booting of the OS is completed. In the context of the case study a terminal is connected to the simulated UART to monitor the boot process. Since the iMX21 does not provide any WiFi functionality, the Internet connection is realized by a WiFi USB stick connected to one of the USB ports inside the real device. In order to run the complete unmodified software stack, a special USB peripheral needs to be modeled inside the VP bridging between the simulated USB hub and the host machine's USB hub. Hence, the VP can communicate with a real WiFi USB stick connected to the host machine. The details of the USB bridge are presented in the following section.

2.2.3 Virtual Connectivity

With the increasing interconnection of embedded systems with other systems and networks, e.g. the Internet, it is important to also offer this possibility while simulating the embedded system using VPs. In the context of this case study the virtual connectivity is described using the Chumby USB connection to the WiFi adapter as an example.

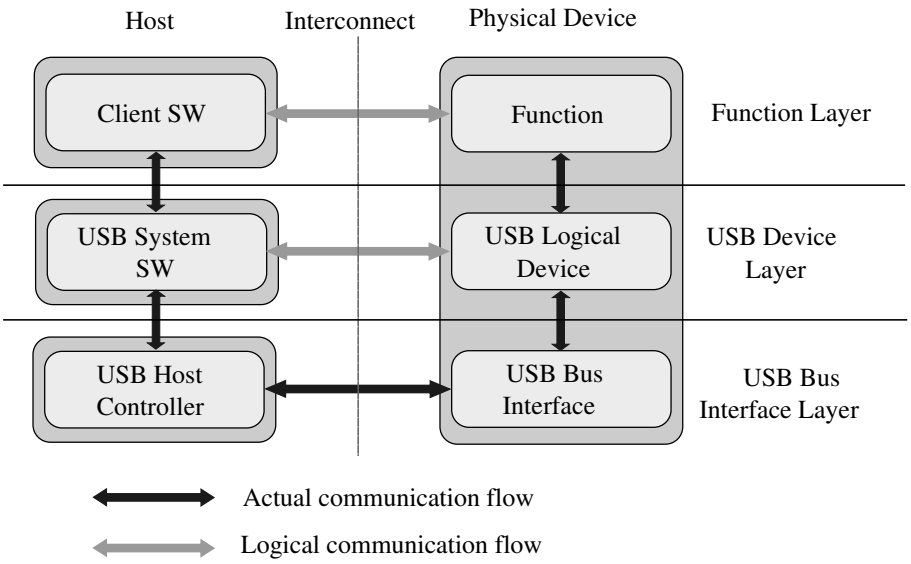


Figure 2.5: USB data flow model

2.2.3.1 USB Stack Architecture

USB provides a communication service between the client software on the host and its USB function on the device. The three different layers of the USB subsystem and their entities are shown in Figure 2.5 [165] and are briefly discussed in the following:

- *USB Physical Device*: A piece of hardware on the end of a USB cable that performs some useful end user function.
- *Client Software*: Software that executes on the host, communicating with the USB device. This client software is typically supplied with the operating system or provided along with the USB device.
- *USB System Software*: Software that supports the USB in a particular operating system. The USB System Software is typically supplied with the operating system, independently of particular USB devices or client software.
- *USB Host Controller (Host Side Bus Interface)*: The hardware that allows USB devices to be attached to a host.

The USB Bus Interface Layer provides physical/signaling/packet connectivity between the host and a USB device. The USB Device Layer provides the possibility to perform generic USB operations with the USB device. On top of this, the Function layer provides additional capabilities to the host via an appropriate client software layer. The Function layer and the USB Device layer perform a logical communication within their layer while the actual communication is using the USB Bus Interface layer to accomplish the data transfer.

2.2.3.2 Virtualization of USB

The goal of the virtualized USB is to directly access a USB device attached to the host machine by the VP. The two key challenges in modeling a virtual USB host are:

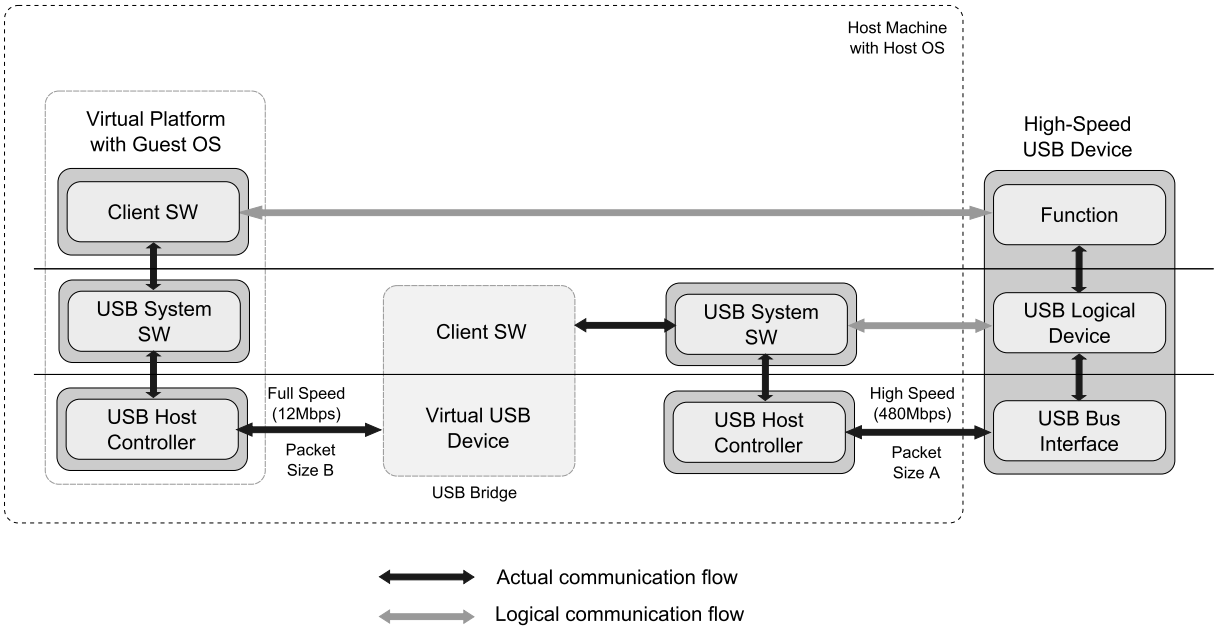


Figure 2.6: Realization of the virtualized USB connection

1. Implementing a communication mechanism between the VP and real USB devices connected to the host machine.
2. Efficiently modeling the data transfer between the host memory and the VP memory and emulating the device state.

Before both challenges are addressed it is important to first understand which parts of the USB functionality are provided by the USB hub of the host system. A USB hub provides an electrical interface between the USB device and the host and is responsible for the following tasks:

- Power management
- Device connect/disconnect detection
- Connectivity behavior
- Device support for high-, full-, and low-speed

Since parts of the USB functionality are already realized by the USB hub of the host machine, only a subset of the USB features need to be modeled inside the VP. For example, the physical communication interface and the power management are transparent to the software running on the VP and hence do not need to be included. Other functionalities like the detection of the device connect/disconnect events need to be modeled inside the VP. By default the host machine has full access rights to the USB device connected to it. In order to keep the changes on the host side minimal, the user manually creates the connection between the VP and the USB device via a command line interface after the device has been connected to the host.

The goal of the virtualized USB is to offer a connection to the USB hardware of the host system. In order to keep the approach as general as possible, no assumptions about the type of USB host controller used in the simulation host can be made. This can lead to a communication scenario in which a USB device configuration is not supported by the simulated USB host. For example, in case of the Chumby device the USB

host controller only supports the full- (12 Mbps) and the low-speed (1.5 Mbps) data transmission but the WiFi adapter also supports the high-speed (480 Mbps) data transmission. During the initialization of the communication the USB controller will select the fastest data transmission mode commonly available on transmitter and receiver sides. In case of the Chumby device the USB host controller selects the full-speed data transmission. However, the situation might be completely different when a USB device supporting high-speed transmissions is directly connected to the simulation host. In this case the USB host available on the host machine will select the appropriate communication speed for the USB device. This can lead to the WiFi adapter being connected to the host via a high-speed connection which is not supported by the simulated USB host. Since not only the data rate differs in both modes but also the packet size, it is not possible to directly move data packets between the simulated USB host and the USB system of the host. This problem can be solved by modifying the system software running on the VP. However, this modification would violate the goal of simulating the unmodified software using a VP. Therefore, a USB bridge is introduced to convert the data packets sent/received by the simulated USB host controller into the data packet format supported by USB system on the host machine as shown in Figure 2.6. This USB bridge increases the USB modeling effort, but it makes the VP independent of the USB hardware available on the simulation host.

2.3 Synopsis

- VPs are available early during the design process and hence allow the concurrent design of the hardware and the software of a system.
- Due to the fact that VPs are just software models of the hardware they can be easily distributed to the different software development teams. Furthermore, they can be easily modified and updated during the course of the development.
- In contrast to real hardware a VP can be halted at any time. Together with the possibility to access virtually all states of the simulated hardware this makes VPs a very good environment for debugging hardware dependent software, e.g. drivers.
- Virtual I/O is of great importance to VPs. Only by offering the possibility to interact with peripherals and other devices the software stack running on the VP can be tested in a realistic environment.

Chapter 3

Related Work

Generally speaking, simulation techniques always trade-off simulation speed versus simulation accuracy. In particular, the possibilities to increase the simulation performance are among others limited by the generality of the selected approach. For small and well defined application areas it is much simpler to create fast simulators because domain specific characteristics can be exploited. General approaches make only few assumptions about the application and hence can not achieve the same level of simulation speed as a specialized simulation technique. Over the past decades, many different simulation techniques have emerged, each with its own strengths and weaknesses.

In this chapter the related work in the field of processor simulation techniques is discussed. First, Section 3.1 classifies the most important simulation techniques and gives a brief description of each technique. All these techniques form the basis for the simulation of virtual platforms. In Section 3.2 a brief overview of the most important simulators and simulation frameworks is given, followed by a survey of the commercially available virtual platform environments in Section 3.3. As far as publicly available the used simulation techniques are also described.

3.1 Overview of Processor Simulation Techniques

The different processor simulation techniques proposed in literature can be classified into two major categories: *fundamental simulation techniques* and *composite simulation techniques*. The fundamental simulation techniques form the basis for all types of processor simulation, whereas the composite simulation techniques are built upon the fundamental simulation principles to further improve the accuracy or the simulation speed. The classification is depicted in Figure 3.1.

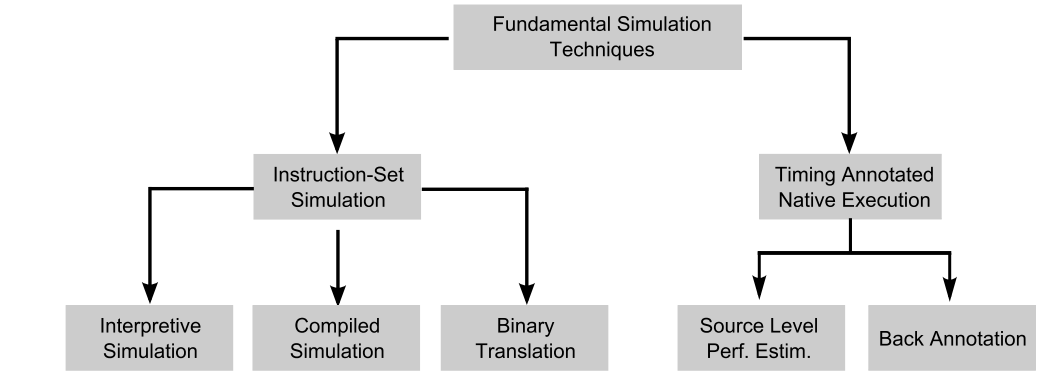
3.1.1 Fundamental Simulation Techniques

The fundamental simulation techniques can be further subdivided into two groups: *instruction-set simulation* and *timing-annotated native execution*.

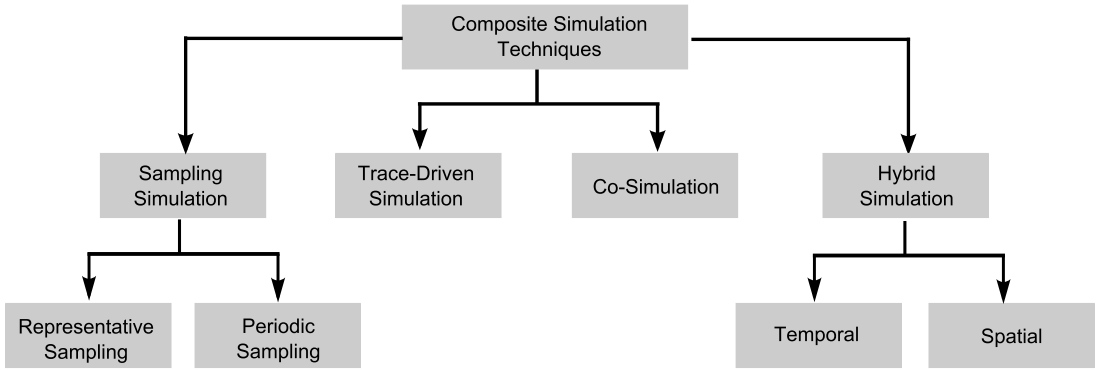
3.1.1.1 Instruction-Set Simulation

Instruction-set simulation (ISS) is widely used in the embedded domain for application simulation, microarchitecture development and profiling purposes.

Different realizations of the ISS based simulation have been developed over time. The most general solution is the *interpretive simulation*. With this technique it is possible to model a wide range of architecture/microarchitecture features. SimpleScalar [21] is a well known example of such an interpretive simulator. For maximum flexibility



(a) Classification of fundamental simulation techniques



(b) Classification of composite simulation techniques

Figure 3.1: Classification of processor simulation techniques

instruction fetching, decoding and execution are simulated. The drawback of interpretive simulation is the low simulation speed caused by the high overhead at runtime.

Compiled simulation has been developed to alleviate the simulation effort by pre-decoding instructions. This is based on the observation that during the simulation the instructions of programs are not subject to change and the results of pre-decoding are likely to be used multiple times. The pre-decoding can be performed either statically [22, 182] (i.e. at compile-time) or dynamically [29, 101, 112, 127, 129, 138] (i.e. , at runtime). The latter is also known as *just-in-time cache compiled* (JIT-CC) [112] or *instruction-set compiled simulation* (IS-CS) [128], which only pre-decodes instructions when they are actually used. It not only supports self-referential and self-modifying code, but also demands less decoding computation. A multi-processing approach [127] is proposed to benefit from the increasingly available multi-processor hosts. In this approach, one processor is used for interpretive simulation, while the other performs dynamic compilation. Thus, the latency of the dynamic compilation is hidden from the user.

The highest simulation speed can be typically achieved by *binary translation* [10,69, 179]. This technique directly translates the target binaries to host executable code. Like compiled simulation, binary translation can also be further classified into *static binary translation* and *dynamic binary translation*. Static binary translation tries to translate the complete application into host code. This approach is extremely difficult, since the

target address of indirect branches might only be known during runtime. Therefore, dynamic binary translation is much more common. Typically a short sequence of the application code, e.g. basic block, is translated into a sequence of host instructions. In order to amortize the translation overhead the translated sequence is cached for the next execution. The target addresses of branches are modified, so that they are pointing to addresses of the already translated code sequence in order to achieve high simulation rates.

To further improve the execution speed optimizations, e.g. inter-basic-block optimizations, are applied [69]. Binary translation is mainly used for instruction accurate simulations. It is less commonly used for cycle-accurate simulation, due to the high complexity of implementation and the difficulty of simulating microarchitecture features. The implementation complexity and the achievable simulation performance directly depend on the similarity of the target instruction-set and the host instruction-set.

3.1.1.2 Timing-Annotated Native Execution

Despite the fact that a lot of effort has been put into improving the simulation performance of ISS based simulations, the speed of directly executing natively compiled applications on the host computers is about two to three orders of magnitude faster. However, native execution does not provide any information about the timing of the application on the target platform. Hence, to estimate performance information through native execution, timing information of each basic block or function has to be assessed and annotated. The timing information can be obtained either through *source code analysis*, *binary analysis*, or *profiling*.

Karuri et al. [72] proposed the *micro-profiler*, which lowers the C source code of an application into an *intermediate representation* (IR) that features low-level operations similar to RISC instructions. In order to resemble more closely the application compiled with the target compiler, the IR is optimized by applying a set of machine-independent optimizations. A cost function is defined for each IR operation, and the costs for each basic block are obtained by accumulating the costs of all corresponding IR statements. An instrumenter inserts extra code for adding up the the estimated costs at each basic block so that the performance of the entire application can be calculated.

Similar to the micro-profiler approach, [62] also analyzes the timing information from the optimized IR. Additionally, [62] applies scheduling to the optimized IR to derive more accurate results. Since the timing information is analyzed directly from the source code, these approaches are also known as *source-level performance estimation*.

In contrast to source level performance estimation the approach proposed in [80] obtains the timing information by decompiling the application binaries to C code. The generated C code (instead of the original source code) can be compiled natively and executed on the host computers. This approach resembles closely a compile time binary translation that is enhanced with timing information. Instead of decompiling the binary, [18,97,139] analyze or simulate target binaries to obtain the timing information, and annotate it back to the original C source code or IR. However, mapping the timing information to the correct location in the source code or in the IR code is a very complex task. Especially compiler optimizations applied to the program can significantly deteriorate the mapping between source code and the target binary.

3.1.2 Composite Simulation Techniques

The composite simulation techniques are based upon the previously described fundamental simulation techniques and can be roughly divided into four groups: *sampling simulation*, *trace-driven simulation*, *co-simulation* and *hybrid simulation*. A comprehensive overview of many composite simulation techniques and benchmarks can be found in [176].

3.1.2.1 Sampling Simulation

Instead of simulating the complete application, sampling simulation selects only portions of the application's execution for detailed simulation. These portions are either selected *periodically* or *analytically*.

SMARTS [174] is a periodical sampling microarchitecture simulator. It utilizes functional simulation to fast-forward the application until the next sampling point is reached. After a warm-up phase for the simulated caches, a detailed cycle accurate simulation is performed on these samples. The obtained performance information is used to extrapolate the performance characteristics of the entire application.

In contrast to SMARTS, Sherwood et al. [143, 144] employ representative (analytical) sampling for performance estimation. The samples are selected by analyzing the similarity of execution traces represented by *basic block vector*s (BBVs). A BBV represents the execution frequency of all basic blocks in the application. It can be obtained from a functional simulation conducted during the preprocessing phase. These BBVs are then clustered into a set of phases by applying machine learning techniques. Henceforth, only one selected representative of each phase needs to be simulated in detail to estimate the overall application performance. Functional simulation or checkpointing [153] facilitate the fast forwarding of the simulation to these representative phases. Recently, sampling simulation has also been introduced in the multi-processor simulation domain [109, 122]. For the multi-processor sampling simulation the phase detection becomes more complex, since the global phases are formed of multiple threads running concurrently. Thus, relating the overall phases to the code fragments executed by the individual threads is very complex.

Sampling based simulation techniques are a very interesting concept which allows fast and accurate performance estimation of applications. They are widely used for microarchitecture design. However, they are not well suited for software development because of the lengthy preprocessing required for discovering the phases. For example, in case of the analytical sampling simulation utilizing checkpointing, the application has to be functionally simulated twice during the preprocessing phase. Once, to discover the representative phases, and then again to create checkpoints before each phase. This kind of preprocessing is time consuming, and each time the application's source code is modified or the compiler's optimizations are changed the complete preprocessing needs to be repeated again. Hence, this simulation technique is not suitable for VP based software development due to high overhead introduced by the preprocessing phase. Furthermore, the user cannot control which part of the application is simulated in detail and which not. Especially for software debugging this behavior is problematic.

3.1.2.2 Trace-driven Simulation

Another commonly used technique is the *trace-driven simulation*. It is utilized to evaluate performance [172], power consumption [164] or memory access behavior [79] of computer systems. A trace is a sequence of system events, e.g. program counter changes or memory accesses, generated from simulation using one of the fundamental simulation techniques (usually functional simulation). After the trace generation, analysis tools are applied to process the traces in detail and derive the desired information. The major problem with trace-driven simulation is that the generated traces often become excessively large. In [40] a technique is proposed to generate short synthetic traces showing the same characteristics as the original traces of applications. For software development trace-driven simulations are not broadly adopted due to the fact that performance information only becomes available after post-processing the trace. In [82] a functional simulator is used to generate a trace which serves as an input for a microarchitectural simulation.

3.1.2.3 Co-Simulation

Co-simulation is mainly used in HW/SW co-design for the simulation of the complete system. Heterogeneous environments [13,177] achieve this by coupling ISS simulators with hardware description language (HDL) simulators, e.g. VHDL or Verilog simulators. The main problem of this approach is the limited simulation performance due to synchronization overhead introduced by the coupling of different simulators. In contrast, homogeneous approaches circumvent the synchronization problem by simulating the software and hardware aspects of the system within a single software environment. Besides the purely software based co-simulation approaches, hardware supported co-simulation approaches [42,171] are also used, in order to achieve higher simulation speed. The hardware supported co-simulation can be subdivided into two different approaches: The usage of dedicated hardware for efficient HDL simulation [42] and the utilization of FPGA based hardware for efficient datapath modeling [171].

3.1.2.4 Hybrid Simulation

In general, hybrid simulation combines two or more different simulation techniques into one simulation environment in order to benefit from the different simulation characteristics, e.g. different levels of accuracy and simulation speed. Hybrid simulation can be classified into two groups: *spatial* and *temporal* hybrid simulation.

Spatial hybrid simulation offers the user the possibility to simulate processors at different levels of abstraction. During design-time of the simulation the developer decides for every processor instance if an abstract simulation is sufficient or if a detailed simulation is necessary. During later design stages the user can then consecutively replace the abstract processor models by specific processor instances. Prominent examples of spatial hybrid simulation are the *virtual processing unit* (VPU) from Kempf et al. [74] and the concept of virtual architecture [68, 178]. Especially for early design space exploration these approaches have proven to be very helpful by using abstract processor models in order to simulate the complete system. It is important to note that VPUs and virtual architectures themselves do not provide spatial hybrid simulation.

Only in combination with other processor simulation techniques, e.g. ISS based, the overall simulation platform offers spatial hybrid simulation.

In contrast to spatial hybrid simulation temporal hybrid simulation offers the user the possibility to switch the simulation technique, i.e. the level of abstraction, during runtime. Hence, based on the actual needs the user can select the appropriate simulation technique. In [83] a hybrid simulation technique is presented which is capable of switching between a detailed CA simulator and a simplified simulation model (functional simulation plus cache simulation) in order to speed up the simulation without sacrificing the accuracy. The first simulation of each basic block is performed with the CA simulator, whereas the later encounters are performed with the simplified simulation model. Muttreja et al. propose a hybrid simulation technique [107, 108] for tackling the performance/energy estimation problem of single processors. In their solution some parts of an application are executed on the native host machine, whereas the rest runs on an ISS. Since native execution is much faster than the ISS, significant simulation speed can be achieved if the natively executed parts are the most frequently executed pieces of code. The performance estimation for the natively executed code requires a training phase to build a function level performance model, and therefore, is not flexible to easily account for software modifications.

At a first glance, sampling based simulation and hybrid simulation seem to be quite similar, because both techniques use two different simulation approaches for a fast but accurate simulation. However, in sampling based simulations the switching point cannot be defined by the user, whereas in most hybrid simulations the user can directly control the switching between the simulation modes. Hence, hybrid simulation techniques are more suitable for application debugging and software development, since they offer the developer the possibility to better control the switching.

3.2 Simulators and Simulation Frameworks

This section presents the most widely used simulation frameworks used in academia for microarchitecture development as well as for full system simulation. Each framework is briefly described with special focus on the simulation techniques that are employed. A comprehensive overview on different simulators and simulation technique used in academia and industry can be found in [157].

Asim – The C++ based, modular simulation framework Asim [41] has been developed by Intel. It enables the user to describe the processor performance model by combining a set of simulation modules. In order to reach high simulation performance the environment uses an event based simulation kernel. The different tools for managing and debugging the the performance models are combined into the *Architect's Workbench*. The configuration of the simulator can be modified either by a GUI-based, interactive tool or manually by directly editing the configuration files.

COTSon – The COTSon simulation framework [7, 43] is jointly developed by HPLabs and AMD. It targets cluster based systems composed of hundreds of commodity processors connected by a communication network.

The COTSon simulation framework is based on the *functional directed* philosophy, which combines functional emulation and timing models for accurate and sufficiently fast simulation to simulate the complete software stack including the OS. For functional simulation AMD's *SimNow* simulator is utilized. In order to be able to simulate clusters with up to 1000 cores the traditional cycle by cycle simulation paradigm has been abandoned in favor of a statistical sampling approach that allows to trade accuracy for speed. The simulation environment is freely available [1] under the MIT open source license.

GEMS – In the context of the multifacet project of the Wisconsin University the general execution-driven multiprocessor simulator *GEMS* [93] is developed. GEMS is designed as a modular simulation infrastructure that decouples the functional and the timing aspects of the simulation. Based on this *timing-first* simulation approach, the full system simulator *Simics* [90, 168] from Virtutech is used for the functional simulation of the system and the *Ruby* memory system simulator is used for the detailed timing simulation of the memory subsystem. To further enhance the timing fidelity for complex processor models Opal [94] is included as detailed processor timing model. This simulation framework is available under GNU public license (GPL) [106], however, a license for the commercial Simics simulator is required.

M5 – The M5 simulator [15] is developed in the context of researching architectures for high-speed TCP/IP network I/O. The full system simulation capabilities provided by the M5 simulator enable the simulation of the complete software stack required for the development of TCP/IP network I/O. Although initially developed for network research this simulation framework can be utilized for microarchitecture and memory research. The framework is C++ based and uses Python for the configuration of the simulator. Due to the strict modular approach it can be easily extended. The M5 simulator is available under an Berkeley-style open source license [160].

PTLSim – The PTLSim simulation [179] framework is a fully cycle accurate full system simulator for x86-64 processors. It models the microcode of the processor, the complete cache hierarchy and the memory subsystem. As a special feature this tool offers the possibility to switch at any time between the full out-of-order simulation mode and the native x86-64 mode. Since the host computer is also using the x86-64 instruction set the switching between both simulation modes can be easily realized. The multi-processor and multi-thread support is realized using KVM/QEMU [12]. The simulator is available under the GPL [126]. Based on PTLsim the *MARSSx86* [92] simulator provides additional support for multicores, coherent caches and on-chip communication.

QEMU – QEMU [12] is an open source machine emulator capable of full system simulation. By using binary translation a high simulation performance is achieved. QEMU solely concentrates on simulation performance and functional simulation, there is no support for timing models.

SimFlex – The SimFlex project [54, 145] provides a full system simulation framework called *Flexus* that builds upon Virtutech's Simics simulator. Similar to the COTSon simulation framework the Flexus framework provides accurate timing models of processors, memories and interconnects. The timing information obtained by those models is used to control the functional simulation. This approach is combined with the statistical sampling approach SMARTS [174] to evaluate microarchitectural properties, e.g. cycle per instruction (CPI) , with the desired accuracy and confidence.

SimNow – The SimNow [11] instruction level simulator developed by AMD provides a configurable full system simulation for x86 and x86-64 multiprocessor platforms. In order to achieve a high simulation performance, binary translation is applied to translate the simulated instructions into a sequence of host instructions. With this approach a slowdown of only $10\times$ with respect to the native execution is achieved.

SimOS – The operating system simulation environment SimOS [132] is specifically designed as testbed for the evaluation and the development of operating systems. The framework provides full system simulation using functional simulation in order to achieve a high simulation performance. Furthermore, it is possible to switch between a functional processor simulator and a detailed processor simulator for accurate timing analysis. In order to store the simulation state a checkpoint/restore framework is included in SimOS.

SimpleScalar – The processor simulator SimpleScalar [21] is widely used in academia for microarchitecture development due to its configurability and extendability. It provides different simulation models, ranging from a fast functional simulation model to a very complex and detailed out-of-order processor model.

SimSnap – The SimSnap [153] framework employs a combination of *application level checkpointing* (ALC) and native execution to fast-forward the simulation. The application's source code is instrumented by the Cornell Checkpoint Compiler C^3 [19,20] in order to save the state of the program at a given point. In the fast-forwarding mode the compiled application is natively executed on the host machine and the checkpointing capability is used to transfer the application's state into the simulator at a switching point. Currently, the proposed approach works only if the instruction set architecture (ISA) of the simulator matches the ISA of the host machine. Ringenberg et al. [130] suggest to use *Intrinsic Checkpointing* to store the state of a simulation. In

contrast to conventional ALC, Intrinsic Checkpointing modifies the application binary itself. Hence, this approach is independent of the availability of the application's source code.

3.3 Commercial Virtual Platforms

The growing complexity of MPSoC platforms combined with the increasing amount of software running on such systems has led to a broad set of new development tools. In this context, the use of VPs for system simulation has gained a lot of attention. As described in Chapter 2, VPs offer the possibility of pre-silicon software development, design space exploration and performance characterization. Over the last decade a new design methodology called *electronic system-level* (ESL) [9] design has evolved to cope with the increased design complexity which cannot be handled efficiently at the RTL level. Several companies have developed *electronic design automation* (EDA) tools to support the user throughout the complete design process. This section focuses on the VPs solutions offered by these companies. Each company and VP product are briefly presented in the following:

Cadence – The *Incisive* [23] design environment from Cadence allows the embedded system designer to model the system architecture using SystemC. The simulation environment is SystemC based and supports the TLM-2.0 modeling approach. A checkpoint environment based on process checkpointing is provided to store the complete simulation state. Furthermore, the *C-to-Silicon* compiler can be used to generate RTL code from C source with low effort. This is especially important for the creation of precise power models that can be incorporated in the system simulation. For the low level RTL based simulations, it is possible to utilize hardware emulation to increase the simulation performance.

Carbon – The *SoC Designer* [24] framework from Carbon allows to design, simulate, debug and analyze VPs. The simulation kernel is based on SystemC and supports the TLM-2.0 modeling approach. In order to ensure a seamless transition between VP design and RTL design, *Carbon Model Studio* offers the possibility to generate fast cycle accurate processor simulators directly from RTL code. Furthermore, a checkpoint based approach is used to provide fast but full hardware accurate behavior of the simulated application. If the behavior of the simulation has already been simulated in the past then the results are replayed without the need for detailed simulation. Only in case of an unknown state a detailed simulation is necessary.

CoWare[1] – The Virtual Platform [33] environment from CoWare can be divided into two parts: The *Platform Architect* for the VP development and the *Virtual Platform Analyzer* for debugging and analyzing the SW running on the VP. The complete simulation environment is based on SystemC, using an optimized SystemC implementation for fast simulation. The usage of SystemC

[1] Acquired by Synopsys

makes it easy to integrate processor simulators into the system simulation by the means of SystemC wrappers. Especially, the ISS based simulators generated by the *Processor Designer* using the LISA language [58, 59] can be seamlessly plugged into the simulation environment. Furthermore, the simulation environment provides the possibility to create checkpoints and to restore the state of a simulation at a later point in time (for more information about checkpointing refer to Chapter 5). It is possible to script the platform creation as well as the simulation environment in order to customize the simulation environment to the needs of the user and to automate tests.

Imperas – The simulation environment from Imperas [64] utilizes a proprietary simulation kernel based on binary translation for high simulation speed. It provides the possibility to incorporate SystemC modules by means of wrapper modules. In 2008 Imperas started the *Open Virtual Platforms* (OVP) initiative [117] and made the simulation models available as open source. These models can be used together with the simulation kernel that is freely available as binary.

Mentor Graphics – The *Vista Architect* [96] design environment from Mentor Graphics allows the SoC designer to model the system architecture of a SoC. Furthermore, it can be used for debugging and analyzing the created VPs. In contrast to other VP environments it is possible to define power policies. Based on these policies, the inner states of a module and the incoming traffic it is possible to derive an early power estimation. The entire simulation environment is SystemC based and uses the TLM-2.0 modeling approach. This design environment can be combined with a high level synthesis approach called *CatapultC* which can be used to directly generate RTL and TLM based SystemC models from a C/C++ model. Thus, it possible to iteratively refine the power model of the VP based on the RTL information.

Synopsys – The integrated virtual platform development environment *Innovator* [151] from Synopsys is based on SystemC. This environment is used for the embedded system design to efficiently develop, run and debug VPs. In conjunction with the *DesignWare System Level Library* it is possible to quickly assemble VPs. This library contains a rich set of predefined building blocks, such as various processor models, buses, interconnects and peripherals, which are all TLM-2.0 compliant. All of those building blocks are modeled at different levels of abstraction, e.g. loosely timed, approximately timed and cycle accurate, allowing the developer to choose the correct abstraction level for his platform.

Tensilica – The Tensilica [158] Xtensa processor is a configurable and extensible application specific instruction set processor (ASIP), that can be tailored towards the needs of the system developer by configuring various parameters, e.g. cache size, size of the register file. To further customize the processor the developer can add user defined extensions that are written in the

Tensilica instruction extension (TIE) [135] language. With the help of this instruction set specialization the performance and energy efficiency of the processor can be greatly improved. The ISS of the Xtensa processor provides two simulation modes: cycle accurate simulation and a fast functional simulation called *TurboXim* [133]. The TurboXim mode is realized by a kind of binary translation called *just-in-time host based code generation*. It is possible to dynamically switch between both simulation modes. This enables the implementation of sampling based simulation approaches for fast but still accurate simulations. Two different system level modeling environments are provided: *XTMP* for C based modeling and *XTSC* for SystemC based modeling. Due to the higher complexity of the XTSC environment the systems modeled in XTMP usually show higher simulation speed.

VaST[1] – Two different tools are provided by VaST [167]: *CoMET* is used for the development of a VP and *METeor* is used for pre-silicon software development. The visualization of the different characteristics and parameters of the VP is performed by the *Metrix* tool, which can be easily configured to the user's needs. A highly optimized simulation kernel based on a proprietary technique is used to simulate complete systems at a cycle accurate level. In order to increase the flexibility it is possible to include SystemC modules into the simulation. Furthermore, fast processor simulators can be modeled using the *Virtual Processor Model-Transformer* (VPM-T) and included into the system simulation.

VirtuTech[2] – The *Simics* [90,168] simulator from VirtuTech is based on a proprietary simulation kernel which offers high simulation speed but limits the possibility to include simulation models from other vendors. Therefore, Simics offers a special SystemC bridge to include standard SystemC modules into the simulation. The simulation environment supports *reverse debugging* and simulation checkpointing. In order to maintain a high simulation speed for complex systems it is possible to distribute the simulation over different processor cores by exploiting the multi-host simulation capabilities of Simics. For modeling peripheral components a *Device Modeling Language* (DML) is used to simplify the modeling process. The simulation environment provides a scripting interface to automate and customize the simulation to the needs of the user. Furthermore, the simulation environment allows injecting faults into the simulation in a reproducible and systematic way.

Table 3.1 shows a summary of the features offered by the different commercial VP environments. Although not all of the environments are based on the SystemC simulation kernel, all of them offer the possibility to incorporate SystemC models for interoperability reasons.

[1] Acquired by Synopsys
[2] Acquired by Intel

Company	SystemC based	SystemC support	Checkpoint/ Restore	Miscellaneous Features
Cadence	✓	✓	✓[b]	C to RTL behavioral synthesis
Carbon	✓	✓	✓	Cycle accurate SystemC generation from RTL code
CoWare VPA	✓	✓	✓[b]	
Imperas	-[a]	✓	-	Open source IP blocks
Mentor Graphics	✓	✓	-	Early power estimation, C to RTL behavioral synthesis
Synopsys Innovator	✓	✓	-	
Tensilica	-[a]	✓	-	Switching between cycle accurate and instruction accurate simulation
VaST	-[a]	✓	-	
VirtuTech Simics	-[a]	✓	✓	Reverse execution, reverse debugging, multi-host simulation, fault injection

Table 3.1: Overview of commercial virtual platform environments

[a] Proprietary simulation kernel
[b] Based on process checkpointing

3.4 Code Instrumentation Frameworks

Code instrumentation plays a central role in the hybrid simulation approach developed in this thesis (see Chapter 6). Furthermore, code instrumentation is of big importance for characterizing the runtime behavior of an application as described in Section 3.1.1. In general, two different approaches can be identified: *binary instrumentation* [89, 110] and *source level instrumentation*. Since the latter approach is used for the hybrid simulation presented in this thesis, a brief overview of two different source level instrumentation frameworks is given in the following.

3.4.1 LANCE

LANCE [84] is a software environment for implementing C compilers. It comprises a C frontend for the generation of the intermediate representation (IR), a C++ based library to interact with the IR and a compiler backend interface for assembly code generation. Furthermore it provides a set of standard IR-level code optimizations, e.g. constant propagation, dead code elimination and common subexpression elimination. The IR is represented as a set of three address code statements, which leads to an assembly like, but machine independent code. The distinctive characteristic of LANCE is the fact, that the IR is represented by a subset of the language C. Hence, it is possible to directly compile and execute the IR code. This feature makes it very easy to create a source level instrumenter using LANCE. However, due to limitations in the frontend it is only possible to process C89 compliant applications with the LANCE environment.

3.4.2 Low Level Virtual Machine

During the last years the low level virtual machine (LLVM) compiler infrastructure [78] has gained a lot of interest due to the rich set of features it provides. The frontend

transforms the application source into a RISC like, machine independent IR in *single static assignment* (SSA) [105] form. One of the specific characteristics of the LLVM IR is that it preserves the high level information of the application, e.g. types and data flow information. Hence, this IR is suitable for performing complex transformations exploiting the preserved high level information.

LLVM provides support for various input languages, e.g. for C/C++ a gcc-based frontend is used. Furthermore, it provides lots of different backends for transforming the IR into machine specific assembly code. For the implementation of a source level instrumenter a special backend is used to generate portable C code from the IR. A unique feature of LLVM is the availability of a JIT-compiler to directly execute the IR code without the need for creating a native binary. Compared to other compiler frameworks it provides a set of powerful optimizations, e.g. link-time optimizations, run-time optimizations and compile-time profile-driven optimizations. Those optimizations techniques lead to very efficient code.

3.5 Synopsis

- ISS based processor simulation techniques have been pushed to their limits. Higher simulation speeds can only be achieved using composite simulation techniques.
- Most composite processor simulation techniques have been developed for characterization and performance analysis of microarchitectural decisions. In general, these techniques are not well suited for software debugging.
- Based on the observation that software developers do not need the same level of accuracy during the entire simulation of their application, hybrid simulations are a good match for pre-silicon software development. In contrast to sampling based simulations they give the user the possibility to control the switching between the different simulation modes. Moreover, no time consuming preprocessing is required.
- Source code instrumentation based approaches are used to model the performance characteristics of the target system without sacrificing the simulation performance of the direct host execution.
- Over the last years SystemC has been established as de-facto standard for modeling VPs, due to the good interoperability between different vendors.

Chapter 4

SHAPES – A Heterogeneous Scalable Multi-Tile Platform

Since the SHAPES platform serves as the main test platform for the new simulation techniques developed in this thesis, the different aspects of this platform will be highlighted in the following. More specifically the hardware aspects, the software toolchain and the main driver applications are presented.

Heterogeneous multiprocessor systems-on-chip (MPSoCs) are becoming a viable alternative to traditional monolithic processor systems for implementing embedded streaming applications. The challenge is to identify a scalable HW/SW design style for future CMOS technologies enabling a high gate count [4, 35, 150]. The two major problems in deep sub-micron technologies are the minimization of wire delay problems [25, 57] and the management of the ever increasing design complexity. The tiled architecture [119, 156] approach addresses these problems by using a high number of simple processing tiles. The simplicity of the individual tiles leads to a minimization of the wire delay problem. In order to achieve the necessary computing power, a large quantity of tiles is combined into a computing system. The inter-tile communication is realized by weaving a distributed routing fabric for on-chip and off-chip packet switching. Due to the tiled nature this approach is considered to be highly scalable and very suitable for highly parallel applications.

The goal of the SHAPES (scalable HW/SW platform for embedded systems) project [120, 142] is to design a Teraflops board, that can be used as a building block for Petaflop-class systems. Due to its flexibility and scalability, the SHAPES architecture is intended to cover the entire range from the commodity market applications (4–8 tiles), classical signal processing application such as audio/video processing, ultra-sound and radar (2 K tiles) up to the numerically demanding, massively parallel scientific simulations (32 K tiles). Figure 4.1 shows the structure of a SHAPES multi-tile platform. In order to harvest the computing power offered by such a platform not only the hardware but also the corresponding software toolchain needs to be considered.

