Jussi Hanhirova

Performance analysis of hardware accelerated scheduling

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 12.11.2014

Thesis supervisor:

Prof. Heikki Saikkonen

Thesis advisor:

D.Sc. (Tech.) Vesa Hirvisalo

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Jussi Hanhirova

Title: Performance analysis of hardware accelerated scheduling

Date: 12.11.2014 Language: English Number of pages:7+71

Department of Computer Science and Engineering

Professorship: Software systems / Embedded systems Code: T-106

Supervisor: Prof. Heikki Saikkonen

Advisor: D.Sc. (Tech.) Vesa Hirvisalo

Performance analysis of heterogeneous MPSoCs (Multiprocessor System-on-Chip) is difficult. The non-determinism of parallel computation, communication delays and memory accesses force the system components into complex interaction. Hardware acceleration is used both to speedup the computations and the scheduling on MPSoCs. Finding a accompanying software structuring and efficient scheduling algorithms is not a straightforward task.

In this thesis we investigate the use of simulation, measurement and modeling methods for analyzing the performance of heterogeneous MPSoCs. The viewpoint of this thesis is in simulation and modeling: How a high abstraction level simulation methodology can be used in modeling and analyzing of parallel systems based on MPSoCs. In particular we are interested in efficient use of hardware accelerated scheduling mechanisms and how they can be analyzed.

Both parallel simulation and simulation of parallel systems contains many different methods, tools and approaches that attempt to balance between competing goals and cope with a specific subset of the problem space. Challenge is that in all approaches most of the simulation and modeling related problems remain and new challenges emerge.

This thesis shows that the resource network methodology and dynamic scheduling models are a viable approach in modeling heterogeneous MPSoCs with accelerators. Concrete contributions are based on upgrading an existing simulation framework to support parallelism. Main contribution is on one hand that modeling concepts have been widened, and on the other hand that the supporting mechanisms have been implemented. The thesis work in progress was published in a peer reviewed international scientific workshop and the final results in a peer reviewed international scientific conference. The toolset has also been used in multi-university organized teaching and by the industry.

Keywords: performance analysis, modeling, simulation, parallelism, hardware accelerated scheduling, MPSoC

AALTO-YLIOPISTO
PERUSTIETEIDEN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Jussi Hanhirova		
Työn nimi: Laitteistokiihdytetyn vuoronnuksen suorituskykyanalyysi		
Päivämäärä: 12.11.2014	Kieli: Englanti	Sivumäärä:7+71
Tietotekniikan laitos		
Professuuri: Ohjelmistotekniikka / Sulautetut järjestelmät		Koodi: T-106
Valvoja: Prof. Heikki Saikkonen		
Ohjaaja: TkT Vesa Hirvisalo		

Heterogeenisten moniydinjärjestelmien suorituskykyanalyysi on haasteellista. Laskennan epä-deterministisyys, kommunikaatioviiveet ja lukuisat muistioperaatiot saattavat järjestelmän komponentit monimutkaisiin vuorovaikutussuhteisiin. Laitteistokiihdytettyjä ajoitusmenetelmiä käytetään nopeuttamaan ajoituspäätöksiä. Sopivan ohjelmarakenteen ja tehokkaiden ajoitusalgoritmien löytäminen ei ole helppoa.

Tässä työssä tutkitaan miten simulointi-, mittaus- ja mallinnusmenetelmiä voi käyttää laitteistokiihdytettyjen moniydinjärjestelmien suorituskykyanalyysiin. Työn näkökulma on simuloinnissa ja mallinnuksesa: Miten korkean abstraktiotason simulointimenetelmät soveltuvat moniydinjärjestelmiin pohjautuvien rinnakkaisten järjestelmien mallinnukseen ja suorituskykyanalyysiin. Erityisen kiinnostuksen kohteena on laitteistokiihdytteisten ajoitusmenetelmien tehokas käyttö sekä analysointi.

Rinnakkaissimulointi pitää sisällään erilaisia menetelmiä, työkaluja ja lähestymistapoja jotka pyrkivät tasapainottelemaan ristiriitaisten tavoitteiden välillä. Haasteena on se että kaikissa lähestymistavoissa simulaation ja mallinnuksen useimmat ongelmat säilyvät ja uusia ongelmia ilmaantuu.

Työn tulokset viittaavat siihen että resurssiverkkopohjainen menetelmä dynaamisen ajoituksen kanssa on toimiva lähestymistapa rinnakkaisten järjestelmien suorituskykyanalyysiin. Työn konkreettiset tulokset pitävät sisällään olemassaolevan simulointiympäristön päivittämisen rinnakkaisuutta tukevaksi. Keskeinen tulos on toisaalta se että mallinnusmenetelmiä on laajennettu ja toisaalta se että näitä tukevat mekanismit on toteutettu. Keskeneräisen työn tulokset on julkaistu vertaisarvioidussa tieteellisessä seminaarissa ja valmiin työn tulokset vertaisarvioidussa tieteellisessä konferenssissa. Simulointiympäristöä on käytetty usean yliopiston järjestämässä yhteisopetuksessa sekä teollisuudessa.

Avainsanat: suorituskykyanalyysi, mallinnus, simulointi, rinnakaisuus, laitteistokiihdytetty vuoronnus, moniydinjärjestelmä

Preface

I would like to thank Prof. Heikki Saikkonen for supervising the thesis and giving me the opportunity to work full-time with the topic.

I am grateful to D.Sc.(Eng.) Vesa Hirvisalo for support, guidance and all the enlightening discussions during the thesis process.

Finally, I want to thank my wife Piia and my daughter Hilla. I am thankful to Piia for her love and never-ending support and encouragement with my studies and the thesis. Hilla I thank for the happy moments of play at home that always make me realise what really matters in life.

Otaniemi, 12.11.2014

Jussi Hanhirova

Contents

Abstract ii

Abstract (in Finnish) iii

Preface iv

Contents v

Abbreviations vii

1 Introduction 1
1.1 Research problem . . . 1
1.2 Contributions . . . 1
1.3 Structure . . . 2

2 Heterogeneous computing systems 4
2.1 Internet of Things . . . 4
2.2 Multi and manycores . . . 5
2.3 High performance embedded computing . . . 6
2.4 Embedded System design . . . 8

3 Performance analysis of computing systems 11
3.1 Performance analysis . . . 11
3.2 Simulation . . . 12
3.3 Parallel simulation . . . 13
3.4 Simulators . . . 15
3.5 Resource networks . . . 17

4 PSE – Performance Simulation Environment 18
4.1 Queuing network simulation . . . 18
4.2 Toolset overview . . . 18
4.3 Modeling workflow . . . 20
4.3.1 Editor tools . . . 20
4.3.2 Compiler tools . . . 23
4.4 Basic building blocks . . . 24
4.5 Monitoring . . . 28
4.6 RNS runtime . . . 28

5 Mechanism for resource network simulation 31
5.1 Modeling hardware accelerated scheduling . . . 31
5.1.1 Pull mode scheduler . . . 31
5.1.2 Push mode scheduler . . . 33
5.1.3 Dynamic scheduling . . . 35
5.1.4 Modeling memory . . . 36
5.1.5 Fork and join . . . 39

5.2 Mapping PSE to hardware 41
5.3 Parallelizing discrete event simulators 42
5.3.1 PSE replicated trials 43
5.3.2 Parallel PSE 45

6 Demonstrative experiment **47**
6.1 Experiment setup 47
6.2 Simulation results 50
6.3 Conclusions on experiment results 52

7 Discussion **54**
7.1 Challenge 54
7.2 Related work 54
7.3 Discoveries 56
7.4 Future work 56

8 Conclusions **59**

Bibliography **61**

Abbreviations

API	Application Programming Interface
CPU	Central processing unit
DATE	Design, Automation and Test in Europe
DES	Discrete event simulation
DSP	Digital Signal Processing
ESM	European Simulation and Modeling
FIFO	First In First Out
GB	Gigabyte
Gbps	Gigabytes per second
GNU	GNU's Not Unix
GNU Pth	GNU Portable threads
GUI	Graphical User Interface
HW	Hardware
IP	Intellectual property
ISA	Instruction Set Architecture
MB	Megabyte
MPSoC	Multiprocessor System-on-Chip
NoC	Network on Chip
NPU	Network Processing Unit
NSN	Nokia Solutions and Networks
OpenEM	Open Event Machine
PSE	Performance Simulation Environment
QoS	Quality of Service
RNS	Resource Network Simulator
RTL	Register Transfer Level
SMP	Symmetric Multi Processing
SoC	System-on-Chip
SW	Software
TLB	Translation Lookaside Buffer
WCET	Worst Case Execution Time

1 Introduction

This thesis investigates the use of modeling, simulation and measurement for analyzing performance of parallel systems implemented on MPSoCs with accelerators. The contribution of this thesis is to present a way to model and simulate these systems. Concrete contributions are based on updating an existing simulation framework and describing mechanisms necessary for modeling hardware accelerated scheduling.

While simulation is a widely studied field and there are many methods that are used for it, both simulation of parallel systems and parallel simulation are hard. PSE (Performance Simulation Environment) is a toolset aimed at performance analysis of hardware and software co-scheduled manycore systems. PSE fits into the active research field of parallel simulators. PSE applies discrete event simulation of models represented using the resource networks methodology. This thesis shows that the resource network methodology and dynamic scheduling models are a viable approach for modeling heterogeneous MPSoCs with accelerators.

1.1 Research problem

Modern MPSoCs (Multiprocessor System-on-Chip) are tightly connected parallel systems with complex interactions. Performance analysis of these systems is hard because the non-determinism of parallel computation, communication delays and memory accesses force the system components into complex interaction. On top of this, the internal data streams on the MPSoC interact with computation and communication resources affecting their behavior. The resource interaction is further mixed with scheduling, which is usually performed at multiple points in the system. Finally, MPSoCs are usually used to process data streams of dynamic behavior. Generally the worst case inputs for MPSoCs are unknown.