4.1 SHAPES Hardware

The main hardware building block of the SHAPES project is a dual-core tile consisting of a general purpose RISC processor and a VLIW DSP. This tile is called a RISC-DSP tile (RDT). Figure 4.2 depicts the block diagram of such a tile and its key components. An ARM926EJ-S serves as general purpose processor for control dominated code, while a mAgic [121] VLIW processor is used for efficiently executing the signal processing parts of an application. During the development of the mAgic DSP care was taken that

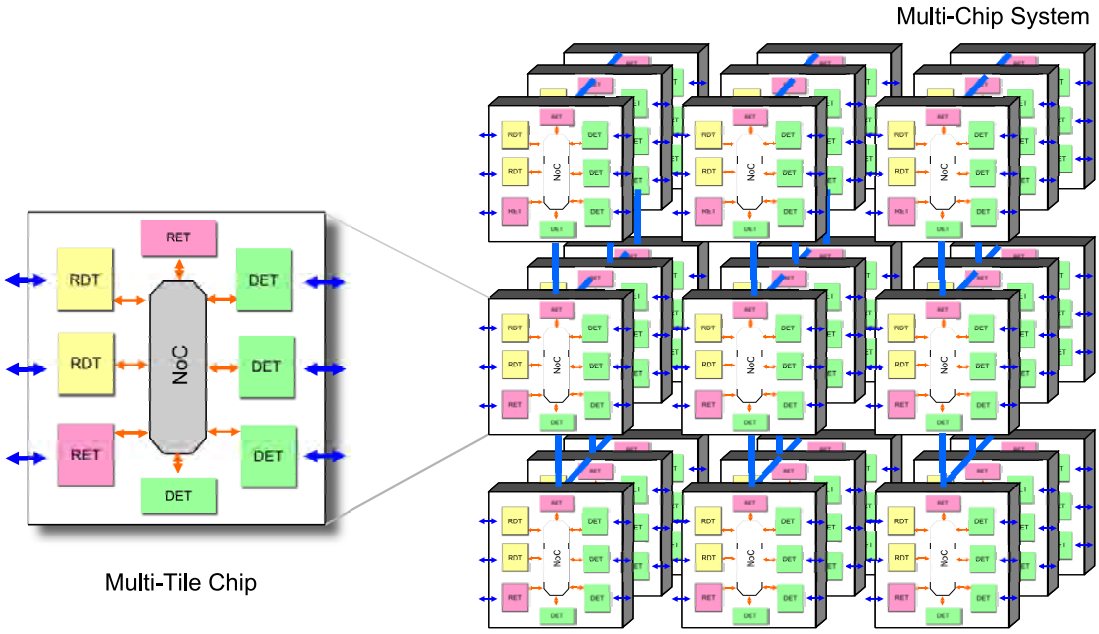


Figure 4.1: Multi-tile SHAPES platform

the processor is fully C programmable without losing performance. The DSP architecture is capable of performing 10 floating-point (40 bit precision) operations in parallel. This high degree of parallelism leads to a peak performance of 1 GFlop at a frequency of 100 MHz (For comparison, the well known processor TMS320C6713B [163] from Texas Instruments performs 1.8 GFlops at 300 MHz). Figure 4.3 shows the structure of the mAgic floating point DSP.

All components including the on-tile memory and the processing elements are connected using a multi-layer AMBA bus for high throughput. Each RDT tile also provides a rich set of peripherals that is referred to as peripheral on tile (PoT) block. The individual peripherals are connected via an AMBA peripheral bus (APB) that in turn is connected to the multi-layer bus via a bridge.

The most prominent peripherals are: timer, interrupt controller, universal synchronous/asynchronous receiver transmitters (USARTs), serial peripheral interface (SPI) and peripheral DMA controller (PDC). In order to efficiently map a wide range of applications to the hardware, a set of different tiles is proposed. All tiles are stripped-down versions of the RDT tile described above, i.e. it is possible to customize the tiles based on the characteristics of the application intended to run on a system. Table 4.1 gives an overview of the different tiles and their respective configuration supported by the SHAPES project.

The RDT tile of the SHAPES project is based on the DIOPSIS MPSoC from Atmel [8]. However, in contrast to the commercially available DIOPSIS platform the SHAPES RDT also incorporates the *distributed network processor* (DNP) for inter-tile communication. A detailed description of the DNP can be found in Section 4.1.1.

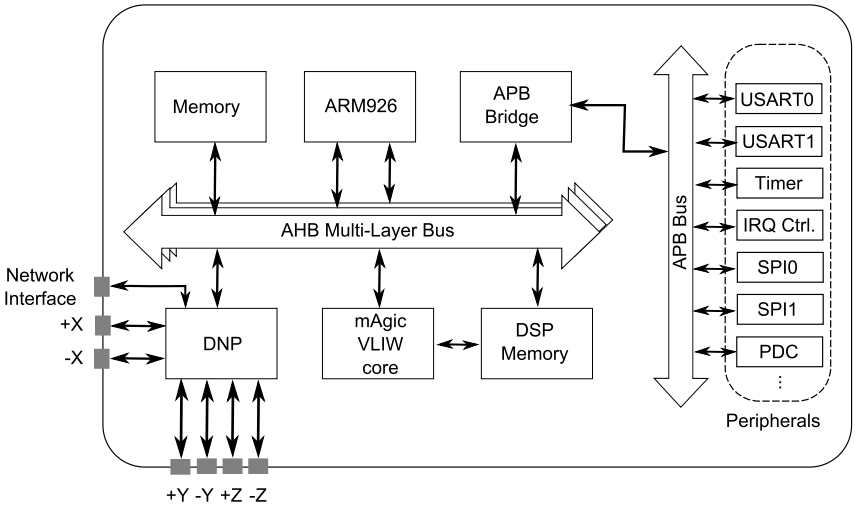


Figure 4.2: Schematic of the RISC DSP Tile

	ARM	DSP	Periph.	Mem.
RISC Tile (RET)	✓	-	✓	✓
DSP Tile (DET)	-	✓	✓	✓
RISC DSP Tile (RDT)	✓	✓	✓	✓
Link Tile (LET)	-	-	✓	-
Memory Tile (MET)	-	-	-	✓

Table 4.1: Overview of the different tiles supported by SHAPES

In order to satisfy a given performance requirement, it is possible to combine a large number of tiles into a multi-tile computing system. This multi-tile system is realized by placing several RDT tiles on a single multi-tile chip. Further on, the multi-tile chips can then be arranged in a 3D mesh topology to build highly scalable computing systems. In case of a multi-tile chip, a network-on-chip (NoC) is used as on-chip interconnect. The individual tiles then use their DNPs to access the NoC for on-chip communication and the DNP's off-chip ports for off-chip communication. Due to this modular design it is very simple to customize the computing environment for the needs of the individual applications. Based upon the application specification and the field of application the exact number of multi-tile chips and the mixture of tiles can be determined.

4.1.1 Distributed Network Processor

The inter-tile communication is realized by the distributed network processor (DNP), which serves as a generalized DMA controller. Thus, the packet processing required for the inter-tile communication can be offloaded from the computing resources to the

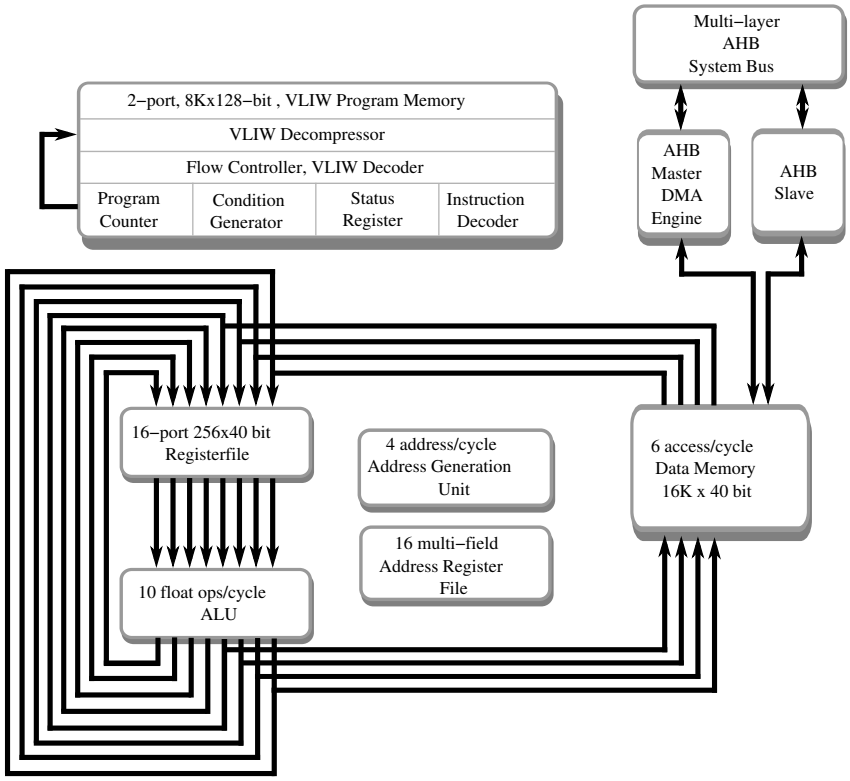


Figure 4.3: Schematic of the mAgic processor [8]

DNP. The DNP architecture is a packet-based crossbar switch with configurable routing capabilities that can be integrated into MPSoCs for on-chip and off-chip communication.

The on-chip inter-tile communication is realized by connecting the DNP to a NoC using the DNP-network interface (NI). On the left side of Figure 4.1 the inter-tile communication on a multi-tile chip is shown. The on-chip communication is realized by connecting the DNP's NI to the Spidergon NoC [31, 32], developed by STMicroelectronics. The Spidergon NoC is based on a regular, scalable point-to-point topology. Packet based communication is performed using *deterministic routing* and *wormhole switching* [34] for efficiency reasons. It connects an even number of nodes $N = 2n$ in a bidirectional ring enriched with cross links between opposite nodes. Figure 4.4 shows an example of the Spidergon topology and the corresponding chip layout. Each node is connected via three unidirectional links: a clockwise link, a counter clockwise link and an across link. In general, the degree of a node determines the number of neighboring nodes that a particular node is connected to. High degree nodes reduce the average path length at the cost of a higher complexity. Spidergon networks – with a degree of 3 – form a good compromise between a ring and a 2D mesh in terms of network diameter, average network distance and complexity [17]. Furthermore, due to its regular nature only a simple HW router is required, resulting in an efficient HW implementation of the NoC. This is an important characteristic for a NoC, since the available area for communication on a chip is limited.

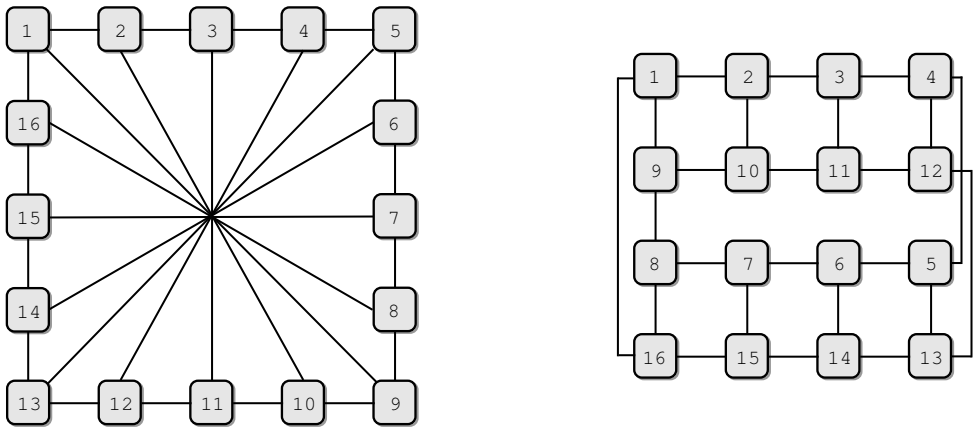


Figure 4.4: Topology of a Spidergon interconnect and chip layout

The off-chip communication is realized via 6 serial off-chip links (+X, -X, +Y, -Y, +Z, -Z), which form a 3D torus topology with the other multi-tile chips as shown on the right side of Figure 4.1. Each of the DNPs knows its position in the 3D torus and thus is able to route the incoming data packets to the destination tile. By using the DNP, the inter-tile communication has been made explicit, since each tile maintains its own memory address space. Therefore, also the programming model needs to make the communication between tiles explicit. Combined with the heterogeneity of a single tile, the usage of a powerful software toolchain is mandatory in order to fully exploit the characteristics offered by this kind of hardware. The SHAPES software toolchain and the realization of the inter-tile communication are presented in Section 4.2.

4.2 SHAPES Software Toolchain

As described earlier, tiled architectures offer a good way for implementing scalable high-performance platforms. However, the characteristics of tiled architectures impose a set of restrictions and challenges on the software development process. In order to develop applications that are capable of fully exploiting the potential offered by tiled architectures, a powerful software toolchain is required. The purpose of such a toolchain is to minimize the effort of the application developer. At the same time the system software must be fully aware of the architectural parameters such as interconnect bandwidth, latency, computing capability and availability of memory.

This clearly indicates that the traditional software development flow is not suitable for heterogeneous multi-tile DSP architectures. Therefore, a new design flow is required for this type of architecture. The new flow needs to cope with the high complexity of the hardware while accommodating the short development time for applications. This flow is facing two contradicting requirements. On the one hand the programming environment is expected to relieve the application developer from the burden to deal with the hardware details, but on the other hand accurate feedback about the system performance is required in order to control and steer the development process. Apart form this, the toolchain must also be suitable for continuous and

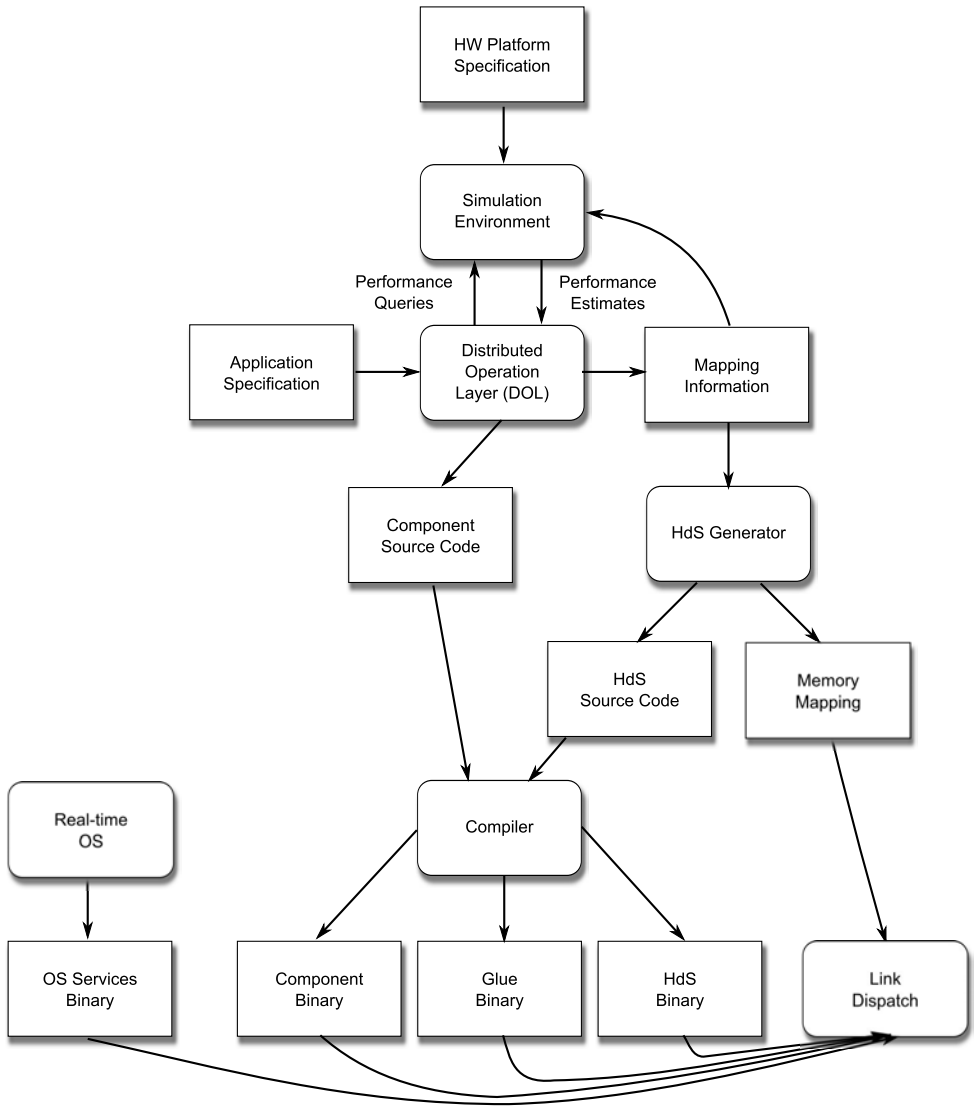


Figure 4.5: Overview of the SHAPES software toolchain and the interaction of the different components [120]

iterative QoS-aware designs, taking the underlying hardware into account as well as the application's structure.

The individual components of the SHAPES toolchain are: the distributed operation layer, the hardware dependent software, the compiler and the simulation environment. Figure 4.5 gives an overview of the different components of the toolchain and illustrates their interaction.

In the following the major components of the toolchain and their interaction are described.

4.2.1 Distributed Operation Layer

The distributed operation layer (DOL) [53, 61, 161] serves as system level framework which processes the application specification and acts as a model compiler. The key

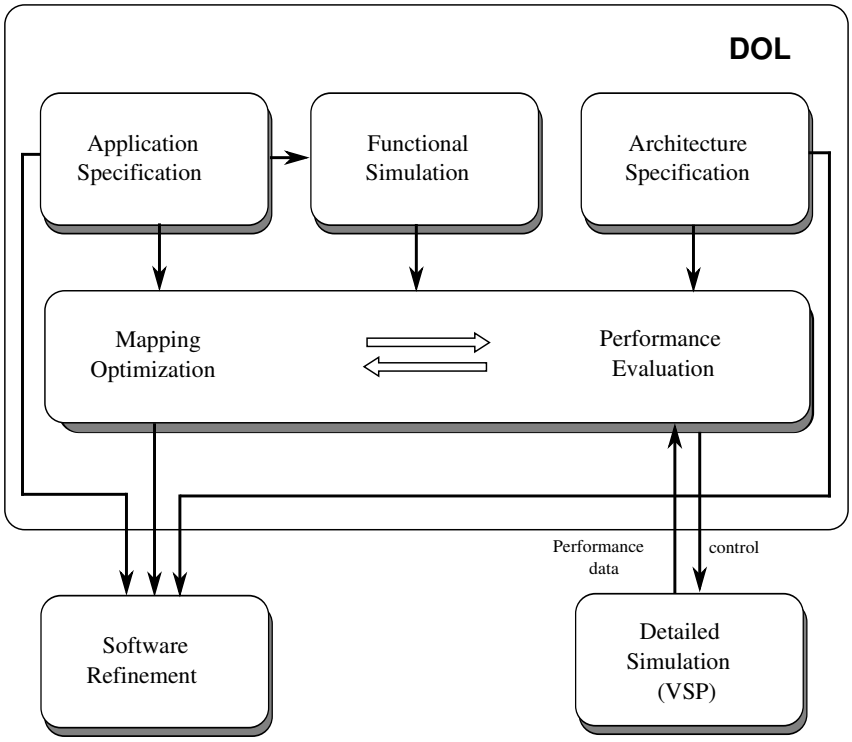


Figure 4.6: Overview of the DOL framework [161]

elements provided by the DOL environment are a programming model, a communication interface, a hardware abstraction, a system level performance analysis and a framework for high level design space exploration. An overview of the DOL framework is depicted in Figure 4.6.

As shown in Figure 4.5 the application specification serves as the main starting point for the software design flow and captures the functional and non-functional aspects of the intended application. The DOL programming model is based on a Kahn process network (KPN) [71], which allows to explicitly model the application tasks, their interaction and the available degree of parallelism. Each task is written in conventional C/C++, which has the advantage that state-of-the-art compilers and their respective optimizations can be applied to the machine code generation. Furthermore, C/C++ is a well known language with a large amount of legacy software available in academia and industry. Thus, the transition from sequential C/C++ code to the DOL programming model can be performed with a moderate porting effort, similar to porting applications to the message passing interface (MPI) [104, 115]. Apart from this, the utilized process network strictly separates communication from computation, thereby minimizing the effort required to create a new mapping of processes to processors. The communication is realized via FIFO buffers with blocking read and write operations. During the mapping phase the most suitable communication channel for each inter-task communication is selected, based on the communication characteristics available from the architecture specification, e.g. throughput and latency.

Once the transition from a sequential application into the DOL programming model is carried out, the DOL framework supports the application developer by automating

a lot of time consuming steps and providing powerful tools. For example, based on the DOL application description it is possible to automatically generate a pure functional simulator. This simulator serves for debugging and testing of the DOL application as well as for obtaining important mapping parameters at the application level, e.g. the amount of data transferred on a given communication link.

In order to fully exploit the computing power offered by a tiled multiprocessor architecture the mapping of the individual tasks to different processing elements is of high importance. However, the mapping of the processing elements alone is not sufficient. The mapping of the communication links to the available communication paths needs to be considered, too. In fact, in order to achieve a good mapping it is important to perform the processing element mapping and communication mapping at the same time since they are interdependent. The DOL framework supports the application mapping by an automatic mapping optimization tool that provides a set of optimized mapping candidates of how the application can be mapped onto the target platform. The optimization process can be subdivided into two steps: *performance evaluation* and *optimization*, both steps are iteratively executed.

The performance evaluation can be conducted by simulation as well as formal analysis. Figure 4.7 depicts the evaluation and optimization cycle. During the design space evaluation DOL needs to analyze various combinations of tasks running on different processing elements, communication links realized through different communication paths and the effects of resource sharing. Therefrom it is obvious that a fast performance evaluation is essential to efficiently perform the design space exploration. In a first high level evaluation, the estimation is performed using a fast and possibly less accurate analytical performance estimation. This analytical estimation is based on the *Modular Performance Analysis* [26, 170] which is based on Real-Time Calculus [162]. After the high-level evaluation has determined a number of promising candidates, a low-level estimation based on simulation is applied in order to obtain more precise performance information. This information can also be used to calibrate the parameters of the analytical model, e.g. determine the runtime of processes on a processing element and estimate the overhead introduced by context switching. This is important in later mapping stages where high accuracy is necessary. In Section 4.2.3 the details about the SHAPES simulator, which is used to obtain detailed performance information, are presented.

The optimization phase determines a good application mapping based on the information collected during the evaluation phase. The optimization objectives are fixed by the designer beforehand. The DOL framework is capable of handling multiple and possibly conflicting objectives. Furthermore, the optimization needs to incorporate constraints on the feasibility of mappings, such as fixed mappings for certain processes or the availability of local memory. The optimization used in DOL is based on evolutionary algorithms [183] and uses the PISA [16] interface. The optimization criteria are computation time and communication time.

The DOL programing model offers a good starting point to migrate an application to the SHAPES platform. Together with the automatic mapping optimization the DOL framework offers the designer a convenient possibility to find good application to hardware mappings. However, due to the high level of abstraction it is not possible to execute a DOL application directly on the SHAPES platform. A hardware dependent

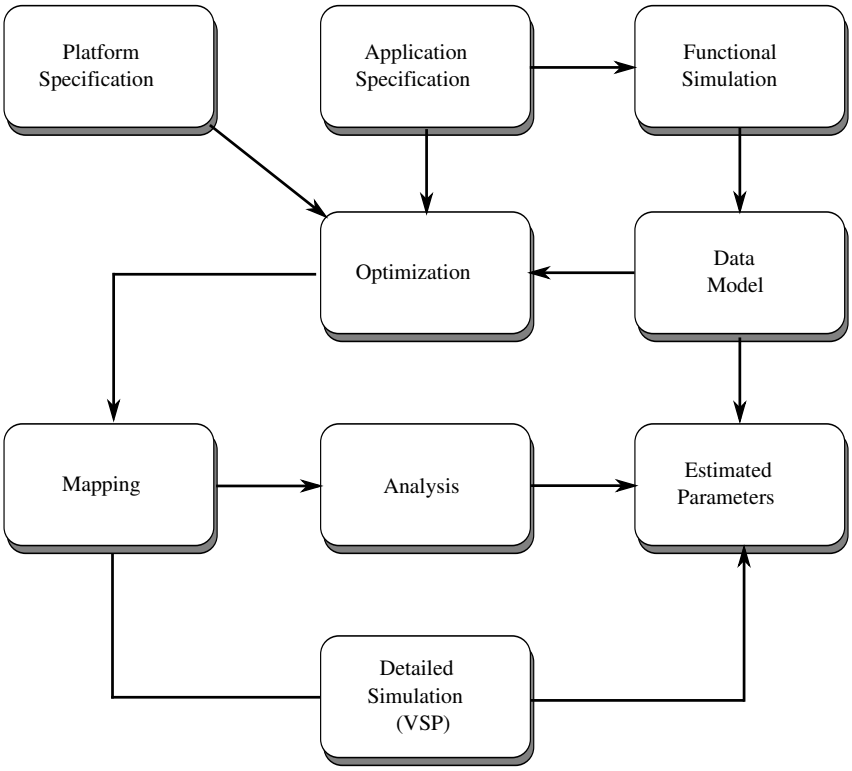


Figure 4.7: DOL evaluation, optimization cycle [161]

software layer is required to fill the gap between DOL and hardware. This software layer and its connection to DOL and the hardware will be presented next.

4.2.2 Hardware Dependent Software

The DOL framework processes the application description and generates a set of mappings from the application to the different heterogeneous processing elements. However, this alone is not sufficient for executing the application on the hardware, because the low level software controlling the hardware specific features of the platform is missing. In order to execute the different application tasks on the processing elements, a complete software stack for each element is required. Due to the heterogeneous nature of the processing elements a specialized and highly optimized version of this software needs to be provided for each type of processing element. The software stack can be roughly divided into the application software and the hardware dependent software (HdS). The HdS encapsulates the inter-process communication software, the operating system (OS) and the hardware abstraction layer (HAL). The operating system and the communication software are responsible for providing the necessary services to the upper layers as well as to manage and share resources. The services include resource management and control, scheduling of processes on the different processing elements, intra-processor communication and inter-processor communication. In fact, each HdS implementation is closely linked with the underlying processing element. Only by ap-

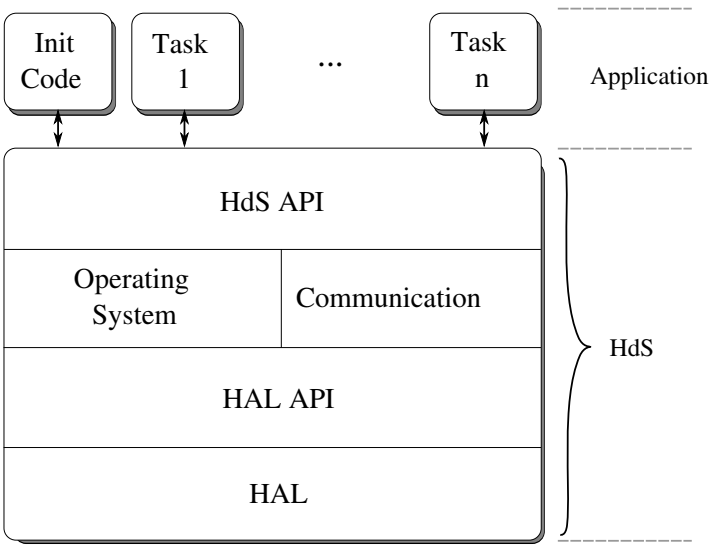


Figure 4.8: Application software stack [49]

plying architecture specific optimizations it is possible to provide high performance to the higher software layers.

Figure 4.8 shows the general structure of a complete software stack running on a processing element. The top layer is formed by the different application tasks. The initialization code ensures that all communication channels and tasks are instantiated and initialized prior to utilization. As described in Section 4.2.1, the DOL programing model requires that the communication between the different tasks is explicit. This inter-task communication is mapped to the HdS communication API. Besides the support for inter-task communication also support for operating systems is provided by the HdS. Only very simple applications do not require OS support, but for complex applications, as intended to be run on the SHAPES platform, the OS support plays a central role. For example, for an efficient application to HW mapping multi-process support of the OS is essential. The access to the underlying hardware is realized via a HAL API that provides a uniform access to the various processing elements.

From the description of the HdS structure above, it is evident that developing an HdS layer is a complex and error prone task. The time spent on developing such a software must not be underestimated, since it accounts for a considerable amount of the entire software development time. Traditionally, this kind of software is provided by the system vendors, so that the software developers can concentrate on the application development. However, in case of newly developed hardware the system designer needs to manually create the HdS layer. In case of the SHAPES platform manually creating the HdS is not a feasible option. Due to the high number of different processing elements the effort spent for creating the HdS would be huge.

Instead, an automatic generation approach has been selected [48–50, 124, 125]. Figure 4.9 gives an overview of the used HdS generation flow. The DOL description and the mapping serve as input for the HdS generation process. Furthermore, the API provided by the communication library and the operating system library are also taken into account while generating the HdS. During HdS generation the abstract commu-

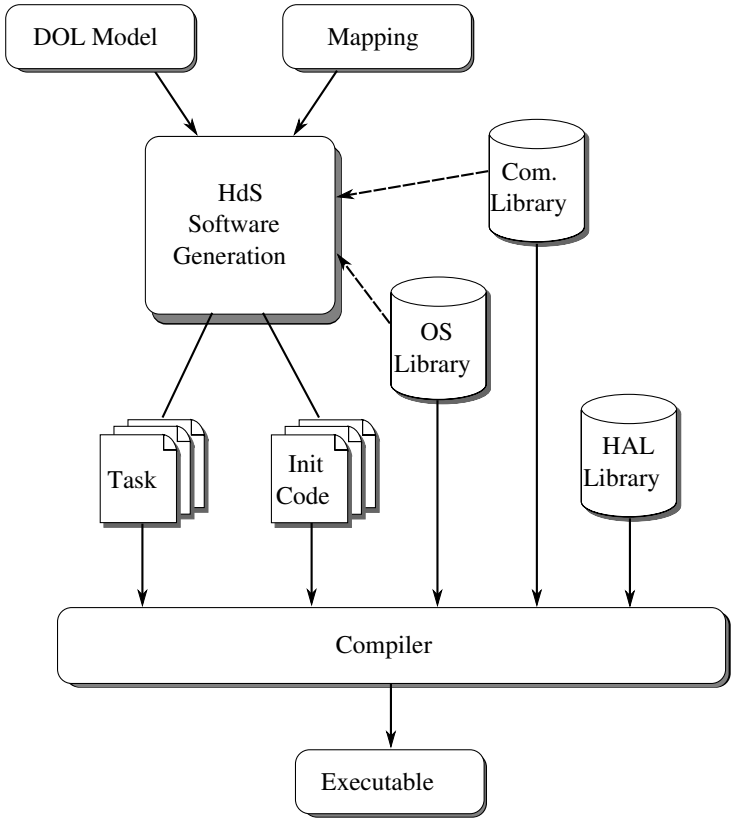


Figure 4.9: HdS generation flow

nication channels specified in DOL are mapped to the corresponding hardware communication links as specified by the mapping information. HdS takes care of selecting the most appropriate data communication scheme, e.g. DMA transfer or shared memory, to realize an efficient communication. Moreover, different optimization steps are performed during the generation to keep the overhead of the HdS layer on the system performance minimal. For example, memory optimizations [49] are applied to remove unnecessary data buffers. Finally, the HdS software generation outputs a set of C code files which can then be compiled and linked with the HdS libraries. For example, for the mAgic processor the Chess/Checkers tool-suite [155] is used to generate the highly optimized binary code for the VLIW DSP.

4.2.3 Simulation Environment

SHAPES is a scalable and highly flexible hardware platform that offers the system designer many degrees of freedom. At a high level the design space can be searched in three dimensions: First, the different characteristics of the basic tiles can be investigated. Second, the number of tiles can be varied to match the applications requirements and third, the different communication schemes can be explored. Due to the high development effort and costs only a hardware prototype of a RDT tile has been developed in the context of the SHAPES project. For all other types of basic tiles and especially for multi-tile configurations this project relies solely on simulation of the

hardware platform. Due to the central role of simulation in this project the key aspects of the created virtual SHAPES platform (VSP) are presented in the following.

4.2.3.1 Use Cases

Within the scope of SHAPES three main use cases of simulation can be identified.

- *Performance Estimation*: Using the simulation environment it is possible to collect detailed performance characteristics for a given application, e.g. context switching overhead. This information is required by the DOL framework in order to optimize the application's mapping.
- *Software development*: The simulation platform enables the execution of the software generated by the SHAPES toolchain. Thus, it is possible to test, debug and optimize the application for the SHAPES hardware platform. In particular for this use case the simulation performance is of great importance since application developers are not willing to wait hours to find out if the application is working or not.
- *Design Space Exploration*: With the virtual SHAPES platform it is possible to quickly evaluate different numbers and combinations of tiles, long before a hardware prototype is ready. Hence, the simulation helps to identify promising candidates for a hardware implementation.

The three use cases clearly show the broad field of applications for the VSP inside the SHAPES project. For the software development the simulation speed is the critical factor, since software developers need a quick feedback during the edit-compile-debug development cycle. For the other two use cases the accuracy plays the dominating role since the information gained by the simulation will be used for design decisions.

4.2.3.2 Simulation Requirements

In general, the simulation speed plays a critical role for the usability of a simulation environment. This is in particular true for such a complex architecture as the SHAPES platform. The simulation speed of a platform is mainly determined by the abstraction level. Classical register transfer level (RTL) simulations provide very detailed information about the simulated system. However, the low simulation speed prohibits an efficient design space exploration and software development. Therefore, the simulation level needs to be at a much higher level than RTL. For the SHAPES platform two levels of abstraction are considered: *cycle accurate* (CA) and *instruction accurate* (IA). The cycle accurate simulation has the advantage that the microarchitecture of the processor is fully simulated, thus leading to a very accurate timing behavior. In contrast, instruction accurate simulation provides much higher simulation speed by neglecting the microarchitectural features. Thus, CA simulation provides a higher accuracy while IA simulation permits shorter development cycles.

For a single tile the virtual SHAPES platform allows both, cycle accurate simulation or instruction accurate simulation, depending on the requirements of the user. The multi-tile VSP clearly focuses on a high simulation speed by using only IA simulation.

Apart from the different abstraction levels of the processor simulation also different abstraction levels for the communication links are possible. They range from

Num. tiles	Sim. time [s]	Memory [MB]	Sim. speed [kHz]
1	762	181	281.9
2	1588	336	135.6
4	3512	646	61.3
8	7613	1280	28.4

Table 4.2: Scaling of simulation time, memory consumption and simulation speed for the LQCD application

very detailed pin-accurate level to a very abstract functional level without any timing information or protocol information [77]. In the context of SHAPES, the transaction level modeling (TLM) [116] provides a good balance between accuracy and complexity. Compared to pin accurate simulation, TLM reduces the number of events that have to be processed during the simulation by only transmitting the data payload while bypassing the communication via low-level constructs like SystemC signals. Hence, the simulation performance can be kept at an acceptable level while scaling up well with the multi-tile SHAPES platform.

4.2.3.3 Realization of VSP

The virtual SHAPES platform is modeled using SystemC and for simulation the CoWare Virtual Platform environment [33] is used. The ARM general purpose processor (GPP) as well as the DSP are modeled as instruction set simulator (ISS) at an instruction accurate level. The ARM processor model includes a cache model and is taken from the CoWare IP library, whereas the floating point VLIW DSP is modeled using the architecture description language LISA [58]. Building upon the single tile simulator it is possible to automatically generate a multi-tile simulator using a generation script. This script uses the number of tiles, their individual types and whether NoC support is required or not as input parameters and then automatically assembles the multi-tile simulator.

At the current state the scalability of the simulator is limited by the fact that the generated SystemC simulator is a 32 bit executable. Therefore, no matter how much memory is physically present in the simulation host system, the usable address space for the simulator is restricted to 4 GB, which allows the instantiation of 16 tiles at maximum. Migrating the entire simulation environment to 64 bit will solve this problem, though it will require quite some reworking of the different simulation components, e.g. the existing IP blocks.

4.2.3.4 Characterization of VSP

Performance measurements of the different multi-tile simulators have revealed that the simulation time grows approximately linear with the number of tiles. Table 4.2 shows the simulation time, the memory consumption and the simulation performance

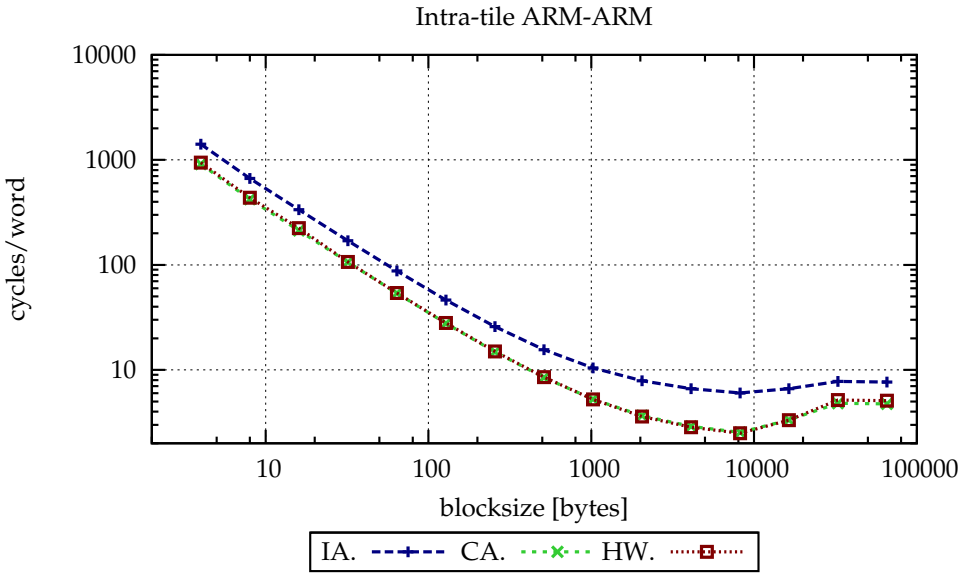


Figure 4.10: Intra-tile ARM-ARM communication

for the lattice quantum chromo-dynamics (LQCD) (see Section 4.3) application using different numbers of tiles [1]. The LQCD application has been scaled with the number of tiles as suggested by Gustafson's law [52]. The simulation speed of the simulator is expressed as equivalent frequency to reach the same performance level using the SHAPES hardware. Measurements show that the simulation performance ranges from approximately $200\ \mathrm{kHz}$ to $519\ \mathrm{kHz}$ for a single RDT tile and it ranges from $21\ \mathrm{kHz}$ to $28\ \mathrm{kHz}$ for an eight tile platform. The high degree of variation in simulation speed on the same host depending on the simulated software is due to the fact that the ratio between simulated time and simulation time (wall clock time) is not constant. This can be explained by the utilization of a *discrete event simulation* (DES). Depending on the number of activations of certain hardware blocks the simulation effort varies over time, e.g. inter-tile communication requires the simulation of the DNP and thus slows down the entire simulation.

In order to assess the accuracy of the created VSP, different series of measurements have been performed to compare the timing information predicted by the simulation with the timings obtained from the hardware prototype. Since only a single tile RDT hardware prototype is available in the context of SHAPES, it is only possible to directly evaluate the accuracy of the single tile simulation. Based on this information the accuracy of the multi-tile simulation can be estimated since it builds upon the single tile simulation. Figure 4.10 and Figure 4.11 show the intra-tile ARM-ARM and ARM-DSP transmission of data blocks for instruction accurate and cycle accurate simulation and the execution on the hardware prototype. In both cases the cycles per data word are plotted for different block sizes. As expected, the IA simulation shows the higher deviation from the results of the real hardware than the CA simulation. However, the deviation between IA and hardware remains mostly constant over the different

[1] All measurements have been performed on an Intel Core2 Quad Q6700 $2.66\ \mathrm{GHz}$ with 16 GB of RAM.

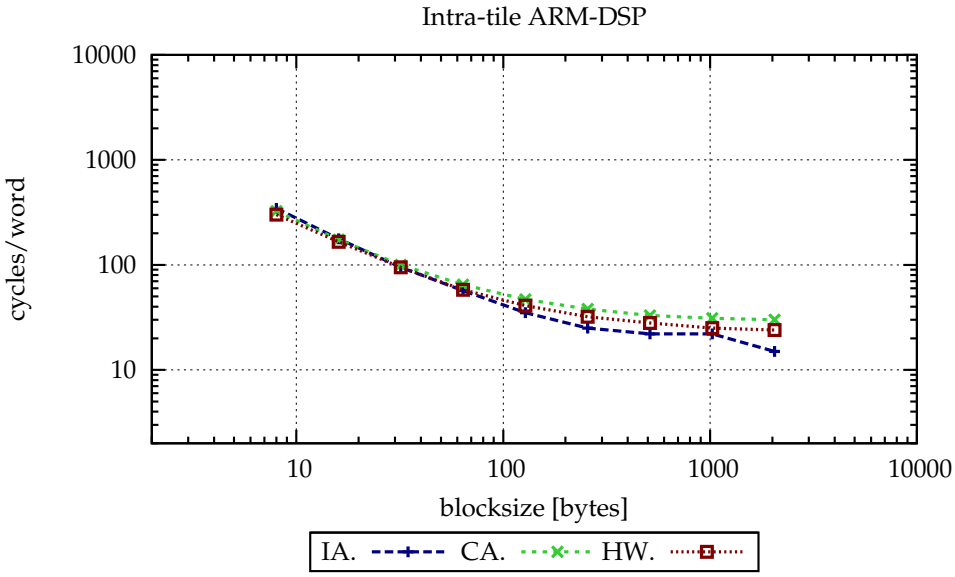


Figure 4.11: Intra-tile ARM-DSP communication

block sizes and thus it is possible to correctly predict the trends with an IA simulation. The error of the instruction accurate simulation ranges between 49% and 140% for the ARM-ARM communication and between 0% and 37.5% for the ARM-DSP communication. In case of a cycle accurate simulation the error ranges between 0% and 7.8% for the ARM-ARM communication and between 5.2% and 25.0% for the ARM-DSP. In Table A.1 and Table A.2 of Appendix A the detailed measurement information are given.

The main reason for this high difference between the cycle accurate and instruction accurate simulation are the different ARM cache models employed in both simulation modes. According to the hardware specification the ARM926EJ-S has 16 Kb data cache and 16 Kb instruction cache. Furthermore, the processor provides a *write buffer* [66] to hide the latency of writing data to the main memory. This write buffer is not simulated in the instruction accurate simulation, which explains the obtained cycle numbers. Particularly for applications that initiate a high number of write requests, e.g. during memory copy, the missing write buffer leads to a pessimistic estimation of cycles. For other applications the impact of the missing write buffer is much smaller. For example the timing difference between CA and IA for booting a 2.6.22 Linux kernel on the ARM is only 8.86%. Compared to the IA simulation the more accurate CA simulation increases the simulation time by a factor of 2.38.

4.3 SHAPES Applications

The motivation of the SHAPES project is to develop a scalable platform which is suitable for performance demanding signal processing applications, e.g. audio and video processing and numerical simulations. This section gives a brief overview of the main driver applications that are used for testing and optimizing the SHAPES platform.

4.3.1 Lattice Quantum Chromo-Dynamics

The quantum chromo-dynamics (QCD) [5] is the prevailing theory of strong interactions formulated in terms of quarks and gluons. Because of the high non-linearities of the strong forces it is hard or even impossible to find a purely analytical solution in quantum chromo-dynamics. By introducing a discrete space-time lattice [51, 173] it is possible to numerically solve this problem using Monte Carlo simulations. In order to obtain significant results the simulation must be repeated many times due to the statistical nature of the solution. The quality of the simulation depends on the size of the lattice as well as on the numerical precision of the underlying hardware. The algorithms related to LQCD (Lattice QCD) are considered as one of the most demanding numerical applications.

This application is a very good test case for the SHAPES platform since the problem can be easily scaled from a single tile up to several thousand tiles, leading to a larger lattice. Furthermore, this type of application relies on good floating point support in order to achieve high performance, which is a characteristic that most of the other available VLIW DSPs are lacking.

4.3.2 Wave Field Synthesis

The wave field synthesis (WFS) [148] is a spatial audio rendering technique that allows to create a 3D sound field in an efficient way. It has been invented at the TU Delft in the Netherlands and has been successfully demonstrated in academia. WFS is based on the wave theory concept of Huygens, that all points of a wave front serve as individual sources for a secondary wave front. This principle can be applied in acoustics by placing a large number of small loudspeakers closely together. The signal for each individual loudspeaker is computed by an algorithm based on Kirchhoff-Helmholtz integrals and Rayleigh's representation theorem [36]. The superposition of the wave fields produced by the loudspeakers forms the wave field, which is an accurate representation of the original wave field. However, the high computational complexity hinders a wide application of this technique.

Current realizations of the WFS are based on standard PCs or DSPs. Due to the high computational complexity these systems are only capable of computing a small number of sound objects in realtime. The SHAPES platform is an ideal target for this application, since it offers a high floating point performance and can be easily scaled. For m sound sources and n loudspeakers $m \cdot n$ processes are required. Each of them consists of a simple convolution filter. An overview of the WFS implementation for SHAPES can be found in [147].

4.3.3 Hilbert-Transformation

Today, ultrasound beam forming is realized by using digital delay lines for dynamic phase correction. In order to increase the flexibility and quality of the beam forming a fully software based solution is envisioned. To realize this massively parallel computing architectures like SHAPES are required to achieve the high data throughput.

In order to obtain the analytical radio frequency (RF) signal required for software based beam forming the real valued input vectors produced by the transducer are transformed by means of the time-domain Hilbert transformation [118] (Equation 4.1).

$$u_{\mathrm{RF}}(k,i) = z_{\mathrm{RF}}(k,i) + j\frac{2}{\pi}\sum_{m=1}^{H_w}\frac{z_{\mathrm{RF}}(k+m,i) - z_{\mathrm{RF}}(k-m,i)}{m}, m = 1,3,5\ldots H_w \quad (4.1)$$

$z_{\mathrm{RF}}(k,i)$ is a sample of the real valued input signal, $u_{\mathrm{RF}}(k,i)$ is a sample of the analytical RF signal and H_w is the length of the spatial Hilbert transformation window. In the context of the SHAPES project a highly optimized version of the Hilbert transformation has been implemented to assess the achievable data throughput and the scalability of this software approach.

4.4 Synopsis

The SHAPES environment is a representative of the emerging heterogeneous, tiled-architectures, which can be easily scaled with the application's size. In order to tame the complexity of the architecture, a complete toolchain has been developed that hides as much as practically possible of the complexity from the application developers.

Inside the SHAPES project the simulation of the hardware plays a central role for performance estimation and early application development. Due to the already high complexity of a basic RDT tile, efficient simulation techniques are of great importance for this project. For this reason, the virtual SHAPES platform is an ideal testing environment to study the impact of the new simulation techniques developed in this thesis.

Chapter 5

Checkpointing

The usability of VPs for software developers heavily depends on the simulation speed. In particular for interactive software components, e.g. GUIs, a simulation speed close to the speed of the final hardware device is desirable for an efficient development process. For the future, more complex MPSoC platforms will be deployed and, hence, an increased demand for simulation performance is anticipated. The *Checkpoint/Restore*(C/R) technique offers the possibility to reduce the number of simulation runs by storing the state of the simulation to a file, and to use this checkpoint image to restart the simulation in exactly the same state at a later point in time. For example, instead of booting the operating system (OS) every time a new application needs to be executed, a checkpoint permits to omit the time consuming boot process and to concentrate on the execution of the application itself. It is important to note that checkpointing does not speed up the simulation itself but it helps the user to reduce the time spent in simulation.

For modeling VPs, SystemC [30,116] has been established as the de-facto standard. It allows to efficiently describe software as well as hardware aspects of VPs. The most important use cases for the C/R feature in SystemC simulations are listed in the following:

Time saving – By creating a checkpoint image at a certain point, the user does not need to re-run the entire simulation in order to reach a given state, but can just re-load the state from disk.

Move around in time – The facility of creating multiple checkpoints at different points in time gives the user the possibility to move around in time. This feature is intended for debugging critical parts of a simulation without re-running the entire simulation.

Periodic checkpointing – Periodic checkpointing allows the user to quickly recover the state shortly before an error has occurred. Hence, the restarted simulation can be employed to investigate the reason for this error.

Simulation transfer – A developer of a VP can transfer a checkpoint image to another developer to request help for a specific problem or to demonstrate a certain behavior of the simulation.

In the context of this thesis a C/R framework taking into account the specific characteristics and needs of SystemC based VPs has been developed. This framework is based on *process checkpointing*. In contrast to traditional process checkpointing the specific requirements of SystemC based VPs are taken into consideration, e.g. the interaction with external components such as debuggers, GUIs, consoles and OS resources. The details of the process checkpointing itself are not presented here. In order to evaluate the impact of the proposed framework on the development and usage of VPs, the C/R mechanism has been integrated into CoWare Virtual Platform [33]. Throughout

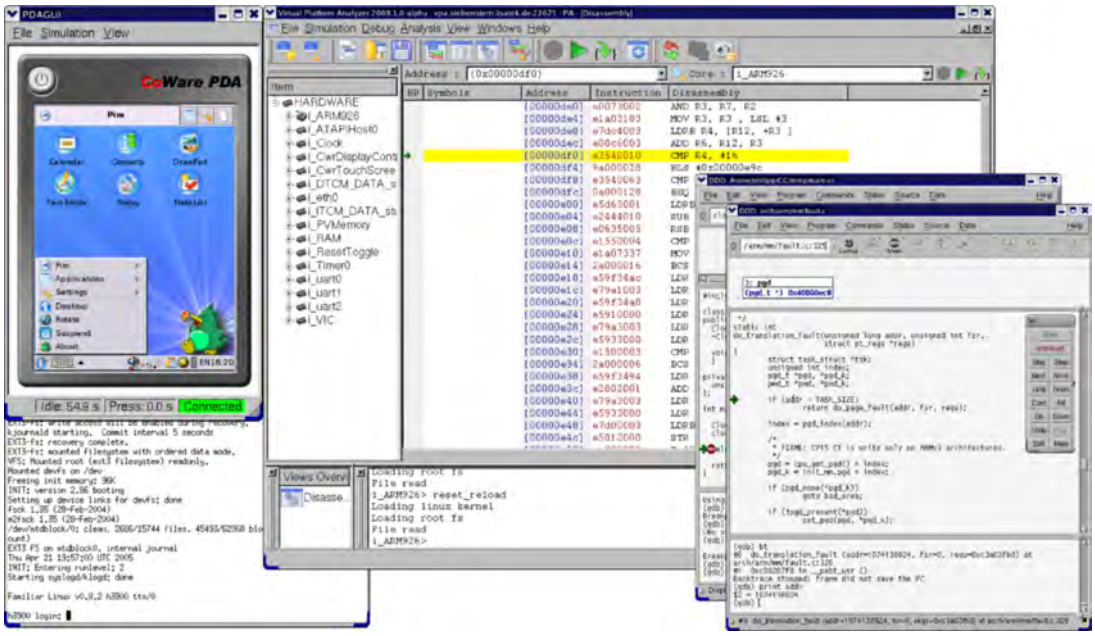


Figure 5.1: Debugging and analyzing a PDA virtual platform

the chapter a personal digital assistant (PDA) platform is used to demonstrate the necessary steps for checkpointing.

The remainder of this chapter is structured as follows: First, in Section 5.1 the PDA evaluation platform is presented to motivate the need for checkpointing. Then the different requirements for checkpointing are discussed. Next, an overview of the related work is given in Section 5.2. The detailed description of the proposed checkpointing framework and the integration of user modules are presented in Section 5.3 and Section 5.4. Afterwards an analysis of the checkpointing performance is presented in Section 5.5. In order to assess the impact of checkpointing on complex VPs a case study is conducted and presented in Section 5.6 using the Virtual SHAPES Platform (for more details refer to Section 4.2.3). Finally, Section 5.7 summarizes the contribution of this work and discusses some possible extensions for the future.

5.1 Motivation

This chapter describes how the C/R mechanism can be used to accelerate the typical edit-compile-debug cycle for software development. The mechanism is explained in the context of a representative application development task: debug and analyze how an application on a PDA interacts with the hardware after a touchscreen event occurred. This task is typical for the kind of problems that developers face when porting an operating system to a new hardware platform. It requires a complete boot of the guest operating system on top of the VP. Furthermore, the application has to be initialized before the actual debug task can be started.

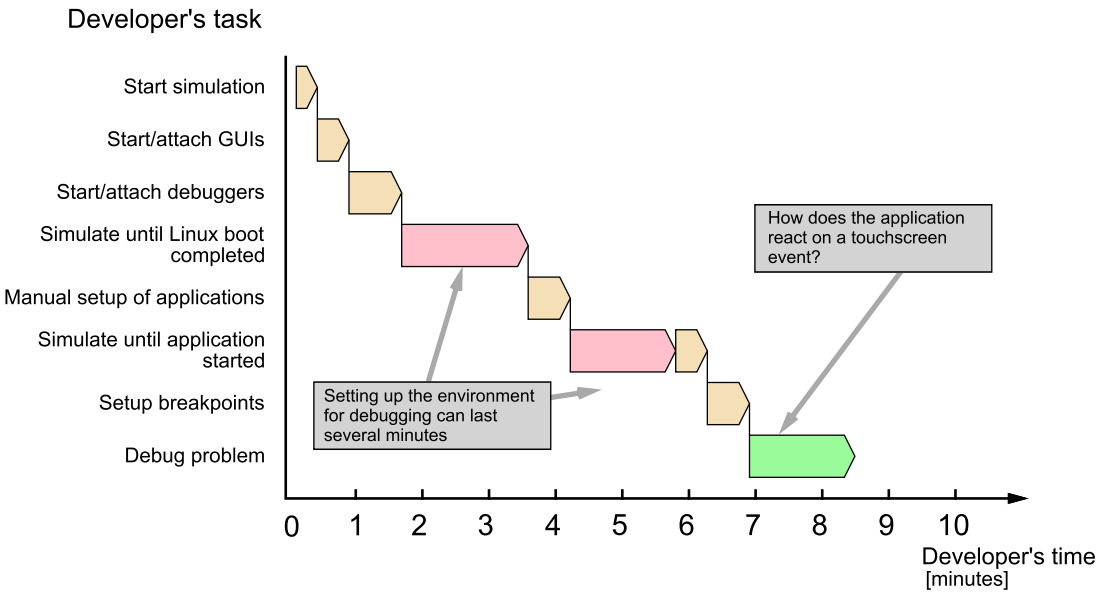


Figure 5.2: Developer time spent in a typical debug session

For the actual debugging task the developer may also have to attach several debuggers to the VP. A typical collection of debug tools for the CoWare Virtual Platform is shown in Figure 5.1. The Virtual Platform Analyzer allows inspecting all details of the hardware platform, while two data display debuggers (DDD) [180] are utilized for source level debugging of the application software and of the Linux kernel software. The emulation of peripherals, e.g. touchscreen and terminal, is in general realized by modeling those components as separate applications that communicate with the VP via a specific API. In case of the PDA example the touchscreen and the boot console are realized as external applications.

Setting up the tools and driving the VP into the desired state for debugging the given problem can become a quite time consuming task. As shown in Figure 5.2 the complete setup procedure can take several minutes. It comprises the starting of the VP and the debugger tools, attaching the debuggers to the platform, setting up breakpoints and loading the software images. Next, the VP has to be executed until the Linux OS is booted. Further user interaction may be required for a Linux user login and the manual start of applications. This procedure needs to be repeated for each new debug cycle. The C/R mechanism enables the developer to shorten the setup phase to several seconds, as shown Figure 5.3. It is possible to create a checkpoint of the VP at any time, e.g. just at the beginning of the actual debug phase. When a checkpoint is restored, the VP is restarted in the exact state of the checkpoint creation time. Furthermore, all available debuggers are re-attached to the restored VP. Additionally, all the external applications for touchscreen and console are also restarted if they are no longer running. The breakpoint settings in the Virtual Platform Analyzer can be configured by a Tcl script that is automatically executed after each restoration of a checkpoint.

From this example it becomes clear that debugging complex VPs can benefit from the checkpointing functionality. However, depending on the specific use case of the

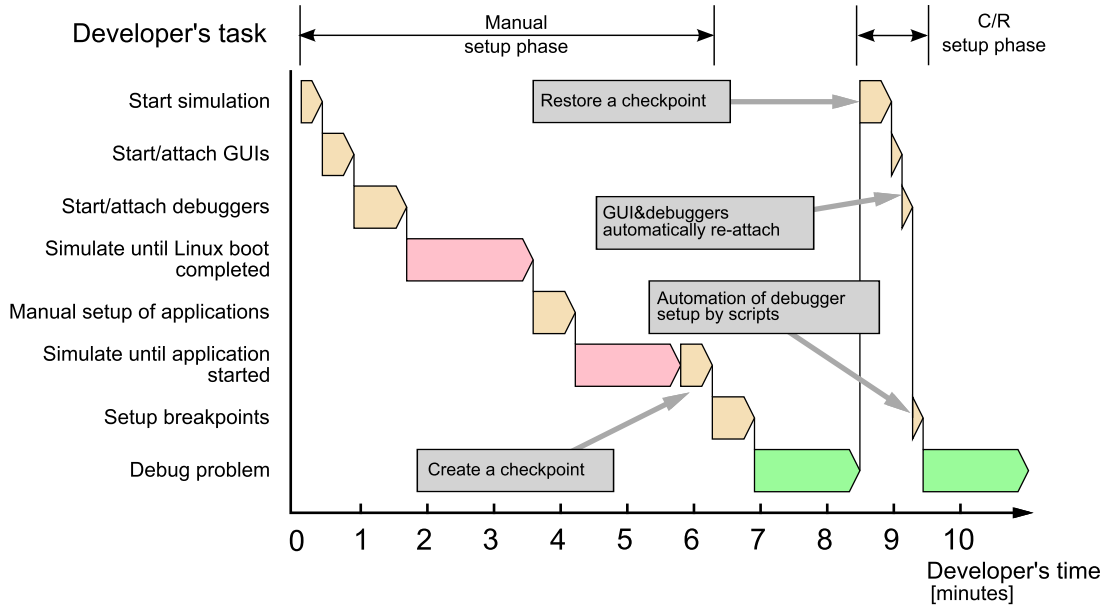


Figure 5.3: Debug cycle is shortened by checkpointing and restoring a virtual platform

C/R, different requirements are imposed on the C/R capabilities. The most important requirements are briefly discussed in the following:

Reliability – Reliability of the C/R functionality is of utmost importance. The reconstructed simulation is required to be in exactly the same state as the original simulation was before checkpointing.

Transparency – In the context of C/R transparency means that the checkpointing feature should be applicable without any modification on the simulated system. The need for complete transparency of the C/R implementation is not required for SystemC simulations, because the modification and re-compilation of the simulation is possible. Nevertheless, in order to minimize the effort for the user, the required adaptation should be as minimal as possible.

Performance – The different use cases impose different performance requirements on the C/R functionality. This includes the speed of checkpointing and restoring as well as the size of the created checkpoint image. For example, in case of periodic checkpointing the time required to checkpoint must be low in order to keep the impact on the simulation performance small. If a high number of images need to be stored at the same time, the size of the checkpoint also matters. In general, performance is more important for the restoration of a checkpoint than for the creation, because a checkpoint will be created only once, but is likely to be restored many times.

Support for external applications – Virtual platforms are often connected to external applications such as GUIs, consoles and debuggers in order to monitor and interact with the simulation, see Figure 5.1. Therefore, the C/R functionality must be capable of re-establishing connections to external ap-

plications in order to ensure seamless usage and debugging of the simulation.

Support for OS Resources – SystemC simulations may contain dependencies to the operating system like open files, I/O streams, shared objects, shared memory or even GUI resources. These resources must be handled separately by the C/R implementation as they are not part of the simulation process itself.

The importance of the different requirements depends on the actual use case and simulation environment. Due to the connection of external debuggers to processor models and the frequent utilization of OS resources for logging data, the support of external applications and OS resources is of high importance for SystemC based VPs. The transparency is ranked only second because VPs can be easily re-compiled.

5.2 Related Work

Checkpointing is commonly used for VHDL and Verilog simulators like Cadence NC-Sim [23], Mentor Graphics ModelSim [95] and Synopsys VHDL/Verilog Simulator [152]. However, extending the existing solutions to support C/C++ and SystemC modules is not trivial due to the way how the internal state is represented in memory. The internal state of a SystemC module is dependent on global variables, local variables, heap values, and OS resources such as file handles. All of them heavily depend on external tools like the compiler and the linker. With respect to checkpointing both compiler and linker show a kind of black box behavior that cannot be influenced. Hence, extracting the necessary state information is a very complex task.

Currently, a non-transparent approach is used to overcome the problem by introducing callback functions. The developers of the C/C++ and SystemC modules need to take care of saving the state. For example this approach is introduced in the Cadence Incisive Simulator starting with version 6.2. However, no special support for external components such as debuggers is provided.

The earliest mentioning of the utilization of checkpointing in a full-system simulator dates back to the mid nineties and is used inside the SimOS simulation environment [132] to switch between the fast and the detailed simulation mode. The switching between both simulation modes is realized by transferring the simulation state between both simulation modes. This kind of simulation state serialization is also used to realize the checkpointing functionality. The checkpointing capabilities of the IBM Mambo [141] simulation system are based on the same principles as in the SimOS simulation system. Closely related to the previously presented checkpointing techniques is the approach utilized by Virtutech for checkpointing SystemC models [102] which are simulated together with the Virtutech Simics [90] simulator. In the context of computer architecture simulation the SimFlex [54, 145] environment shows an ambitious use of checkpointing for full system simulation.

In the Virtutech simulation environment the user needs to manually specify the data which represents the state of a SystemC module by utilizing a mechanism called *attributes*. This kind of *model state serialization* approach leads to comparatively small checkpoints that can easily be exchanged between different computers. However, this

approach is currently limited to SystemC methods. It is not possible to checkpoint modules that make use of SystemC threads. Therefore, the applicability of this approach is rather limited, since most VPs rely on SystemC threads for modeling their behavior. Furthermore, marking all the relevant states in a complex module, e.g. a processor simulator, is a time consuming and error prone task that can only be performed by a developer with expert knowledge of this module.

Process checkpointing is another approach to implement C/R for SystemC simulations. The term *process checkpointing* describes a technique for storing the complete state of a process. The created checkpoint image can be used to reconstruct the process state at a later point in time, even on another machine. The image usually comprises the processor register set, parts of the process address space like stack, heap and global data, and the state of allocated operating system resources.

Process checkpointing is mainly used for process migration and fault tolerance and can be subdivided into two groups: *kernel space implementation* and *user space implementation*. Kernel space implementations such as CRAK [181] and BLCR [39, 67, 136] have the advantage that they can directly access the state of all OS resources allocated by the target process. Hence, kernel space checkpoint implementations are capable of delivering completely transparent checkpointing solutions to the user. However, additional effort must be spent to create the required kernel modules in order to access all OS states.

In contrast to kernel space process checkpointing implementations, the user space implementations usually require that the checkpointed application can be re-compiled. The most prominent implementations are Condor [87], Libckpt [123] and WinCkp [2, 28, 86]. Condor and Libckpt are both implemented for UNIX whereas WinCkp is a process checkpoint implementation for Windows NT 4.0.

The evaluation of the existing process checkpointing implementations showed that there is actually no solution available which would support the complete set of possible use cases or applications. Most solutions target one special use case or application type and support only a certain set of OS resources. Especially the support of interprocess communications to other user processes, such as external debuggers in the VP environment, is missing completely in all solutions.

Operating system checkpointing stores the state of the entire OS and all its applications. The main advantage of this approach is the complete transparency for the simulator developer. No special care needs to be taken for OS resources inside the simulator and debuggers, since the simulation and the OS are checkpointed together. Operating system checkpointing has become widely available over the last years especially by the usage of VMWare [169]. However, the drawbacks of this approach are the prohibitively large checkpoint images and the lack of control of the checkpointing process by the simulator.

5.3 Virtual Platform Checkpoint Framework

The VP simulation process is a normal C++ executable composed of C++ classes modeling hardware components, linked against the SystemC C++ class library and the Sys-

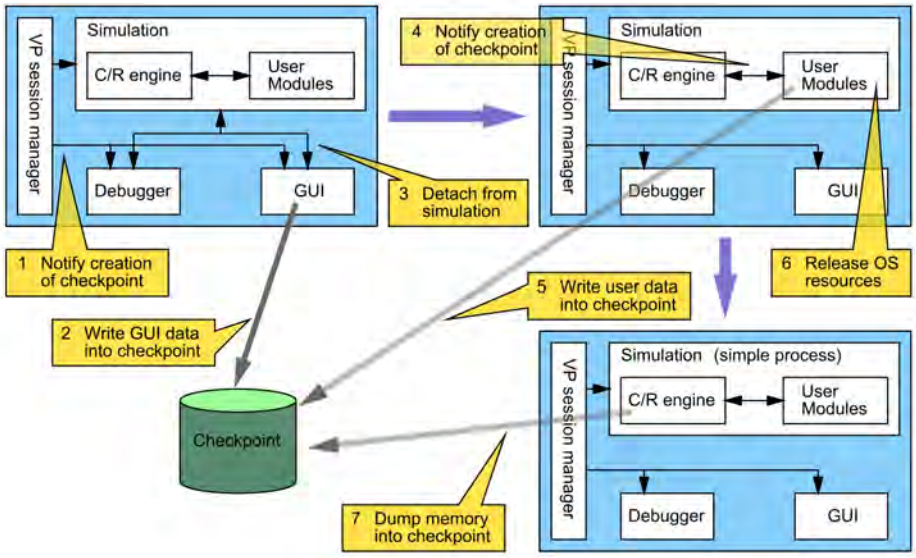


Figure 5.4: Checkpoint procedure: Releasing OS resources

temC scheduler. Due to its dependency on external tools like C++ compiler and linker, which can be seen in this context as black box components, the proposed C/R framework is based on a process checkpointing mechanism [131] (see Section 5.3.1). Thus, the state of the whole process including stack, heap and global data as well as the content of all processor registers is saved into a checkpoint image on the file system. A process restore mechanism is able to reconstruct the simulation process state based on the content of the checkpoint image at a later point in time. Moreover, the utilization of process checkpointing simplifies the integration of the C/R framework into a SystemC based simulation, because no dependencies on the underlying SystemC kernel exist.

Since the process checkpointing mechanism is storing the complete simulation memory automatically, all C++ variables and objects of the hardware model are automatically checkpointed without the necessity to adapt the platform source code. The only exception is the utilization of inter-process communication channels and operating system resources such as open files, sockets, threads or mutexes, which all form gateways from the simulation process to the outside environment. Thus, the checkpointing framework has either to ensure that these resources are re-established during restore, or if this is not possible, closed before checkpointing and re-opened after restoring. For example, the utilization of files is common in SystemC modules for tracing data or for reading configuration information. Sockets, threads and mutexes are sometimes used for processor model debugger interfaces.

The central problem of handling OS resources and inter-process communication is solved by driving the simulation into a safe state where process checkpointing is applicable. Reaching this state is accomplished by providing a framework that releases all open OS resources and closes all communication channels before the actual initiation of the process checkpoint and re-establishes them afterwards and after restoring the checkpoint respectively. The main components involved in this process are the

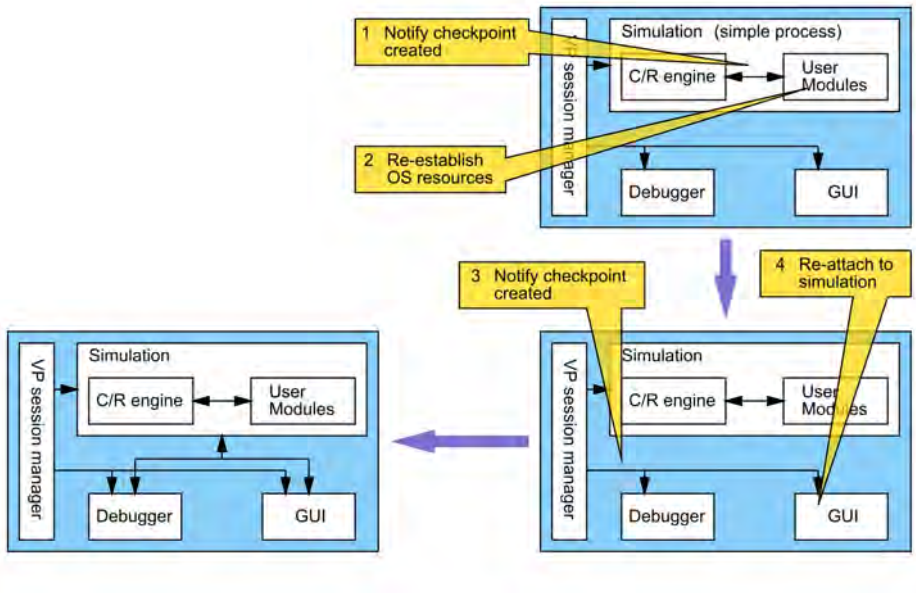


Figure 5.5: Checkpoint procedure: re-establishing OS resources

Virtual Platform Session Manager and the C/R engine. The VP session manager is a separate process that is responsible for controlling the checkpointing procedure. This process drives the communication between the debugger and GUI tools while they are detached from the simulation process, or while no simulation process is running at all. By introducing this background process, it can be assured that the debuggers are not affected by the checkpointing procedure, since they remain attached to the VP Session Manager all the time. Third-party debuggers require a plug-in in order to connect to the VP Session Manager. The C/R engine is part of the simulation process and performs the process checkpointing once all external connections are closed.

The checkpointing procedure is illustrated in Figure 5.4. First, the debuggers and the GUIs are notified (1). If necessary the GUIs transfer data (2) to the checkpoint image. Then the debuggers and GUIs are detached from the simulation (3) to permit the release of sockets, pipes and other interprocess communication resources. In the next step, the user modules of the simulation will be notified of the checkpointing (4). After writing their user data (5) to the checkpoint image, the user modules release all OS resources (6). Now, the simulation process has no external dependencies anymore and regular process checkpointing can be applied to store the state of the process (7).

Figure 5.5 shows how the simulation is re-connected to the external resources after the checkpointing process has completed. First, the user modules inside the simulation are notified (1) that the process checkpointing has been completed. Then each user module re-establishes its OS resources (2). In the next step the debuggers and GUI are notified (3) and re-attached to the simulation process (4).

The procedure for re-establishing the OS resources after the restoration of a checkpoint is comparable to that after the creation of a checkpoint, see Figure 5.6. First a new simulation process is created (1). Then the data from the checkpoint image is used to restore the memory of this process (2). In the subsequent step the user modules are

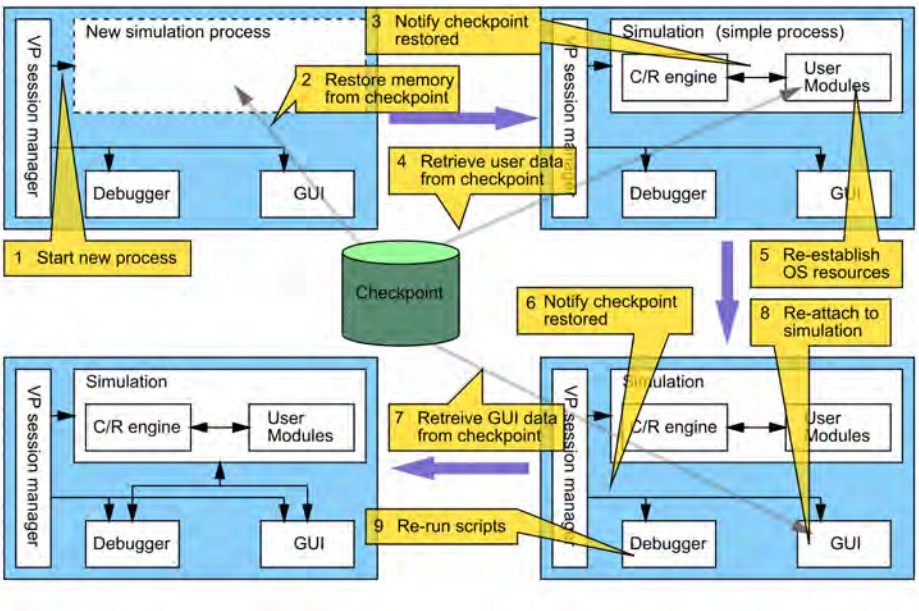


Figure 5.6: Restore procedure: Recreation of the simulation process and re-establishing of the OS resources

notified (3) and retrieve the stored user data from the checkpoint image (4). After loading the data, the OS resources are re-established (5). Next, the debuggers and GUIs are notified (6). The GUIs retrieve their data from the checkpoint (7). Then the debuggers and GUIs are re-attached to the simulation (8). Finally, the debugger scripts are re-run (9) to set all necessary breakpoints.

5.3.1 Process Checkpointing

In this section the details of the employed process checkpointing are discussed on the basis of a Windows XP virtual memory layout as shown in Figure 5.7. Before the simulation data can be written to a file, the virtual memory space of the simulation process must be completely identified and decided which part of it needs to be stored. The first step is to define the part of the virtual memory address space which needs to be taken into account for checkpoint/restart. The address space is divided in two parts: The user space, starting at address `0x0000000` and ending at `0x7FFFFFFF` and the kernel space which ranges from `0x80000000` to `0xFFFFFFFF`. The kernel space is not accessible by the user and is excluded from checkpointing. As shown in Figure 5.7, the user space is flanked on both sides by not accessible and unused memory areas. These areas define the address frame for the identification process: It starts at `0x0010000` and ends at `0x7FFF0000`. The next step is to divide this frame into different memory regions, where a region is defined as a range of memory pages which share the same attributes.

The resulting array containing the identified memory regions is used to analyze if a memory region should be part of the checkpoint or not. This decision depends on two conditions:

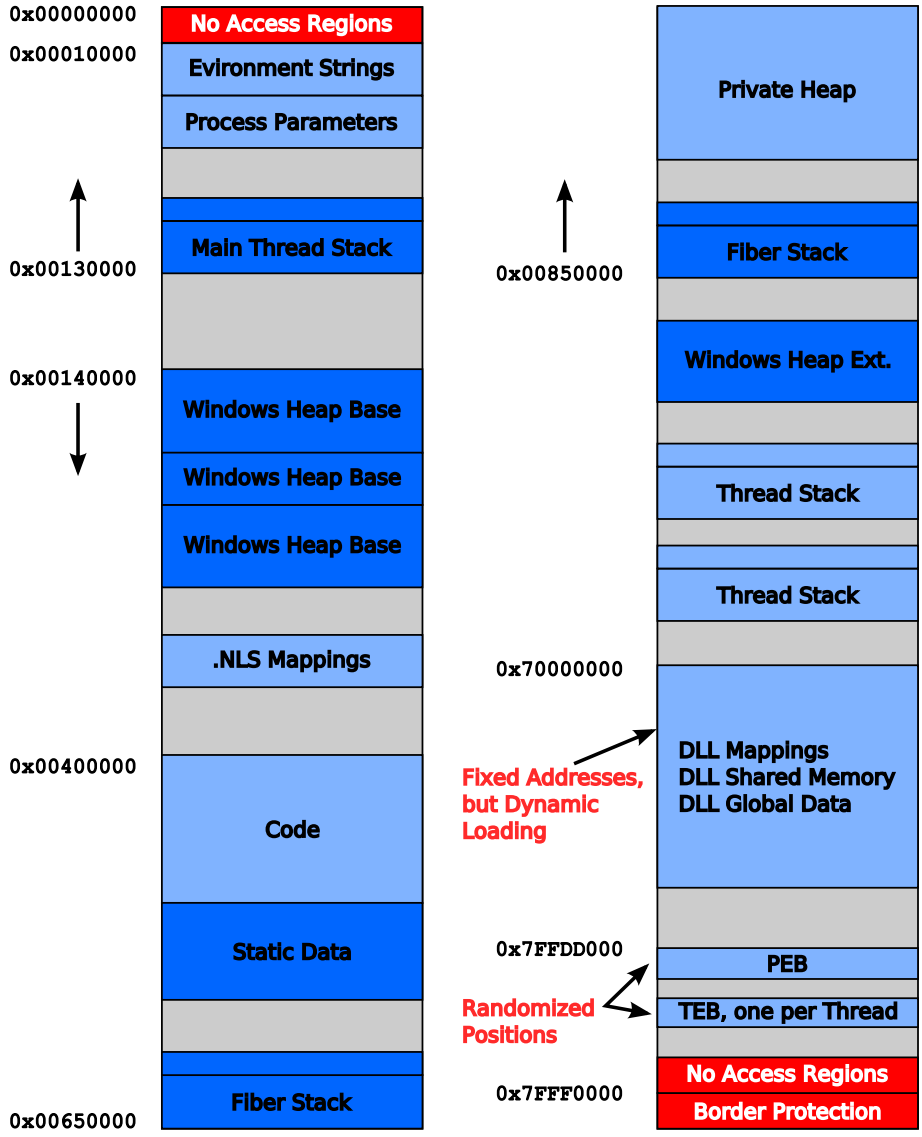


Figure 5.7: Example of a Windows XP virtual memory layout

1. If a region is never changed during a simulation run, it does not need to be stored. This is usually the case for regions with read-only protection.
2. Only data belonging to the simulation itself is allowed to be checkpointed. The infrastructure data must not be touched.

Based on these conditions the different code regions that need to be stored are identified. Further code size reduction can be achieved by analyzing in detail which code regions contribute to the process state. This step is very much dependent on the OS that is used.

Process checkpointing relies on the fact that the memory layout of a process does not change between runs. However, this assumption is not valid anymore with the introduction of *address space layout randomization* ASLR [140]. This feature of the loader randomizes the memory layout every time an application is loaded in order to complicate buffer overflow induced attacks. This randomization certainly aggravates the

```
1 #include <CwrBase/SimBase/CheckpointRestoreObserver.h>
2
3 class MyModule : public nCoWare::nCheckpointRestore::Observer {
4    ...
5 private:
6    virtual std::string handleBeginOfCheckpointing(...) {
7       copyTraceFileToCheckpointDirectory();
8       closeTraceFile();
9       return "";
10   }
11
12   virtual void handleEndOfCheckpointing(...) {
13      reopenTraceFile();
14   }
15
16   virtual void handleEndOfRestore() {
17      copyTraceFileFromCheckpointDirectory();
18      reopenTraceFile();
19   }
20   ...
21 }
```

Listing 5.1: Adaptation of a user SystemC module: Writing user data into a checkpoint

process checkpointing. Further research is required to analyze if process checkpointing in its current form can be utilized together with ASLR. In the context of this thesis ASLR has been deactivated.

5.4 Integration of User-Defined Modules

As described in the previous section, the framework for handling external communication channels and OS resources during the checkpoint and restore procedure allows the integration of user modules, debuggers and GUIs. This section focuses on the integration of user-defined SystemC modules, which is the most common task for model designers. For external applications such as GUIs and debuggers an API is provided. The details of this API are out of the scope of this thesis and are therefore not discussed.

The user-defined modules are notified about checkpoint and restore events by inheriting from a special observer object provided by the C/R framework API. This observer object internally connects to the VP session manager. The C++ code example in Listing 5.1 illustrates how this observer is utilized for a SystemC module that continuously writes trace data into a file. The trace data that was produced before a checkpoint is created shall further be available when restoring the checkpoint. Therefore, the user module closes the file during the checkpoint procedure and copies its content into the checkpoint image. After the checkpoint was created, the file is re-opened. In case a checkpoint is restored, the module first restores the original content of the trace file by

```
 1 #include <CwrBase/SimBase/CheckpointBlocker.h>
 2
 3 void MyModule::readConfiguration(...) {
 4    ...
 5    { // limit scope of variable config_file
 6       nCoWare::nCheckpointRestore::
 7       ScopedCheckpointBlocker cp_blocker(
 8          "MyModule reading config"
 9       );
10       std::ifstream config_file(config_filename);
11           // read configuration
12       ...
13    }
14    // checkpointing is possible again
15    ...
16 }
```

Listing 5.2: Adaptation of a user SystemC module: Temporarily blocking the creation of checkpointing while a file is opened

copying it back from the checkpoint image. Then, it re-opens the file to allow further dumping of data when the restored VP resumes the simulation.

The presented C/R framework does not handle temporary OS resources at all. Instead, the creation of a checkpoint is blocked for the short time when such resources are in use. Either the creation of a checkpoint is delayed until the appropriate resource is freed, or a message is printed to the GUI to inform the user why the creation of a checkpoint is not possible at the moment.

Very often files or other OS resources are only used for a short period of time. Examples for a temporary file access are the reading of configuration information or loading of a binary image to the memory of a processor. For this kind of temporary OS resource utilization the C/R framework API provides a simple mechanism that blocks the creation of checkpoints while the simulation is inside such an unsafe section. This is achieved by temporarily instantiating a checkpoint blocker object, as illustrated by the code example in Listing 5.2. As long as the instance of the blocker is active checkpointing is not possible. The developer can pass a string to the constructor of the blocker object, which is used to inform the user at runtime, why the creation of checkpoints is temporarily disabled.

In case that not all OS resources had been released before the process checkpoint is created, the produced checkpoint image would be corrupted. As a result a restored simulation based on a corrupted checkpoint would fail when accessing a previously unreleased OS resource. Usually, a debugger will point the user to the source code location that uses the incorrectly handled OS resources. However, to ease the development of checkpointing enabled platforms the checkpointing environment supports the model developer by scanning the simulation process for open OS resources immediately before the process checkpoint is created. All open files, sockets and threads are

reported to the user and the creation of a potentially corrupted checkpoint is aborted. This feedback helps the user to locate the incompatible code portion and correct it.

5.5 Performance of the Checkpoint/Restore Mechanism

In general, the time required for the creation and restoration of a checkpoint is proportional to the memory size of the simulation process. For example, the PDA platform shown in Figure 5.1 consists of an ARM926 single core processor model and requires an overall size of approximately 400 MB. Furthermore, the creation time of a checkpoint depends on the underlying file system. In case of storing the checkpoint image on a local disk of a Windows XP host, the checkpoint and restore procedure requires 10 seconds each. In contrast, on a Linux host the caching effects of the file system have a very positive effect on the creation of checkpoints, so that creating a checkpoint only takes 2.5 seconds while it is restored after 6 seconds.