Parallelism is hard to analyze directly using traditional computer system methods, but using different abstraction levels systems based on MPSoCs can be modeled and simulated. Detailed monitoring of the simulation is required to be able to retrieve metrics and traces for performance analysis.

The viewpoint of this thesis is in simulation and modeling: How a high abstraction level simulation methodology can be used in modeling and analysis of parallel systems based on MPSoCs. The research question is following: how to efficiently use hardware accelerated scheduling mechanisms and how hardware accelerated scheduling can be analyzed?

1.2 Contributions

This thesis presents a way how MPSoCs can be modeled using a resource reservation based mechanism. Focus of this thesis is in scheduling, how to model and analyze hardware, software and hybrid scheduled systems. Performance analysis is done by constructing executable models of systems with adjustable monitoring mechanisms. Modeling is based on the use of graphical editor tools which allow model description using basic building blocks. Models are simulated using discrete event based approach.

Concrete contributions of this thesis are based on upgrading an existing simulation framework to support parallelism. Main contribution is on one hand that modeling concepts have been widened and on the other hand that the supporting mechanisms have been implemented. Work done in the thesis has been presented to international scientific community and it has also been used in the university teaching and by the industry.

Contributions can be summarized as follows:

- Upgrading of an existing simulation framework (PSE) to support modeling and simulation of parallel systems.
- Implementation of fundamental models required for performance modeling of hardware scheduled systems.
- The thesis work in progress was published in a peer reviewed scientific workshop 3PMCES at the DATE2014 conference (on 28th March 2014 in Dresden, Germany).[1]
- The results of the thesis were published in a peer reviewed scientific conference ESM'2014 (on 22-24th October 2014 in Porto, Portugal).[37]
- The PSE toolset has been used in multi-university organized teaching [86] during the ParallaX project[1].
- PSE has been used by the industry partners of the ParallaX project.

The result of the first contribution is an upgrade to an Open Source toolset PSE (Performance Simulation Environment). The upgrading consisted of the implementation and testing of the fork-join mechanism required for modeling simultaneous resource requests. The second contribution includes concrete models and documented description on how they can be used to model the key structures of heterogeneous MPSoCs. The thesis work in progress was presented in form of an abstract paper and a poster at the 3PMCES workshop at the DATE2014 conference. The results of the thesis were presented in form of a regular paper in the ESM'2014 conference. Contribution for the multi-university organized teaching, the ParallaX education day, was to create performance analysis learning material package and teach the basics of PSE modeling workflow for the participants.

1.3 Structure

The structure of this thesis is as follows. First an overview to the context of this thesis is given in Chapter 2 which is titled as Heterogeneous computing systems. Chapter 2 begins by introducing the concept of Internet of Things. Next the multi- and manycores are presented, after which the term high performance embedded

[1] ParallaX project is an industry-driven research consortium lead by finnish universities. It is focused in research of parallel systems.[66]

computing is explained. Chapter 2 is concluded by a view to embedded system design.

Chapter 3 takes a wider look to the performance analysis of computing systems. The Chapter 3 begins with an introduction to performance analysis, after which the simulation methodology is presented. Chapter 3 also presents main approaches for parallel simulation and review some recent simulators. Chapter 3 concludes in opening up the resource networks concept.

Chapter 4 presents the PSE (Performance Simulation Environment) toolset. First an overview to PSE is made and the basic modeling workflow is described. Next the basic building blocks of PSE are introduced, after which the monitoring functionality and the RNS runtime are presented.

In Chapter 5 the mechanisms for resource network simulation are introduced and problems related to parallelizing the simulator are discussed. In particular concrete examples are given how hardware accelerated scheduling and memory systems can be modeled.

In Chapter 6 a demonstrative experiment using PSE is presented. The demonstrative experiment shows how PSE can be used in early design space exploration of networked computing systems.

Chapter 7 sets PSE into a wider context, underlines the main problems facing the field of parallel simulation and discusses possible solutions for the problems.

The conclusions are presented in Chapter 8 which forms the last Chapter of this thesis.

2 Heterogeneous computing systems

In this chapter the context of this thesis is presented. This chapter begins with a view to the Internet of Things (IoT) and the main technologies behind its evolution. Next, implications of IoT to the requirements in computational performance are presented. To conclude this chapter the challenges in embedded system design are presented.

The interested reader can refer to [89] for a hardware point of view to parallelism and to [69] for a software point of view on programming heterogeneous parallel systems, [53] can be used as a reference for performance analysis and [40] for parallel simulation. Additional materials have been refered to with the introduction of the topics.

2.1 Internet of Things

Internet of Things (IoT) refers to connecting objects as independently identifiable entities to the Internet. In year 2008 the number of devices connected to the Internet outnumbered the human population, and in 2013 the number had reached 13 billion. According to Cisco there will be about 50 billion Internet-connected devices in 2020. [38]

These IoT objects have a large variance in size, in the type of data they produce or process, and in the real-time performance requirements. The objects can be anything from vehicles like boats or cars to agricultural field sensors, or from medical implants to everyday things such as food packages, furniture and clothes. What these objects share in common is that they are embedded systems equipped with sensors and Internet connection.[38]

The massive boom in the number of new Internet-connected devices has been enabled by the development in chip and sensor manufacturing technologies [52] and in the wireless communication technologies [39], such as software defined radio [69]. Chips and sensors are not just unprecedentedly cheap to make, but they also consume negligible amounts of energy. Multiple small devices can be interconnected to form wireless sensor networks and using energy harvesting technologies they can be made self-sustaining.[106, 113]

An important branch of IoT is the ITS (Intelligent Traffic System) applications. The concept of ITS is based on collecting data with sensors installed in vehicles and the surrounding infrastructure, processing and extraction information out of that data, and then sharing that through different communication means. Many of the ITS applications are hard-real time systems, where missing a deadline in computation could lead into fatal consequences.[99]

Another branch of IoT is the industrial Internet, where physical machines and networked sensors are integrated to form complex control systems. An overview of IoT and related technologies can be found from [8]. Current trends and future visions of IoT can be read from [24]. A survey to (energy efficient) wireless communication technologies is in [39], wireless sensor networks are described in [113], energy harvesting technologies are surveyed in [113, 106] and the concept of Intelligent Traffic

Systems is presented in [99].

In general IoT applications can be seen as systems that process sensor data streams. The amount of data being transmitted in the Internet is constantly growing. The data stream behavior, which can be static or dynamic, is central to the computational requirements of the system. In case of dynamic data streams system scheduling is a key issue. Hardware scheduling is used in systems that process dynamic data streams because hardware by nature is parallel and can see the system as whole. This contrasts to software which needs to browse through data structures step-by-step.[89]

In contrast to general IoT applications, many applications in the High Performance Computing (HPC) domain are considered tightly connected. They generate vast amounts of data that needs to be aggregated on-the-fly near the sensors. Software does not seem to scale for this kind of work, instead hardware schedulers and supporting hardware structures need to be used.[20]

2.2 Multi and manycores

Moore's law still holds today – more and more transistors can be packed to a smaller area.[74] Dennard scaling, introduced in 1974, has been the driving force behind Moore's law. Dennard scaling argumented that the performance per watt of processors is growing exponentially at roughly the same rate as transistor density in Moore's law.[32] But Dennard scaling no longer holds – at small sizes current leakage power becomes significant and leads to the so-called *power wall* [89]. The power wall and other challenges related to chip manufacturing techniques has lead chips to the age of *dark silicon* [97]. Dark silicon refers to current chip architectures which utilize dynamic voltage and frequency management to power up only certain regions of a chip at a time. As power has become the main limiting factor in computation, more power efficient implementations of hardware functionalities translate directly to better computational performance.[89]

Multicores are systems in which from two to slightly over ten processors are connected to resources via a shared bus. Manycores refer to systems with much more processors or simple cores and accelerators, that are connected using some other interconnect than traditional bus. A SoC, System-on-Chip, integrates all computer components on one chip and a MPSoC, multiprocessor System-on-Chip, is a SoC that uses multiple processors. All MPSoC components are connected with on-chip interconnects.[110]

The ability to pack more and more transistors on a chip makes it possible to implement special hardware units for different chip functionalities to minimize energy consumption. This has lead to the development of heterogeneous manycores, which contain different types of cores and special purpose accelerators. Examples of such systems are MPSoCs with a traditional multicore processor connected to a GPU such as the AMD APU (Accelerated Processing Unit) [4] or a NPU (Network Processing Unit) MPSoC with a pipeline of special purpose hardware accelerators and an array of MIPS cores such as the Cavium Octeon family.[22] The miniaturization of the digital functions on a chip is also referred to as *More Moore* and the diversification of chip functionalities is referred as *More-than-Moore*.[52]

2.3 High performance embedded computing

High-performance computing (HPC) is traditionally associated to the field of scientific applications requiring large amounts of computations. Embedded computing on the other hand has been associated with embedded systems requiring real-time performance. High-performance embedded computing (HPEC) refers to real-time embedded systems with high-performance requirements and strict power and cost budget requirements.[110]

After the processor manufacturers have hit the power wall, heterogeneous manycore SoCs have become their main product in attempts to increase computational performance.[110] The most efficient systems at the moment use pipelined processors of five-to-eight stages, which is th most efficient design in terms of performance per joule and silicon area.[7]

The large set of HPEC applications and the strict power and cost budgets has lead to a huge selection of many different MPSoC designs.[110] They all have their own peculiarities and need tailored software to fully benefit from the available hardware. Problem is how to program these platforms efficiently, and will such methods scale and offer code and performance portability.[55]

Traditionally the computer system stack has consisted of applications described using a programming language on top of a runtime, which is being managed by an operating system providing hardware resources through an ISA interface.[55]

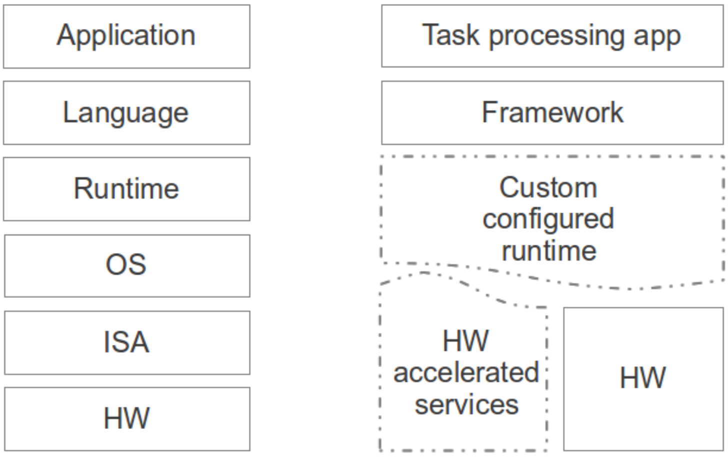


Figure 1: The traditional view to the computer system stack on the left and on the right one alternative for the evolvement needed.