Due to the underlying process checkpointing, the checkpoint images have shown to be quite sensitive to recompilations of the VP. Any change to the simulation executable of the VP invalidates the checkpoint image. Additionally, its portability is limited. Usually it is possible to restore a checkpoint on a host with identical installation. But already slight differences in the patch level of the operating system may prohibit to restore a checkpoint.

Recompilations or other changes of the embedded software which is executed by the processor models of the VP do not corrupt the checkpoint image. After a checkpoint is restored, the VP always contains the original software, e.g. the OS, and behaves like the checkpointed platform. The user is now able to load the recompiled software image, e.g. an application, into the processor model and continue the simulation of the VP.

The comparatively large size of the checkpoints based on process checkpointing increases the checkpointing time and limits the number of images that can be kept on a hard disk. For this reason the framework incorporates a stream-based compression utilizing the deflate algorithm [37] to compress the data before writing it to the hard disk. The deflate algorithm is a combination of the lossless Lempel-Ziv-Storer-Szymanski algorithm [149] and Huffman coding. This algorithm offers a set of parameters that enable the user to trade-off the compression rate for compression speed. This feature is very useful to determine the optimal working point for the compression in the context of checkpointing.

The impact of the compression on the checkpoint time is twofold: First, the computational overhead introduced by the compression/decompression increases the checkpointing time. Second, less data needs to be written/read from the hard disk, which leads to a faster checkpointing. Depending on which effect is dominating the checkpointing becomes faster or slower. Independent of the checkpointing time, compression clearly increases the number of checkpoints that can be stored simultaneously on the hard disk.

# tiles	checkpoint [s]	restore [s]	image size [MB]
1	3.00	10.20	319
2	5.34	18.30	568
4	12.33	38.80	1094
8	50.00	75.00	2115

Table 5.1: Duration and image size of uncompressed checkpointing for different numbers of SHAPES tiles

5.6 Case Study

In this case study the proposed checkpointing framework is applied to a complex VP in order to study its applicability and impact. As driver the Virtual SHAPES Platform (VSP) that is presented in Chapter 4 has been used.

Before applying the checkpointing to the VSP, some modifications of the platform were required. More specific, the creation of log files by the different peripheral modules needed to be extended to support closing of the OS resources during the checkpointing. Updating all peripherals and succeeding validation required only two days for a developer who is already familiar with the SHAPES platform. Two test applications were used: (1) a Linux kernel (version 2.6.22) boot and (2) a *Lattice Quantum Chromo-Dynamics* (LQCD) [51,173] (refer to Section 4.3) simulation from the domain of theoretical physics, specifically optimized for the SHAPES platform. Booting the Linux kernel on a single SHAPES tile and mounting the file system requires 68.1s. Checkpointing this system takes 2.8 seconds and consumes 261 MB of disk space. Restoring the system requires 3.2 seconds. Hence, by using the checkpoint instead of booting the Linux the desired state can be reached faster by a factor of approximately 20×.

Besides reaching a certain state of an application the user also needs to setup the debuggers, the different views and the breakpoints. This overhead is not necessary if the application is checkpointed. Hence, an even better ratio between simulation plus setup of the environment and restoring can be expected.

Due to its excellent scalability the LQCD application is executed on different numbers of SHAPES tiles to evaluate the scalability of the checkpointing in terms of simulation size. Table 5.1 shows the checkpointing time and the uncompressed image size for different numbers of tiles. The case study is conducted on an Intel Quad Core Q6700 based system with 2.66 GHz and 16 GB of RAM using Scientific Linux 5.0 as operating system.

The obtained results show that the checkpoint image size is less than linearly growing with the number of tiles simulated. The time required for checkpointing is growing approximately linearly. Caching of the underlying operating system allows fast creation of checkpoints. However, caching only works if the checkpoint size does not exceed the disk cache size. In case of the LQCD application, the 8 tiles version requires 4.1× instead of 2× longer than the 4 tile version due to this effect.

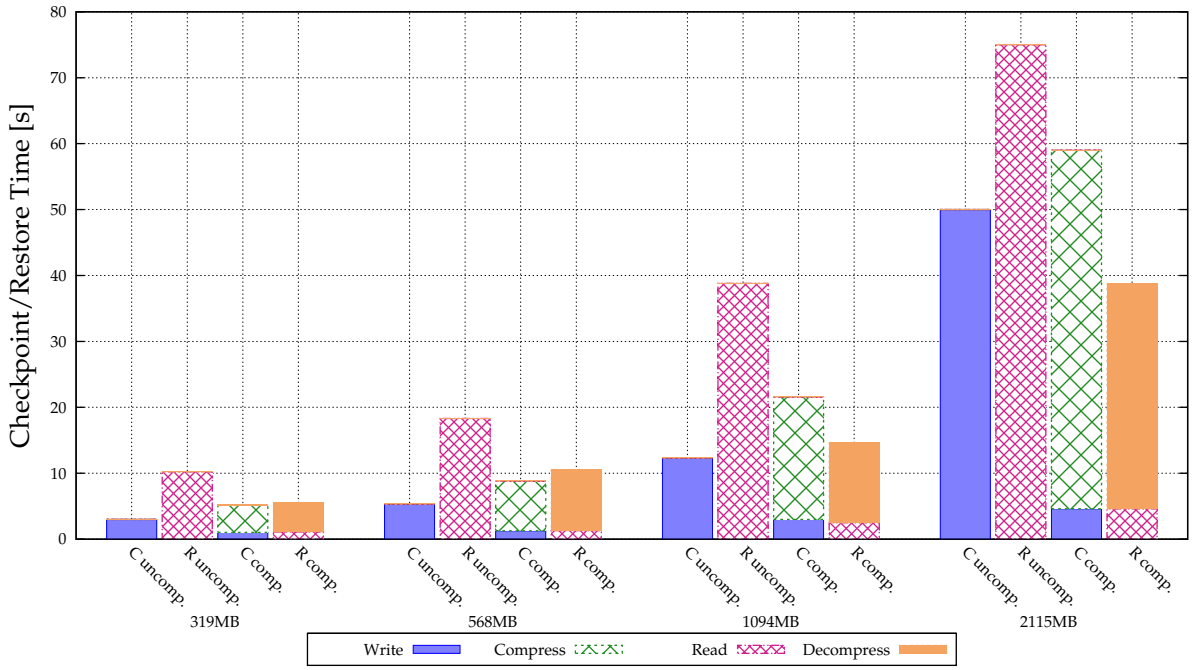


Figure 5.8: Checkpoint/restore time for compressed and uncompressed checkpoints

During the development of the SHAPES LQCD application for the 8 tile platform a bug in the remote DMA driver (RDMA) caused the LQCD to produce wrong results. Driving the application in detailed simulation mode into the state where this problem was suspected took 16min 21s. Naturally, this state had to be generated many times before it was possible to locate the reason for the bug. With checkpointing it takes only 32.1s to store the state of the simulation. Restoring the checkpoint takes around 34 seconds. This clearly shows that debugging of applications running on VPs benefits from checkpointing.

In order to analyze the implications of compressing/decompressing the checkpoints at runtime, the LQCD application running on multiple tiles is used as test case. Figure 5.8 shows the measured times required for checkpointing and restoring compressed (comp.) and uncompressed checkpoints (uncomp.). The simulation complexity and respectively the checkpoint size has been varied by running the LQCD application on one, two, four and eight tiles, leading to uncompressed checkpoints ranging from 319 MB to 2115 MB. All experiments have been conducted using a local hard disk capable of storing data with approximately 42 MB/s. Experiments varying the compression rate have revealed that the fastest compression mode shows the best overall performance. Fast compression is obtained at the cost of a reduced compression ratio. However, on average the size of the compressed images can be still reduced by a factor of 10.

The measured results for the checkpoint creation show that the compression overhead is not fully compensated by the reduced image size. This effect can be explained by the combination of a fast local hard disk and the caching effects of the underlying

file system. In case of slow storage media, e.g. laptop hard disk, the picture is completely different. Here, the compression is capable to reduce the checkpoint size and the checkpointing time. This is completely different for the checkpoint restore process. In this case the compression reduces the restoration time significantly. The reason for the completely different behavior of checkpointing and restoring is twofold. First, the read-ahead caching of file system does not improve the loading of the checkpointing significantly. Hence, reducing the amount of data required to be read from the hard disk size has a direct impact on the time necessary for restoring a checkpoint. Moreover, the deflate algorithm belongs to the group of asymmetric compression algorithms, which requires much more time for compression than for decompression.

5.7 Conclusions and Outlook

The presented case study demonstrates that the C/R feature provides a valuable contribution to the software developers' productivity by shortening their typical edit-compile-debug cycle. To evaluate the applicability and the impact of this framework on complex VPs it has been integrated into the CoWare Virtual Platform environment. Compared to other checkpointing approaches this framework is specifically tuned for SystemC by supporting the checkpointing of SystemC simulations with debuggers and GUIs connected. Thus, the full debug scenario around a VP is supported. Complete transparency has not been enabled by the proposed approach, however the changes to achieve full compatibility require only minimal effort and usually consist of only a few lines of C++ code. The checkpointing procedure has been proven to be fast and reliable. In order to mitigate the large checkpoint size, the checkpoints can be compressed at runtime before being stored. On average a compression factor of 10 is reached. Depending on the underlying file system and on the speed of the storage medium the checkpointing time may or may not benefit from the compression. Because of the asymmetric behavior of the deflate algorithm the checkpoint restore process always benefits from compression. Due to the utilization of process checkpointing the presented approach is sensitive to changes of the VP and fails after a recompilation of the VP. Therefore C/R feature is most useful for an engineer who is developing software on top of a VP.

Further improvements of the checkpoint size are possible by the implementation of incremental checkpoints. Only the differences between two subsequent checkpoints are stored and thus the memory consumption can be further reduced. However, it can be expected that the restore time will increase due to the fact that several incremental checkpoint images have to be read to restore the state of the simulation.

Chapter 6

Hybrid Simulation

As discussed in the previous chapters the simulation performance is of high importance, especially in the context of VPs. For software developers VPs can only be considered as a viable solution for early software development, if the simulation performance is sufficiently high, so that the software development turnaround times are not significantly affected compared to a real hardware platform. Furthermore, in order to cope with the growing software complexity faster simulation techniques are required to ensure the future usability of software development using VPs.

A detailed analysis of VPs used for software development reveals that the major building block affecting the overall simulation speed is the processor simulation. This observation triggered a considerable amount of research activities to improve the simulation performance of ISSs. In Chapter 3 the most prominent simulation approaches developed over the last years are briefly discussed. Reviewing the different solutions proposed to tackle the simulation speed problem shows that ISS based simulation techniques have been exhausted to the extreme. Not much speedup can be expected from this traditional approach in future. Hence, it is necessary that new simulation techniques will replace today's prevailing ISS based processor simulation. Despite all the development efforts in the recent years the simulation speed is still very low compared to real hardware. In fact, not only the simulation techniques improved but also the complexity of the systems being simulated was growing at the same time, partly compensating the achieved speedups.

In general, the speed of a processor simulation is inversely related to the level of abstraction, due to the reduced amount of states that needs to be computed. Hence, the software developers have the possibility to trade-off simulation performance with accuracy depending on their requirements. For example, debugging a complex application requires first to simulate the application until the point in time is reached where the problem is expected. Then, in a second step the developer will examine the behavior of this code fragment carefully to localize the problem. Clearly, during the first step the simulation speed is of primary concern, whereas in the second step the emphasis is on the simulation accuracy.

Based on the observation that the required simulation performance and accuracy feature a high variability over time, e.g. during application debugging, a hybrid simulation framework called *HySim* is proposed. In contrast to most other simulation techniques, this technique clearly focuses on the requirements of software developers rather than the hardware centric requirements of system designers. This framework

offers the possibility to switch between an abstract simulation (AS) mode and an ISS based simulation mode at runtime. The main objectives of the proposed solution are:

Performance – High simulation speed is achieved by native code execution on the host machine. All applications with platform independent code, e.g. ANSI C code, can benefit from the native execution and can be simulated using the abstract simulation technique.

Performance estimation – During the abstract simulation a runtime performance estimation is applied to provide the user with approximate performance figures of the application without the need for a detailed simulation. In particular for multi-processor simulation the timing information is important to maintain the relative order of tasks on different processors.

Compatibility – It is important to keep the simulation framework as generic as possible, i.e. all types of programs should be supported by the framework, without the need for modifying the binary and independent of the presence of debug information. Additionally, assembly code as well as third party libraries should be supported to make the framework adaptable to a wide range of applications.

Transparency – The framework placed on top of an existing ISS should be transparent to the ISS as well as to the other components of the simulated system, e.g. peripheral devices, buses and other processor simulators. This is important especially in the context of system simulation, where the processor simulator is only one building block of the entire system.

Retargetability – In order to apply the HySim framework to a wide range of processor simulations it must be independent of the underlying ISS.

The rest of this chapter is organized as follows. In Section 6.1 the basic concept of the hybrid simulation is presented before the details of the code instrumentation are discussed in Section 6.2. Section 6.3 shows how an application can be partioned to make best use of HySim. Afterwards, Section 6.4 discusses the integration of this technique into an industrial simulation framework. Next, the integrated performance estimation framework is presented in Section 6.5. In Section 6.6 experimental results are presented and discussed. Finally, this chapter is summarized and some future work is discussed in Section 6.7.

6.1 Concept of Hybrid Simulation

This section describes the concept of the proposed retargetable, hybrid simulation approach, which defines a hybrid processor simulation architecture and utilizes an abstract simulation to accelerate the instruction set simulation.

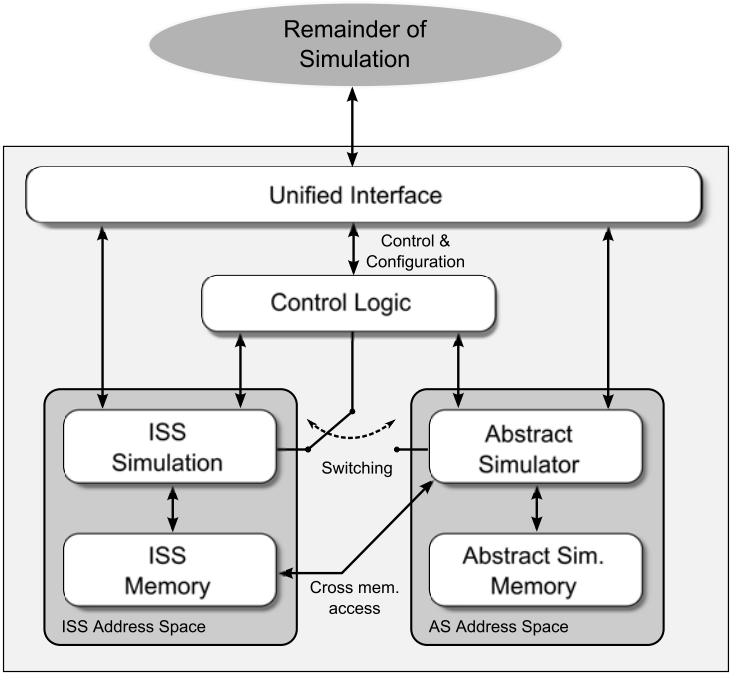


Figure 6.1: Architecture of the simulation system

6.1.1 Architecture of the Hybrid Processor Simulator

The key idea of the HySim framework is to speed up the overall simulation by directly executing parts of the application on the host machine. Unlike binary translation approaches the HySim framework enables the user to decide which parts of the application are simulated in detail and which parts are simulated at a high abstraction level.

From the perspective of the ISS the native code execution can be seen as an abstract simulation which is transparent to all the other components of the simulation system. Thus, this concept can be easily applied to an ISS without affecting the rest of the system simulation. This feature is especially important for large system simulations, since it allows to improve the simulation performance without having to modify other components of the system, such as buses, memories and peripherals. Figure 6.1 shows the structure of a hybrid processor simulator consisting of a target simulator, i.e. a traditional ISS simulator, and an abstract simulator (AS). To ensure the seamless switching between both types of simulation an external control logic is introduced. Due to the fact that both simulators execute in a mutually exclusive way, no special synchronization mechanism is required for data exchange between both simulators. Each simulator has its own memory to store the state of the application. To ensure the coherence of both simulators the abstract simulator has the ability to access the memory of the target simulation [1]. Details about the switching and the required instrumentation can be found in Section 6.1.3.

[1] In the following the terms *target simulation* and *ISS based simulation* are interchangeable.

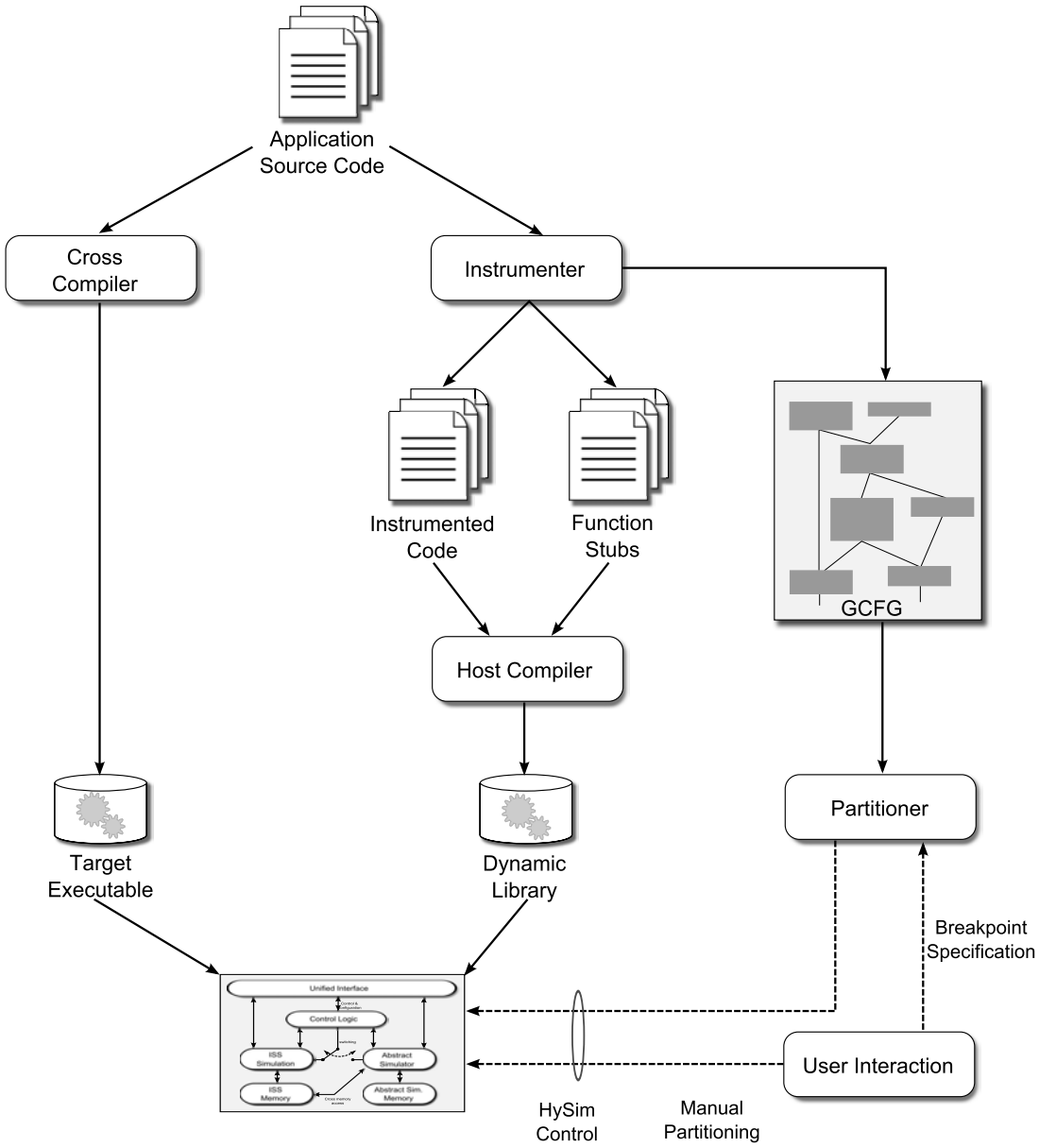


Figure 6.2: HySim software flow concept

6.1.2 Software Flow

In order to utilize the proposed hybrid simulation framework the traditional software flow needs to be extended as shown in Figure 6.2. Two clearly separated branches starting from the application source code can be identified. The left branch represents the standard cross compilation flow which is required to generate the binary for the target simulation. The right branch shows the tool flow required to generate the abstract simulator. Memory accesses in the application source code are automatically instrumented to access the target simulator memory in order to ensure data consistency during the abstract simulation. Furthermore, for each function in the source code a function stub is generated to guarantee seamless switching between both simulation

modes. All generated files are compiled using the available host compiler to produce a dynamic library containing the abstract simulator. Additionally, the instrumenter analyzes the application and generates a *global control flow graph* (GCFG). This GCFG serves as input to the automatic partitioner, together with a breakpoint specified by the user. Based on these inputs, the partitioner will segment the application between the ISS based simulation and the abstract simulation in such a way that the specified breakpoint is reached as fast as possible. More details on the partitioning algorithm can be found in Section 6.3. Furthermore, it is also possible for the user to bypass the partitioner and directly control the hybrid simulation.

6.1.3 Abstract Simulator

The abstract simulator forms the heart of the hybrid simulation by offering high simulation speed. In the following the details of the AS are presented. Currently, the binary translation technique reaches up to 500 MIPS. However, its application is mainly limited to RISC like processor architectures. In comparison to the available raw computing performance of today's PCs even the fastest simulations are one to two orders of magnitude slower than the native target execution. This gap mainly stems from the overhead introduced by simulating the target architecture. The simulation of a single target instruction requires on average between 10–100 host instructions for instruction accurate simulations. Because of this simulation overhead, a hybrid simulation capable of switching between an ISS based simulation and native code execution is expected to achieve even higher simulation rates. Apart from this, the proposed abstract simulator is independent from the simulated target architecture. Hence, a wide range of processor types can be equipped with this technique.

As presented in Chapter 3 there exist different hybrid simulation approaches. The main differentiators between those approaches and the HySim framework are the applied switching level and the supported type of instruction set architectures (ISAs). Most hybrid simulators offer the user the possibility to switch between cycle accurate and instruction accurate simulation, e.g. Tensilica. PTLSim [179] provides the possibility to switch between a CA simulation and native code execution. However, in case of PTLSim this switching is greatly simplified by the fact that the ISA of the host and of the simulation target are identical.

In the domain of processor and microarchitecture development IA simulations are considered to be abstract since most of the microarchitectural features are not simulated, e.g. pipeline and bypassing. In this domain IA simulations are used as an instrument to drive a simulation quickly into a state of interest. This kind of hybrid simulation is clearly tailored towards the needs of architecture developers. Because in the light of an efficient software development the switching between CA and IA simulation is not adequate. For the software development the simulation performance is of high importance, therefore HySim provides the user with the possibility to switch between an IA level simulation and native code execution. It is important to note that hardware developers and software developers have a different notion of functional simulation. Hardware oriented developers already consider IA level simulations as functional since the obtained timings are not fully accurate. In contrast, software de-

velopers refer to functional simulation if no timing information at all can be obtained from the simulation, e.g. native code execution. For the majority of the applications and software modules developed, the accuracy of IA simulations is sufficient for the evaluation of the functionality and for the debugging. In that sense the HySim framework offers software developers the possibility to switch between IA simulation and functional simulation.

From all hybrid simulation approaches in literature the *Virtual Processing Unit* (VPU) proposed by Kempf et al. [73] bears the highest resemblance of modeling a processor simulator at a very abstract level. However, the VPU technology offers the user the possibility to perform *spatial* hybrid simulation by combining ISS based processor simulation with VPU based processor simulation, i.e. before simulation the user can decide for each processor simulator if this instance should be simulated at a high abstraction level or in detail. In contrast, the HySim framework provides the possibility of *temporal* hybrid simulation, i.e. at runtime the user can decide for each processor simulator whether he would like to perform abstract simulation or IA level simulation.

In general, switching between an ISS based simulation and an abstract simulation is a very complex and architecture dependent task (see Equation 6.1), since the entire state of the simulated program must be transferred to the abstract simulator to ensure a coherent state.

R_{CPU}	:	All registers describing the CPU state
R_{PC}	:	Program counter
R_{param}	:	Registers containing the function parameters
M_{text}	:	Text section contains the application code
M_{data}	:	Data section stores all global and static variables in the data section
M_{stack}	:	Stack section stores the local data and function parameters
M_{heap}	:	Heap section contains global data that is allocated and deallocated at runtime

$$ApplicationState = R_{\mathrm{CPU}} \times M_{\mathrm{text}} \times \underbrace{M_{\mathrm{data}} \times M_{\mathrm{stack}} \times M_{\mathrm{heap}}}_{\mathrm{Data}} \tag{6.1}$$

Due to compiler optimizations a one-to-one relation between variables in both simulators can not be assumed, e.g. some variables might be optimized away and their respective values are held in registers. In order to establish a mapping between both simulators additional information is required, for example the debug information included in the program could be used. However, relying on debug information has two weaknesses: First, this limits the approach to processors with toolchains supporting debug information and applications with debug information available, which might not be the case for processors under development. Additionally, different debug formats, such as DWARF [159] and STABS, need to be supported to make this approach retargetable. Second, in case of compiler optimizations the accuracy of the debug information is greatly diluted, rendering this approach infeasible.

$$ApplicationState = R_{\mathrm{PC}} \times R_{\mathrm{param}} \times M_{\mathrm{text}} \times \underbrace{M_{\mathrm{data}} \times M_{\mathrm{heap}}}_{\text{Global Data}} \times M_{\mathrm{stack}} \tag{6.2}$$

The problem of a state transition between both simulators can be greatly simplified by restricting the switching to function borders (see Equation 6.2). At a function border the entire state can be described by the program counter (R_{PC}), the function parameters passed via registers (R_{param}), the global memory and via the stack (M_{stack}). Local variables do not need to be considered since their life range is limited by the begin and the end of a function. Thus, this approach circumvents the problem of register-to-variable mapping without the need for debug information. The coherence of the global variables is guaranteed by maintaining only one single instance of them. The abstract simulation can directly access the simulation memory of the IA simulation by a set of API functions (see cross memory access in Figure 6.1).

In the following the details of the instrumentation step necessary for the global memory accesses are discussed.

6.2 Source Code Instrumentation

This section discusses the required source code instrumentation and its implication on the different language constructs. The source code instrumentation is necessary to implement the cross memory access of the AS. Without the instrumentation it would be impossible to keep the global variables located in the ISS address space in sync while running the AS. Apart from this, the details of the switching between both simulation modes are presented in this section due to its close relation to the application analysis and instrumentation.

6.2.1 Bidirectional Invocation

In order to switch between both simulation modes during runtime two concepts are utilized: The transition from ISS based simulation to AS is realized by a *forward invocation* and the opposite direction is realized by an *inverse invocation*. The reason for the two different approaches are the different boundary conditions for both simulation modes. For compatibility reasons no modifications on the application binary are possible. However, this restriction does not apply to the instrumented program code.

The forward invocation is realized by using the concept of stubbing. For each source code function the instrumenter automatically generates a function stub (see Figure 6.4). This stub forms a bridge between both simulation modes and translates the function calls. While starting the simulation a link between the functions on the ISS and their corresponding functions on the abstract simulator are created. Based on the knowledge of the calling conventions of the target compiler it is possible to extract the function parameters and transfer them to the abstract simulator. The generated stubs and the associated instrumented functions together form the abstract simulator.

The concept of stubbing cannot be applied for the transition from AS to ISS because no modifications on the target binary are allowed. Having the application source code

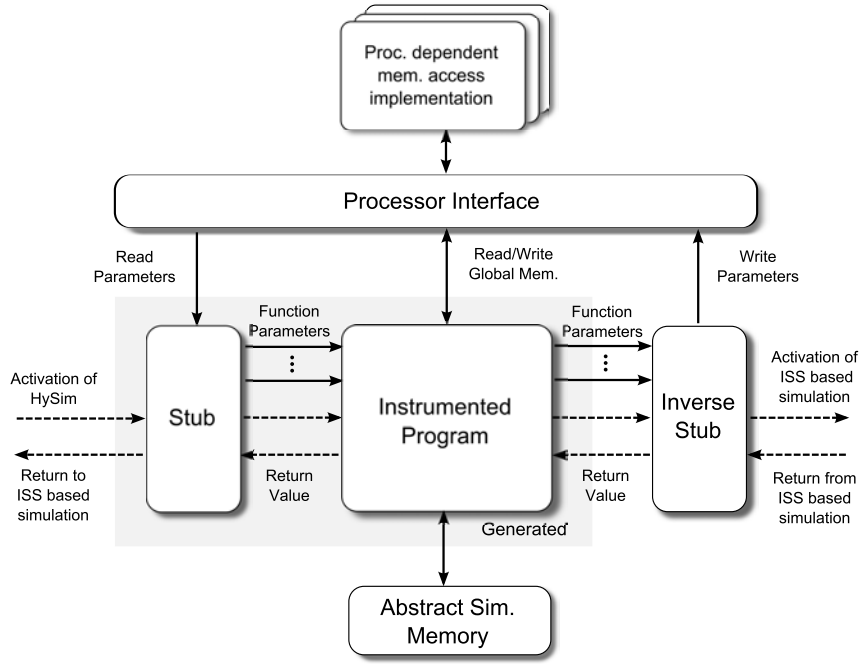


Figure 6.3: Invocation of abstract simulation from the ISS based simulation and vice versa

```
1 void foo() {
2    // Stub for foo()
3    int Param1;
4    int Return;
5
6    Param1 = GetPARAM_int(1); // Transfer parameter from ISS
7    Return = AS_foo(Param1);  // Execute foo() on AS
8    SetRETURNVALUE(Return);
9 }
```

Listing 6.1: Example of an automatically generated stub running on the AS

at hand, it would be possible to add the stub directly into the source code and recompile the application binary. However, this approach works only if the complete source code including the different libraries are available. In order not to restrict the hybrid simulation the concept of inverse stubbing has been selected for the transition from AS to ISS. This approach does not require any changes on the target binary itself. In case that a function `f()` executing on the abstract simulator needs to execute a function, e.g. `printf()`, on the ISS, the inverse stub is called first. It transfers the function parameters into the ISS and monitors the execution of the ISS. Once the execution of the function finishes, the result value is transferred back to the abstract simulator and the abstract simulation is continued. The inverse stubs are mainly used to execute C standard library function calls, e.g. `malloc()`, inside code running on the AS.

```
1 void* LIBC_malloc(size_t size) {
2    // Inverse stub for malloc()
3    long ReturnPC;
4    void* ReturnValue;
5
6    StoreContext();     // Store processor context
7
8    SendArg(0,size);    // Transfer argument to ISS
9
10   ReturnPC = Jal(LIB_TP_malloc); // Set PC to address
11                                  // of function malloc
12   RunUntil(ReturnPC); // Run ISS
13
14   ReturnValue = (void*) GetReturnValue();
15   RestoreContext();   // Restore processor context
16
17   return ReturnValue;
18 }
```

Listing 6.2: Example of an inverse stub running on the AS

Figure 6.3 depicts the forward and inverse invocation. The stub, the inverse stub and the instrumented program access the function parameters and the global memory via an API in order to keep the components independent of the used architecture and ISS. As an example Listing 6.1 shows a generated stub transferring one function parameter from the ISS to the AS and after execution of the function `foo()` the stub copies the computed return value back to the ISS. Exemplary, Listing 6.2 shows the inverse stub for the `malloc()` function. The inverse stub executes the `malloc()` function on the ISS, hence, it will change the ISS context, i.e. PC and registers will be modified. Therefore, the inverse stub must first store the original context of the ISS in order to restore the state of the ISS later on. In the next step the argument `size` is copied to the ISS. The ISS simulator is executed until the execution returns from the `malloc()` function. Next, the return value is copied from the ISS and the previously saved ISS state is restored. Finally, the obtained result value is returned.

6.2.2 Instrumentation

The instrumenter is the heart of the HySim framework. It ensures that a consistent state is maintained between both simulation modes. This is achieved by analyzing the application source code and then instrumenting it. So that the AS can access the ISS address space in order to keep the global variables consistent in both simulation modes.

As discussed in Section 6.1, both simulators maintain their own, separate address space (see Figure 6.1) called the *ISS address space* and the *AS address space*. However, to guarantee a consistent state, it is important that the abstract simulator can access the ISS address space. A direct access to the ISS address space from the AS is not

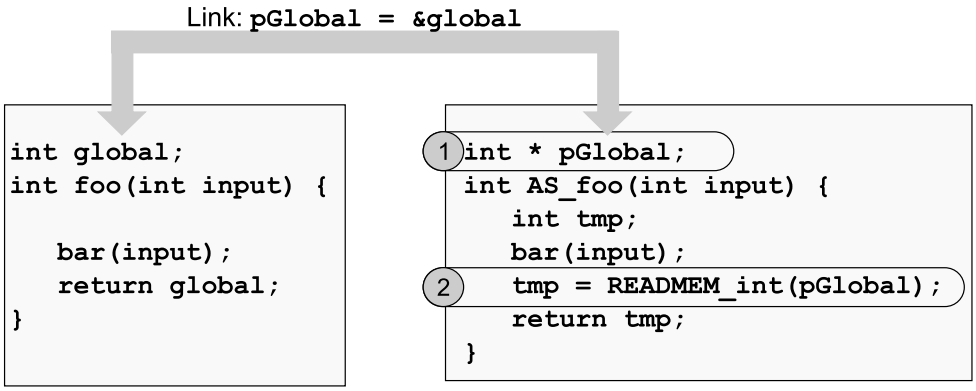


Figure 6.4: Example of the required instrumentation to enable switching between both simulation modes

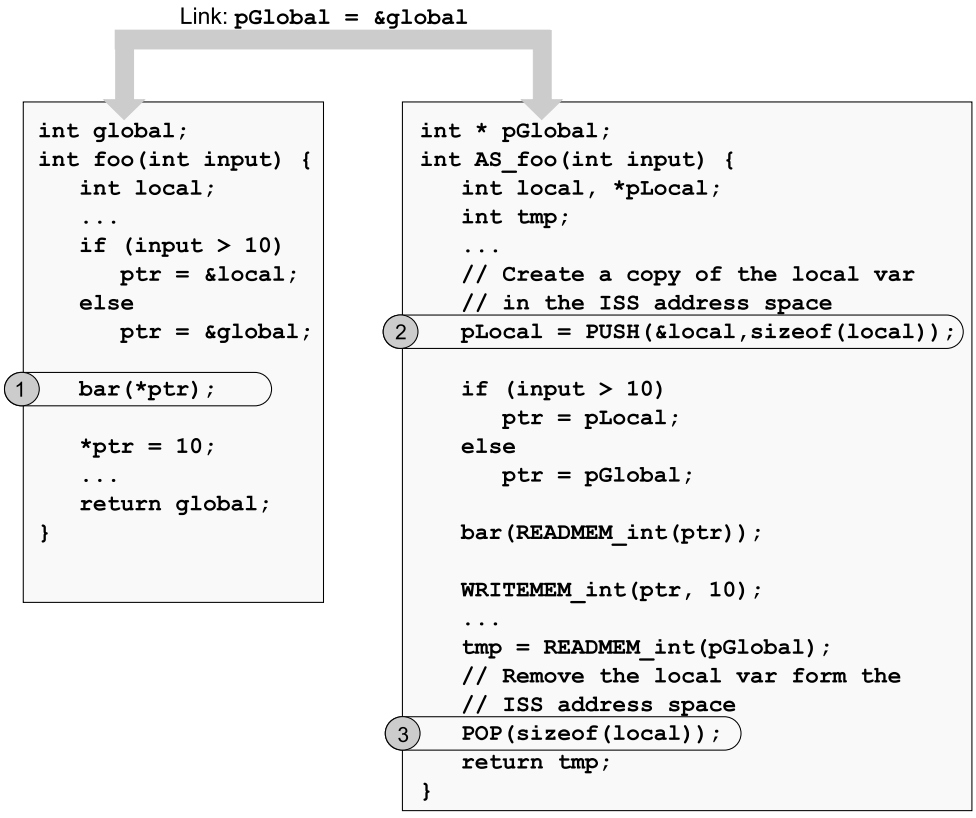


Figure 6.5: Resolving the pointer conflict introduced by using the address of a local variable

possible, because dereferencing a pointer to an ISS memory location does not access the correct data and might even access an invalid address in the AS address space and thus cause a segmentation fault. Therefore, the memory access is realized via a set of helper functions. These functions can access the memory of the ISS and offer the AS the possibility to directly manipulate the content of the ISS memory.

The instrumenter takes care of automatic transformation of all memory accesses into calls to the appropriate helper functions. Most frequently the ISS address space is accessed via global variables and pointers. In the following, the different language constructs that need to be handled by the instrumenter are discussed:

Global variables – Global variables and local static variables can be accessed either by names or via pointers in C [75]. In the first step the instrumenter declares a unique global pointer for each of these variables and assigns the pointer to the address of the corresponding variable in the ISS address space. The address of the global variable in the ISS address space is determined by reading the memory map (e.g. using objdump) of the target binary. This link between both address spaces ensures that a value change in one address space is also reflected in the other. In the second step the instrumenter replaces all read/write accesses to these variables by a function call to the appropriate helper functions.
Figure 6.4 shows how the instrumentation is performed for a simple example. On the left side of the figure the original source code and on the right side the instrumented code is shown. Function `foo()` returns the value of the global variable `global`. Since the value of this variable can not be determined at compile time in general, the instrumenter will create a global pointer `pGlobal` and fills it with the address of the variable `global` in the ISS address space (1). This is done while loading the abstract simulator. In a subsequent step the read access to the global variable is replaced by a function call to the helper function `READMEM_int()` (2).
Using helper functions instead of direct memory access introduces a runtime overhead to the abstract simulation. Thus, unnecessary instrumentation needs to be avoided. For example, for global or static variables declared with the `const` keyword, an additional copy of the variable is created in the AS address space. In this case the instrumenter can directly access the local copy without the need for helper functions.

Local variables – Local variables only reside in the AS local memory, so accessing them directly is safe. However, accessing the local variables through pointers introduces ambiguity because during instrumentation time it can not be determined in which address space the pointer is located. It is even possible that the address space of the pointer depends on the function call. For example, the pointer could point to a global or to a local variable. This problem is solved by copying all local variables that are accessed by pointers, e.g. local arrays, to the ISS address space. Hence, this ensures that pointers can only point to elements residing in the ISS address space. Figure 6.5 gives an example of the described pointer conflict. Depending on the function parameter `input` the pointer `ptr` is pointing to a local variable or to a global variable. Both variables are residing in different address spaces. This conflict cannot be resolved during compile time (1), therefore, the local variable is copied into the ISS address space using the `PUSH()` helper function (2). The `PUSH()` function puts the data on top of the processor stack and returns a pointer to this address location to the AS. Using this pointer `pLocal` in-

stead of the original local variable `local` resolves the pointer conflict, since now `pGlobal` and `pLocal` are both pointing to the ISS address space and both can be accessed using the same helper function. At the end of the function `foo()` the local variable is removed from the ISS stack by calling the `POP()` function (3).

Aggregated data structures – The specification of the language C does not define the layouts of aggregated data structures like `struct` or `union`, different compilers may produce different memory layouts for aggregated data structures. Special helper functions are required to translate between the different memory layouts. In the current implementation of the instrumenter this problem is avoided since the target compilers employed for the case study have an identical data layout as the host.

Floating point – Floating point computation is complicated to handle by the instrumenter because it is highly platform dependent, e.g. the rounding strategy is not defined in the IEEE 754 [63] standard. Hence, by switching between both simulation modes the precision of the results might be affected. In case that exactly the same precision is required in both simulation modes, all floating point operations must be instrumented and the required precision is then emulated on the host. However, this reduces the expected speed-up of the native code execution and should not be necessary in the general case.

C Standard Library – In general, there are two possibilities to handle calls to the C standard library: First, map the function calls to the corresponding function calls on the host machine. Second, switch back to the detailed simulation to execute the function call. The latter approach is more complicated to realize since it requires an inverse stub (see Listing 6.2) for each function call. However, this approach is the more general solution, because it also allows to execute functions with side effects. For example, mapping the function call `fabs()` to the host machine is not a problem since this function has no side effects. But for function calls like `malloc()` this approach is not feasible, since they have side effects such as reducing the available memory on the heap. In order to keep the HySim framework as general as possible all calls to the C standard library are executed on the ISS based simulation at the cost of a reduced simulation speed.

6.3 Application Partitioning

The HySim framework offers the user two possibilities of partitioning an application: *manual partitioning* and *automatic partitioning*, as depicted in Figure 6.2. The manual partitioning gives the user the complete freedom to choose which functions will be executed on the AS and which functions remain on the ISS based simulator. However,

no sanity checking is performed, the user is responsible to ensure the correctness of the partitioning.

In case of automatic partitioning, the user only marks a code location with a special breakpoint called *fast-forward breakpoint* (FF breakpoint). Based on the GCFG the partitioning is then automatically performed in such a way that the marked code location is reached as fast as possible, i.e. the partitioner tries to execute as many functions as possible on the AS. Naturally, executing the complete program on the AS would result in the fastest possible partition to reach the desired code location. However, the trivial solution is not feasible in general, because not all types of functions can be executed on the AS without compromising the switching capability. For instance, all functions contributing to the actual call stack on the ISS must be simulated on the ISS based simulation, otherwise it is not possible to continue the execution of the application after switching back from the AS.