With the new manycore SoCs and parallel programming models this traditional stack structure is breaking. The traditional thread based programming methods and the software based thread scheduling is ineffective in modern MPSoCs. Threads induce too much overhead to these systems in the form of context-switches and surplus context information. Also the concurrent programming model associated with software threads is not scaling with increasing core count.[46, 64]

Instead of redesigning the whole computer system stack from scratch, evolution of existing components is needed, as the existing computer and embedded systems are full of legacy code that is too laborious and in practise impossible to be rewritten from scratch.[7]

Figure 1 illustrates the evolvement needed in the traditional way of abstracting the computer system stack. In the traditional programming stack (left), the application is a single process and the operating system abstracts other concurrent processes away by creating a virtual machine for the process. Adding multithreading has happened by making all the layers of the picture thread safe (i.e., cache coherency in hardware, memory consistency models at ISA, safe-grained locking at OS level, threading support in runtimes, and concurrency in languages). But the traditional methods do not scale with increasing core counts. Instead hardware accelerated functionalities must be used (right side of the picture) and the software and runtime must be configured according to the underlying hardware.

Different approaches for describing applications have emerged to allow software to scale with increasing core counts. The new approaches access resources using concepts such as worker threads, run-to-completion loops and describe applications using work units such as microthreads, events, jobs, tasks or tags.[78, 85, 7] But the main problem remains: how to allocate the computing resources to get maximal utilization with given dependencies and timing constraints?

Traditional software based scheduling is not efficient with the new work units. Evolution is needed close to the hardware at OS and ISA levels. In order to open the ISA bottleneck[2], hardware functionality needs to be expanded and exploited in new manners.[89]

Hardware-based or hardware-accelerated scheduling and runtime operation must be used to accelerate access to the resources. Hardware accelerated scheduling is based on the use of special hardware to access simultaneously and in parallel different structures in a MPSoC. Scheduling is restricted by computational dependencies and hardware latencies.[89] Power efficient scheduling and related power management issues can be found in [109, 100, 114]. Regarding schedulers there exists pure software implementations, pure hardware implementations and mixtures of both worlds. Choice between software and hardware implementations is always a trade-off between numerous factors, for example between performance and general applicability. Examples of hardware schedulers include: Carbon [60], Isonet [65], task-superscalar [36], ADM [93], TMU [96] and the tag based approach [23].

Real-time scheduling on a manycore platform requires executing tasks so that their time constraints are always met. While the increased number of cores generally

[2] ISA bottleneck, or scalability bottleneck at ISA level, relates to architectures performing computation using the fetch-decode-execute cycle and a limited number of processor registers. Limited number of registers and the complex logic involved e.g. in out-of-order execution and branch prediction form a performance hindrance between the SW and the HW. HW behind the ISA is forced to make guesses about the SW. More communication between SW and HW is needed and more and smaller cores with simpler datapath would help to solve these issues.[89]

means better average case execution time, the worst-case execution time (WCET) might get worse because of the more complex interactions. Measuring the average performance of heterogeneous systems is hard and their cycle-accurate simulation is too slow. Traditional methods such as WCET analysis are not automatically working on manycores.[2] Although some research has been done for example with GPUs [49], embedded streaming applications with data-dependent tasks [10], and multicore processors with shared caches and bus [27], accurate performance prediction of multi- and manycores is still considered unfeasible to make.[2] A survey on hard real-time scheduling for multiprocessor systems in done in [31].

Using memory over interconnects is the normal method of communications needed in parallel computation.[47, 33] Transaction memories are an attempt to hide the difficulties from application programmers, but the field has still many problems to solve.[63, 62] Current approaches are towards memory consistency.[44] How the memory models have influenced programming languages is studied in [3].

Main methods of parallel programming can be divided into implicit and explicit approaches. In the explicit approach the programmer is in control to expose parallelism in the code (e.g. OpenMP [81], MPI [75], OpenCL [80], CUDA [79]). The implicit way relies to for example parallelizing compilers [57] or speculative multithreading for speeding up sequential code [51]. Parallelizing compilers have been studied extensively, but currently effective parallelism extraction still requires domain-specific knowledge.[57] Hardware aware approach to the programming of manycores is presented in [17]. A through-out view to the programming of heteogeneous MPSoCs is given in [69].

2.4 Embedded System design

MPSoC platform usefulness is determined by a large set of different aspects, e.g. supported programming languages and compiler technologies, portability of code and performance and available runtime support. Usually most of the final product customization comes from software needs. That is why the software development environment plays such a crucial role in defining the usefulness of a MPSoC platform.[55]

Embedded System design allows implementation of system functionalities either in software, in hardware or as a mixture of both. HW/SW co-design techniques can be used to explore the design space of systems based on MPSoCs. The HW/SW partitioning is in key role when performance goals have to be met with chip area, power dissipation and total system cost constraints. Methods for iterative HW/SW co-design have been described e.g. in [56, 45, 43].

The HW/SW co-design space is huge, with many interactions between and within hardware and software. Design Space Exploration (DSE) techniques are used to determine the minimum number of resources needed to schedule the applications. Depending on the application and target platform complexity, DSE may take much effort. This is especially true in designing a system running multiple applications with hard-real time constraints. Higher abstraction levels are used to narrow the search. Some methods that employ simulators at early design space exploration are for example in [11, 107, 73].

Modern embedded MPSoCs have tight constraints: They have real-time guarantees to fulfill, they have limited resources and they have a limited power budget. Model-driven development restricts an application to a certain model of computation, which facilitates quantitative analysis with respect to e.g. schedulability and worst-case execution time.[35]

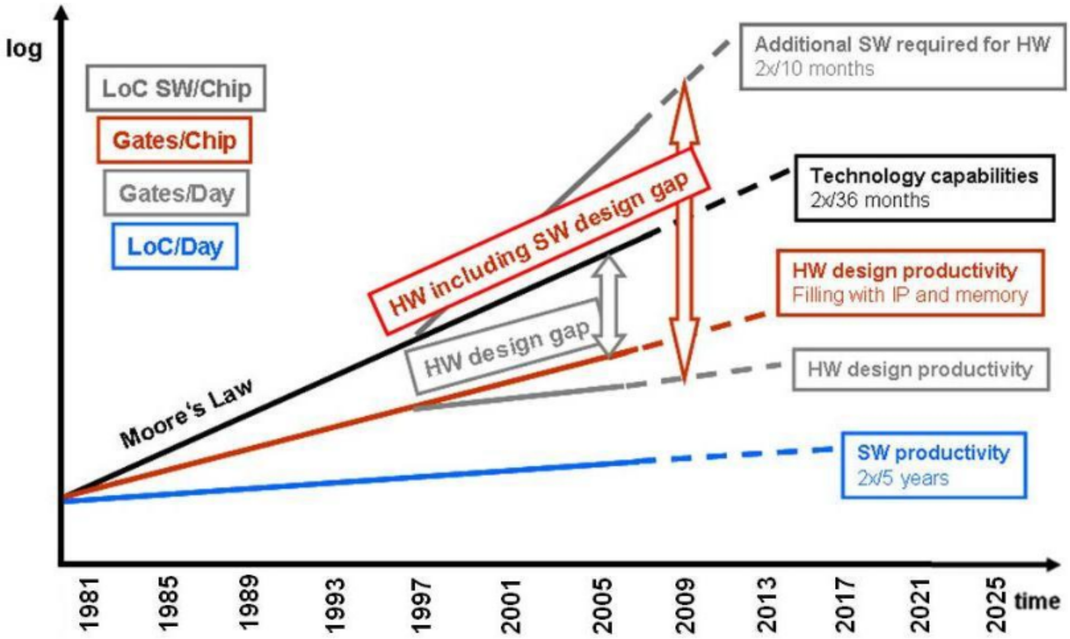


Figure 2: The HW-SW design gap is caused by the fact that the productivities and demands of HW and SW are growing at different pace with regard to Moore's law. Although the HW design productivity has been improved in recent years with the IP component reuse, the HW design productivity still lags behind the technological capabilities dictated by Moore's law. New HW designs demand new SW in increasingly growing numbers, but the productivity especially for HW-dependent SW is far behind. The red arrow shows the SW-HW design gap. Adapted from ITRS [52].

In Figure 2 the advancements in SW and in HW design productivity is portrayed with technological capabilities dictated by Moore's Law. Figure 2 shows a gap between the requirements and current productivity of MPSoC related technologies. New methods and tools are needed to help the system designers in making critical design decisions. To simplify the specification, verification and implementation of MPSoCs, and to enable more efficient design space exploration, a new level of abstraction is needed above the familiar register-transfer level.[52]

Designing large scale systems for the Internet of Things widens the already extensive design space of MPSoCs. Central questions when designing IoT application focus on partitioning the required computation between different system elements. In a hypothetical IoT application computation could be partitioned for example between a sensor's microcontroller, a mobile device, a node of the communication infrastructure and a cloud server. Optimization problem includes minimizing energy consumption and cost of all components, using only a limited set technologies at the consumer devices and a limited budget for developing new software or hardware. Early design space exploration of above described systems calls for higher levels of abstraction. High-level simulation models and automated design space exploration can be seen as an essential element in designing these new systems.[52]

3 Performance analysis of computing systems

This thesis studies how the performance of hardware accelerated scheduling and parallel systems in general can be analyzed. The chosen approach is modeling and simulation using resource networks as the modeling basis.

First in this chapter performance analysis is defined. Next a general view to modeling and simulation is done. Finally the resource network methodology is presented.

3.1 Performance analysis

In computation performance has always been a key issue. In general the goal is to find the best cost-performance trade-off and acquire the highest performance at the lowest cost.[53]

Depending on the system under study there are different performance goals that need to be minimized/maximized. E.g. Maximizing throughput and minimizing latency of a packet processing system [23] or minimizing power consumption while providing a satisfying level of functionality for a mobile device.[100] On some fields, such as the vehicle industry, the main performance metrics can be for example maximizing reliability and fault tolerance with minimal manufacturing costs.[99]

Performance evaluation can be done in many different ways. The three main approaches are analytical modeling, simulation and measurement. Analytical modeling uses mathematical models to abstract main characteristics of a system and analyzes or predicts it's behavior. Analytical models require assumptions and generalizations in order that the analysis methods can be used. Simulation is based on modeling the system and then measuring its behavior when it is executed. Direct measurements can be used when an executable system exists. In general the three main methods of performance analysis: measurement, analytical modeling and simulation, should complement each other and be used to validate results of other methods, but in practice this is not always possible.[53]

Selecting an evaluation technique and suitable metrics for it are the main first steps in performance evaluation. Availability, or the life cycle stage, of the system under study poses constraints for the methods that can be applied to it. Measuring a yet not existing system is not possible, whereas it can usually be modeled for simulation or for analytical modeling. Another constraint is set by the time available for evaluation. Analytical modeling is in general considered to require least time of the three, while for simulation and measurement the time varies greatly. Simulation and measurement require special tools that have their strengths and weaknesses and a learning and application curve to take. Both simulation and measurement can be made more accurate using more time and resources, but in general a suitable balance between time and accuracy is sought. There exists no recipe to mechanically conduct a successful performance evaluation. One could argue that performance analysis is a form of art, that requires considerable amount of insight and experience for successful completion.[53]

3.2 Simulation

Simulation is about mimicking the behavior of a defined system. Simulations are based on models, which describe the system at an appropriate level of detail. Simulation of computation includes simulating hardware, software and their co-operation with regard to inputs for the system. Simulation of complex systems usually requires the use of several abstraction layers.[83]

Simulation models are interpreted by executing them, alas by computing changes in the model with respect to possible input and time change. The model can be continuous or discrete and its interpretation can also be either of these. A dynamic model reacts to time while a static model represents a steady-state model that is time-invariant. Models can have an uncountable number of instances or states, or they can be enumerable. Similarly the model can be deterministic or stochastic. A deterministic model's states can be uniquely determined by parameters in the model and by its previous states, while in a stochastic model the transitions between different model states are also affected by probabilities. Input for a simulation model can be based on random number generation or on using fixed traces.[83]

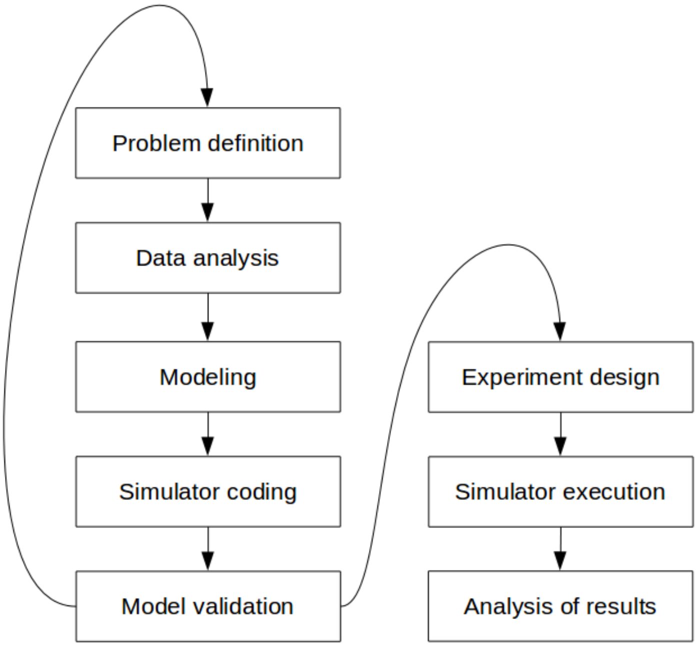


Figure 3: Simulation is an iterative process involving multiple steps. Here an overview to the different phases is presented. After problem description and data analysis follows the modeling of the system. An executable simulator program is created from the model. The model and the simulator are refined until they can be validated. Experiments are designed and then executed using the simulator. Monitoring the execution of simulation experiments produces traces for analysis.[83].

Simulation is an iterative process involving multiple steps, Figure 3 summarizes the process. After initial problem definition and analysis of system data, the system is modeled. After the model is ready a simulation program is prepared using automated tools or manually. Successful validation usually requires iteration by going back to previous phases of the process. After the simulation program is successfully validated the simulation experiments can be designed. Experiments are executed on the simulator and the execution is measured to obtain results for analysis.[83]

There are different designs for building simulation models. In [83] three types of sequential simulation model designs are identified: Event-advance, unit-time advance and activity based design. Event-advance and unit-time advance designs model the system using events that occur and change its state. They differ on how the global simulation time is advanced. In event-advance design the time is advanced (and the state of the system is changed) each time an event occurs. In unit-time advance design the global simulation time is advanced in fixed increments of time and events occur when their trigger time is met. Activity-based designs model the system as a collection of activities or processes and with conditions determining when the processes start and end.[83]

3.3 Parallel simulation

Simulation of parallel systems is possible using the sequential simulation approaches described in the previous chapter and by updating the global simulation time only after all parallel events have been processed. Parallel simulation on the other hand refers to executing a single simulation program in parallel. Aim is to improve the execution speed of the simulation. Parallel discrete event simulation (PDES) has a long research tradition dating back to late 1970's.[25, 16]

Basic approaches for PDES can be categorized into *functional decomposition*, *replicated trials*, *time-stepped / synchronous approach* and the *asynchronous approach.*[40]

In the functional decomposition approach the tasks that the simulation program does are split into parallelizable entities, e.g. random number generation, event list processing and monitoring are performed in separate threads. This approach suffers from scaling issues as the functions to be parallelised are limited.[40]

In the replicated trials approach the simulator is executed as independent instances using multiple threads. This approach can be used for example to reduce variability of results or to explore a wider set of parameters. This approach has the potential to scale well with increasing core count as the instances are independent of each other. But on the other hand replicating the simulator instances easily leads to excessive use of memory which can form a bottleneck for the scalability.[40]

In the synchronous designs the entire design is controlled by a global simulation time and the simulation is proceeded and updated in lock-stepped intervals. Global time can only be advanced when all parallel processes have finished, which requires frequent and and centralized communication. This approach suffers also from the fact that events generally occur at irregular intervals. Maximization of the number of events for each interval requires assumptions in the timing model, which can easily lead to poor performance.[40]

In the asynchronous approach the simulation design is partitioned into disjoint processes (logical processes) that may advance asynchronously. Each logical process maintains a local pending event queue and carries out simulation.[40]

Specification and partitioning of the simulation models in a way to exploit parallelism and properly executing them on parallel hardware is not a straight forward task. The fundamental problem of parallel execution is to decide whether event A can be executed in parallel with event B, when we cannot know for certain if A will affect B without executing it first. (A can affect B through a complex sequence of events.) On a larger scale this means that each component of a simulator will produce state updates that are possibly relevant for other components. These updates need to be distributed and their ordering in time must be preserved.The asynchronous PDES approach requires that the local causality constraint has to hold. This means that the parallel execution must yield the same results as sequential execution of the simulation program.[40]

Processing of PDES events is generally of low computational complexity, as each event only updates state variables and possibly schedules new events. PDES is difficult to parallelize because of the fine grained nature of the application and the frequent communication and the complex dynamic dependencies involved. On shared memory multi-core systems PDES simulators easily generate pressure on the memory subsystem.[111]

Discrete event simulation parallelization strategies can be categorized into conservative and optimistic approaches. In conservative approaches the causality errors are not tolerated. This is achieved by using different strategies to determine when it is safe to process an event. The optimistic approaches use detection and rollback mechanisms. An optimistic simulation is proceeded in parallel and upon a causality error the state is recovered by re-execution. Conservative methods can easily end in a deadlock situation and require to use different lookahead mechanisms to avoid them. Their performance depends on the degree to which lookahead can be used an on the messaging overhead involved in the deadlock avoidance and recovery. Optimistic methods risk on spending a lot of time in recovering and re-executing events sequentially. They also require to save a history of the model states to enable rollback, which on large models can greatly increase the underlying resource usage.[40]

The type of the modeled system greatly influences how much events interact and change each others states, which affects whether the conservative or optimistic approach is better suited. Similarly application specific knowledge can be used in building a simulation program so that parallelism would be maximized.[40]

Parallel simulation is generally considered to be a very hard problem. The obstacles that are faced are common to all parallel programming and execution. Research done within the field of parallel simulation can have impact and direct applications on general parallel computation. For example, the time stamped events have an analogy in the indexes of an iteration of a loop.[40]