6.3.1 Global Control Flow Graph

The global control flow graph $GCFG = (N, E)$ is a *directed graph* where each node $n \in N$ represents a basic block (BB) and each edge $(n_i, n_j) \in E$ depicts a control flow transition from BB n_i to BB n_j. The GCFG can be seen as an extension of the traditional control flow graph (CFG) [3] which represents the control flow between basic blocks at function level. The GCFG is obtained by merging all local CFGs together, so that:

$$N = \bigcup_{i=1}^{K} N_{\mathrm{CFG}_i} \tag{6.3}$$

$$E = \Big[\bigcup_{i=1}^{K} E_{\mathrm{CFG}_i} \Big] \cup E_{\mathrm{call}} \tag{6.4}$$

where E_{call} denotes the set of edges from the function call to the entry node of the callee and K is the number of functions in the application. The GCFG is essential for program partitioning since it contains all information about the function execution sequence.

On the function level the GCFG can be described as a directed graph $GCFG_{\mathrm{func}} = (M, D)$:

$$M = \{1, ..., K\} \tag{6.5}$$

$$D = \{(m_1, m_2) | \, \exists (n_1, n_2) \in E, n_1 \in N_{CFG_{m_1}}, n_2 \in N_{CFG_{m_2}}\} \tag{6.6}$$

where M represents the the set of functions described by the GCFG. An edge $(m_1, m_2) \in D$ denotes that a function m_1 contains a call to a function m_2.

In Figure 6.6 the GCFG generation is exemplified using a simplified version of the GCFG. The GCFG comprises eleven CFGs ($K = 11$), one for each function, and starts with the `main()` function. Inside the `main()` function four function calls can be seen, each function call points to the CFG of the corresponding function. Hence, the GCFG

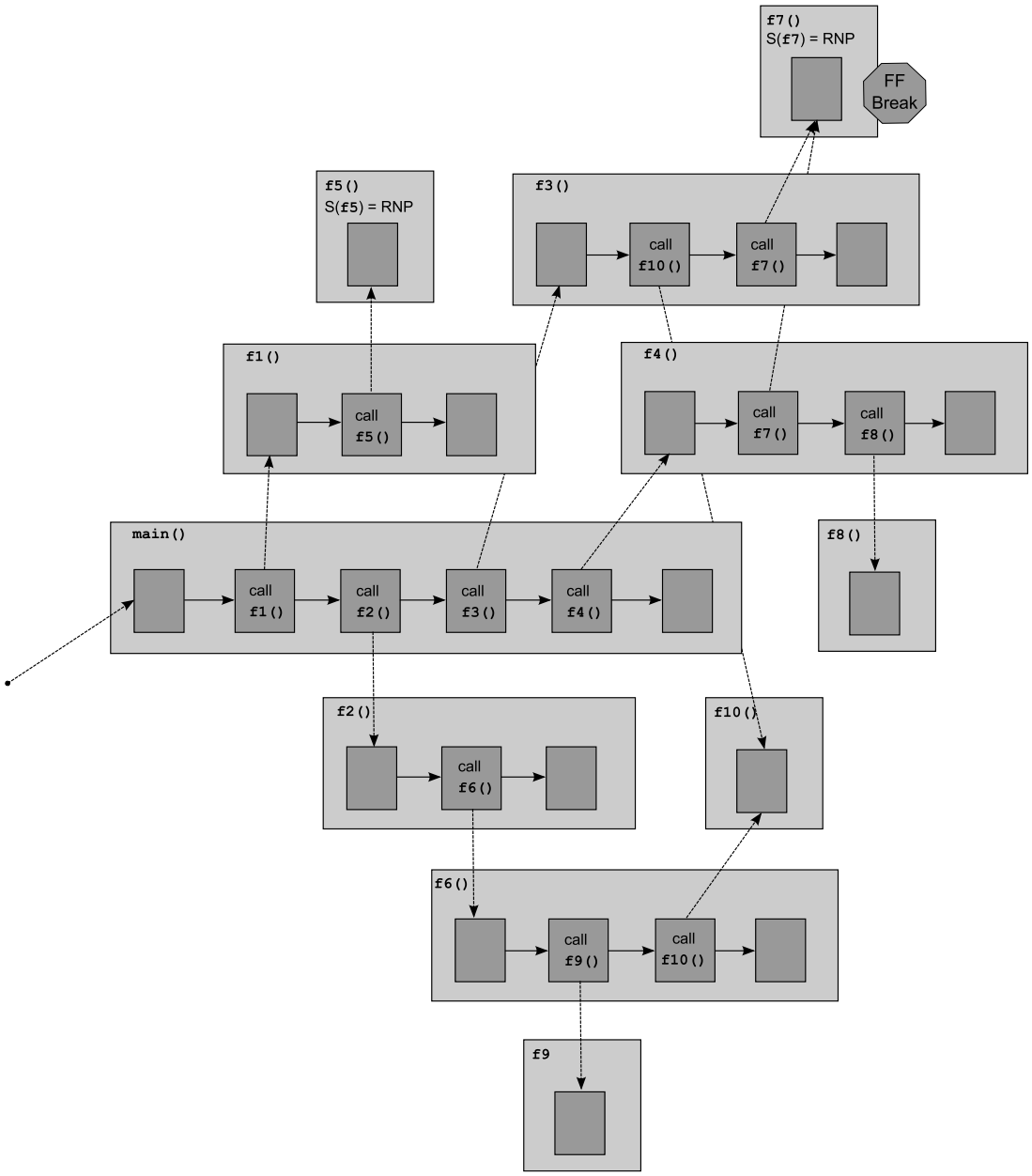


Figure 6.6: Example of a GCFG with a FF breakpoint

provides information about all possible calling sequences of functions and about the possible call stack for a given function.

In the GCFG each function is associated with a status flag $S_m \in \{\mathrm{P}, \mathrm{NP}, \mathrm{RNP}\}, m \in \{1, ..., K\}$. This flag can indicate three different states: *partitionable* (P), *non-partitionable* (NP) and *recursive-non-partitionable* (RNP). By default all functions are assumed to be partitionable. The states NP and RNP both indicate that a function cannot be executed on the AS but needs to be executed on the ISS simulator instead. The difference between NP and RNP is that RNP effects not only the marked function but also all functions contributing to the call stack of the marked function. Hence, this flag can be utilized to ensure that the function call stack is not corrupted by switching between

Status flag S_m	Description
P, Partitionable	By default all functions are considered to be partitionable.
NP, Non-Partitionable	All incomplete functions, i.e. where only the function prototype is available, are marked as NP.
RNP, Recursive-Non-Partitionable	Functions containing function pointers, direct stack manipulation or marked by the FF breakpoint are considered to be RNP.

Table 6.1: Overview of the partitioning flag S_m

the simulation modes. Table 6.1 summarizes the different states possible for the status flag S_m.

As shown in Figure 6.2 the GCFG generation is part of the analysis and instrumentation phase. During the GCFG generation each function of the application is also analyzed and the flag S_m is set to the corresponding value. In the following, the different code features relevant for the GCFG generation are presented and their handling is described:

Static functions – The visibility of `static` functions is limited to a compilation unit [75], therefore multiple static functions belonging to different compilation units may possibly have the same name. In order to resolve this ambiguity the creation of the GCFG is performed in two steps: First, all CFGs belonging to one compilation unit are merged together. Static functions are prefixed with the name of the compilation unit for a non-ambiguous identification. Second, the resulting compilation units CFGs are merged together to form the GCFG.

Function pointers – In C it is possible to call functions using function pointers. In general it is impossible to determine the possible targets at compile time. Therefore, a conservative approach is selected and all functions containing function calls via function pointers are marked as RNP.

Direct stack manipulation – The C language also supports a set of functions that directly modify the stack and change the control flow such as `setjmp()` and `longjmp()`. Those functions are used for saving the stack context for non-local gotos. In general it is not possible to capture the control flow introduced by those functions in the GCFG. Therefore, all functions containing those language constructs are automatically marked as RNP.

Third party libraries – Third party libraries normally only expose the declaration of their functions. Since the function source code is absolutely necessary for the generation of the AS, it is not possible to execute such functions

on the AS. For each incomplete function, i.e. only the function prototype is available, a pseudo body is created and marked as NP. This pseudo body is required to correctly model the GCFG of the application.

6.3.2 Problem Formulation

The goal of the automatic partitioning algorithm is to maximize the number of functions $|functions_{\mathrm{AS}}|$ being executed on the AS, while adhering to a set of boundary conditions. The problem of automatically partitioning the GCFG graph $GCFG_{\mathrm{func}}$ can be formulated as follows:

$$functions_{\mathrm{AS}} \subseteq M \tag{6.7}$$

under the following constraints:

$$\forall m \in M : S_m = \mathrm{NP} \Rightarrow m \notin functions_{\mathrm{AS}} \tag{6.8}$$

$$\forall\, m \in M : [\exists\, m' \in M : S_{m'} = \mathrm{RNP} \wedge (m, \ldots, m') \in \mathrm{path}(D)] \Rightarrow m \notin functions_{\mathrm{AS}} \tag{6.9}$$

All function that are marked as NP are excluded from the execution on the AS (Equation 6.8), as well as all functions that are marked as RNP or have a path to a function being marked as RNP (Equation 6.9).

$$\begin{aligned} \mathrm{path}(D) := \{(m_1, \ldots, m_i) |\, & \forall j = 1, \ldots, i : m_j \in M \\ & \wedge \forall j = 1, \ldots, i-1 : (m_j, m_{j+1}) \in D\} \end{aligned} \tag{6.10}$$

6.3.3 Automatic GCFG Partitioning

The automatic partitioning of the GCFG is controlled by the user through defining a fast-forwarding breakpoint at a certain function. An FF breakpoint can be set in any function independent of the availability of the C source code. The function with the FF breakpoint is called *FF function*. The goal of the partitioner is to find a good partitioning, considering the fact that not all functions can be executed on the AS. Algorithm 1 shows the partitioning algorithm in pseudo code.

This algorithm uses the graph $GCFG_{\mathrm{func}}$ and the function status flags $S_1 \ldots S_K$ as input. Those flags are obtained by analyzing all functions in the graph and setting the status flag according to Table 6.1. In the first step, the index of all functions been marked as $S_m = \mathrm{NP}$ and $S_m = \mathrm{RNP}$ are copied into the sets $functions_{\mathrm{NP}}$ and $functions_{\mathrm{RNP}}$ respectively. In the subsequent step, all functions marked as RNP are considered. For those functions, also their ancestors have to be marked as RNP. This is done by iteratively marking all predecessors of the functions already marked as RNP. A predecessor m_2 of a function m_1 marked as RNP is found if there is an edge from a function m_2 being not marked RNP to the function m_1. In the last step, all functions that can be executed on the AS are determined by computing the set difference of the set M and the union of the $functions_{\mathrm{NP}}$ and $functions_{\mathrm{RNP}}$ set.

The partitioning is exemplified using the GCFG shown in Figure 6.6. Suppose that `f5()` performs a function call by using a function pointer. Since a function pointer points potentially to all functions, this function is marked as $S_{f5} = \mathrm{RNP}$. The user sets a FF breakpoint within `f7()`, therefore `f7()` is also marked as $S_{f7} = \mathrm{RNP}$ in order

Algorithm 1 Partitioning algorithm

Input: $GCFG_{\text{func}}(M, D), S_1 \dots S_K$
Output: set of $functions_{\text{AS}}$ that can be executed on the AS

```
1: functions_NP ← ∅
2: for all m ∈ M do
3:    if S_m = NP then
4:       functions_NP ← functions_NP ∪ m
5:    end if
6: end for
7: functions_RNP ← ∅
8: for all m ∈ M do
9:    if S_m = RNP then
10:      functions_RNP ← functions_RNP ∪ m
11:   end if
12: end for
13: while ∃ m_1 ∈ functions_RNP, m_2 ∈ (M \ functions_RNP), (m_2, m_1) ∈ D do
14:   functions_RNP ← functions_RNP ∪ m_2
15: end while
16: functions_AS ← M \ (functions_NP ∪ functions_RNP)
```

to maintain the call stack for later simulation on the ISS simulator. The set M contains all functions of the $GCFG_{\text{func}}$ M ={`main()`, `f1()`, `f2()`, `f3()`, `f4()`, `f5()`, `f6()`, `f7()`, `f8()`, `f9()`, `f10()`}. In this example the set $functions_{\text{NP}}$ is empty: $functions_{\text{NP}} = \emptyset$. Initially two functions have been marked as RNP: $functions_{\text{RNP}} =$ {`f5()`,`f7()`}. After computing all ancestor functions being alive the set is $functions_{\text{RNP}} =$ {`main()`,`f1()`,`f3()`,`f4()`,`f5()`,`f7()`}. All functions contained in the $functions_{\text{NP}}$ and $functions_{\text{RNP}}$ sets can not be executed on the AS. The other functions can be executed on the AS, in this example the functions `f2()`, `f6()`, `f8()`, `f9()` and `f10()`. Hence, by executing those functions on the AS the function `f7()` is reached as fast as possible. Figure 6.7 shows the obtained partitioning.

6.4 Integration into a Commercial Simulation Framework

The proposed hybrid simulation technique focuses on the simulation of single processor cores. However, due to the increasing complexity of embedded systems more and more MPSoC architectures are employed. Hence, the simulation of complete MPSoC virtual platforms is of great importance for software development during early design stages. For system simulation SystemC [116] has been established as the de facto standard for modeling.

Due to the fact that the hybrid simulation technique only requires changes within a processor simulator, it can be integrated into an existing system simulation environment with moderate effort. As shown in Figure 6.1 the abstract simulator is transparent

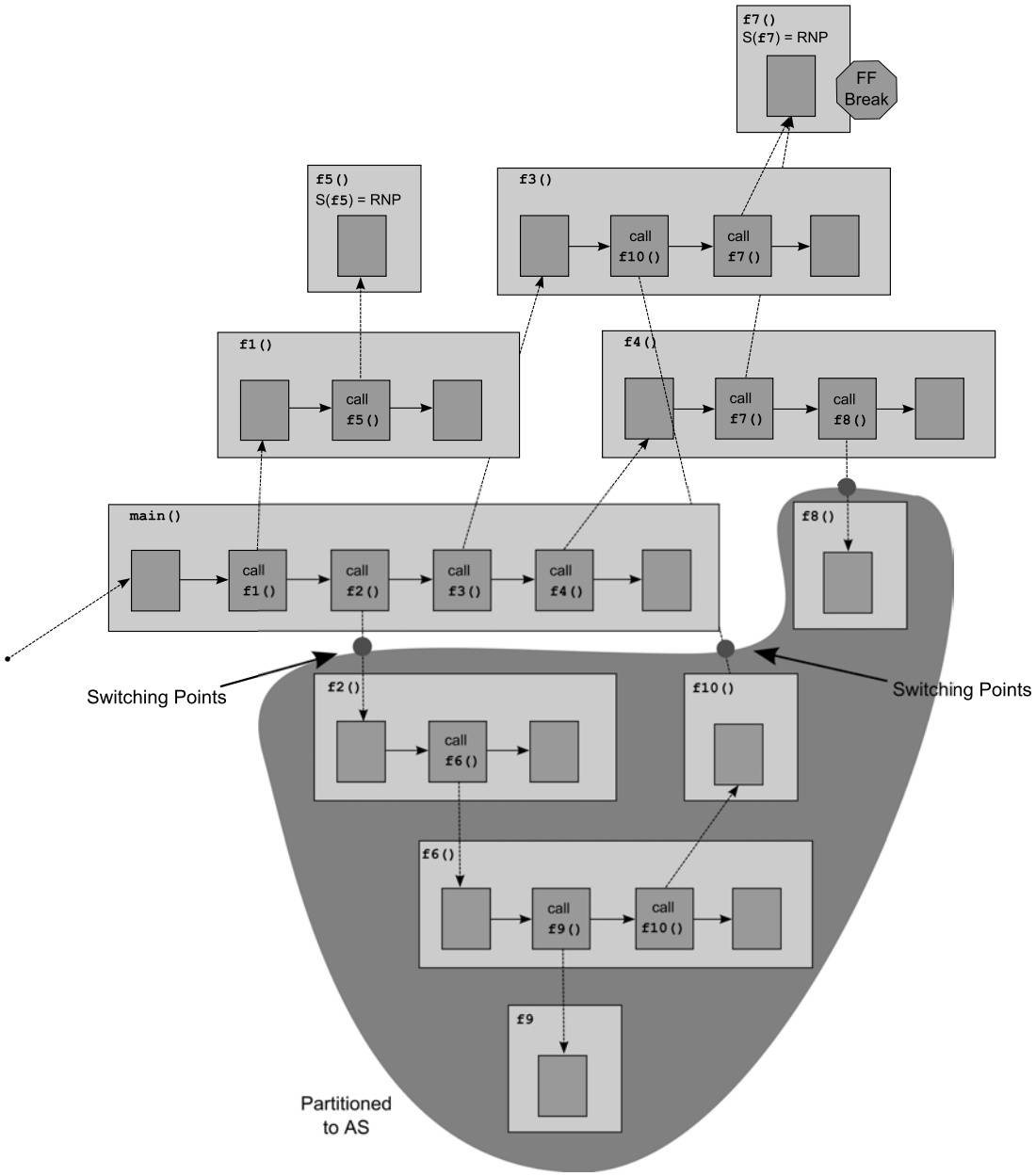


Figure 6.7: Example of a partitioned GCFG to reach function `f7()` as fast as possible

to other system components outside of the processor simulator. A one time effort needs to be spent to integrate a framework that supports for cross memory accesses of the AS.

In order to assess the impact of hybrid simulation on complex system simulations the proposed HySim framework has been integrated into the SystemC based simulation framework from CoWare [33]. Figure 6.8 shows the structure of the realized integration. The hybrid simulation functionality has been distributed over different modules in order to separate the processor dependent and independent parts. All processor independent parts are combined into the *HySim Device*, which is responsible for the switching between both simulation modes, controlling the abstract simulator and

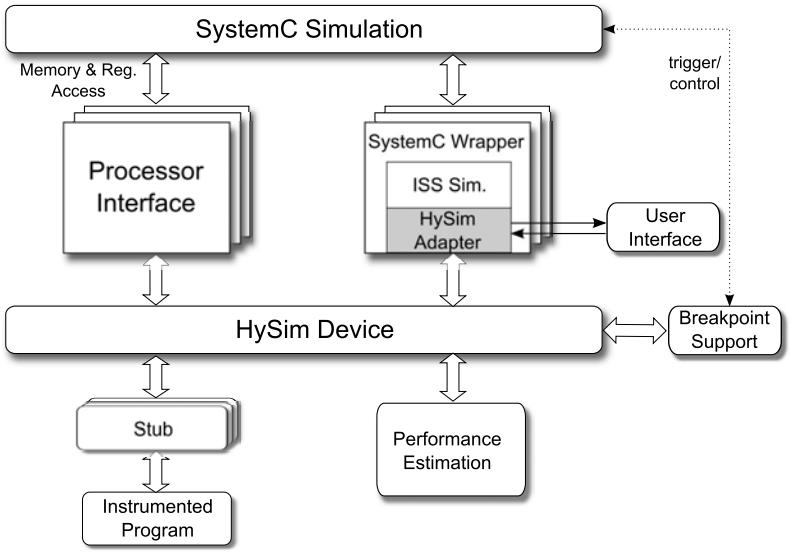


Figure 6.8: Schematic of the HySim integration into a SystemC simulation environment

the performance estimation. The HySim Device is compiled into a library that is linked to the SystemC simulation. The core of the abstract simulation is realized by the instrumented application source code and the corresponding generated stubs. The processor dependent parts are implemented in the *Processor Interface* module and in the *HySim Adapter*. The latter is part of the SystemC processor wrapper and collects information about the ISS simulation that is necessary for switching between both simulation modes. Furthermore, it extends the user interface of the ISS simulation, so that the user can directly interact and control the HySim framework at runtime. The Processor Interface module is responsible for mapping the implementation independent function primitives used for function parameter and memory access to the corresponding processor dependent access functions. In essence, the Processor Interface together with the HySim Adapter capture the processor and compiler dependent characteristics, e.g. used calling conventions, name of the stack register and memory addressing. Hence, for each processor type utilized in the system simulation, a specific Processor Interface and a HySim Adapter needs to be implemented. While all other parts of HySim remain unmodified.

The notification that a switching point is reached during simulation is implemented by setting an ISS breakpoint to the corresponding location in the application code. In order to minimize the overhead introduced by switching, a lightweight instrumentation framework called *instrumentation point* (IPT) is employed to notify the HySim Device. The minimization of the switching overhead is crucial for achieving high speedups with hybrid simulation.

A comprehensive evaluation of the different factors influencing the achievable simulation performance of the abstract simulator is given in the next sections.

Parameters	Description	Depends on
C_{sim}	Average number of host cycles to execute one ISS instr.	Simulator
C_{switch}	Average switching overhead in host cycles	
C_{mem}	Average host cycles to access an ISS memory location	
N_{mem}	Num. of memory accesses to the ISS mem.	Application
N_{func}	Num. of host cycles to execute the function on the host	
N_{ISSinstr}	Num. of ISS cycles to execute the function on the ISS	
R_{mem}	Num. of memory accesses per ISS instruction	
α	Proportionality factor between host and ISS cycles	Proc. type

Table 6.2: List of parameters, their symbols and dependencies

6.4.1 Abstract Simulation Performance Considerations

In order to evaluate the overall speedup with hybrid simulation, it is essential to understand all parameters influencing the simulation speed of the abstract simulation first. Equation 6.11 calculates the achievable simulation speedup in case that a function is executed on the AS instead of using an ISS based simulation. The constants C_{sim}, C_{switch} and C_{mem} denote the average number of host cycles required to execute one ISS instruction, the switching overhead introduced by the HySim framework and the average number of host cycles necessary to access the ISS memory respectively. N_{ISSinstr} denotes the number of instructions to execute the function on the ISS, N_{func} denotes the number of cycles required to execute the function on the host processor and N_{mem} denotes the the number of memory accesses to the ISS memory. Table 6.2 summarizes the different parameters.

$$Speedup = \frac{C_{\text{sim}} \cdot N_{\text{ISSinstr}}}{N_{\text{func}} + C_{\text{switch}} + N_{\text{mem}} \cdot C_{\text{mem}}} \tag{6.11}$$

Assuming that there is a linear relation between N_{func} and N_{ISSinstr} (Equation 6.12) and using the memory ratio R_{mem} (Equation 6.13) it is possible to obtain an approximate speedup equation (Equation 6.14). Table 6.3 shows the proportionality factor α measured for different applications. Typically, the values for α vary over one order of magnitude $(0.33 \ldots 3)$.

$$N_{\text{func}} \approx \alpha \cdot N_{\text{ISSinstr}} \tag{6.12}$$

$$R_{\text{mem}} = \frac{N_{\text{mem}}}{N_{\text{ISSinstr}}} \tag{6.13}$$

$$Speedup \approx \frac{C_{\text{sim}} \cdot N_{\text{ISSinstr}}}{\alpha \cdot N_{\text{ISSinstr}} + C_{\text{switch}} + C_{\text{mem}} \cdot R_{\text{mem}} \cdot N_{\text{ISSinstr}}} \tag{6.14}$$

For the virtual SHAPES platform measurements have shown that the average switching overhead is $C_{\text{switch}} = 1953$ host cycles independent of the processor type. Table 6.4

	des	G721	susan	crc	mandelbrot
α_{ARM}	0.48	0.6	0.33	0.99	0.41
α_{mAgic}	2.14	2.81	-[a]	1.84	1.9

Table 6.3: Measured proportionality factor α for different applications

[a] Not possible to execute the benchmark directly on the mAgic due to I/O constraints

	ARM [host cycles]	mAgic [host cycles]
C_{mem}	500/81 [a]	94
C_{sim}	7660	4382

Table 6.4: HySim cost table for the VSP

[a] Memory access cost using the direct memory interface (DMI)

summarizes the costs for accessing the global memory from the AS via the Processor Interface and the subjacent SystemC simulation and the costs of simulating one ISS instruction. These numbers have been obtained by accessing and averaging the internal counter registers of the host processor over a large set of measurements.

Figure 6.9 visualizes the speedup prediction (Equation 6.14) depending on the number of simulated instructions (N_{ISSinstr}) for different memory to instruction ratios R_{mem}, ranging from 10% to 80%. This figure shows that already comparatively small functions can benefit from the execution on the abstract simulation. Functions with more than 200 simulated instructions achieve already high speedups. Moreover, this figure indicates that the speedup is limited by the number of memory accesses and the corresponding cost to access the ISS memory (Equation 6.15).

$$Speedup \approx \frac{C_{\text{sim}}}{\alpha + \frac{C_{\text{switch}}}{N_{\text{ISSinstr}}} + C_{\text{mem}} \cdot R_{\text{mem}}} \overset{N_{\text{ISSinstr}} \to \infty}{\approx} \frac{C_{\text{sim}}}{\alpha + C_{\text{mem}} \cdot R_{\text{mem}}} \tag{6.15}$$

Hence, high speedups can be either achieved by reducing the number of memory accesses or by reducing the overhead introduced by each memory access. To a certain extent it is possible to reduce the number of memory accesses itself, however, the majority of accesses cannot be optimized away because they are necessary to maintain the simulation context between both simulation modes. Therefore, it is important to minimize the cost per memory access by creating a lightweight memory access interface between AS and ISS.

In case of the SHAPES system simulation framework the cost per memory access for the ARM processor is very high, about 500 host cycles per access are required, due to the high number of abstraction layers that encapsulate the ISS memory. The overhead can be greatly reduced by directly accessing the simulated memory using the *direct memory interface* (DMI) [116] . Using the DMI it is possible to bypass the complete

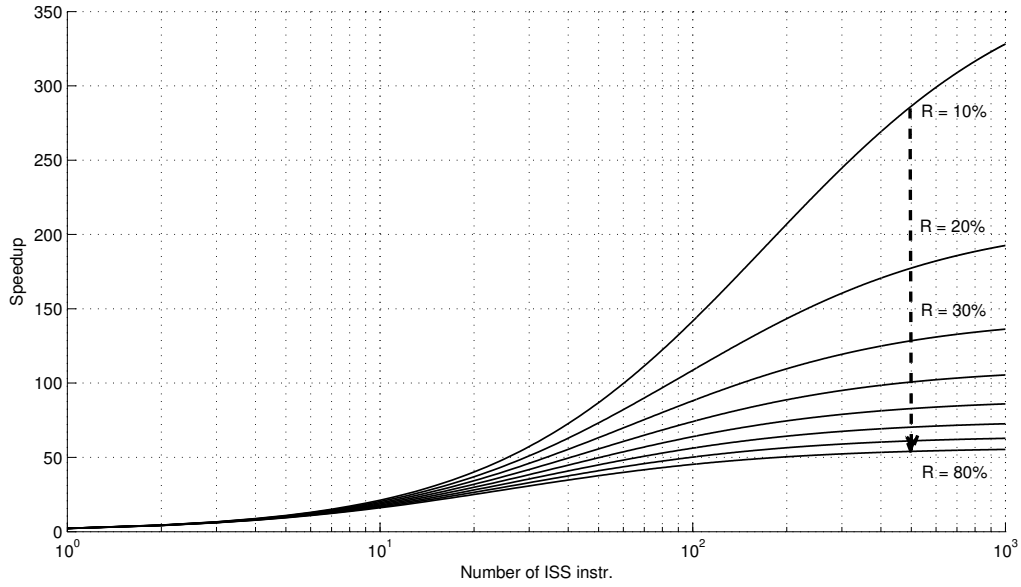


Figure 6.9: Predicted speedup for the mAgic processor with $\alpha = 2.0$, $C_{\text{switch}} = 1953$, $C_{\text{sim}} = 4382$ and $C_{\text{mem}} = 94$

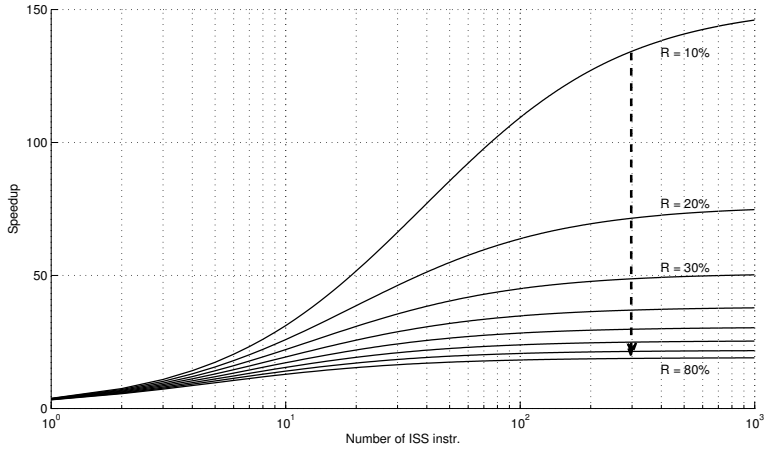
encapsulation and to directly dereference memory pointers in the AS. In case of the ARM processor this reduces the overhead from 500 host cycles to 81 host cycles (cf. Table 6.4). 81 cycles are in the range of a L3 cache miss on the host computer. Hence, all the overhead introduced by the abstraction layers can be fully bypassed. Further improvement is only possible by optimizing the memory layout of the application in order to reduce the number of cache misses. Figure 6.10a shows the predicted speedup for the ARM processor with $C_{\text{mem}} = 500$ and Figure 6.10b shows the predicted speedup using the DMI ($C_{\text{mem}} = 81$ host cycles) to reduce the memory access overhead. Using the direct memory access the achievable speedup can be significantly increased. However, only functions in the range of 1000 simulated instructions can exploit the full simulation speed.

6.4.2 HySim Performance Considerations

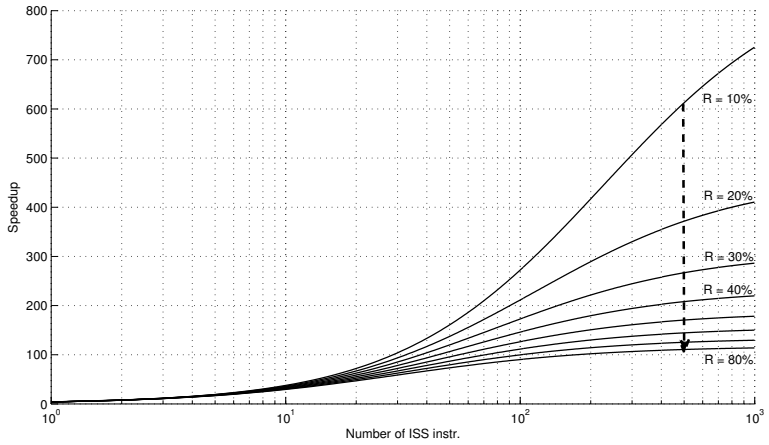
As discussed in Section 6.4.1 the achievable speedup of the AS is mainly determined by the number of memory accesses and their corresponding overhead. The overall speedup for the HySim framework can be calculated by Amdahl's Law [6] as shown in Equation 6.16.

$$Speedup_{\text{App}} = \frac{1}{\frac{Fraction_{\text{HySim}}}{Speedup_{\text{HySim}}} + (1 - Fraction_{\text{HySim}})} \tag{6.16}$$

The overall speedup reachable by the hybrid simulation approach is determined by the speedup gained by executing a function on the abstract simulation (Equation 6.14) and by the fraction of application code that is amenable for the abstract simulation.



(a) Memory access in host cycles $C_{\mathrm{mem}} = 500$



(b) Memory access in host cycles $C_{\mathrm{mem}} = 81$

Figure 6.10: Predicted speedup for the ARM processor with $\alpha = 0.5$, $C_{\mathrm{switch}} = 1953$ and $C_{\mathrm{sim}} = 7660$

Unlike other processor simulation approaches, e.g. ISS based, it is hard to estimate the achievable simulation speed using a hybrid simulation. This is due to the strong dependency of the overall speedup on the application, i.e. the number of memory accesses and the fraction of code that can be executed on the abstract simulator. The measured speedups in Section 6.6 can be used to roughly estimate the impact of the hybrid simulation technique on the simulation performance.

6.5 Performance Estimation

Using the abstract simulator it is possible to reach a given application state quickly, but no target dependent information is provided. Among others, information about the execution time on the target hardware is missing. In case that hybrid simulation

is used for application debugging the missing timing information is not of great importance. However, in a multiprocessor environment the lack of timing information might influence the task schedule of the application [70]. Nevertheless, the user can always obtain the execution time of the application by switching back to an ISS based simulation. In many cases a quick and rough estimate of the execution time would be sufficient. For the above mentioned reasons a performance[2] estimation framework is included in the abstract simulator.

For performance estimation several different techniques have been suggested in literature. Section 3.1.1.2 gives an overview of the different techniques that are all based on timing annotated native execution. The prevalent technique is to annotate timing information to the application source code. The timing information can be either obtained by analyzing the application source code, e.g. μ-Profiler [72], or the user needs to manually specify the required information. For RISC based target architectures this approach yields good timing estimation results. However, this approach is not well suited for complex DSP like architectures, because the timing is strongly influenced by microarchitectural characteristics of the processor, e.g. instruction level parallelism, special purpose registers, register allocation and predicated instructions. Gao et al. have proposed a micro-checkpoint based technique called *Cross Replay* [46] to overcome this problem in the context of hybrid simulations. The basic idea is to execute a given function on the AS and to use a lookup table to determine the timing depending on the executed control path. If there is no entry for a specific control path then the control path is replayed on a separate ISS to record the timing. Although the proposed approach yields good estimation results, it is not possible to apply this approach for the hybrid simulation integrated into a system simulation framework as described in Section 6.4. For the replaying a micro-checkpoint it is necessary to instantiate a separate ISS in order not to interfere with the rest of the simulation. Within the system simulation framework not only a separate instance of the ISS but also instances of all peripheral memories need to be created, which leads to a very high overhead.

Therefore, a more integration friendly performance estimation framework is applied based on periodically sampling the ISS execution as shown in Figure 6.11. Instead of executing all instances of a given function on the abstract simulator, some of the instances are executed on the ISS while the number of simulated cycles are recorded. The estimator uses the obtained information to produce an estimated value of the timing. Invoking the detailed simulation directly affects the simulation performance of the hybrid simulation, but at the same time it allows the estimator to adapt itself to varying conditions during runtime. Hence, a good trade-off between simulation performance and accuracy needs to be found. In the following the statistical properties of sampling are discussed to analyze the accuracy of the obtained estimations. Furthermore, different estimators are presented and their advantages and disadvantages are discussed.

[2] In the context of this thesis performance estimation is referring to timing estimation if not stated otherwise.

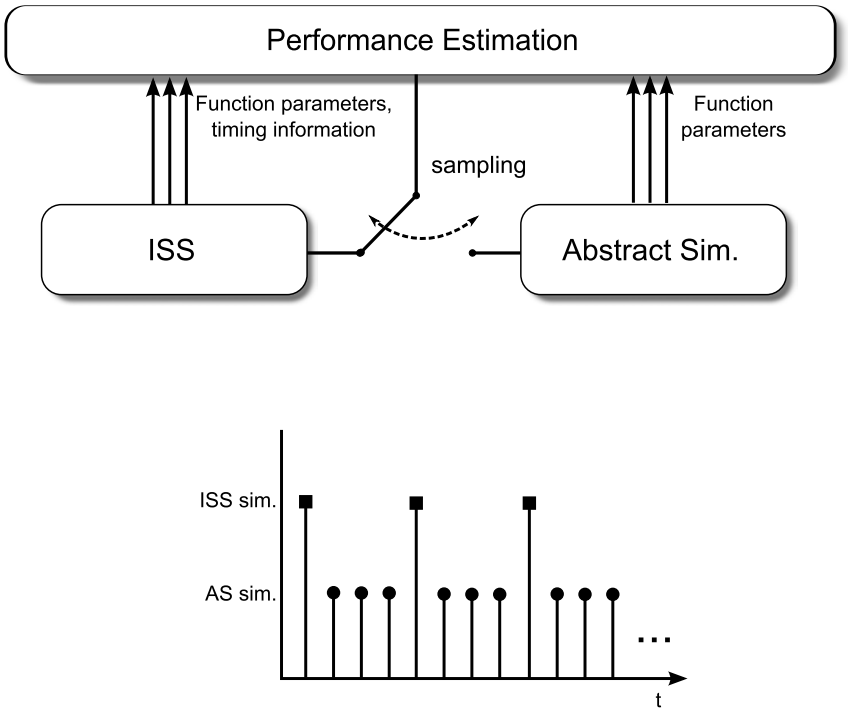


Figure 6.11: Performance estimation using a sampling based approach

6.5.1 Evaluation of Statistical Sampling

The field of statistical inference offers well defined procedures to quantify the quality of selected samples. The goal of statistical sampling is to estimate a given cumulative property of a population by measuring only a subset of the population [85]. The theory of sampling is concerned with choosing the minimal sample size that is sufficient to estimate a given property with the desired accuracy and precision. Furthermore, the theory does not make any assumption about the distribution of the population. This is important, because the distribution of the execution times of a given function will heavily depend on the application and on the selected input parameters. Based on statistical sampling it is possible to quantify the error margin and the confidence of the obtained estimate. This approach has been successfully applied in SMARTS [174] to estimate the average cycles per instruction (CPI). In the following the key equations are presented for calculation of the minimum sample size.

Table 6.5 gives an overview of the variables and terminology relevant to describe the sampling process. *Simple random sampling* selects a sample of n elements randomly out the population with N elements. For a sufficiently large n ($n > 30$, central limit theorem) this sample can be used to estimate the characteristics of the whole population. In particular the estimation of the mean μ of the population based on the mean of the samples $\hat{\mu}$ is of interest. The coefficient of variation v (Equation 6.17) represents the normalized standard deviation. The probability that the sample reflects the population well increase with the sample size n and decreases with v.

$$v = \frac{\sigma}{\mu}, \qquad \hat{v} = \frac{\hat{\sigma}}{\hat{\mu}} \tag{6.17}$$

Population variables	Sample variables
N size	n size
μ mean	$\hat{\mu}$ mean
σ standard deviation	$\hat{\sigma}$ standard deviation
v variation coeff.	$\hat{v}$ variation coeff.
	$(1-\alpha)$ confidence level
	$\pm\epsilon\mu$ confidence interval
	k systematic sampling interval

Table 6.5: Overview of the sampling variables

The confidence into the estimate of the mean $\hat{\mu}$ can be formally described by two independent terms: the *confidence level* $(1-\alpha)$ and the *confidence interval* $\pm\epsilon\mu$. The confidence level describes the fraction of measurements that produce $\hat{\mu}$ within the confidence interval $\pm\epsilon\mu$. Assuming $N \gg n \gg 1$ the confidence interval can be described by Equation 6.19. Equation 6.18 describes the percentile of the normal distribution.

$$z = 100\left(1 - \frac{\alpha}{2}\right) \tag{6.18}$$

$$\pm\,\epsilon\mu = \pm\left(z\frac{\sigma}{\sqrt{n}\mu}\right)\mu \tag{6.19}$$

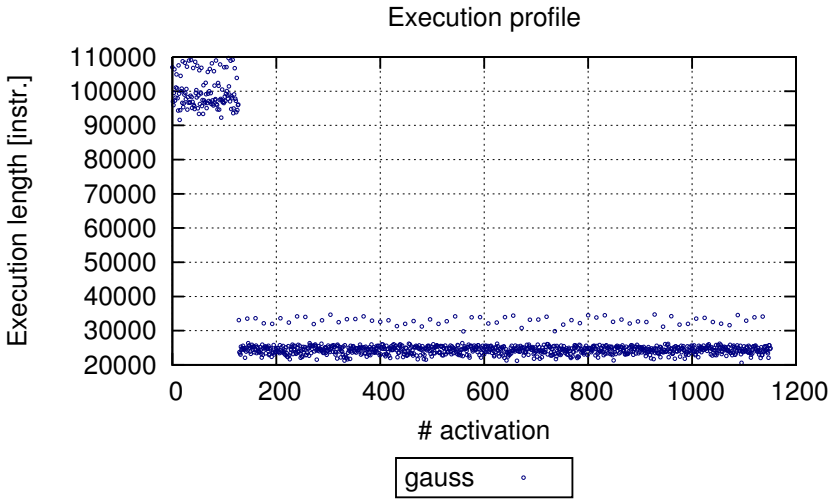
Given the standard deviation σ, the mean μ and the sample size n it is possible to calculate the confidence interval for a given confidence level or vice versa. For designing a sampling based estimation framework it is important to determine the size of the sample for a given confidence level and confidence interval. However, the true standard deviation and mean are rarely available in practice. Therefore n is approximated by using the estimated standard deviation $\hat{\sigma}$ and mean $\hat{\mu}$ as shown in Equation 6.20.

$$n \geq \left(\frac{z\hat{\sigma}}{\hat{\mu}\epsilon}\right)^2 \tag{6.20}$$

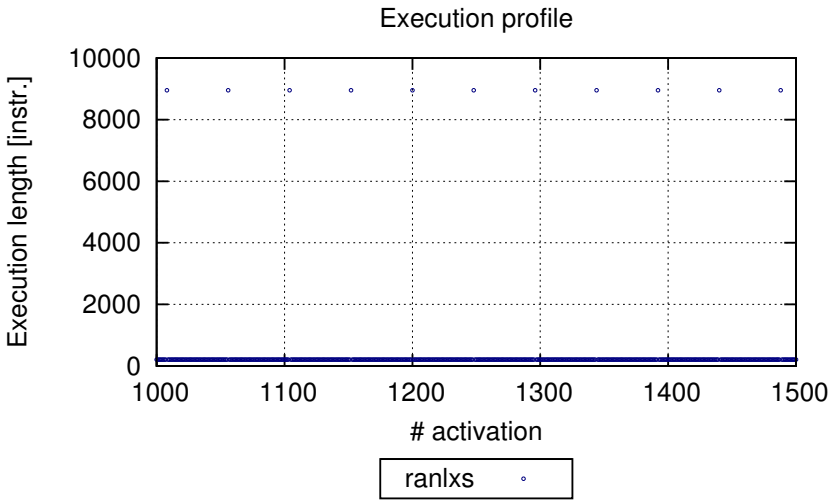
6.5.2 Evaluation of Sample Based Performance Estimation

In practice random sampling is often approximated by *systematic sampling*. In this approach every k-th element is selected such that the sample size is $n = N/k$. Systematic sampling works well if the measured property does not show periodic behavior. In particular, the property should not vary cyclically with the same periodicity as k or multiples of k.

Sampling based performance estimation is a very efficient method to derive estimates with high fidelity as described in the previous section. Inside SMARTS [174] and the COTSon [7] framework sampling has been successfully applied for estimating the



(a) Execution profile with moderate variation $v = 0.71$



(b) Execution profile with periodic variation $v = 6.55$

Figure 6.12: Example execution profiles

timing. However, for the HySim framework additional constraints need to be considered while applying sampling. In order to obtain estimates with good fidelity the size of the sample needs to be sufficiently large, depending on the variation coefficient.

Measurements have shown that the number of calls to a certain function are usually in the range of thousands. This limits the sample size that can be maintained without a large negative impact of the simulation performance. Two example execution profiles are shown in Figure 6.12. Figure 6.12a depicts an example trace with a moderate variation coefficient $v = 0.71$, whereas Figure 6.12b shows an execution trace with a very high variation coefficient of $v = 6.55$. The high variation coefficient and the periodic nature of the outliers shown in Figure 6.12b lead to an unacceptable estimation error. For execution traces that show a smaller variation coefficient, e.g. Figure 6.12a,

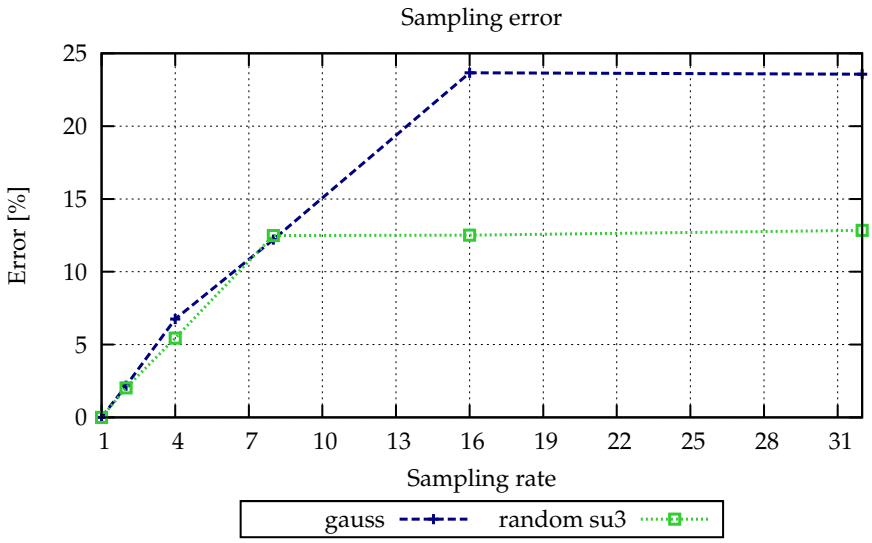


Figure 6.13: Sampling error depending on the sampling rate

the estimation error is sufficiently small. Figure 6.13 shows the estimation error for two example functions in dependence of the sampling rate k.

In summary, sampling based timing estimation is a powerful methodology for obtaining estimates with good fidelity. However, in case of the hybrid simulation framework the accuracy of the estimates is bounded by the limited number of switches, especially since a good partition would try to minimize the number of switches between both simulation modes. Furthermore, the measurements taken indicate a high periodicity of the function execution trace that can significantly dilute the estimation accuracy. One possibility to improve the estimation quality would be to incorporate information about the function parameters as suggested in [107].

6.6 Experimental Results

In this section, experimental results are presented. The virtual SHAPES platform is used as a test platform to evaluate the impact of hybrid simulation. The benchmarking is conducted on an Intel Quad Core Q6700 based system with 2.66 GHz and 16 GB of RAM using Scientific Linux 5.0 as operating system.

As benchmark a set of six different embedded applications was selected, covering different application domains: encryption (*des*), audio processing (*G721*), theoretical physics (*LQCD*), compression (*Linux decompress*), complex dynamics (*Mandelbrot set*) and signal processing (*hilbert*).

In a first experiment, the HySim framework is used to execute the application as fast as possible by partitioning the maximum of functions to the AS. Clearly, this is not the typical use case of this framework but it allows estimating the maximum achievable acceleration. The baseline for this measurement is the simulation of the SHAPES platform utilizing ISS-based processor simulation only. For comparison reasons the speedup reachable with dynamic binary translation (DBT) is also shown.

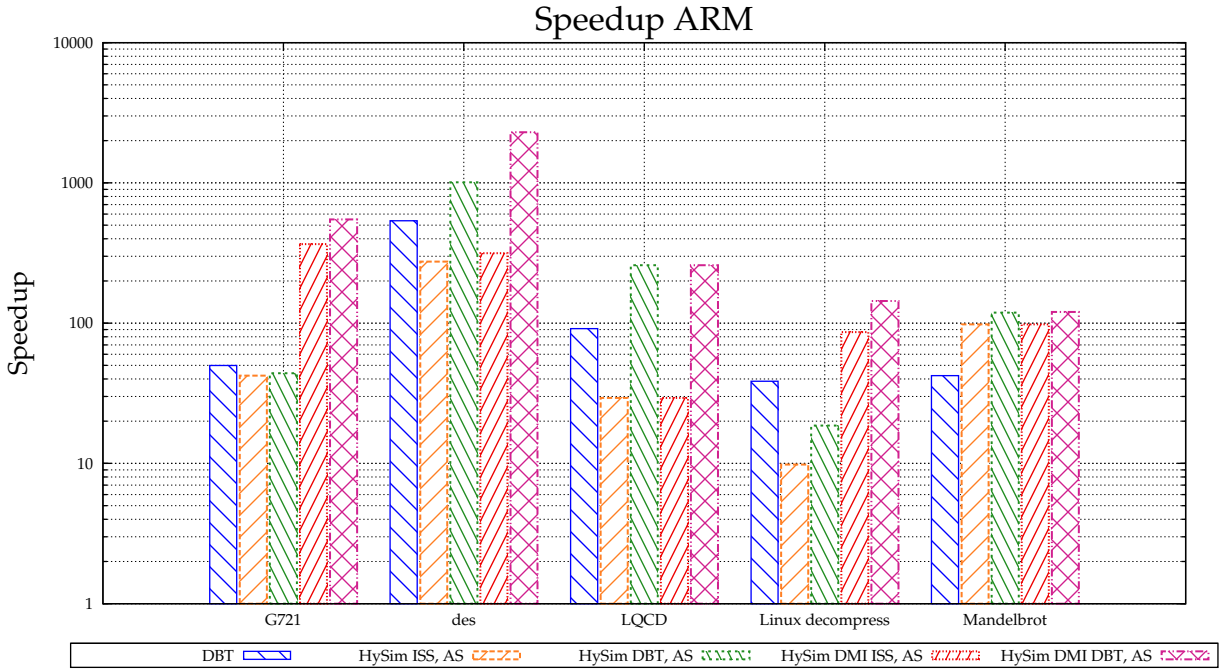


Figure 6.14: Speedup results for the ARM processor

Four different hybrid simulations are used in this experiment. The first configuration utilizes the HySim framework to switch between the AS and the ISS based simulation. The second configuration combines the binary translation with the hybrid simulation by switching between the AS and the dynamic binary translation (DBT) mode for the detailed simulation. Configurations three and four are identical to the configurations one and two respectively except that they make use of the DMI interface to access the global memory. Figure 6.14 shows the speedups achievable with the different configurations for the ARM processor and Figure 6.15 for the mAgic processor.

The obtained results indicate that hybrid simulation benefits in most cases from the combination with the binary translation. In all of these cases the hybrid simulation is bounded by the simulation speed of the ISS-based parts of the simulation (see Section 6.4.2). Thus, a higher speedup can only be reached if the slow simulation is accelerated. By applying binary translation to those parts the overall simulation speed can be increased. Due to the fact that the global memory accesses are very expensive ($\sim$500 host cycles) on the ARM processor, the utilization of DMI further increases the simulation speed. However, the gain of this modification greatly varies between the different applications. For example, the Linux decompress and the G721 application show a significant speedup using DMI, whereas the LQCD application shows no significant improvement. The main reason for this behavior is the memory ratio R_{mem} of the different applications. With a memory ratio $R_{\text{mem}} = 2\%$ the LQCD application is not bounded by the global memory accesses and hence does not benefit from the adoption of the DMI scheme.

Because of the good applicability of binary translation to simple RISC processors such as the ARM processor the hybrid simulation alone is not capable of outperform-

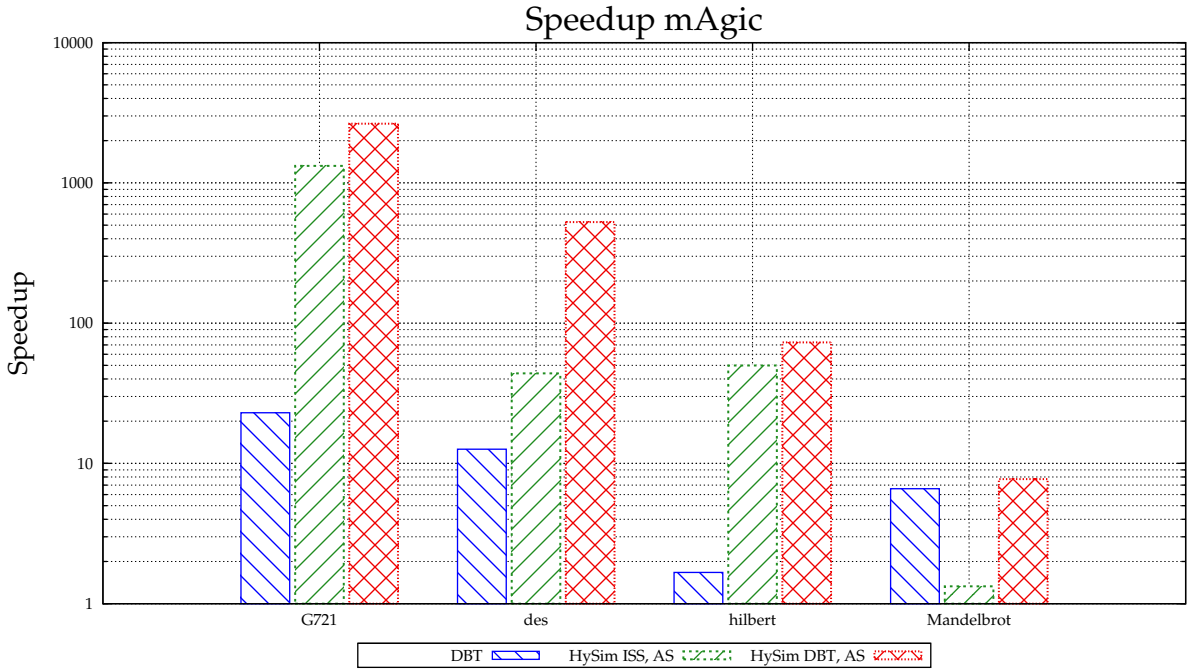


Figure 6.15: Speedup results for the mAgic processor

ing the binary translation on the ARM. In contrast to the ARM processor the mAgic processor benefits much less from the binary translation due to its complex microarchitecture. In case of the mAgic the hybrid simulation alone can outperform the binary translation. Combining hybrid simulation and binary translation leads only to a moderate speedup. Overall, the combination of binary translation and hybrid simulation achieves speedups ranging from 1.2 to 114 compared to binary translation alone.

A typical use case for the hybrid simulation is to reach a certain state of an application as fast as possible in order to debug this state using detailed simulation. For example, using hybrid simulation the decompress of the Linux kernel takes only 1 second instead of 216 seconds for ISS based simulation. The LQCD application requires a lengthy data preprocessing before the actual computation starts. Using the hybrid simulation it is possible to reduce the time from 17 seconds to 6 seconds.

6.7 Conclusions

Different hybrid simulation approaches have been developed for simulation, e.g. Tensilica enables the user to switch any time between CA and IA simulation and PTLSim offers the possibility to switch between CA simulation and native code execution for x86 ISAs. This chapter presents a hybrid simulation framework that leverages the hybrid simulation concept, so that it is possible to switch between IA simulation and native code execution independent of the ISA of the target processor. Unlike other simulation frameworks this framework has been specifically tailored towards the needs of software developers by providing a fast simulation mode and enabling the user to con-

trol the switching between both simulation modes. Furthermore, no time consuming pre- and post-processing is required for this approach.

With the HySim framework it is possible to reach the desired state of an application quickly and then switch to a slow but detailed simulation. The partitioner takes care of moving as many functions as possible to the AS for fastest possible execution. Target dependent code like assembly will be executed on the ISS to ensure correct behavior. The fact that the ISS is not aware of the AS and does not need to be modified, makes the HySim framework a versatile instrument which can be employed for a wide range of ISSs. Moreover, the hybrid simulation has been integrated into a SystemC based simulation framework commercially available from CoWare to analyze the integration effort and the impact on the system simulation.

The obtained results show that the concept of hybrid simulation is capable of achieving high simulation speedups, especially if it is combined with binary translation for the IA simulation. In case of complex VLIW architectures a high effort is required for to create suitable binary translation enabled simulation model. For such complex processor architectures the hybrid simulation approach offers software developers high simulation speeds with much less implementation effort.

Chapter 7

Timing Approximate NoC Simulation

Simulating MPSoC systems with a large number of processing elements requires not only efficient simulation techniques for the processing elements (cf. Chapter 6), but also a scalable and efficient network-on-chip (NoC) simulation environment. Simply ignoring the effects of the NoC during early stages of the design space exploration might be a solution for small systems running only a single application. However, for multi-threaded applications and multi-application scenarios neglecting the timing delays introduced by the NoC might completely change the relative task execution order and thus is not an option, even during early design stages. Hence, it is important to consider the NoC effects throughout the different design stages of the software development.

This chapter introduces a hybrid simulation approach for the NoC of the simulated system. Conceptually this approach is comparable with the hybrid simulation of the processing elements discussed in Chapter 6. However, in contrast to the simulation of processing elements the impact of the NoC simulation on the simulation speed stems from the high number of generated events and not from the computational complexity of the interconnect simulation in the *discrete event simulation* (DES). Therefore, a different approach has been selected to cope with this characteristic.

The rest of this chapter is organized as follows. In Section 7.1 the general approach for the hybrid NoC simulation is presented. Section 7.2 gives a brief overview of related work. In Section 7.3 the assumptions and key characteristics of the analytical NoC model, which is going to drive the abstract NoC simulation, are discussed and the applied model is presented. In the following the simulation environment including the simulated hardware and software components are introduced in Section 7.4. The results based on this simulation environment are presented and evaluated in Section 7.5. Finally, this chapter is summarized and a short outlook is given in Section 7.6.

7.1 General Approach

The characterization of communication networks is a well established research area and is of particular interest for the domains of telecommunication, traffic engineering and computer networks. Simulators are commonly used to assess the key characteristics, throughput and latency, of communication networks with high fidelity. However, this approach is very time consuming and does not scale well. For this reason analytical NoC models have been proposed to complement the simulation based approaches. These models offer fast and reasonably accurate estimations of the NoC characteristics. Moreover, these approaches scale very well and can be applied to very large communication networks.

The basic method of delay estimation is to determine the hop-count between the source and the destination node, while neglecting the latency introduced by contention at the input buffers of the routers. However, this contention induced latency can have a significant impact on the overall network performance, especially for medium and high traffic loads. The goal of the analytical models is to predict this contention delay. In general, the analytical models can be divided into two groups [111]: *probabilistic models* [88] and models based on *queuing theory* [38, 81]. In the context of the proposed hybrid NoC simulation an analytical model based on queuing theory is used.

Queuing theory allows modeling the stationary state of communication systems based on a set of key parameters, e.g. arrival rates λ and service times T. The parameters are modeled as *random variables* (RVs) and approximated by appropriate probability density functions (PDFs). These PDFs can be obtained by measurement of a real system or by making educated guesses on the properties of the PDF.

In queuing theory a router – the central component of each communication network – is modeled as a single server and the input buffers are treated as queues. This type of analytical modeling has proven to yield very good results for the stationary state of a queuing system. However, queuing theory is not capable of capturing the transition states in highly time variant systems. In contrast to analytical model driven approaches, simulation based approaches are able to capture also the transition states of a communication system at the price of a higher complexity and a lower simulation speed.

In the domain of VP based system development, the simulation based NoC approaches are currently dominating because of various reasons. First, VPs have evolved bottom up, from low level RTL based simulations which simulates the complete HW. Hence, it is natural to simulate also the NoC part. Second, up to now the complexity of on-chip communication was pretty much limited to a set of buses.

However, with the emergence of complex MPSoC systems as well as highly scalable system, e.g. the SHAPES platform (see Chapter 4), the complexity of the interconnect becomes an important factor for determining the overall simulation speed. Even though the computational overhead related to the simulation of a NoC transaction is small, the impact on the overall simulation performance is quite significant due to the high number of simulation events being generated. For each event the SystemC kernel needs to be involved which incurs a high overhead for context switching.

In summary, analytical NoC models offer a good alternative to quickly characterize complex NoCs without the need of lengthy simulation runs. However, in contrast to the simulation based approaches the accuracy of this approach is limited. The key idea of the hybrid NoC simulation approach is to combine the advantages of analytical NoC models with the advantages of simulation based approaches. Figure 7.1 presents the general concept of the proposed hybrid NoC simulation approach. The network simulator consists of two different NoC models, a detailed NoC model based on simulation and an analytical model of the NoC. The detailed simulation model is used to sample the parameters necessary to tune the analytical model. Once those parameters have been determined the hybrid NoC simulator switches to the analytical model. In this mode, an ideal functional router is used to provide direct point-to-point connections between the source and the destination nodes bypassing all intermediate communica-

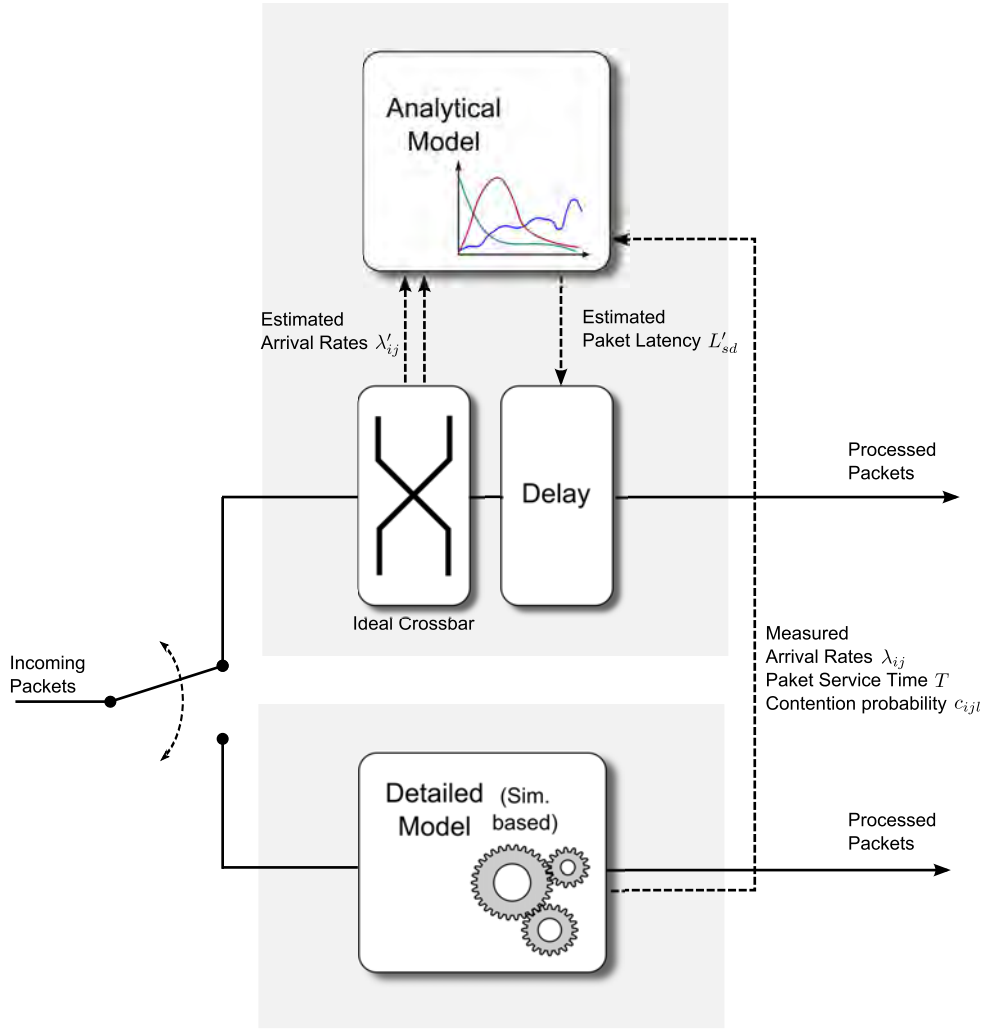


Figure 7.1: General concept of the hybrid NoC simulation

tion nodes. In order to maintain the timing of the overall system, a delay is introduced after each source-destination transaction, mimicking the effects of the real NoC. This delay information is provided by the analytical NoC model. By switching back to the detailed NoC simulation it is possible to update the parameters of the analytical model and hence ensure a good quality of the NoC behavior estimation.

7.2 Related Work

With the increasing complexity of MPSoCs the need for scalable on-chip interconnections is growing. The classical bus-based interconnects only work efficiently for a very limited number of masters. For larger systems the interconnect quickly becomes the bottleneck and severely limits the system performance. Benini et al. [14] therefore discuss the different possibilities of building scalable interconnects. The advantages and disadvantages of the different approaches as well as the challenges are highlighted. A

comprehensive overview of the outstanding research problems and challenges of NoC design can be found in [91].

Using NoCs adds another layer of complexity on top of the MPSoC development. The efficient utilization of the NoC is a cross layer challenge. Only if the hardware and the corresponding software stack are tuned together the full potential of the interconnect can be exploited. In this context the NoC simulation plays an important role. Since it offers the developers to perform design space exploration and evaluation of different NoCs. In the following the most important use cases for NoC simulation are briefly presented.

Due to the large set of parameters provided by NoCs finding the correct configuration of the hardware and of the protocol stack is non-trivial. In [175] an online configuration simulation framework for large scale networks is proposed using an recursive random search algorithm to determine a good configuration of the different parameters. In [114] an queuing theory based analytical NoC model is used to predict potential bottlenecks of the interconnects, e.g. the starvation of different links. This information is then used by the prediction based flow control in order to improve the average packet latency of the NoC.

In order to better understand potential performance issues of the on-chip interconnect, traffic modeling is an important step. It allows to use traffic generators to quickly test a large number of different communication scenarios. Furthermore, the usage of traffic generators can help to reduce the simulation complexity and hence to increase the simulation performance. In [146] an empirically derived statistical traffic model is presented. It uses three parameters – the burstiness, the hop count and the packet injection distribution – to describe the temporal and spatial variations of the NoC traffic. This model is based on the self-similarity of the NoC traffic. In [137] multi-phase traffic generation is presented based on automatic phase detection. This approach is based on the principles developed in the context of sampling simulation and aims to identify the different phases of the NoC communication. The obtained information can be used as input for a stochastic traffic generator.

Besides the characterization of NoC traffic, queuing theory based models serve also as low overhead predictors for modeling contention of shared resources in MPSoCs as described in [134]. Based on the characteristics of the application memory access pattern synthetic traces are generated which are then used for estimating the contention of the shared resources. In order to keep the overhead of the predictor low an interval analysis is applied. The resource contention is back-annotated to the different sources by using the *spreading factor*. This factor calculates the fraction of the delay effecting the different input queues due to contention while accessing the shared resource. For the Spidergon NoC, which as been specifically developed for MPSoCs, an analytical performance model has been developed in [99] to reduce the need of lengthy simulations during the design space exploration.

7.3 Analytical NoC Model

In this section the analytical router model applied in the hybrid NoC simulation is discussed. This analytical router model forms the basic building block for analytically

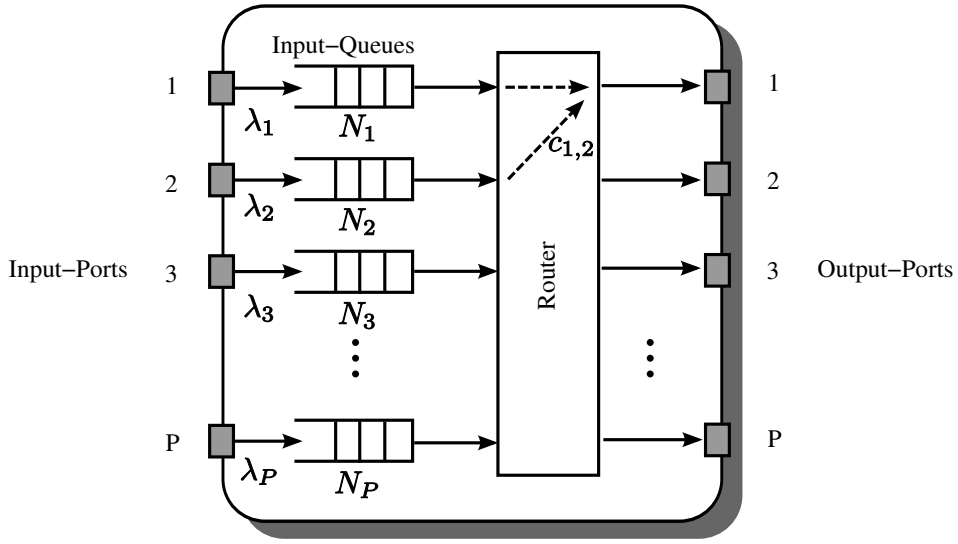


Figure 7.2: Analytic view of a router

modeling the entire NoC. This NoC model is based on the work of Ogras et al. [113], which has been developed for performance analysis of wormhole routing based NoCs with arbitrary packet size, finite buffers and deterministic routing. Furthermore, this model does not make any assumptions on the network topology, which makes it a versatile analytical model for NoCs. It extends the M/G/1 router representation [76] and assumes a Poisson process at the input of every router. The computation of the *blocking probability* due to the finite buffer size and the average queue length is based on the work of Takagi [154]. The core model for computing the number of packets waiting at a certain input port of a certain router – which directly delivers also the waiting time – is in essence the queuing model from Takagi. However, this model only considers a single router with a finite buffer size. The effects induced by the router being part of a larger network and the contention effects are nicely modeled in the contention probability parameter presented in the work from Ogras et al. [113]. These concurency effects due to the presence of multiple routers have been captured by the proposed *effective service time* and introduced in the earlier mentioned analytical model from Takagi. A detailed description of the analytical model and the effective service time can be found in Section 7.3.2.

First, the basic assumptions made by the analytical model and the important parameters are presented, followed by a detailed description of the key aspects of the analytical router model.

7.3.1 Assumptions and Parameter Description

In the following the assumptions and parameters used for the analytical router model are presented. A router with buffered inputs and P input-ports and output-ports is considered, as shown in Figure 7.2. Furthermore, the analytical model assumes wormhole control flow using a deterministic routing algorithm.

In general the packet size **S** (bytes) is application dependent and can vary over time. Hence, the packet size **S** is modeled as a random variable. H_s denotes the service time for the header flit and BW denotes the bandwidth (bit/s) of a single link between two routers. The packet service time **T** accounts for the traversal of all flits belonging to one packet through the observed router excluding the the queuing delay. The service time **T** can be calculated as follows [113]:

$$\mathbf{T} = H_s + \left\lceil \frac{\mathbf{S}}{BW} \right\rceil \tag{7.1}$$

Consequently, also the packet service time **T**, being dependent on the packet size **S**, is modeled as a random variable. The traffic rate x_{sd} denotes the packet rate (packets/s) being transmitted from a source node at router s to destination node at router d. The arrival rate of the header flits at router j and input port i is given by λ_{ij} (packets/s). Table 7.1 summarizes the all parameters and their dependencies used in the analytical model.

The model assumes a Poisson distribution of the arrival process of the header flits for each router. This is a common assumption in many analytical router models and is known to work well for service times that are correlated with the packet size. However, Nikitin et al. [111] have shown that for deterministic service times (M/D/1) this assumption leads to an overly pessimistic prediction of waiting times.

7.3.2 Analytical Router Model

This section focuses on the analytical model of a single router i with P buffered inputs. In order to simplify the model a fixed packet size, and hence a fixed service time, are assumed. The load ρ of the router can be computed by Equation 7.2.

$$\rho = \lambda \cdot T \tag{7.2}$$

The analytical router model is based on the M/G/1/K model from the queuing theory [154]. This model assumes a finite buffer size, maximum K packets can be stored inside the system. The maximum number of packets K in the system is composed of the packet currently being processed plus the B packets stored in the input queue.

$$K = B + 1 \tag{7.3}$$

π_k denotes the probability that k packets are in the router, i.e. in the queue and currently being processed. The steady state equation for the state transition is given by Equation 7.4. The p_{jk} denotes the probability that number of packets inside the system changes from j to k during one service time T.

$$\pi_k = \sum_{j=0}^{K-1} \pi_j p_{jk},\ K \geq 1 \tag{7.4}$$

The Equation 7.5 computes the probability a_k that k packets arrive during the service time T for a given average arrival rate λ assuming a Poisson distribution of the arrival process.

Symbol	Description	Depends on
$\mathbf{S}$	RV describing the packet size	Application
x_{sd}	Packet transmission rate from source node s to destination node d	
H_s	Service time for the header flit	Router
BW	Network channel bandwidth	
B_{ij}	Size of the input buffer at router i, channel j	
P	Number of input and output queues on a router, including the one from the local tile	
$\mathbf{T}$	RV describing the packet service time.	Application, router
f_{ijl}	Forwarding probability from input-port j to output-port l in router i	
c_{ijl}	Contention probability between channel j and l in router i	
λ_{ij}	Arrival rate at router i, channel j	Topology, application, routing
$\hat{\lambda}_{ijl}$	Arrival rate for packets flowing from input-port j to output-port l of router i	
N_{ij}	Average number of packets in input buffer j of router i	
W_{ij}	Average waiting time in input queue j of router i	
L_{sd}	Average packet latency on the path from router s to router d	

Table 7.1: List of parameters, symbols and their dependencies

$$a_k = e^{-\lambda T}\frac{(\lambda T)^k}{k!},\ k \geq 0 \tag{7.5}$$

Instead of directly computing the probabilities π_k using Equation 7.4 a recursive approach [154] is selected to compute π'_k. Equation 7.6 shows the relation between π_k and π'_k.

$$\pi_k = \pi_0 \cdot \pi'_k \tag{7.6}$$

Equation 7.7 and Equation 7.8 show how to compute π'_k recursively using the probabilities a_k.

$$\pi'_0 = 1 \tag{7.7}$$

$$\pi'_{k+1} = \frac{1}{a_0}\left(\pi'_k - \sum_{j=1}^{k} \pi'_j a_{k-j+1} - a_k\right),\ k \geq 0 \tag{7.8}$$

π_0 can be directly computed from π'_k as shown in Equation 7.9.

$$\pi_0 = \left(\sum_{k=0}^{K-1} \pi'_k\right)^{-1},\ K \geq 1 \tag{7.9}$$

Having computed π_k and π_0 it is possible to calculate the probability P_k that k packets are present in the system at an arbitrary time, as shown in Equation 7.10. In contrast to the probability π_k (Equation 7.4) the probability P_k indicate the probability that k packets are in the system while taking the finite buffer size of the system into account.

$$P_k = \frac{\pi_k}{\pi_0 + \rho},\ k = 0 \ldots K \tag{7.10}$$

For evaluating the NoC performance the average number of packets per input buffer of a router plays a central role. The average number of packets N_{ij} inside the input queue j of router i can be computed by Equation 7.11.

$$N_{ij} = \sum_{k=1}^{K} k P_{ijk} = \frac{\sum_{k=1}^{K-1} k\pi_k}{\pi_0 + \rho} + K\left(1 - \frac{1}{\pi_0 + \rho}\right) \tag{7.11}$$

Equation 7.11 computes the average number of packets in a system with finite input buffers. However, this model needs to be extended in order to take the multiple queues of a router into account. Especially, the contention caused by multiple data packets being routed to the same output port needs to be considered. For this the *forwarding probability* f_{ijl} and the *contention probability* c_{ijl} proposed by Ogras et al. [113] are used to extend the analytical router model.

The forwarding probability f_{ijl} describes for router i the probability that a packet coming from the input-port j is going to be forwarded to the output-port l. The forwarding probabilities of a router i can be expressed as as shown in Equation 7.12 assuming a *round-robin* scheduling in case of contention. For the sake of brevity, the arrival rates $\hat{\lambda}_{ijl}$ (cf. Table 7.1) are used to calculate the forwarding probabilities. With the knowledge about the routing algorithm it is easily possible calculate the arrival rates $\hat{\lambda}_{ijl}$ from the arrival rates λ_{ij}.

$$f_{ijl} = \frac{\hat{\lambda}_{ijl}}{\sum_{k=1}^{P} \hat{\lambda}_{ikl}},\ 1 \leq j, l \leq P \tag{7.12}$$

Based on the forwarding probability it is possible to compute the contention probability c_{ijl} between channel j and channel l of router i. In Equation 7.13 $c_{ijj} = 1$ accounts for the fact that a packet has in any case to wait for all the packets in its own input queue.

$$c_{ijl} = \sum_{k=1}^{P} f_{ijk} f_{ilk},\ j \neq l,\ 1 \leq j, l \leq P,\ c_{ijj} = 1, \tag{7.13}$$

In order to take the effects due to resource contention into account the *effective service time* T_{eff} is introduced in Equation 7.14. For each router input-port the effective service time T_{eff} is calculated based on the contention probability c_{ijl} of the different router channels. Although it is assumed that the service time T of the router is constant the effective service time T_{eff} per input channel will vary over time dependent on the contention in the router. The concept of the effective service time allows to seamlessly combine the router models of Takagi [154] and Ogras et al. [113] into an analytical router model suitable for the hybrid NoC simulation.

Based on the effective service time it is possible to compute the effective probabilities $a_{\text{eff},k}$ that k packets are in the system (Equation 7.15).

$$T_{\text{eff},ij} = T_{ij}\left(1 + \sum_{k=1}^{P} c_{ijk}\right),\ j \neq k \tag{7.14}$$

$$a_{\text{eff},k} = e^{-\lambda T_{\text{eff}}} \frac{(\lambda T_{\text{eff}})^k}{k!} \tag{7.15}$$

Hence, it is possible to reformulate Equation 7.11 taking the effective service time into account as shown in Equation 7.16

$$N_{\text{eff},ij} = \sum_{k=1}^{K} k P_{\text{eff},ijk} = \frac{\sum_{k=1}^{K-1} k\pi_{\text{eff},k}}{\pi_{\text{eff},0} + \rho} + K\left(1 - \frac{1}{\pi_{\text{eff},0} + \rho}\right) \tag{7.16}$$

All information required to calculate the average packet queue length $N_{\text{eff},ij}$ can be derived from the arrival rates λ_{ij}. Together with the knowledge about the routing algorithm it is possible to compute the arrival rates for each router directly from the traffic rates x_{sd} as shown in Equation 7.17.

$$\lambda_{ij} = \sum_{\forall s} \sum_{\forall d} x_{sd} \mathcal{R}(s, d, i, j) \tag{7.17}$$

The function $\mathcal{R}$ returns a 1 in case a packet from router s to router d is sent via router i and input channel j.

$$\mathcal{R}(s, d, i, j) = \begin{cases} 1 & : \text{ is routed via router } i \text{ and channel } j \\ 0 & : \text{ is not routed via router } i \text{ and channel } j \end{cases} \tag{7.18}$$

In order to use this analytical model inside the hybrid NoC simulation approach, the NoC simulation model only needs to provide the traffic rate x_{sd}, the service time T respectively the packet size and information about the routing algorithm. All this information can be easily extracted from the detailed as well as from the fast NoC simulation.

7.3.3 Analytical NoC Model

With the average number of packets per queue and the arrival rates for every router, the average waiting time W_{ij} for each router i and input queue j can be computed by using Little's theorem [38] as shown in Equation 7.19.

$$W_{ij} = N_{\text{eff},ij} / \lambda_{ij} \tag{7.19}$$

Based on the average waiting times per router and input channel it is possible to compute the average latency L_{sd} for each source-destination pair (Equation 7.20) by summing all waiting times of the traversed routers $\Pi_{s,d}$. Whereas W_s denotes the waiting time at the source node and $\Pi_{s,d}$ denotes the set of routers traversed by the packet.

$$L_{sd} = W_s + \sum_{(i,j)\in\Pi_{s,d}} W_{ij} \tag{7.20}$$

The average packet latencies for a source-destination pair obtained by Equation 7.20 are used to control the delay added to each packet transaction while running the abstract NoC simulation (see Figure 7.1).

During the fast simulation mode the latency of a complete message passing interface (MPI) data transfer, i.e. a sequence of packets, is approximated by Equation 7.21, where N_{pkt} denotes the number of packets and N_{hops} denotes the number of hops associated with this transaction. The second term of this equation calculates the average latency per hop assuming that the packets of a transaction are equally distributes over the different hops. This heuristic approximates the time required to transfer a complete packet sequence assuming a pipelined behavior of the MPI transaction.

$$L_{\text{transaction},sd} = L_{sd} + (N_{pkt} - 1)\,\frac{L_{sd}}{N_{\text{hops}}} \tag{7.21}$$

7.4 Simulation Environment

In this section the simulation environment that served as a basis for the development and evaluation of the hybrid NoC simulation is presented. This environment makes use of a tiled architecture approach and is closely related to the SHAPES platform (see Chapter 4). However, in contrast to the SHAPES platform this platform is based on the OSCI SystemC kernel [116] and does not require any third party IP. Having full control over the source code makes this simulation environment a perfect platform for testing new simulation techniques.

In order to assess the results of the hybrid NoC simulation presented in Section 7.5, it is important to understand first the key characteristics of the simulated HW presented in Section 7.4.1 as well as of the SW stack running on top of it (Section 7.4.2).

7.4.1 Simulated Hardware

The basic structure of the HW architecture is closely related to the SHAPES architecture. Both systems employ the tiled architecture concept in order to obtain an easily scalable HW platform for massively parallel applications.