3.4 Simulators

Simulator is a program that executes a simulation model. Typically a simulator uses three data structures: state variables, an event list and a global simulation clock. State variables are used to describe the state of the system. The event list is used to store all pending events that have not yet occurred. The global simulation clock stores how far the simulation has proceeded.[83]

A discrete event simulators can be summarized using the following pseudocode, which abstracts the main loop of the program.

```
while simulation is in progress
  remove smallest time stamped event from event list
  process the event:
     (a) update state variables
     (b) schedule new events
```

General purpose simulators do not exist, instead a simulator is always built with some goals in mind. Some metrics to evaluate different simulators include performance, flexibility, and detail. Performance determines the amount of workload the model can process given the computing resources available for simulation. Flexibility indicates how the models are constructed and how easy it is to modify and vary different designs. Detail defines the level of abstraction used in the models. The choice of modeling detail and abstraction level mostly dictates both the execution time of the simulation and also the time it takes to model a system.[9]

There exists a wide variety of different simulator tools and frameworks. Broadly simulators can be classified into cycle accurate, functional and high level simulators. The most accurate level is used when microarchitectural design decisions are sought and when hardware designs are evaluated and verified. At that level the models are generally described using hardware description languages, such as VHDL or Verilog, and the simulations are executed cycle accurately.[55] Cycle-accurate simulators give precise numbers of the hardware performance, but they are too slow to execute any larger application. Typically a cycle-accurate simulator runs 1000-100000 times slower than what the native execution would be.[72] Besides being slow, they are tedious to update and do not necessarily produce significantly more accurate results than e.g. functional simulators. Problems related to cycle-accurate simulators have been investigated in [105].

Functional simulators model the hardware on a more abstract level than cycle-accurate simulators. They implement what programmers see of the system architecture and generally allow execution of real program binaries. Functional simulators are generally used to investigate how different application behave on a certain hardware. Examples of functional simulators include GPGPU-Sim and Barra, which are simulators of NVIDIA GPUs, both capable of executing CUDA code.[67] Authors in [103] use GPGPU-Sim with gem5 [41] to simulate a CPU-GPU SoC. Several other functional simulators aimed for simulating heterogeneous systems are surveyed in [67].

Cycle-accurate simulators are vital for microarchitecture exploration and for making detailed design decisions, similarly functional simulators are a fundamental tool for software architects. But for early high-level exploration both are too slow and can actually produce misleading results. For high-level simulation the abstraction level of the simulated hardware can be raised. Also the simulated application does not need to be represented as an instruction stream, instead some higher level of abstraction can be adapted.[90]

High-level abstraction is generally used in early design space exploration of future systems. In the early design space exploration it is not cost effective to model a system on a too detailed level, instead it is desirable to be able to get coarse results from several different designs to then further direct the exploration. High-level simulators can be used to examine how a model behaves using different parameter sets for the hardware and software configurations.[72]

Heterogeneous MPSoCs can be modeled and simulated at different levels of details. SystemC [82] and SpecC [34] are design languages and methodologies which are intended for specification and design of SOCs or embedded systems including software and hardware. They can use fixed platforms, integrate systems from different IPs, or synthesize the system blocks from programming or hardware description languages. SystemC and SpecC support system modeling and simulation at different levels of abstraction, from pure functional un-timed models to cycle-accurate register transfer level models. SystemC is a C++ class library based language, while SpecC is a super-set extending ANSI-C.[19] While SystemC and SpecC allow high-level modeling of computer systems, their main use is the iterative development and model refinement into register transfer level models, ready to be deployed on silicon.

Simulator toolsets suitable for modeling and simulating parallel target architectures are numerous. Some examples include: Proteus [15], RSIM [50], SimOS [91] and SimpleScalar [9]. The previous simulators are sequential discrete event simulators, whereas the following simulator examples employ parallelism: BigSim [112], COTSon [6], GEMS [68], Graphite Multicore Simulator [72], ISE [42], SlackSim [28], SimFlex [108], SimNow [5], Sniper [21], TaskSim [90], Wisconsin Wind Tunnel (WWT) [88] and Wisconsin Wind Tunnel II (WWT II) [76].

With simulators there is always a tradeoff between accuracy and speed. Some of the simulators support execution on multiple levels of abstraction. For example TaskSim uses four levels of abstraction. Applications can be modeled on the highest level of abstraction as computation and MPI calls, on the second highest abstraction level as computation and required synchronizations, on the second lowest level of abstraction using memory access list and on the lowest abstraction level as instruction list.[90]

Two recent and still actively developed simulators are the Graphite Multicore Simulator by Carbon research group, MIT and Sniper simulator from Ghent University. Graphite Multicore Simulator is a distributed, parallel simulator for design-space exploration of large-scale multicores and application research.[72] It has been recently updated with a support for runtime power modeling.[61] Sniper is a multicore simulator based on the Graphite simulation infrastructure and the interval core model that allows exploring different homogeneous and heterogeneous multi-core architectures. Sniper also supports power modeling of multi-core architectures.[21]

3.5 Resource networks

Modeling of computer systems makes extensively use of different networks. Networks are useful because with them it is easy to bring structures into the model. Networks used in modeling computer systems include e.g. Petri nets [84], Queuing networks [30] and Markov Chains [14].

Aim of this thesis is to prototype a way to model parallel systems and allow monitoring the simulated execution at different levels of detail. For example Queuing Networks do not directly support detailed monitoring. On the other hand they can be solvel analytically to obtain some key metrics of the behavior.[14] Detailed monitoring of simulation execution can reveal model behavior that would not be deducible from some general metrics of the execution. For example monitoring the queue length of tasks waiting a hardware resource can have a average mean of N, but when the queue lengths are plotted with regard to time a periodic behavior might be observed. This observed periodicity can be more crucial for understanding the model behavior than a set of metric values.

Resource network [30] is a modeling concept suitable for high-level modeling and simulation of parallel computer systems which supports monitoring at different levels of detail. Resource networks concept is based on describing the resources and the resource usage of a computer system using a resource provision network and a resource utilization network. Resource networks use the modeling abstraction of active and passive resources.[30]

The resource provision networks are directed graphs where nodes represent different resources of the system and edges represent their interconnections. Resource usage network is also a directed graph where edges represent possible paths for events and nodes represent resource requests. Both the resource description network and the resource usage network can contain loops. Workload model is a directed acyclic graph which generates events that enter and use the resource usage network.

Resource networks support fork-join queue [58] methodology which allows modeling of simultaneous resource requests. With dynamic scheduling the resource network models can employ load balancing schemes. The resource network methodology can be modeled and simulated using discrete event simulation.

4 PSE – Performance Simulation Environment

This chapter presents PSE – Performance Simulation Environment. PSE is based on an in-house toolset QNS (Queuing Network Simulation). First the motivation to use QNS as base and update it to support modeling and simulation of parallel systems instead of a large variety of other possible simulation tools is done. Next an overview of the PSE toolset and the modeling workflow is done, after which the basic building blocks of PSE models are described. After that a look at the monitoring system of PSE is done. Finally the PSE runtime system RNS – Resource Network Simulator is presented.

4.1 Queuing network simulation

The PSE simulator toolset is based on an older simulation toolset QNS (Queuing network simulator) [48]. The QNS toolset was upgraded during this thesis into PSE. PSE adds new functionality to QNS that is needed to support the simulation of parallel systems.

There exists many alternative modeling and simulation tools that could have been used to model and analyze the performance of parallel systems. These have been briefly surveyed in the previous chapter. QNS was chosen because it is a moderately sized simulator software (QNS consists of about 30k lines of code, while e.g. Graphite consists of 140k lines of code [95]). Besides of the moderate size of QNS, it being an in-house tool supported the choice.

4.2 Toolset overview

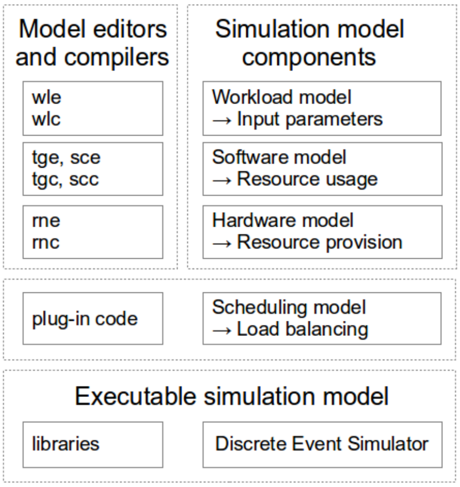

Model editors and compilers
wle
wlc
tge, sce
tgc, scc
rne
rnc
Simulation model components
Workload model → Input parameters
Software model → Resource usage
Hardware model → Resource provision
plug-in code
Scheduling model → Load balancing
Executable simulation model
libraries
Discrete Event Simulator

Figure 4: PSE contains a set of model editor and compiler tools that are used to create individual simulation model components. With simulator engine libraries and optional plug-in scheduling and timing code these files are turned into an executable discrete event simulation program.

PSE (Performance Simulation Environment) is a heterogeneous modeling and simulation environment. It is an extendable environment where various components can be changed or added. The PSE toolset is integrated by using a resource reservation based mechanism as the modeling basis. PSE runs on commodity Linux hardware.

Evaluation of PSE models is based on simulation. The default simulator in PSE is a discrete event simulator that has been tuned for simulating parallel processing systems. The simulation monitor is a central part of the simulator as it produces the simulation output for post-processing tools.

Figure 5 presents a high-level view of the PSE tools. The basic set-up of PSE consists of several integrated tools. Model editors *wle, tge, sce* and *rne* are graphical user interface tools which are used to describe different components of the simulation model. Compiler tools *wlc, tgc, scc* and *rnc* turn the model files into C code. Optional plug-in C code can be used to provide additional scheduling and timing details. The model files with PSE libraries are compiled into an executable discrete event simulator program.

The main components of a system model are a resource network for providing the resources and a usage network for utilizing the resources. In addition to these, PSE uses a workload model and optional scheduling and timing models.