A configurable number of processing elements (PEs) are connected in a 2D mesh network using a packet switched fabric. The different PEs are interconnected by a network processor module. This module provides 4 external links for the communication with the neighboring PEs and one local link for the communication with the PE. As routing algorithm deterministic XY routing [100] is applied. An example configuration of the HW with 6×6 PEs is shown in Figure 7.3. Depending on the intended

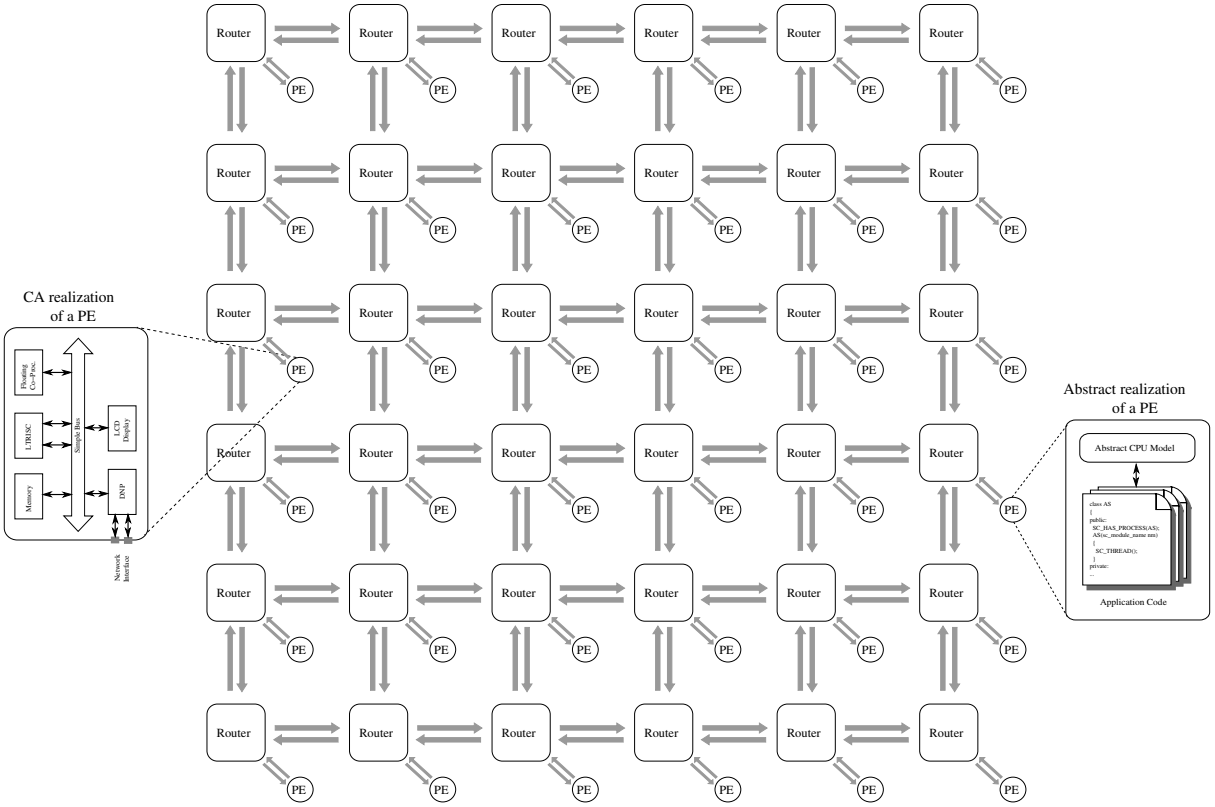


Figure 7.3: Example configuration, 2D-mesh with 6×6 routers

use case of the simulation system different types of PE simulations can be used. For instance, for a detailed analysis of the system timing behavior, the PEs are modeled as a complete subsystem including a CA processor model, a bus and memories. For the CA processor models, LISA [58, 59] generated processors are used which are the only components that are not available in source code.

In case that the NoC and the routers are the main focus of the analysis, abstract processor simulators can be used as PEs. In favor of high simulation speed the abstract processor simulator executes the application directly on the host neglecting all HW details. In order to roughly model the timing of the application it is possible to annotate the timing by means of SystemC `wait()` statements to the application code. This kind of abstract processor simulators is well suited for stimulating the NoC with realistic traffic scenarios while at the same time its impact on the simulation speed is low. Hence, this configuration is well suited for evaluating the characteristic and scalability of the NoC subsystem using a high number of nodes. It is even possible to combine different types of PE simulators inside one simulation system, i.e. a few nodes are simulated in detail whereas the remainder of the nodes use the abstract processor simulator to generate the NoC stimulus.

In Figure 7.4 the realization of the network processor is shown. It is composed out of two main components: the *bridge module* and the *router module*. The bridge module is responsible of communicating with the local tile by moving data from the local tile to the router and back. Furthermore, all control commands (e.g. send request, data receive) sent by the local tile to the network processor are interpreted inside the bridge

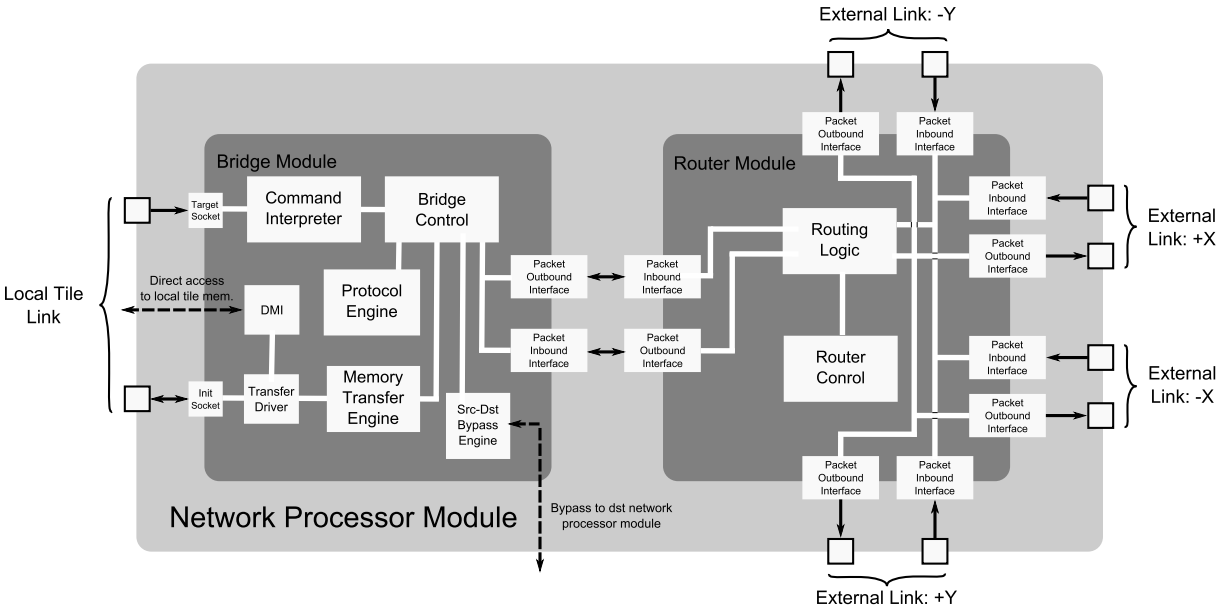


Figure 7.4: Realization of the network processor module

and the appropriate control signals are then forwarded to the router. The router module is responsible of forwarding the incoming packets to their destination according to the selected routing algorithm. Data packets for the local tile are forwarded to the bridge module which copies the data to the appropriate memory location of the tile. Due to the modular design, different components of the network processor can easily be extended or replaced, e.g. the routing algorithm, without affecting other components.

During the analytical simulation of the NoC the router module is deactivated, instead the source bridge module will transfer the data directly to the destination module using the bypass engine. The destination bridge module will annotate the delay of the transaction according the data estimated by the analytical NoC model. Only bypassing the the NoC results in very a limited simulation speedup since the data transfer from and to the local tile is becoming the bottle neck. In order to circumvent this problem a *direct memory interface* (DMI) has been integrated into the bridge module. This enables the bridge to directly copy the data to the desired memory location bypassing the complete communication hierarchy of the PE.

7.4.2 Software Stack

In order to exploit the parallelism offered by such a scalable HW platform an appropriate programming model and the corresponding SW stack are required. The well established MPI based programming model [104, 115] has been selected to keep the porting efforts of the application developers low. This programming model assumes a separate address space for each process. The inter-process communication is explicit and needs to be added by the application developer.

Using the MPI programming model has the advantage that the application developer can run the application directly on the host during the first development steps.

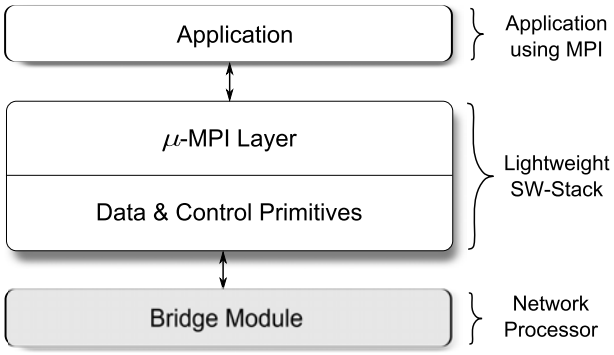


Figure 7.5: SW Stack running on top of the HW

```
 1 int MPI_Init(int *argc, char ***argv);
 2 int MPI_Finalize();
 3 int MPI_Abort(MPI_Comm comm, int errorcode);
 4 int MPI_Get_processor_name(char *name, int *resultlen);
 5 int MPI_Comm_rank(MPI_Comm comm, int *rank);
 6 int MPI_Comm_size(MPI_Comm comm, int *size);
 7
 8 // Blocking communication
 9 int MPI_Ssend(void *buf, int count, MPI_Datatype datatype,
10               int dest, int tag, MPI_Comm comm);
11 int MPI_Recv(void *buf, int count, MPI_Datatype, int source,
12               int tag, MPI_Comm comm, MPI_Status *status);
13
14 // Non-blocking communication
15 int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
16               int dest, int tag, MPI_Comm comm,
17               MPI_Request *request);
18 int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
19               int source,int tag, MPI_Comm comm,
20               MPI_Request *request);
```

Listing 7.1: Functions supported by the μ-MPI layer

Later, the application can be easily ported to the simulated platform. The translation of the MPI primitives into control and data primitives for the network processor is performed by the μ-MPI layer as shown in Figure 7.5.

Currently, the μ-MPI layer supports blocking and non-blocking MPI data transfers. The non-blocking communication is restricted to at maximum four parallel transactions from four different senders per receiver node. Each sender node supports only one transaction at a time. These limitations have been introduced for the sake of simplicity of the μ-MPI layer and the reduced memory consumption of the system. Listing 7.1 shows the interfaces of all functions supported by the μ-MPI layer.

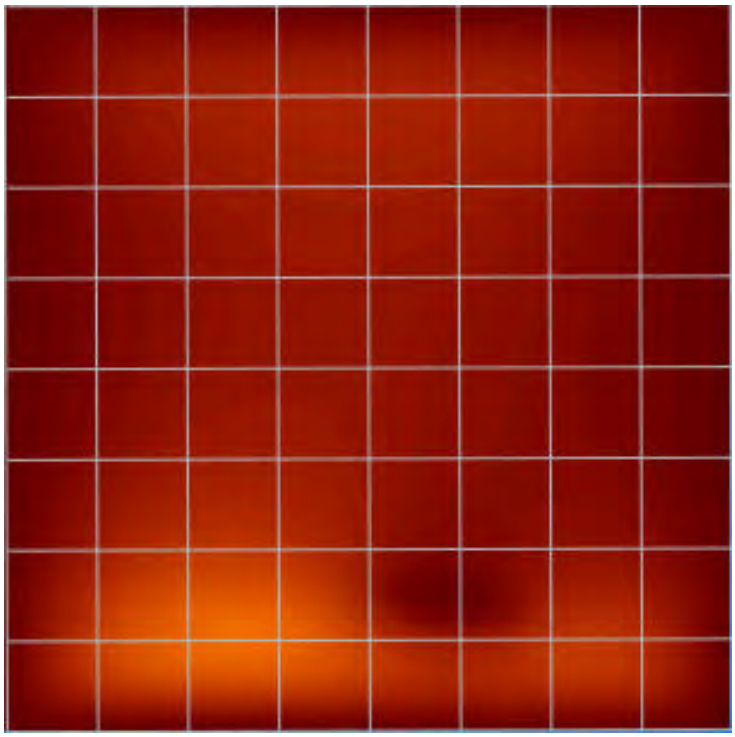


Figure 7.6: Screenshot of an 8×8 FDM heat simulation

7.4.3 Application

In principle, all kinds of applications using blocking MPI data transfers can be ported to the simulation platform. In the context of this thesis, a 2D heat spread simulation has been selected as main driver application. This application numerically solves the partial differential equation for computing the heat spread on a thin 2D plate. The computation is realized by applying the *finite difference method* (FDM) for solving the partial differential equation. This application has been selected because it can be easily parallelized and shows good scalability with respect to the number of computation nodes. The parallelized version of the FDM heat computation is based on [60] and uses MPI primitives for inter-tile communication.

Figure 7.6 shows a screenshot of the heat spread simulation using 8×8 PEs. Each PE computes the heat spread for a single tile. After each iteration the tile will communicate with the neighboring tiles to update the boundary conditions for the next iteration. The master PE controls the simulation process and polls the results from each worker node after a configurable time interval. The individual results are assembled into one result picture and are displayed via a simulated display device.

During the start up phase and the result polling phase the inter-tile communication is purely sequential from/to the master PE to/from the worker PEs. All worker PEs are blocked until the communication with the master PE is finished. During the computation phase all worker PEs are communicating with their neighbors in parallel.

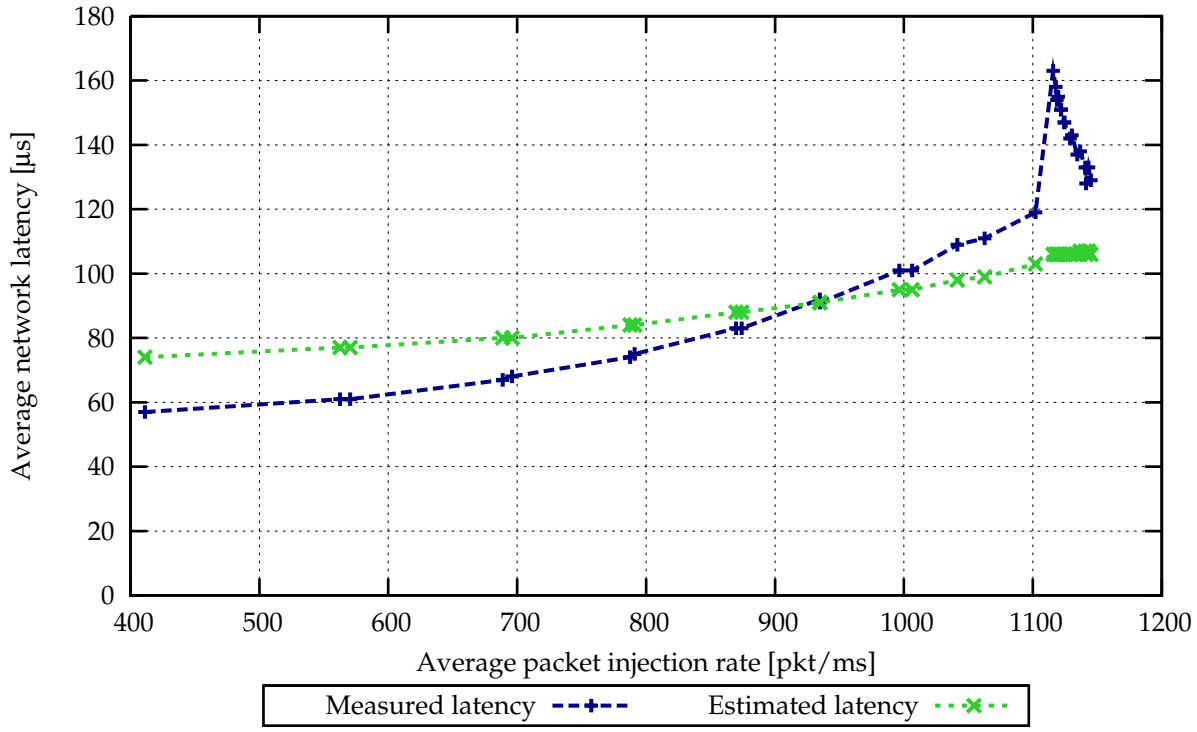


Figure 7.7: Average network latency $\bar{L}$ for an 8×8 system with two data transfers in parallel per receiving node

7.5 Experimental Results & Evaluation

In this section, experimental results obtained the above mentioned simulation environment and the developed analytical NoC model are presented. All experiments have been conducted on an Intel Quad Core Q6700 based system with 2.66 GHz and 16 GB of RAM using Scientific Linux 5.0 as operating system. The NoC system is using a fixed packet size of 1 kB. Each packets contains 64 bytes of control data and 960 bytes of payload. A synthetic application that instantiates $N/2$ communication pairs in a 2D mesh with N nodes serves as test application. The goal of this application is to generate a configurable amount of congestion in the NoC in order to evaluate the analytical NoC model. For flexibility reasons the communications pairs are loaded from a configuration file during the application initialization. This file is generated by a random number generator. At run-time the communication pairs will exchange a configurable amount of data using the μ-MPI layer of the simulation environment. Depending on the number of nodes and the generated communications pairs the congestion of the different links in the NoC will vary. Furthermore, the number of data packets being sent over the network can be controlled by varying the amount of data being transferred for a source-destination pair. In order to further increase the pressure in the NoC, it is possible to perform up to four data transfers in parallel to a single receiver node by using the non-blocking data transfers offered by the μ-MPI layer. Hence, the probability of creating contention in different routers increases significantly.

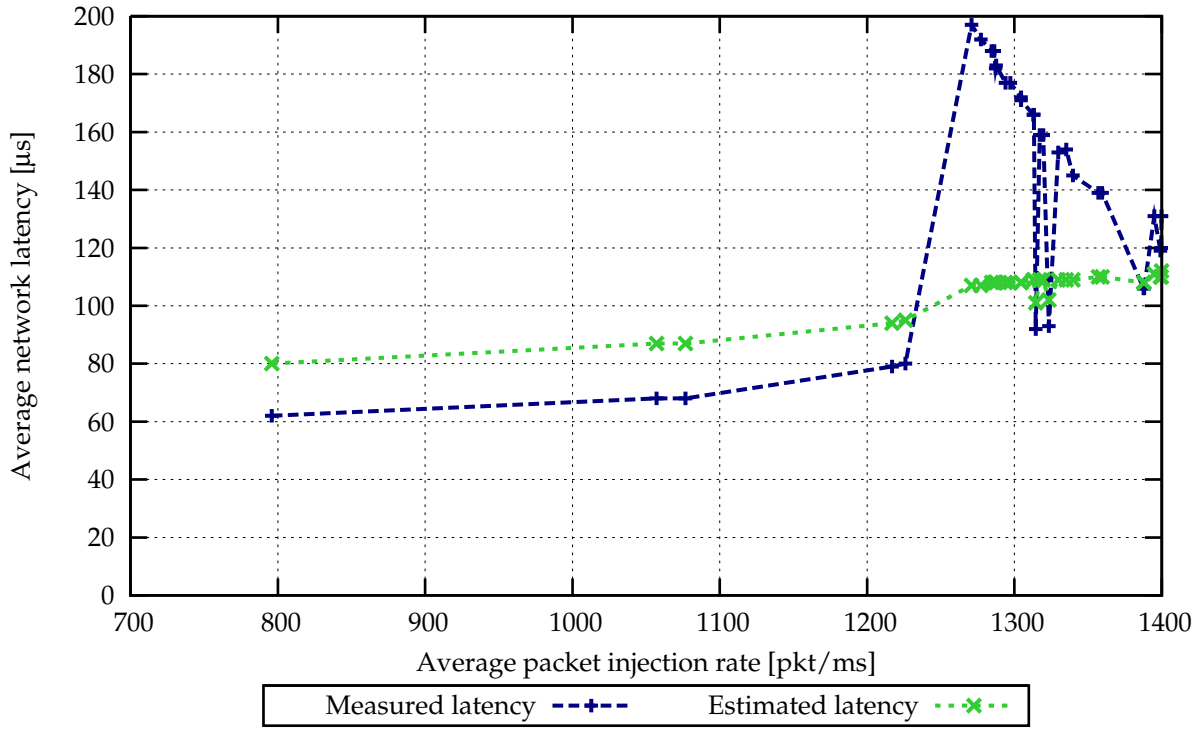


Figure 7.8: Average network latency $\bar{L}$ for an 8×8 system with four data transfers in parallel per receiving node

7.5.1 Average Latency

In the first set of experiments the accuracy of the predicted latency is compared to the simulation results by plotting the *average latency* $\bar{L}$ (μs) versus the *average packet injection rate* (pkt/ms). The average latency $\bar{L}$ is the weighted average of all source destination latencies L_{sd} as shown in Equation 7.22.

$$\bar{L} = \frac{\sum_{\forall(s,d)} L_{sd} \cdot x_{sd}}{\sum_{\forall(s,d)} x_{sd}} \tag{7.22}$$

Since the communication traffic is generated by the test application running on top of the μ-MPI layer and not directly by a traffic generator connected to the NoC, it is not easily possible to directly influence the packet injection rate. However, by varying the computation to communication ratio, the data size being transferred during one MPI transaction and the number of parallel transactions per receiving node, it is possible to indirectly control the average packet injection rate of the system.

Figure 7.7 shows the estimated and measured average packet latency $\bar{L}$ for an 8×8 system with two transactions in parallel per receiving node and Figure 7.8 depicts the average latency for same simulation system with four parallel transactions per receiving node. Overall, the analytical NoC model presented in Section 7.3 is capable of capturing the trends of the average network latency $\bar{L}$ for different packet injection rates. However, for high packet injection rates the measured latency shows an unexpected behavior that is not well reflected by the estimator. In case of the four parallel

transactions per receiver node (Figure 7.8), it can be seen that the deep local minima are also predicted by the estimator. However, the dynamic range of the predicted latency is much lower than the measured dynamic range. It is important to note, that for low and medium packet injection rates the analytical NoC model works pretty well. For high packet injection rate scenarios the analytical NoC model is overly optimistic. Further analysis of this behavior has shown that there are two reasons for this mismatch between the analytical NoC model and the simulation. The first effect contributing to the extremely high average latencies is the burstiness of the communication. The μ-MPI layer splits the data to be transferred into a sequence of packets of size 1kB. The high packet injection rates are achieved by a large data size which leads to long data bursts being transferred between the nodes. This type of communication is likely to create a lot of contention in the different routers and hence high latencies are the consequence. Approaches like [76] try to model this kind of bulk transfers in order to mitigate the estimation error in case of long data transfers. The second reason for this high deviation between analytical NoC model and simulation is the saturation on the interface from the destination router into the local tile. The analytical model presented in Section 7.3 does not consider the local tiles. It assumes that all packets reaching the destination router can directly flow into the local tile. For low and medium traffic this assumption is correct. However, in case of a high number of parallel transactions and long bulk transfers the modeled local tile interface is likely to saturate. Even if only one local tile saturates, the complete NoC will be affected due to the back pressure of the halted packets.

7.5.2 Critical Path Latency

Considering the average packet latency $\bar{L}$ as an indicator for the accuracy is commonly used in the literature about the queuing systems. However, from application perspective the absolute NoC timing is not of highest priority, as long as the relative execution order of the different application tasks remain the same. In order to evaluate this, a second set of experiments has been conducted. In these experiments, the critical path of the application has been measured, i.e. the simulated time required to complete the application. As reference a simple estimator has been used which estimates the latency between a source-destination pair purely based on the number of hops required to reach the destination. In case of the analytical NoC model, 10% of the application's runtime is used to train the estimator before running the entire application. Figure 7.10 depicts the switching between the detailed NoC simulation and the fast simulation mode. Before using the fast NoC simulation mode there is an initial training phase to determine the packet rates for the analytical model. Once the analytical NoC model is trained, the fast NoC simulation mode can be applied. Due to the time variant behavior of many applications it might be necessary to re-train the NoC model from time to time. In order to measure the fidelity of the NoC model the simulation switches in regular intervals back to the detailed simulation mode for a short period of time. During this detailed simulation the training parameters such as packet rates and contention probabilities are compared with the current parameters. In case that the parameters have changed significantly the simulation will remain in the detailed simulation mode to re-train the parameters and then switch back to the fast simulation mode. In case

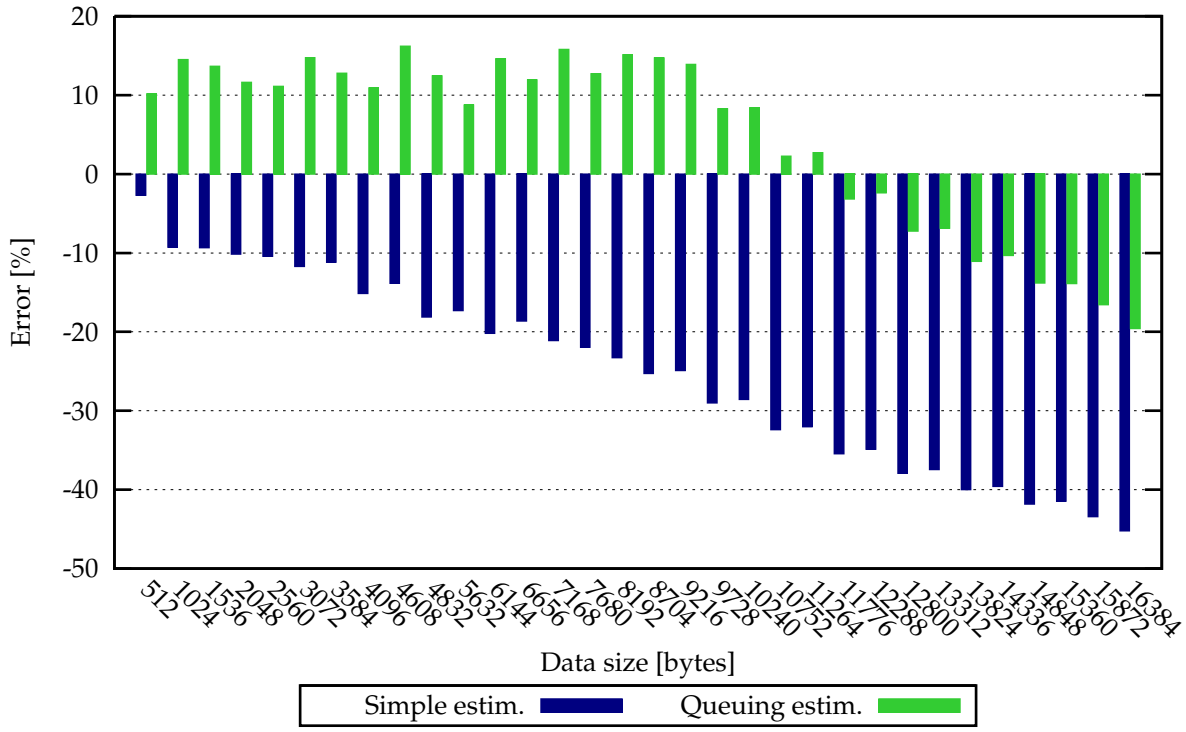


Figure 7.9: Critical path latency of an 8×8 system with two data transfers in parallel per receiving node

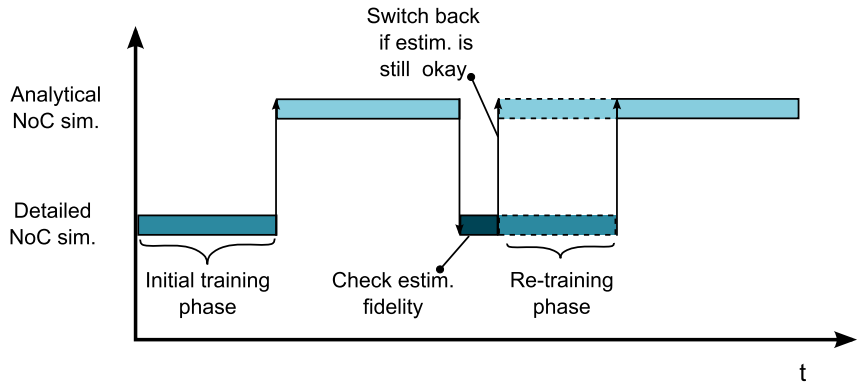


Figure 7.10: Training of the analytical NoC model during runtime

the training parameters are still valid the simulation directly switches back to the fast simulation mode. Hence, the time spent in detailed simulation mode is minimized.

Figure 7.9 and Figure 7.11 show the relative error of an 8×8 system for estimating the critical path of the application for two transactions and respectively four transactions in parallel per receiving node. The simple estimator is underestimating the simulated time and the error is growing with the transferred data size. With the growing data size the fraction of time related to congestion in the NoC is becoming more dominant and therefore the fidelity of the simple estimator drops significantly. The error of

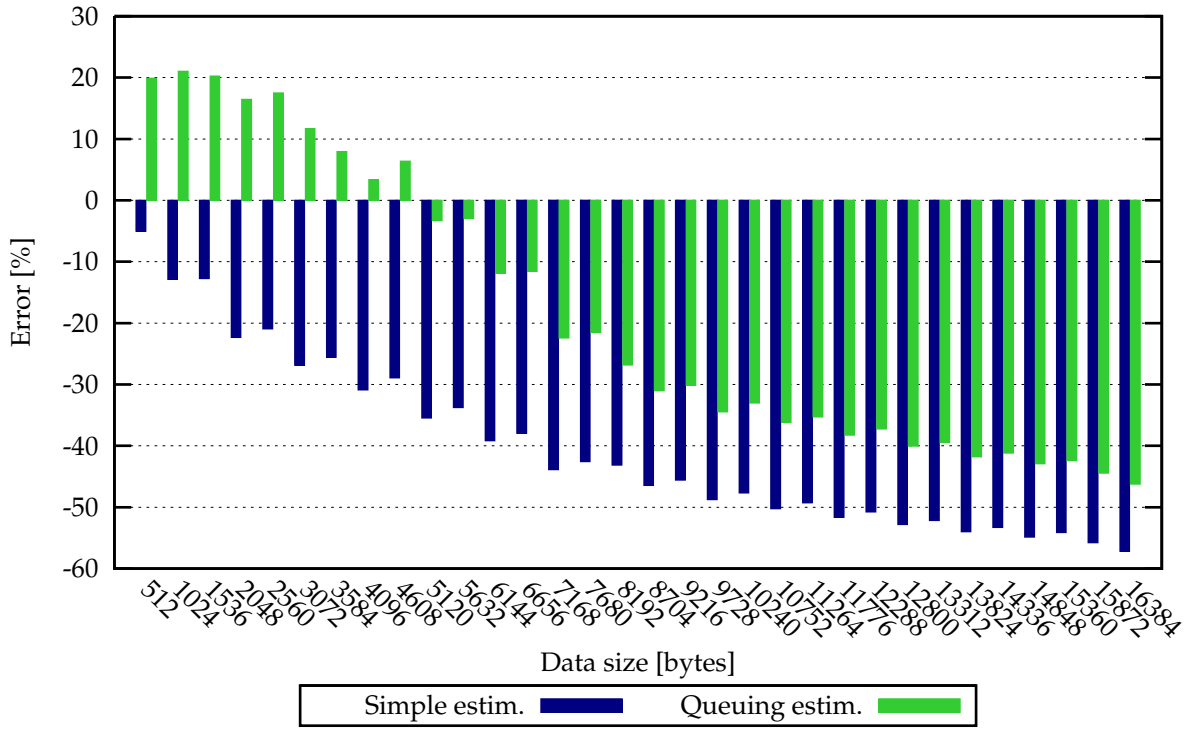


Figure 7.11: Critical path latency of an 8×8 system with four data transfers in parallel per receiving node

the simple estimator ranges from -2.7% to -45.22% for two transactions in parallel and ranges from -5.0% to -57.18% in case of four transactions. In general, the analytical NoC model shows a better estimation fidelity than the simple estimator, especially for the cases with high contention. For test cases which only transfer a low amount of data the analytical NoC model is overly pessimistic. This behavior has been described by Nikitin et al. [111] and is due to fact, that for fixed service times the assumption that the input of all routers are Poisson distributed does not hold. For medium sized data transfers the estimation accuracy yields goods results. But with the increasing data size of the transfers the accuracy of the analytical NoC model drops significantly. However, it is always more accurate than the simple reference estimator. The above described behavior indicates that the approximation used in Equation 7.21 is too optimistic. For medium sized data transfers the estimation error is compensated by the pessimistic estimation of L_{sd}, but for larger transfers $(N_{pkt} - 1)\,\frac{L_{sd}}{N_{\text{hops}}}$ is dominating the estimated overall latency. Equation 7.21 assumes a pipelined behavior for the packets belonging to the same MPI transaction. In general, this assumption is correct, however, in the case of a highly loaded NoC the chances that some packets of a MPI transaction will be delayed because of other transactions increases significantly. This effect is not reflected in Equation 7.21 and hence the estimation is too optimistic for large data transactions.

7.5.3 Simulation Speedup

In a third set of experiments the speedup of the fast NoC simulation mode is being evaluated and compared to the detailed NoC simulation. In order to study the effect of the simulation size on the simulation speedup the simulation complexity has been varied from 16 to 256 nodes. For all experiments the nodes are arranged in a 2D mesh as shown in Figure 7.3. Since this set of experiments focuses on the NoC simulation only, an abstract processor simulator is used to minimize the time required for simulating the computational part of the application. All measurements of the time required to complete the NoC simulation (wall clock time) have been conducted on an Intel i7 920 based system with 2.67 GHz and 12 GB of RAM using Scientific Linux 5.0 as operating system. The following parameters have been used for the speedup measurements: four parallel transactions per receiving node, 8192 bytes being transferred for a single source-destination pair and iteration. In order to obtain reliable timing results, it is important to select a sufficiently large problem size. Especially for the small test cases it is important to select a sufficiently high iteration count in order to get reliable timing results. For this set of experiments a simulation run consists of 10,000 iterations.

Figure 7.12 summarizes the measured simulation times for the different numbers of simulated nodes. The simulation duration has been plotted for three different configurations: the detailed simulation, the simple estimator and the analytical NoC model. As expected, the detailed simulation mode is the slowest simulation type. The simple estimator yields the best results with speedups ranging from $2.7\times$ to $5.8\times$ compared to the detailed simulation. The analytical NoC model is slightly slower than the simple estimator by about 22%, which is due to the increased complexity of the estimator. It is important to understand that the reduced simulation time obtained by both estimators is only partly due to the bypassed NoC. The fact that the μ-MPI layer running on top of the NoC is using *spinlocks* [56], i.e. active waiting, to poll the status of the routers makes the complete system very sensitive on the simulated communication time. Hence, the underestimation of the NoC timing – as shown in the Figures 7.9 and 7.11 – reduces the time spent polling significantly. Thus, the measured speedup can be partly explained by the underestimation of the NoC timing. Therefore, a simple and too optimistic estimator will always be faster than an estimator with good estimation accuracy because less time will be simulated. In conclusion, only if taking the accuracy of the NoC estimator into account it is sensible to compare different NoC simulation speeds. This is especially true for complex software stacks with non-trivial dependencies between the software and the NoC.

In order to obtain a fair comparison between the different estimators the simulation time has been normalized using the detailed simulation as reference. An accuracy factor is calculated by dividing the critical path latency obtained by the estimators by the reference critical path latency obtained by the detailed simulation. The simulation time is then multiplied by this factor in order to normalize the simulation time with respect to the simulation accuracy. Hence, it is possible to calculate the true speedup caused by bypassing the NoC. The normalized simulation times are depicted in Figure 7.12 and show a true speedup ranging from $1.25\times$ to $2.30\times$ for the simple estimator and ranging from $2.34\times$ to $3.18\times$ for the analytical NoC model. Thus, bypassing the NoC while using an analytical NoC model allows to speed up the simulation by approximately

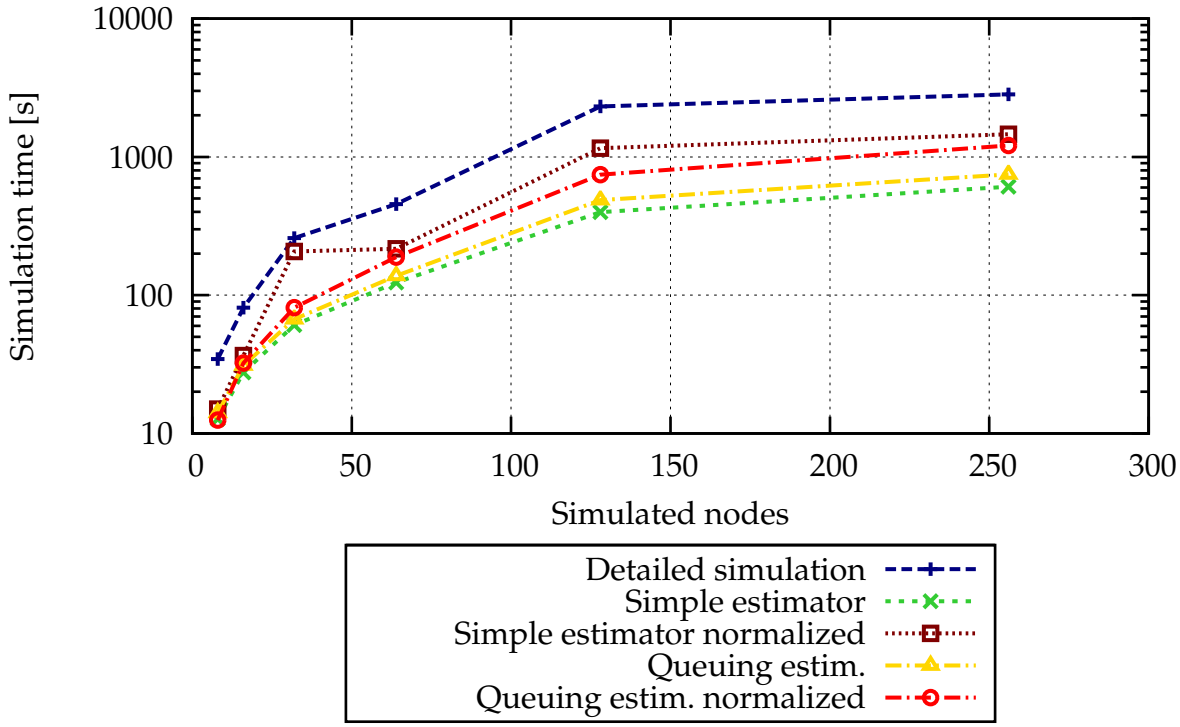


Figure 7.12: Simulation time depending on the number of simulated nodes

a factor of $2 - -3\times$. Compared to the speedups of the hybrid processor simulation (cf. Chapter 6), the speedups achieved by bypassing the NoC are only moderate. This can be explained by two different reasons. First, the employed router model has been optimized for high simulation speed and low complexity during the development. It has been taken care that only very few context switches into the SystemC kernel are required. Second, the router model itself is very simple compared to NoC routers used in industry. Therefore, the time spent inside the router is limited and hence the maximum speedup as well. More complex router models are expected to show higher speedups when being bypassed. Furthermore, with 256 nodes the largest test system has only a moderate size, larger test systems will further increase the fraction of simulation time spent simulating the NoC.

7.6 Conclusion

The growing importance of complex NoCs for the overall performance of MPSoCs has a direct influence on the system simulation by adding additional complexity to the simulation system. In order to mitigate the simulation complexity a hybrid simulation approach for the NoC simulation has been proposed similar to the hybrid processor simulation presented in Chapter 6. During the fast simulation mode the NoC is bypassed and the timing effects of the interconnect – especially the delays due to data contention – are approximated by an analytical NoC model. This NoC model is based on queuing theory and combines the finite buffer size model from Takagi with the

router model from Ogras et al. by introducing the effective service time. The concept of the effective service time allows to summarize all the effects due to resource contention into a single parameter, leading to a time variant service time which is dependent on the actual topology, the router and the application. By using the effective service time the models from Takagi and Ogras et al. can be easily combined into a single analytical NoC model.

The developed hybrid NoC simulation performs well for low and medium loads on the NoC. However, for highly loaded NoCs the analytical NoC model underestimated the timing. This is because of saturation effects on the local tiles which are not considered by the applied NoC model. Furthermore, burst transfers are not well reflected by this model and hence this leads to a too optimistic timing estimation for scenarios with a lot of burst transactions.

A configurable SystemC based simulation environment serves a test platform for the hybrid NoC simulation. On top of the simulated hardware a lightweight software stack is running including a μ-MPI layer which serves as an interface between the application and the routers. Hence, all applications adhering to a subset of the widely used MP-interface can be directly executed on the test platform. This test environment allows to investigate the influence of fast but less accurate NoC simulation on complex software stacks.

The experiments conducted within this thesis demonstrate that speedups between $2\times$ and $3\times$ can be achieved by bypassing the NoC simulation while still considering the timing effects due to data contention. Furthermore, the experiments have shown that for complex software applications it is important not only to measure the achieved speedup but at the same time also to consider the estimation accuracy. Considering one quantity without the other leads possibly to inaccurate predictions of the application behavior. This strong dependency between software and simulated timing stems from software constructs like spinlocks used for polling the status of certain registers. Therefore it is essential to normalize all measured simulation times to the reference simulation before calculating the simulation speedups in order to obtain a fair comparison of the different NoC simulation techniques.

Chapter 8

Summary and Outlook

Over the last decade embedded systems have pervaded nearly all aspects of our daily lives. The main drivers for this development have been the advancements in digital information technologies and the big improvements in deep submicron technologies. These technological improvements have offered the system developers the possibility to integrate complete systems on a chip. However, the downside of this development is the drastic increase of NRE costs because of the huge complexity of modern embedded systems. Managing the complexity of designing an embedded system poses a lot of challenges and risks to the developers. This situation has been further aggravated by the fast growing complexity of embedded software. Already today, the complexity of embedded software has outpaced the HW complexity when comparing lines of code (LoC) written in C and HDL, leading to the so-called embedded software gap. In order to mitigate this gap, new and efficient software design tools are required.

Traditionally, the development of embedded software can only be started once a first hardware prototype of the system is available. This leads to a purely sequential design flow that cannot cope with the decreasing time-to-market observed especially in the consumer market. The increasing complexity of embedded systems combined with the strong requirements, e.g. short time-to-market and reduced NRE costs, require a hardware/software co-design methodology. Only by optimizing the hardware and software simultaneously it is possible to deliver the envisioned functionality while adhering to the various and often contradicting design constraints such as power budget, computing power, design time and costs.