An essential part of PSE models are the monitoring probes. Probes can be attached to all model components on all model layers. The probes produce traces of the simulation execution, which can then be further analyzed using different post-processing tools.

In general, resource provision models are used to describe the underlying hardware and resource usage models are used to describe the software. But in practice the modeling of HW/SW co-scheduled systems requires using both the resource provision and the resource usage models to capture the functionality of the mixed HW/SW components.

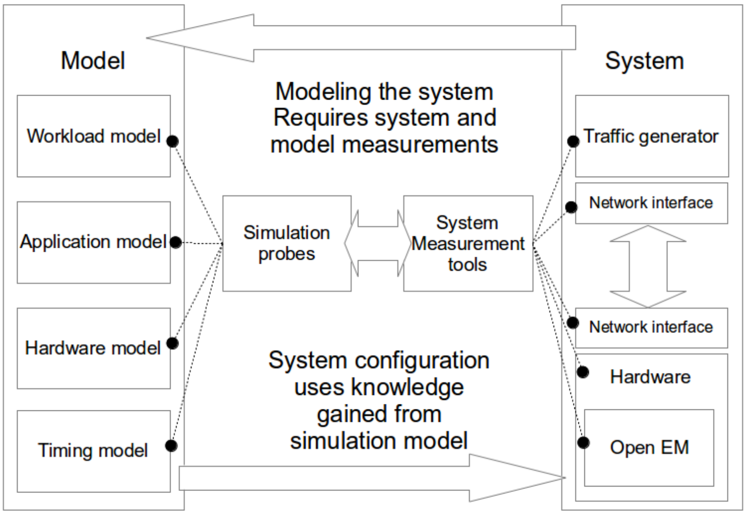


Figure 5: System model measurement loop.

4.3 Modeling workflow

In a larger context PSE fits the iterative modeling-measurement-configuration loop described in Figure 5. System and model measurements are needed to model the system. The knowledge gained from experiments with the model are used to configure system parameters in order to attain better system performance. The modeling and system configuration can be done on different levels and at different system design phases.

The example presented in Figure 5 refers to the configuration of a packet processing system based on the OpenEM programming framework. The application and runtime configuration is an iterative process where PSE can be used to explore the large design space of different runtime configurations and application partitionings.

4.3.1 Editor tools

PSE models are created using model editor tools. The PSE editor tools are created using Tcl/Tk[98]. Figure 6 represents the graphical user interface of the workload model editor, Figure 8 represents the graphical user interface of the task graph editor and Figure 9 represents the graphical user interface of the resource network editor. All editor tools have drop down menus from which model files can be loaded and saved or new design can be started. The dropdown menu is also used to enable a grid onto the drawing canvas and to export model files in postscript format. The editor tools have a toolbar (on the left) that contains the supported drawing commands and manipulation tools. Models are constructed on the drawing canvas using these tools.

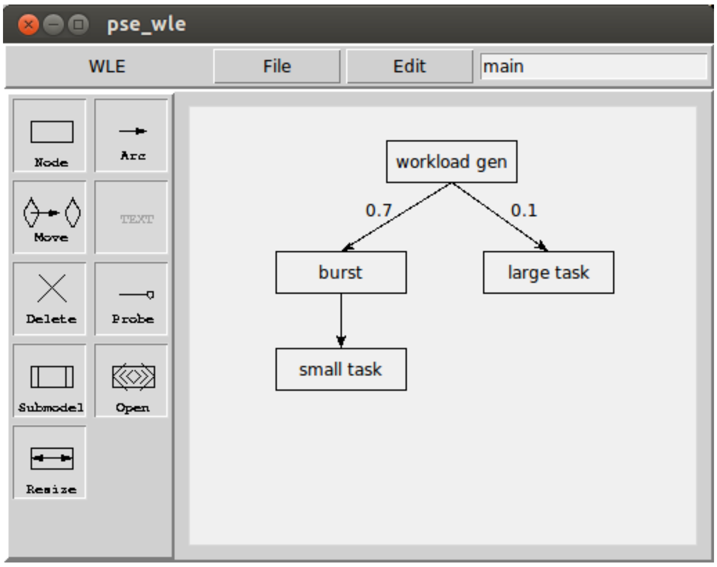
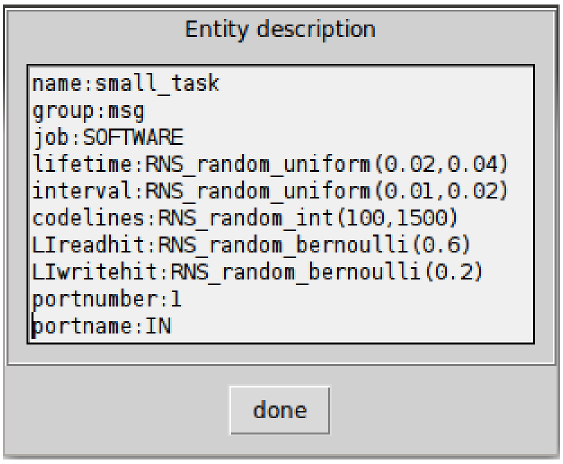


Figure 6: The graphical user interface of the workload model editor. The various tools and symbols are used to create and modify PSE workload model files.

The workload model is a directed graph, where each node represents an action to be taken and arcs represent invocations for new actions. The arcs can be equipped with a probability that determines whether the child node is activated upon activation of the parent or not. With the *text* tool the contents of nodes and arcs can be edited.



```
name:small_task
group:msg
job:SOFTWARE
lifetime:RNS_random_uniform(0.02,0.04)
interval:RNS_random_uniform(0.01,0.02)
codelines:RNS_random_int(100,1500)
LIreadhit:RNS_random_bernoulli(0.6)
LIwritehit:RNS_random_bernoulli(0.2)
portnumber:1
portname:IN
```

Figure 7: Example contents of a node in a workload model.

Figure 7 shows the contents of an example node. The key parameters are the *job, lifetime and interval.* Parameter *job* defines the application model that the generated event will enter. Parameter *lifetime* determines the duration that the node will be kept alive after invocation. *Lifetime* parameter can also be omitted which means that the node is invoked just once. Parameter *interval* determines how often the node is activated (during its lifetime). Other parameter *codelines, LIreadhit and LIwritehit* are user specified variables that can be referenced from the resource provision and resource usage models. The parameters *portnumber and portname* determine the entry point in the application model determined by the

Task graphs are used to model the resource usage of a system. Figure 8 represents the graphical user interface of the task graph editor. Task graphs are directed graphs consisting of arcs connecting different types of nodes. Arcs define the paths an event can take while in the model. With the *text* tool the contents of nodes can be edited.

The resource provision network consists of different types of nodes connected with arcs. Figure 9 represents the graphical user interface of the resource network editor. The nodes of the resource network can be edited using the *text* tool.

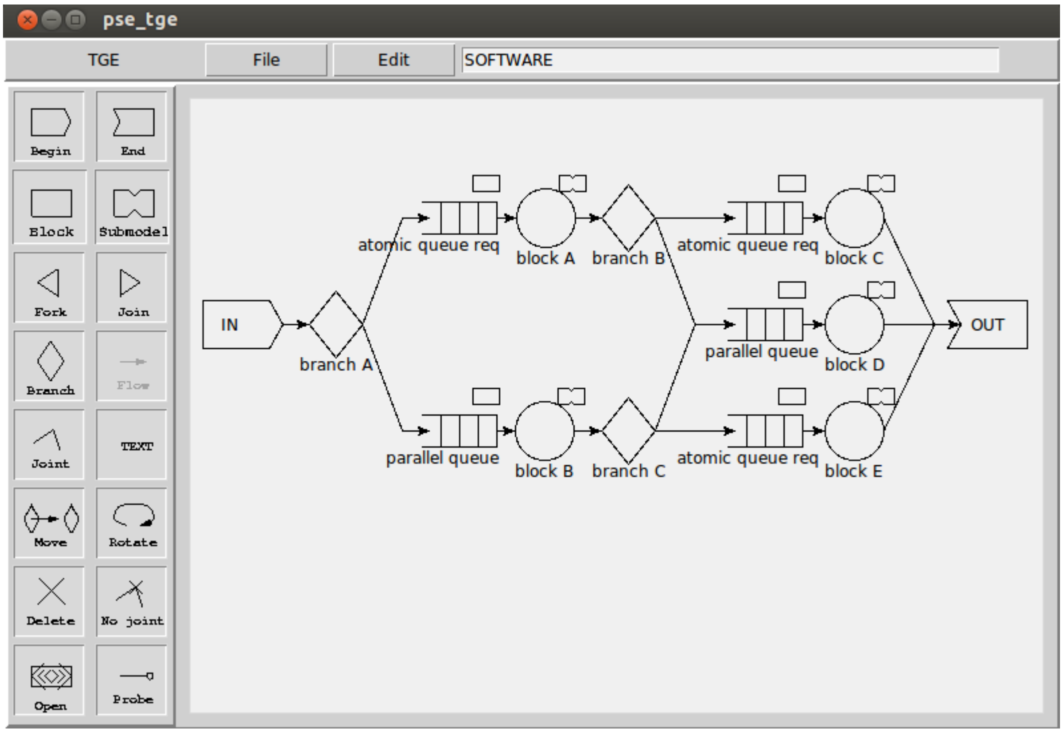


Figure 8: The graphical user interface of the task graph model editor.The various tools and symbols are used to create and modify PSE task graph model files.

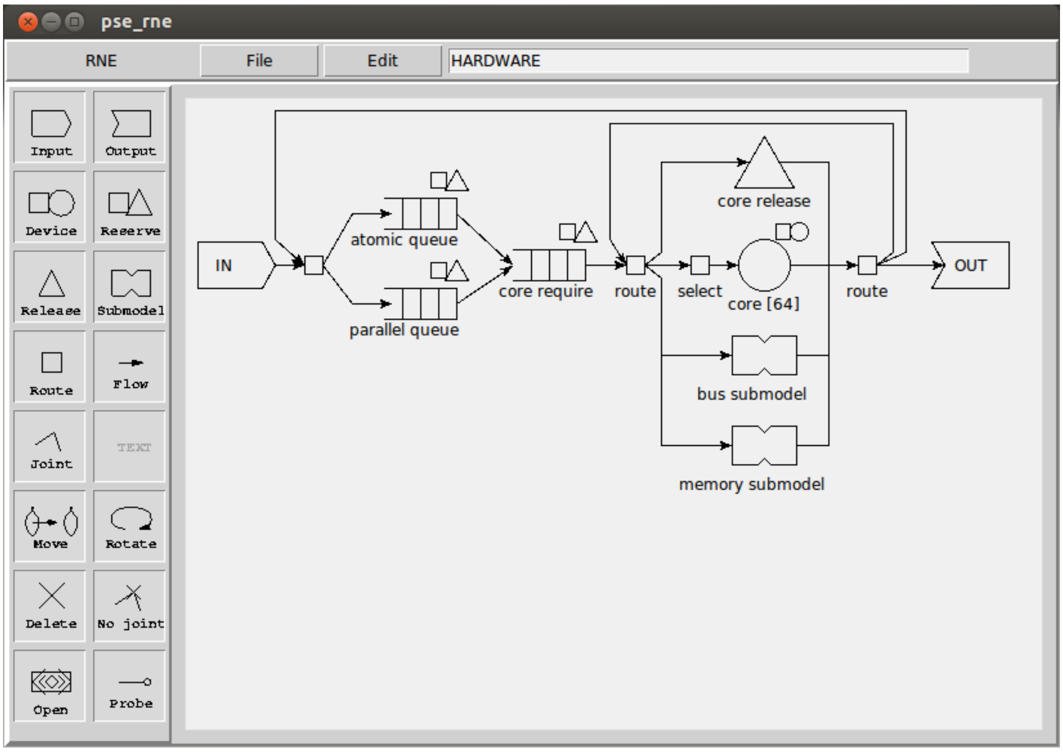


Figure 9: The graphical user interface of the resource network model editor. The various tools and symbols are used to create and modify PSE resource network model files.

4.3.2 Compiler tools

The editor tools of PSE use a textual representation for the models. The model files are compiled into C code using the corresponding compiler tools. In Figure 10 the workflow of creating a complete PSE model is presented. A hardware model created using the *rne (resource network editor)* tool is compiled using the *rnc (resource network compiler)*. The application model is created using the *tge (task graph editor)* tool. The *tgc (task graph compiler)* is used to compile the application model. The *wle (workload editor)* tool is used to model the system workload and the *wlc (workload compiler)* generates code from the model file. Finally with a reference to RNS libraries *gcc* is used to compile all model files into an executable simulation.

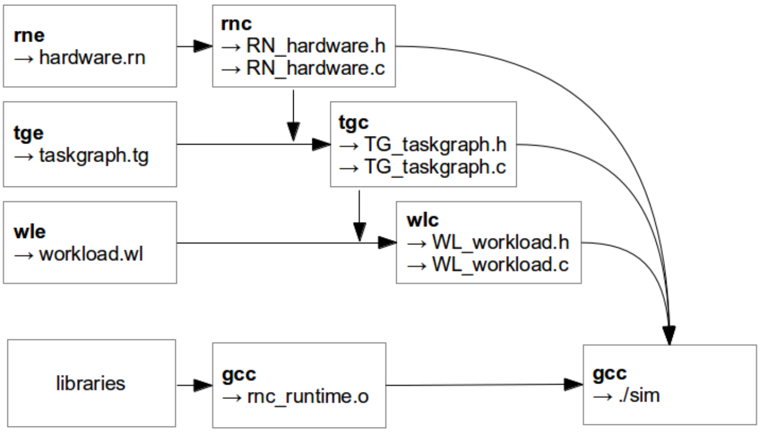


Figure 10: The workflow for creating an executable simulator program begins by describing the model components using the graphical editor tools. The model files are compiled into C code using the PSE compilers. All model files are compiled together with RNS libraries into an executable simulator.

4.4 Basic building blocks

PSE modeling philosophy is based on combining elementary building blocks to model more complex system structures and functionalities. Models formed using the simple building blocks can be grouped into into submodels for semantical clarity and functional reuse.

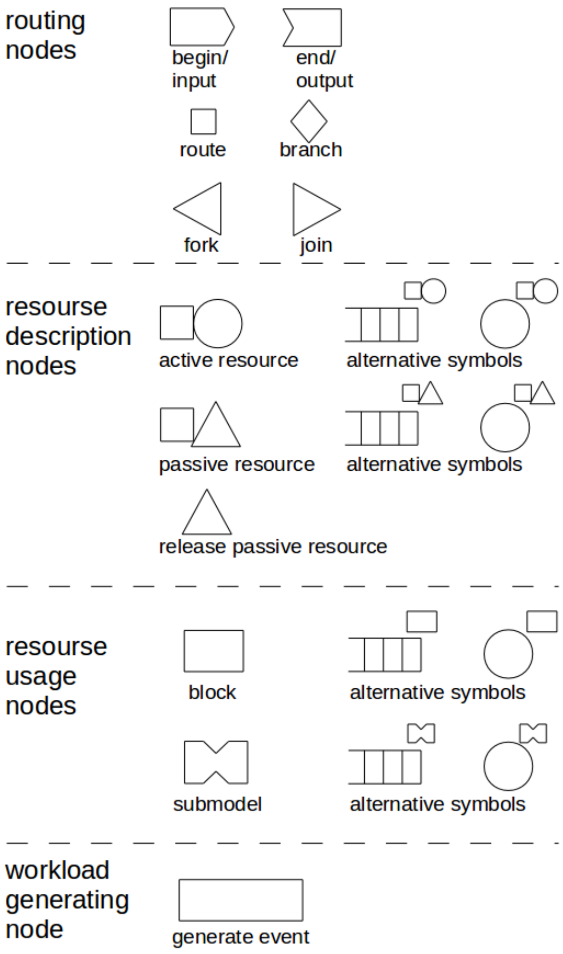


Figure 11: PSE building blocks. Routing nodes define the flow of tasks in the application models. Resource description nodes are used to describe the resources (usually the HW) of a system. Resource usage nodes define the resource requests from the application layer to the hardware layer of the model. Workload generating nodes are used to generate input for the PSE models.

The PSE building blocks can be categorized into routing nodes, resource description nodes and resource usage nodes. The PSE building blocks are presented in the Figure 11. Routing nodes include the *begin* and *end* (*input* and *output* in the RNE) nodes that define the entry and exit points for models. The *Route* and *branch* nodes are used to direct events to target devices or paths using rules defined in the models. The *fork* node allows splitting the flow into two simultaneously taken paths and the *join* gathers both of the flows before it is allowed to proceed. There are two kinds of resource types in PSE, active and passive. *Active resources* have a speed parameter that defines how quickly they are able to fulfill requests. Active resources are requested for service for a certain amount and they return a service delay depending on the amount requested. *Block* is the node that is used to define the type and amount of a resource request. *Passive resources* can be used to provide exclusive access to regions in models. They can be acquired by requesting service from them and they must be released afterwards. Both the active and the passive resources can have *count* and *capacity* parameters defined. These are used to specify the amount of parallel tasks that can be served simultaneously. *Submodel* nodes are used to group different models together to form hierarchical models.

In Figure 11 alternative symbols are shown for the nodes representing active and passive resources, use resource and submodel. These function similarly as the main symbol, but can be used to underline the semantical meaning.

The parameters of the building blocks can be edited with the *text* tool. Clicking on a node opens a dialog window where parameters are written. Example contents of a branch node are presented in Listing 1.

Listing 1: *Example contents of a branch node.*

```

cache hit?
name: LIcachehit
expr: LIreadhit
```

In this example the *LIreadhit* variable defined in the workload model (see e.g. Figure refsmalltask) evaluates to zero or one. The two nodes following the brach node must contain identifiers *tag:0* and *tag:1* to direct the flow based on the value of LIreadhit.

Resources are defined using the attributes presented in in Listing 2.

Listing 2: *Resource description attributes.*

```
name        value type  optional  applies to nodes
----        ----------  --------  ----------------

name        alphanum    no        active and passive resource,
                                  release, submodel
count       integer     yes       active and passive resource,
                                  release, submodel
discipline  alphanum    yes       active and passive resource,
capacity    integer     yes       active and passive resource,
group       alphanum    yes       active and passive resource,
speed       float       yes       active resource
```

Count creates an array of the resources. *Discipline* is used to specify the queueing policy of a resource (default is FCFS, First Come First Served). *Capacity* defines how many tasks can be served concurrently. *Group* is used to form monitoring groups. *Speed* defines the service per time of the resource.

Resource usage requests are defined using the attributes presented in Listing 3.

Listing 3: *Resource usage attributes.*

name	value type	optional	applies to nodes
type	alphanum	no	(see below)
name	alphanum	no	all
port	integer	no	enter
time/amount	float	no	active resource
pc	integer	yes	active and passive resource
group	alphanum	yes	active and passive resource
exits	alphanum	yes	submodel
enters	alphanum	yes	submodel
tag	aplhanum	no	following a branch

The type of resource usage is defined using the *type* attribute. Valid values are: *device* for active resource, *resource* for passive resource, *release* for releasing a passive resource and *submodel, exits or enters* to control movement in and out of submodels. *name* attribute is used to identify the requested resource. *Port* attribute can be used to classify entering events of a model. The *time* and *amount* attributes are used to determine the amount or time requested from an active resource. Attribute *pc* defines the priority of a request. *Group* is used to form monitoring groups. Attributes *exits* and *enters* define the movement inside the submodel hierarchy. *Tag* is used after a branch node to identify the target for an evaluated branch expression.