Virtual platforms (VPs) are executable software models of hardware systems which allow simulating the complete system long before a hardware prototype is ready. Therefore, the utilization of VPs offers the system designer the possibility to efficiently develop and test the embedded software long before the hardware is ready. This co-design approach not only reduces the development time, but it also offers the possibility to feed information obtained during the software development back to the hardware developers. Furthermore, VPs offer superior debugging capabilities compared to real hardware by being completely deterministic, offering good observability of internal states and easy distribution among developers. The acceptance of a VP based methodology clearly depends on the achievable simulation speed. Only if developers are capable of simulating the complete software stack within a reasonable amount of time the VP based approach is of interest for software development. In order to cope with the increasing complexity of embedded systems new and efficient simulation techniques are required.

In the context of this thesis three different techniques exploiting the particular requirements of embedded software simulation are proposed to speed up the simulation

of VPs.

A checkpointing technique has been developed that takes the specific characteristics and requirements of SystemC-based VPs into account. It is important to note that the application of checkpointing does not improve the simulation speed itself, but it reduces the time spent simulating by restoring previously saved states of the whole simulation environment. The proposed checkpointing framework is based on *process checkpointing* since this approach imposes fewer constraints on the SystemC simulation models to be checkpointed than *model state serialization*. In order to apply process checkpointing the simulation process needs to be driven into a safe state by removing all external dependencies, e.g. connections to external debuggers and GUIs. Furthermore, all OS dependent resources such as file handles, sockets and threads need to be released before checkpointing. This is achieved by providing a set of callback functions to the user, such that the user defined simulation modules can be extended to support checkpointing.

Compared to model state serialization based approaches process checkpointing based approaches lead to significantly larger image sizes since the complete process state is stored. In order to mitigate this problem a code compression scheme based on the deflate algorithm has been integrated into the framework and the impact on the image size and checkpointing time has been studied. Using the SHAPES platform as case study for the proposed checkpoint/restore framework, checkpoint compression rates of approximately $10\times$ have been measured. The impact of the compression on the checkpointing time is twofold: First, the computational overhead introduced by the compression increases the checkpointing time. Second, less data needs to be stored, which speeds up checkpointing. For checkpoint restoration the time saving due to a smaller image size dominates and thus restoring a compressed checkpoint is significantly faster than restoring an uncompressed checkpoint. However, the overhead introduced by the compression cannot be fully compensated during the checkpoint creation, hence the creation of compressed checkpoints requires more time than uncompressed checkpoints.

Most simulation techniques have been developed for designing and improving microarchitectural features of processors. However, these techniques are suboptimal for the development of software, because of the completely different requirements on the simulation techniques. A hybrid simulation approach has been developed in the context of this thesis which takes the particular characteristics of VPs for software development into account. In contrast to the hardware-centric microarchitectural design space exploration, simulation techniques that require lengthy pre- and post-processing are ill-suited for efficient software debugging.

The proposed hybrid simulation technique offers the embedded software designer the possibility to combine the characteristics of two different simulation techniques: ISS-based simulation and native code execution on the host. Using the hybrid simulation it is possible to fast forward the application by using the native code execution. Once the application has reached the desired state the user can switch back to the traditional ISS-based simulation for detailed inspection of the application. In particular for software debugging scenarios this hybrid simulation technique is of high value, since it

allows the user to quickly reach the faulty state. Different hybrid simulation techniques have been proposed in literature, however most of them concentrate on switching between cycle accurate and instruction accurate simulations. Hence, they are mainly of interest for hardware developers. In order to significantly improve the simulation performance, the proposed hybrid simulation switches between an instruction accurate simulation mode and native code execution. Compared to other approaches, the proposed hybrid simulation technique is agnostic about the ISA of the simulated processor. In particular, the simulated processor and the host processor do not necessarily need to have the same instruction set.

In order to analyze the impact of hybrid simulation on VPs, the SHAPES virtual platform has been used as case study. In the context of this case study the hybrid simulation has been successfully integrated into the virtual platform simulation environment commercially available from CoWare. Measurements have shown that the hybrid simulation technique is capable of speeding up the simulation. In particular, the combination of binary translation and native code execution reaches very high simulation speeds, with speed-ups ranging from $1.2\times$ to $114\times$ compared to binary translation alone have been measured. The obtained results indicate that complex processor architectures, e.g. VLIW and DSP, greatly benefit from this approach, because binary translation approaches do not work well for such complex processor architectures due to the complex mapping of the simulated instruction set to the host instruction set.

The growing complexity of the on-chip interconnect has a direct impact on the simulation performance. In order to mitigate the NoC simulation complexity a hybrid NoC simulation has been developed. This hybrid simulation technique allows to switch between a detailed NoC simulation and a bypassed NoC simulation. During the bypass of the NoC an analytical NoC model is used to estimate the timing effects due to congestion. The developed analytical model is based on queuing theory and combines the finite buffer model of Takagi with the router model proposed by Ogras et al. by introducing the concept of *effective service time*. The effective service time models the service time seen by each router channel due to routing conflicts inside the router.

A configurable SystemC based simulation environment has been developed in order to study the effects of the hybrid NoC simulation. This simulation platform offers a configurable 2D mesh network interconnecting a tiled architecture. In order to better observe the effects of the hybrid NoC simulation an abstract processor simulator is used for the simulation of the computational parts on the application. On top of the simulated hardware a lightweight software stack is running including a μ-MPI layer which serves as an interface between the application and the routers. Hence, all application adhering to the defined subset of the widely used MP-interface can be directly executed on this test platform.

Experimental results have shown that speedups between $2\times$ and $3\times$ can be achieved by bypassing the NoC simulation while still considering the timing effects due to data contention. Furthermore, the experiments have shown that for complex software applications it is important not only to measure the achieved speedup but at the same time also to take the estimation accuracy into account. The software behavior shows a strong dependency to the simulation timing due to certain software constructs, such as spinlocks. For example, ignoring the NoC contention will lead to an overly optimistic

NoC estimator and hence the software will spend less time within the spinlocks. In most cases this modified timing behavior will have no direct impact on the application correctness since a distributed application should be insensitive to changes of the timing. However, it is very important to consider this effect in order to do a fair speedup comparison of different NoC estimators. This can be achieved be normalizing the measured simulation time with the achieved accuracy of the estimator. The hybrid NoC simulation offers the software developers the possibility to estimate the effects of the NoC without the need for a detailed simulation. Especially, during early software development phases this is very helpful since it allows for fast software prototyping.

Future research directions in the context of fast and efficient software simulations are the development of less operating system dependent checkpointing approaches in order to enable checkpointing for the huge amount of legacy IP available. Furthermore, the combination of hybrid NoC and hybrid processor simulation is expected to provide high simulation speeds. Depending of the actual use case the software developer can select the hybrid NoC simulation, the hybrid processor simulation or a combination of both techniques if the simulation speed is of highest importance. With the growing amount of software running on embedded systems combined with the trend of heterogeneous MPSoCs the simulation becomes more and more important for the application debugging. Therefore, it can be anticipated that in future not only the simulation speed but also good debugging support will be of great interest for the software developers. With the proliferation of complex multicore architectures traditional debugging will not be feasible any more due to the high system complexity. Future simulation frameworks will support the developers by collecting and preprocessing information about the application behavior in order to localize potential software bugs such as race conditions and process starvation. Moreover, a tighter cooperation between algorithm development and software development will be required in order realize an efficient algorithm to system mapping and to localize algorithmic problems early in the software development. A first step towards the combination of algorithm debugging with the traditional software debugging flow has been demonstrated in [47].

Appendix A

SHAPES Measurements

This appendix shows the detailed comparison between the simulated single tile SHAPES platform and the hardware prototype. In Table A.1 the measurements for the ARM-ARM intra-tile communication is shown. The benchmark application copies data blocks of different sizes between two processes mapped on the ARM processor. As expected the IA simulation shows a higher deviation from the hardware prototype than the CA simulation. However, IA simulation is capable to predict the trend correctly.

Block size [bytes]	IA sim. [s]	CA sim. [s]	D940 board [s]	IA error [%]	CA error [%]
4	1409.6	912.0	942.4	49.6	-3.2
8	666.0	423.2	436.8	52.3	-3.1
16	335.0	212.6	224.0	49.6	-5.1
32	169.7	106.0	106.5	59.3	-0.5
64	87.4	53.7	54.1	61.7	-0.7
128	46.4	27.7	28.0	65.7	-1.1
256	25.8	14.9	14.9	73.2	0.0
512	15.6	8.5	8.5	83.5	0.0
1024	10.4	5.3	5.2	100.0	2.0
2048	7.9	3.7	3.6	119.4	2.8
4096	6.6	2.9	2.8	135.7	3.6
8192	6.0	2.5	2.5	140.0	0.0
16384	6.6	3.3	3.3	100.0	0.0
32768	7.8	4.8	5.1	52.9	5.9
65536	7.6	4.7	5.1	49.0	7.8

Table A.1: Timing comparison between the SHAPES hardware and the VSP for ARM-ARM intra-tile communication

Table A.2 shows the results obtained by transferring data blocks of different size between the ARM processor and the DSP on the same tile.

Block size [bytes]	IA sim. [s]	CA sim. [s]	D940 board [s]	IA error [%]	CA error [%]
8	345.0	325.0	301.0	14.6	7.9
16	177.0	174.0	166.0	6.6	4.8
32	95.0	100.0	95.0	0.0	5.2
64	57.0	65.0	58.0	-1.7	12.0
128	35.0	47.0	41.0	-14.6	14.6
256	25.0	38.0	32.0	-21.8	18.8
512	22.0	33.0	28.0	-21.4	17.9
1024	22.0	31.0	25.0	-12.0	24.0
2048	15.0	30.0	24.0	-37.5	25.0

Table A.2: Timing comparison between the SHAPES hardware and the VSP for ARM-DSP intra-tile communication

Glossary

Acronyms

APB	AMBA Peripheral Bus
AS	Abstract Simulation/Simulator
ASIP	Application Specific Instruction Set Processor
BBV	Basic Block Vector
CA	Cycle Accurate
CPI	Cycles per Instruction
DBT	Dynamic Binary Translation
DES	Discrete Event Simulation
DMI	Direct Memory Interface
DNP	Distributed Network Processor
DOL	Distributed Operation Layer
EDA	Electronic Design Automation
ESL	Electronic System Level
FDM	Finite Difference Method
GCFG	Global Control Flow Graph
HAL	Hardware Abstraction Layer
HdS	Hardware dependent Software
HW	Hardware
IA	Instruction Accurate
IR	Intermediate Representation
ISA	Instruction Set Architecture
ISS	Instruction Set Simulation/Simulator
JIT-CC	Just-in-Time Cache Compiled
KPN	Kahn Process Network
LoC	Lines of Code
MIPS	Million Instruction Per Second
MPI	Message Passing Interface
MPSoC	Multiprocessor Systems on Chip
NI	Network Interface
NoC	Network-on-Chip

NRE	Non Recurring Engineering
PDF	Probability Density Function
PE	Processing Element
PoT	Peripheral on Tile
RDT	RISC-DSP Tile
RISC	Reduced Instruction Set Processor
SoC	System-on-Chip
SSA	Single Static Assignment
SW	Software
TLM	Transaction Level Modeling
VLIW	Very Long Instruction Word
VP	Virtual Platform
VPU	Virtual Processing Unit
VSP	Virtual SHAPES Platform

List of Figures

List of Tables

Bibliography

[1] *COTSon: Infrastructure for full system simulation*, 2010. [Online]. Available: http://cotson.sourceforge.net/

[2] H. Abdel-Shafi, E. Speight, and J. K. Bennett, "Efficient user-level thread migration and checkpointing on Windows NT clusters," in *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, WA, USA, July 1999.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[4] A. Allan, D. Edenfeld, J. Joyner, W.H., A. Kahng, M. Rodgers, and Y. Zorian, "2001 technology roadmap for semiconductors," *IEEE Computer*, vol. 35, no. 1, pp. 42–53, January 2002.

[5] G. Altarelli, "A QCD primer," 2002. http://arxiv.org/abs/hep-ph/0204179v1

[6] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*. New York, NY, USA: ACM, 1967, pp. 483–485.

[7] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: infrastructure for full system simulation," *SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 52–61, January 2009.

[8] ATMEL, "DIOPSIS: Dual Inter Operable Processor in A Single Silicon". http://www.atmel.com

[9] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification*, W. Wolf, Ed. San Francisco, CA, USA: Morgan Kaufmann, 2007.

[10] D. Bartholomew, "QEMU: A multihost, multitarget emulator," *Linux Journal*, vol. 2006, no. 145, pp. 3–9, May 2006.

[11] R. Bedicheck, "SimNow: fast platform simulation purely in software," in *Hot Chips 16*, August 2004.

[12] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)*. Berkeley, CA, USA: USENIX Association, February 2005, pp. 41–41.

[13] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, "SystemC cosimulation and emulation of multiprocessor SoC designs," *IEEE Transactions on Computers*, vol. 36, no. 4, pp. 53–59, April 2003.

[14] L. Benini and G. De Micheli, "Networks on chips: a new SoC paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70–78, January 2002.

[15] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, July 2006.

[16] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, "PISA - a platform and programming language independent interface for search algorithms," in *Evolutionary Multi-Criterion Optimization (EMO 2003)*. Lugar, Portugal: Springer, April 2003, pp. 494–508.

[17] L. Bononi and N. Concer, "Simulation and analysis of network on chip architectures: Ring, Spidergon and 2D mesh," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, vol. 2, Munich, Germany, March 2006, pp. 154–159.

[18] A. Bouchhima, P. Gerin, and F. Pétrot, "Automatic instrumentation of embedded software for high level hardware/software co-simulation," in *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2009, pp. 546–551.

[19] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated application-level checkpointing of MPI programs," in *Proceedings of the Principles and Practice of Parallel Programming (PPoPP)*. New York, NY, USA: ACM Press, June 2003, pp. 84–94.

[20] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Collective operations in application-level fault-tolerant MPI," in *Proceedings of the 17th annual International Conference on Supercomputing (ICS)*. New York, NY, USA: ACM Press, June 2003, pp. 234–243.

[21] D. C. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," Tech. Rep. CS-TR-1997-1342, June 1997.

[22] M. Burtscher and I. Ganusov, "Automatic synthesis of high-speed processor simulators," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 55–66.

[23] Cadence Design Systems, Inc., *NC-Sim*, San Jose, CA. http://www.cadence.com/

[24] Carbon Design Systems, Inc., *Carbon Model Studio*, Acton, MA. http://www.carbondesignsystems.com/

[25] L. Carloni and A. Sangiovanni-Vincentelli, "Coping with latency in SoC design," *IEEE Micro*, vol. 22, no. 5, pp. 24–35, September/October 2002.

[26] S. Chakraborty, S. Künzli, and L. Thiele, "A general framework for analyzing system properties in platform-based embedded system designs," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Munich, Germany, March 2003, pp. 190–195.

[27] Chumby Inc., San Diego, CA, USA. http://www.chumby.com

[28] P. E. Chung, W.-J. Lee, Y. Huang, D. Liang, and C.-Y. Wang, "Winckp: A transparent checkpointing and rollback recovery tool for Windows NT applications," in *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (FTCS)*. Washington, DC, USA: IEEE Computer Society, June 1999, pp. 220–223.

[29] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 22, no. 1, pp. 128–137, May 1994.

[30] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "Platform-based behavior-level and system-level synthesis," in *Proceedings of IEEE International SOC Conference*, Austin, Texas, USA, September 2006, pp. 199–202.

[31] M. Coppola, M. D. Grammatikakis, R. Locatelli, G. Maruccia, and L. Pieralisi, *Design of Cost-Efficient Interconnect Processing Units: Spidergon STNoC*. CRC press, Taylor and Francis group, 2008.

[32] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, and A. Scandurra, "Spidergon: A novel on-chip communication network," in *Proceedings of the International Symposium on System-on-Chip (SoC)*, Tampere, Finland, November 2004, pp. 15–26.

[33] CoWare Inc., *CoWare Virtual Platform*, San Jose, CA. http://www.coware.com/products/virtualplatform.php

[34] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Transactions on Computers*, vol. 36, no. 5, pp. 547–553, May 1987.

[35] W. Dally and S. Lacy, "VLSI architecture: Past, present, and future," in *Advanced Research in VLSI, 1999. Proceedings. 20th Anniversary Conference on*, Atlanta, Georgia, USA, March 1999, pp. 232–241.

[36] D. De Vries, "Sound reinforcement by wave field synthesis: Adaptation of the synthesis operator to the loudspeaker directivity characteristics," *Journal of the Audio Engineering Society (AES)*, vol. 44, no. 12, pp. 1120–1131, December 1996.

[37] P. Deutsch, *DEFLATE Compressed Data Format Specification version 1.3*. United States: RFC Editor, 1996.

[38] B. Dimitri and G. Robert, *Data Networks*. Englewood Cliffs, NJ, USA: Prentice-Hall, Inc., 1992.

[39] J. Duell, "The design and implementation of Berkeley lab's Linux checkpoint/restart," Paper LBNL-54941, Berkeley, CA, USA, April 2005.

[40] L. Eeckhout, K. de Bosschere, and H. Neefs, "Performance analysis through synthetic trace generation," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, Texas, USA, April 2000, pp. 1–6.

[41] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan, "ASIM: A performance model framework," *IEEE Computer*, vol. 35, no. 2, pp. 68–76, February 2002.

[42] Eve Inc., *Zero Bug (Zebu)*, San Jose, CA. http://www.eve-team.com

[43] A. Falcón, P. Faraboschi, and D. Ortega, "An adaptive synchronization technique for parallel simulation of networked clusters," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Washington, DC, USA: IEEE Computer Society, April 2008, pp. 22–31.

[44] J. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. San Francisco, CA, USA: Morgan Kaufmann, 2005.

[45] Freescale Inc., *MC94MX21*. http://www.freescale.com

[46] L. Gao, K. Karuri, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr, "Multiprocessor performance estimation using hybrid simulation," in *Proceedings of the Design Automation Conference (DAC)*. New York, NY, USA: ACM, June 2008, pp. 325–330.

[47] B. Geiser, S. Kraemer, J. Weinstock, R. Leupers, and P. Vary, "An integrated algorithm and software debugging tool for signal processing applications," in *Proceedings of System, Software, SoC and Silicon Debug Conference (S4D)*. Sophia Antipolis, France: ECSI, September 2009.

[48] P. Gerin, X. Guérin, and F. Pétrot, "Efficient implementation of native software simulation for MPSoC," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Munich, Germany, March 2008, pp. 676–681.

[49] X. Guérin, K. Popovici, W. Youssef, and F. Rousseau, "Flexible application software generation for heterogeneous multi-processor system-on-chip," in *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, vol. 1, Beijing, China, July 2007, pp. 279–286.

[50] X. Guérin and F. Pétrot, "A system framework for the design of embedded software targeting heterogeneous multi-core SoCs," in *Proceedings of the IEEE International Conference on Application Specific Systems, Architectures, and Processors (ASAP)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2009, pp. 153–160.

[51] R. Gupta, "Introduction to lattice QCD," 1998. http://arxiv.org/abs/hep-lat/9807028v1

[52] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, pp. 532–533, May 1988.

[53] W. Haid, M. Keller, K. Huang, I. Bacivarov, and L. Thiele, "Generation and calibration of compositional performance analysis models for multi-processor systems," in *Proceeding of the International Conference on Systems, Architectures, Modeling and Simulation (SAMOS)*, Samos, Greece, July 2009, pp. 92–99.

[54] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "SimFlex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 31–34, March 2004.

[55] J. Henkel, "Closing the SoC design gap," *IEEE Computer*, vol. 36, pp. 119–121, September 2003.

[56] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[57] R. Ho, K. Mai, and M. Horowitz, "The future of wires," *IEEE Transactions on Computer-Aided Design*, vol. 89, no. 4, pp. 490–504, April 2001.

[58] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language," *IEEE Transactions on Computer-Aided Design*, vol. 20, no. 11, pp. 1338–1354, November 2001.

[59] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with Lisa*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[60] V. Horak and P. Gruber, "Parallel numerical solution of 2-D heat equation," in *Proceedings of Parallel Numerics '05: Theory and Applications (ParNum05)*, M. Vajteršic, R. Trobec, P. Zinterhof, and A. Uhl, Eds., Portoroz, Slovenia, April 2005, pp. 47–56.

[61] K. Huang, I. Bacivarov, J. Liu, and W. Haid, "A modular fast simulation framework for stream-oriented MPSoC," in *Proceedings of the IEEE Symposium on Industrial Embedded Systems (SIES)*, Lausanne, Switzerland, July 2009, pp. 74–81.

[62] Y. Hwang, S. Abdi, and D. Gajski, "Cycle-approximate retargetable performance estimation at the transaction level," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Munich, Germany, March 2008, pp. 3–8.

[63] IEEE Standards Committee 754, *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754*. Institute of Electrical and Electronics Engineers, New York, 1985, reprinted in ACM SIGPLAN Notices, 22(2):9–25, 1987.

[64] Imperas, Ltd., Oxfordshire, U.K. http://www.imperas.com/

[65] International Technology Roadmap for Semiconductors, *SoC Design Cost Model*, 2007. http://www.itrs.net

[66] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, third Edition.

[67] E. R. Jason Duell, Paul Hargrove, "Requirements for linux checkpoint/restart," Paper LBNL-49659, Berkeley, CA, USA, May 2002.

[68] A. A. Jerraya, A. Bouchhima, and F. Pétrot, "Programming models and HW-SW interfaces abstraction for multi-processor SoC," in *Proceedings of the Design Automation Conference (DAC)*. New York, NY, USA: ACM Press, June 2006, pp. 280–285.

[69] D. Jones and N. Topham, "High speed CPU simulation using LTU dynamic binary translation," in *Proceeding of the Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, 2009.

[70] J. Jung, S. Yoo, and K. Choi, "Fast cycle-approximate MPSoC simulation based on synchronization time-point prediction," *Design Automation for Embedded Systems*, vol. 11, no. 4, pp. 223–247, December 2007.

[71] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing '74: Proceedings of the IFIP Congress*. North-Holland Publishing Co, 1974, pp. 471–475.

[72] K. Karuri, M. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr, "Fine-grained application source code profiling for ASIP design," in *Proceedings of the Design Automation Conference (DAC)*, Anaheim, California, USA, June 2005, pp. 329–334.

[73] T. Kempf, M. Dörper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout, "A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Munich, Germany, March 2005.

[74] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr, "A SW performance estimation framework for early system-level-design using fine-grained instrumentation," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. Munich, Germany: European Design and Automation Association, March 2006, pp. 100–106.

[75] B. W. Kernighan and D. Ritchie, *The C Programming Language (2nd Edition)*. Prentice Hall PTR, March 1988.

[76] L. Kleinrock, *Queuing Systems, Volume 1: Theory*. Wiley, 1975.

[77] T. Kogel, M. Doerper, A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, and S. Goossens, "A modular simulation framework for architectural exploration of on-chip interconnection networks," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. New York, NY, USA: ACM, October 2003, pp. 7–12.

[78] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. Washington, DC, USA: IEEE Computer Society, March 2004, pp. 75–87.

[79] M. Laurenzano, B. Simon, A. Snavely, and M. Gunn, "Low cost trace-driven memory simulation using SimPoint," *SIGARCH Computer Architecture News*, vol. 33, no. 5, pp. 81–86, December 2005.

[80] M. T. Lazarescu, J. R. Bammi, E. Harcourt, L. Lavagno, and M. Lajolo, "Compilation-based software performance estimation for system level design," in *Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT)*, 2000, p. 167.

[81] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queuing network models*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984.

[82] J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S. Han, "FaCSim: a fast and cycle-accurate architecture simulator for embedded systems," in *Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. New York, NY, USA: ACM, June 2008, pp. 89–100.

[83] W. Lee, K. Patel, and M. Pedram, "B2Sim: A fast micro-architecture simulator based on basic block characterization," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. New York, NY, USA: ACM, October 2006, pp. 199–204.

[84] R. Leupers, *Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[85] P. Levy and S. Lemeshow, *Sampling of populations: methods and applications*, 3rd ed. New York, NY: Wiley, 1999.

[86] D. Liang, P. E. Chung, Y. Huang, C. M. R. Kintala, W.-J. Lee, T. K. Tsai, and C.-Y. Wang, "NT-SwiFT: Software implemented fault tolerance on Windows NT," in *Journal of Systems and Software*, vol. 71. New York, NY, USA: Elsevier Science Inc., April 2004, pp. 127–141.

[87] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of UNIX processes in the condor distributed processing system," University of Wisconsin Madison Computer Sciences, Tech. Rep. 1346, April 1997.

[88] O. Lysne, "Towards a generic analytical model of wormhole routing networks," *Microprocessors and Microsystems*, vol. 21, no. 7–8, pp. 491–498, November 1998, IEEE 1355.

[89] J. Maebe, M. Ronsse, and K. D. Bosschere, "DIOTA: Dynamic instrumentation, optimization and transformation of applications," in *Proceedings 4th Workshop on Binary Translation (WBT)*, Charlottesville, VA, USA, September 2002.

[90] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, February 2002.

[91] R. Marculescu, U. Ogras, L.-S. Peh, N. Jerger, and Y. Hoskote, "Outstanding research problems in NoC design: System, microarchitecture, and circuit perspectives," *IEEE Transactions on Computer-Aided Design*, vol. 28, no. 1, pp. 3–21, January 2009.

[92] MARSSx86, *Micro-Architectural and System Simulator for x86-based Systems*, 2010. http://www.marss86.org/

[93] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, November 2005.

[94] C. J. Mauer, M. D. Hill, and D. A. Wood, "Full-system timing-first simulation," *SIGMETRICS Performance Evaluation Review*, vol. 30, no. 1, pp. 108–116, June 2002.

[95] Mentor Graphics Inc., *ModelSim*, Wilsonville, OR. http://www.mentor.com/

[96] Mentor Graphics Inc., *Vista Architect*, Wilsonville, OR. http://www.mentor.com/vista

[97] T. Meyerowitz, A. Sangiovanni-Vincentelli, M. Sauermann, and D. Langen, "Source-level timing annotation and simulation for a heterogeneous multiprocessor," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Munich, Germany, March 2008, pp. 276–279.

[98] T. Miyamori, "MPSoC architecture trade-offs for multimedia applications," in *MPSoC'07: 7th International Forum on Application-Specific Multi-Processor SoC*, June 2007.

[99] M. Moadeli, A. Shahrabi, W. Vanderbauwhede, and M. Ould-Khaoua, "An analytical performance model for the Spidergon NoC," in *Proceedings of the 21st International Conference on Advanced Information Networking and Applications (AINA)*, Niagra Fall, Ontario, Kanada, May 2007, pp. 1014–1021.

[100] P. Mohapatra, "Wormhole routing techniques for directly connected multicomputer systems," *ACM Computing Surveys*, vol. 30, no. 3, pp. 374–410, September 1998.

[101] W. S. Mong and J. Zhu, "Dynamosim: A trace-based dynamically compiled instruction set simulator," in *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, November 2004, pp. 131–136.

[102] M. Montón, J. Engblom, and M. Burton, "Checkpoint and restore for SystemC models," in *Proceedings of Forum on specification & Design Languages (FDL)*, Sophia Antipolis, France, September 2009.

[103] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, April 1965.

[104] MPI, *Message Passing Interface Forum*, 2010. http://www.mpi-forum.org/

[105] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[106] Multifacet GEMS, *General Execution-driven Multiprocessor Simulator*, 2010. http://www.cs.wisc.edu/gems/

[107] A. Muttreja, A. Raghunathan, S. Ravi, and N. Jha, "Hybrid simulation for energy estimation of embedded software," *IEEE Transactions on Computer-Aided Design*, vol. 26, no. 10, pp. 1843–1854, October 2007.

[108] A. Muttreja, A. Raghunathan, S. Ravi, and N. K. Jha, "Hybrid simulation for embedded software energy estimation," in *Proceedings of the Design Automation Conference (DAC)*. New York, NY, USA: ACM, June 2005, pp. 23–26.

[109] J. Namkung, D. Kim, R. Gupta, I. Kozintsev, J.-Y. Bouget, and C. Dulong, "Phase guided sampling for efficient parallel application simulation," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. New York, NY, USA: ACM, October 2006, pp. 187–192.

[110] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, San Diego, California, USA, June 2007, pp. 89–100.

[111] N. Nikitin and J. Cortadella, "A performance analytical model for network-on-chip with constant service time routers," in *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. New York, NY, USA: ACM, November 2009, pp. 571–578.

[112] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann, "A universal technique for fast and flexible instruction-set architecture simulation," in *Proceedings of the Design Automation Conference (DAC)*. New York, NY, USA: ACM Press, June 2002, pp. 22–27.

[113] U. Y. Ogras and R. Marculescu, "Analytical router modeling for networks-on-chip performance analysis," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. San Jose, CA, USA: EDA Consortium, April 2007, pp. 1096–1101.

[114] U. Y. Ogras and R. Marculescu, "Analysis and optimization of prediction-based flow control in networks-on-chip," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, no. 1, pp. 1–28, January 2008.

[115] Open MPI, *Open Source High Performance Computing*, 2010. http://www.open-mpi.org/

[116] Open SystemC Initiative (OSCI), *IEEE 1666-2005 Standard, Open SystemC Language Reference Manual*, IEEE, New York, April 2005. http://www.systemc.org/

[117] Open Virtual Platforms (OVP). http://www.ovpworld.org

[118] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-time signal processing (3rd ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1999.

[119] P. S. Paolucci, "The diopsis multiprocessor tile of SHAPES," in *6th International Forum on Application-Specific Multi-Processor SoC MPSOC'06*, Colorado, USA, August 2006.

[120] P. S. Paolucci, A. A. Jerraya, R. Leupers, L. Thiele, and P. Vicini, "SHAPES: A tiled scalable software hardware architecture platform for embedded systems," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Seoul, Korea, October 2006, pp. 167–172.

[121] P. S. Paolucci, P. Kajfasz, P. Bonnot, B. Candaele, D. Maufroid, E. Pastorelli, A. Ricciardi, Y. Fusella, and E. Guarino, "mAgic-FPU and MADE: A customizable VLIW core and the modular VLIW processor architecture description environment," *Computer Physics Communications*, vol. 139, no. 1, pp. 132–143, September 2001.

[122] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong, "Detecting phases in parallel applications on shared memory architectures," in *IEEE International Parallel and Distributed Processing Symposium*, June 2006.

[123] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *Proceedings of USENIX Winter 1995 Technical Conference*, New Orleans, LA, USA, January 1995, pp. 213–224. http://citeseer.ist.psu.edu/plank95libckpt.html

[124] K. Popovici, X. Guérin, F. Rousseau, P. Paolucci, and A. Jerraya, "Efficient software development platforms for multimedia applications at different abstraction levels," in *Proceedings of the IEEE International Workshop on Rapid Systems Prototyping (RSP)*, Porto Alegre, Brazil, May 2007, pp. 113–122.

[125] K. Popovici, X. Guérin, F. Rousseau, P. S. Paolucci, and A. A. Jerraya, "Platform-based software design flow for heterogeneous MPSoC," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 4, pp. 1–23, July 2008.

[126] PTLsim, *x86-64 cycle accurate processor simulation design infrastructure*, 2010. http://www.ptlsim.org/

[127] W. Qin, J. D'Errico, and X. Zhu, "A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Seoul, Korea: ACM Press, October 2006, pp. 103–198.

[128] M. Reshadi, P. Mishra, and N. Dutt, "Instruction set compiled simulation: A technique for fast and flexible instruction set simulation," in *Proceedings of the Design Automation Conference (DAC)*. New York, NY, USA: ACM Press, June 2003, pp. 758–763.

[129] M. Reshadi, P. Mishra, and N. Dutt, "Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 8, no. 3, pp. 1–27, April 2009.

[130] J. Ringenberg, C. Pelosi, D. Oehmke, and T. Mudge, "Intrinsic checkpointing: A methodology for decreasing simulation time through binary modification," *Performance Analysis of Systems and Software*, pp. 78–88, March 2005.

[131] E. Roman, "A survey of checkpoint/restart implementations," Lawrence Berkeley National Laboratory, Berkeley, CA, USA, Berkeley Lab Technical Report LBNL-54942, July 2002.

[132] M. Rosenblum and M. Varadarajan, "SimOS: A fast operating system simulation environment," Stanford, CA, USA, Tech. Rep. CSL-TR-94-631, July 1994.

[133] C. Rowen and S. Leibson, *Engineering the complex SOC : fast, flexible design with configurable processors*. Upper Saddle River, N.J. : Prentice Hall PTR ; London : Pearson Education, 2004.

[134] A. Sahu, M. Balakrishnan, and P. R. Panda, "A generic platform for estimation of multi-threaded program performance on heterogeneous multiprocessors," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. San Jose, CA, USA: EDA Consortium, April 2009, pp. 1018–1023.

[135] H. A. Sanghavi and N. B. Andrews, "TIE: An ADL for designing application-specific instruction set extensions," in *Processor Description Languages*, P. Mishra and N. Dutt, Eds. Burlington: Morgan Kaufmann, 2008, pp. 183–216.

[136] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," *International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, Winter 2005.

[137] A. Scherrer, A. Fraboulet, and T. Risset, "Automatic phase detection for stochastic on-chip traffic generation," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Seoul, Korea, October 2006, pp. 88–93.

[138] E. Schnarr and J. R. Larus, "Fast out-of-order processor simulation using memoization," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. New York, NY, USA: ACM Press, December 1998, pp. 283–294.

[139] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance timing simulation of embedded software," in *Proceedings of the Design Automation Conference (DAC)*, Anaheim, California, June 2008, pp. 290–295.

[140] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*. New York, NY, USA: ACM Press, October 2004, pp. 298–307.

[141] H. Shafi, P. J. Bohrer, J. Phelan, C. A. Rusu, and J. L. Peterson, "Design and validation of a performance and power simulator for PowerPC systems," *IBM J. Res. Dev.*, vol. 47, no. 5-6, pp. 641–651, September 2003.

[142] SHAPES - Scalable Software Hardware computing Architecture Platform for Embedded Systems. http://www.shapes-p.org

[143] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *IEEE Micro*, vol. 23, no. 6, pp. 84–93, December 2003.

[144] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*. New York, NY, USA: ACM Press, October 2002, pp. 45–57.

[145] SimFlex, *Fast, Accurate & Flexible Computer Architecture Simulation*, 2010. http://parsa.epfl.ch/simflex/

[146] V. Soteriou, H. Wang, and L. Peh, "A statistical traffic model for on-chip interconnection networks," in *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, September 2006, pp. 104–116.

[147] T. Sporer, M. Beckinger, A. Franck, I. Bacivarov, W. Haid, K. Huang, L. Thiele, P. Paolucci, P. Bazzana, P. Vicini, J. Ceng, S. Kraemer, and R. Leupers, "SHAPES - a scalable parallel HW/SW architecture applied to wave field synthesis," in *Proceedings of the AES 32nd International Conference*, Hillerød, Copenhagen, Denmark, September 2007, pp. 175–187.

[148] S. Spors, R. Rabenstein, and J. Ahrens, "The theory of wave field synthesis revisited," in *Proceedings of the AES 124th Convention*, Amsterdam, Netherlands, May 2008.

[149] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *Journal ACM*, vol. 29, no. 4, pp. 928–951, October 1982.

[150] D. Sylvester and K. Keutzer, "Impact of small process geometries on microarchitectures in systems on a chip," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 467–489, April 2001.

[151] Synopsys Inc., *Innovator*, Mountain View, CA. http://www.synopsys.com/Tools/SLD/VirtualPlatforms/

[152] Synopsys Inc., *VHDL and Verilog Simulation Software*. http://www.synopsys.com

[153] P. K. Szwed, D. Marques, R. M. Buels, S. A. McKee, and M. Schulz, "SimSnap: Fast-forwarding via native execution and application-level checkpointing," in *Proceedings on the 8th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-8)*. Madrid, Spain: IEEE Computer Society, February 2004, pp. 65–74. http://simsnap.csl.cornell.edu

[154] H. Takagi, *Queuing Analysis, Volume 2: Finite Systems*. Amsterdam, The Netherlands: North-Holland, 1993.

[155] Target Compiler Technologies. http://www.retarget.com

[156] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, April 2002.

[157] O. Temam and R. Leupers, Eds., *Processor and System-on-Chip Simulation*. Berlin, Heidelberg: Springer-Verlag, 2010.

[158] Tensilica, Inc., Santa Clara, CA. http://www.tensilica.com/

[159] The DWARF Debugging Standard. http://dwarfstd.org/

[160] The M5 Simulator System, *A modular platform for computer system architecture research*, 2010. http://www.m5sim.org/

[161] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems," in *Proceedings of the Seventh International Conference on Application of Concurrency to System Design (ACSD)*, Bratislava, Slovakia, July 2007, pp. 29–40.

[162] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 4, Geneva, Switzerland, May 2000, pp. 101–104.

[163] TMS320C67 Texas Instruments. http://www.ti.com/

[164] Tony D. Givargis and Frank Vahid and Jörg Henkel, "Trace-driven system-level power evaluation of system-on-a-chip peripheral cores," in *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)*. New York, NY, USA: ACM, January 2001, pp. 306–312.

[165] USB Implementers Forum, Inc., *Universal Serial Bus Specification Revision 2.0*, 2009. http://www.usb.org/developers/docs/

[166] C. H. K. van Berkel, "Multi-core for mobile phones," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Nice, France, April 2009, pp. 1260–1265.

[167] VaST Systems Technology Corporation, Sunnyvale, CA. http://www.vastsystems.com/

[168] VirtuTech Inc., *Simics*, San Jose, CA. http://www.virtutech.com/whatissimics.html

[169] VMWare Inc., *VMWare Server*, Palo Alto, CA. http://www.vmware.com/products/server/

[170] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse, "System architecture evaluation using modular performance analysis: A case study," *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 6, pp. 649–667, July 2006.

[171] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. Hoe, D. Chiou, and K. Asanovic, "RAMP: Research accelerator for multiple processors," *IEEE Micro*, vol. 27, no. 2, pp. 46–57, March-April 2007.

[172] T. Wild, A. Herkersdorf, and R. Ohlendorf, "Performance evaluation for system-on-chip architectures using trace-based transaction level simulation," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, vol. 1, Munich, Germany, March 2006, pp. 248–253.

[173] K. G. Wilson, "Confinement of quarks," *Phys. Rev. D*, vol. 10, no. 8, pp. 2445–2459, October 1974.

[174] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, San Diego, California, USA, June 2003, pp. 84–95.

[175] T. Ye, H. T. Kaur, S. Kalyanaraman, and M. Yuksel, "Large-scale network parameter configuration using an on-line simulation framework," *IEEE/ACM Transactions Networking*, vol. 16, no. 4, pp. 777–790, August 2008.

[176] J. J. Yi and D. J. Lilja, "Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations," *IEEE Transactions on Computers*, vol. 55, no. 3, pp. 268–280, March 2006.

[177] Y. Yi, D. Kim, and S. Ha, "Fast and accurate cosimulation of MPSoC using trace-driven virtual synchronization," *IEEE Transactions on Computer-Aided Design*, vol. 26, no. 12, pp. 2186–2200, December 2007.

[178] S. Yoo, I. Bacivarov, A. Bouchhima, Y. Paviot, and A. A. Jerraya, "Building fast and accurate SW simulation models based on hardware abstraction layer and simulation environment abstraction layer," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. Washington, DC, USA: IEEE Computer Society, March 2003.

[179] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, vol. 0, pp. 23–34, April 2007.

[180] A. Zeller, "Datenstrukturen visualisieren und animieren mit DDD," *Informatik - Forschung und Entwicklung*, vol. 24, no. 3, pp. 65–75, June 2001. http://www.st.cs.uni-sb.de/publications/details/zeller-ife-2001/

[181] H. Zhong and J. Nieh, "CRAK: Linux checkpoint/restart as a kernel module," Department of Computer Science, Columbia University, Tech. Rep. CUCS-014-01, November 2001.

[182] J. Zhu and D. Gajski, "An ultra-fast instruction set simulator," *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, vol. 10, no. 3, pp. 363–373, June 2002.

[183] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, November 1999.

Curriculum Vitae

Name:	Stefan Kraemer
Date of birth:	July 17th, 1977 Heidelberg, Germany

October 2010	Draeger Safety, AG & Co. KGaA, Luebeck
Aug. 2004 – Aug. 2010	Research Assistant at the department for Software for Systems on Silicon, Prof. Dr. Rainer Leupers, RWTH Aachen University
Jul. 2004	Graduation as Dipl.-Ing.
Jan. 2004 – Jul. 2004	Diploma (Master) Thesis at the Institute for Integrated Signal Processing Systems, RWTH Aachen University, "Investigations in the Automated Customization of Instruction-Sets for Application Specific Processors"
Oct. 1998 – Jul. 2004	Study of Electrical Engineering and Information Technology with focus on Information and Communication Engineering
June 1997	Abitur
Aug. 1988 – Jun. 1997	Gymnasium Martin Luther Schule in Marburg, Germany

Luebeck, July 2011 14th *Stefan Kraemer*