The basic building blocks are used to model larger entities. Figure 12 represents an example PSE model. The model shows how OpenEM based applications can be modeled.

In the example events are generated by the workload model. The generated events enter the application model where they make resource requests at the nodes of the model. The resource requests are served from the hardware model. Depending on the amount of service requested and the internal state of the hardware model the events are delayed for a specified amount of time.

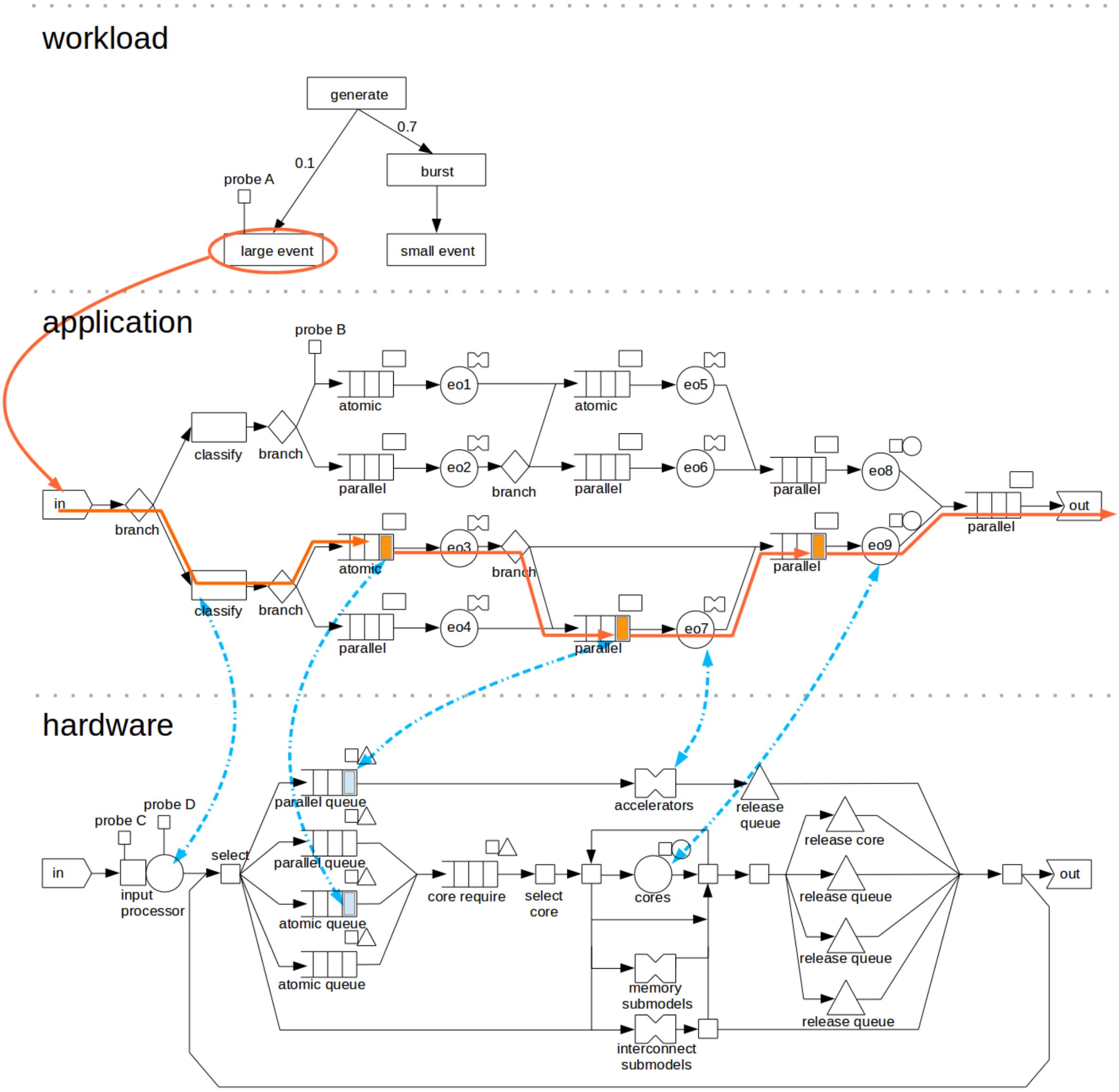


Figure 12: An example of a layered PSE model, which represents how OpenEM[78] based and other similar systems can be modeled using the PSE tools. The workload layer model generates events that enter the application layer model. The events flow in the application model and are placed into queues until access to resources is granted. A resource request is made for example in the *classify* node in the application model, this resource request is for the resource named *input processor* in the hardware model. Similar resource requests are made in the EO submodels. Depending on the state of the hardware layer model, each resource request induces a timing delay to the flow of the events in the application model.

4.5 Monitoring

When simulations are used for performance estimation time is in a key role. Performance simulations use time stamping to keep things ordered with regard to time. Simulators use monitoring techniques to obtain metrics of the simulation behavior. Basic approaches for this are trace-based and on-the-fly approaches. Trace-based simulator produces a trace that is saved for post-processing. An on-the-fly monitored simulator produces metrics of the simulation execution. If only certain metrics of the simulation are under study then collecting a trace file is not necessary.

PSE offers a comprehensive set of simulation monitoring tools. Monitoring is based on probes, which can be attached to the graphically presented models. Probes are used to generate traces or to measure statistics from the simulation execution. In PSE *trace probes* produce trace files that track certain events, whereas *metric probes* compute statistics from the measured target.

In resource description models probes can be attached to resources for measuring utilization or to queues to measure queue length. Trace probes produce a trace file where either every change in the queue length or in the resource utilization is recorded. Metric probes produce statistics of the queue length or the utilization. The statistics produced by metric probes are the mean, standard deviation, mininimum, maximum, sum, and total number of events.

In resource usage models *time probes* can be attached to the edges of the graph. Time probes can be used to produce a trace with a timestamp whenever an event passes the measurement point. The timestamps can be a list of absolute times or times relative to process start. Time probe can also measure the average time of all events relative to process start. In workload models probes can be used to group together measurements from other probes.

Trace probes write tracefiles, whereas metric traces write a single line containing average values. Each probe produces one trace file. If traces are not needed then metric probing should be used as the trace files can easily become very large and the required file operations could saturate the system.

Probes are represented graphically in the PSE models. This is illustrated In Figure 12 where four probes (A,B,C and D) are attached to different parts of the system models. Probe A is attached to the workload model, probe B to an edge of the application model, probe C is attached to the queue of the *input processor* node and probe D measures the utilization of the same node.

4.6 RNS runtime

RNS (Resource Network Simulator) is a discrete event simulator engine designed to execute the PSE models. RNS keeps track of the global time of the simulation, schedules simulation events and manages the monitoring of simulation execution. RNS provides the abstractions and interface used by the model file compilers. Main components of RNS are the event scheduler, the process abstraction with accompanying service routines and the monitoring system.

The RNS scheduler runs in a loop picking the event with the smallest trigger time to execution. Events are generated by the workload model. They can be either system events controlling the whole simulator or workload events which are used to model the workload of the system. System events control for example the total execution time of the simulation and possible resetting of the monitoring metrics. Workload events contain a set of defined parameters and a reference to a resource usage model they will enter. The resource usage model is a process which executes code that uses the RNS interface to make requests for the runtime system. The code executed by the processes is generated by compiling the application models with references to the workload model, hardware models and the runtime libraries.

The main interface of RNS is the service routines. These are used to implement the active and passive resources of the resource networks. The main interface is presented in Listings 4, 5, 6, 7 and 8.

Listing 4: *RNS_demand_device*

```
void RNS_demand_device(RNS_Device *d, double service_amount,
                       char *group, uint64_t pc) {
    ...
    RNS_use_device(d, service_amount/d->speed, group, pc);
    ...
}
```

RNS_demand_device, summarized in Listings 4, translates the demanded service amount from active resource into service time based on the speed of the modeled device. The service time is used as parameter for the RNS_use_device function.

Listing 5: *RNS_use_device*

```
void RNS_use_device(RNS_Device *d, double service_time,
                    char *group, uint64_t pc) {
    ...
    RNS_reserve_resource(d->resource, group, pc);
    RNS_delay_process(d->resource->name, service_time);
    RNS_release_resource(d->resource, group);
    ...
}
```

RNS_use_device presented in Listings 5 wraps the reserve, delay and release functions of RNS together.

Listing 6: *RNS_reserve_resource*

```
void RNS_reserve_resource(RNS_Resource *r, char *group, uint64_t pc) {
    ...
    set_client(r->queue, r->queue_size,
        RNS_current_process, usage_group, pc);
    r->queue_size++;
    ...
```

RNS_reserve_resource presented in Listings 6 adds the currently running process to the resource queue.It also updates the queue length and utilization of the resource.

Listing 7: *RNS_delay_process*

```
void RNS_delay_process(char *name, double seconds) {
    ...
    event.trigger_time = RNS_simulated_time + seconds;
    event.process = RNS_current_process;
    RNS_Heap_insert(event);
    RNS_yield();
    ...
}
```

Listings 7 presents a fragment of the RNS_delay_process function, which is used to delay the given process for a defined amount of time and to generate an event that will be triggered when the requested service has ended.

Listing 8: *RNS_release_resource*

```
void RNS_release_resource(RNS_Resource *r, char *group) {
    ...
     event.trigger_time = RNS_simulated_time;
     event.process = r->queue[new_index].process;
     RNS_Heap_insert(event);
    ...
}
```

RNS_release_resource in Listings 8 selects a process from the resource queue for execution. The process selection is done on basis of the resource queuing discipline. The new process is put to the event list for immediate scheduling.

5 Mechanism for resource network simulation

In this chapter the mechanisms for resource network simulation are presented at different levels of abstraction. First the mechanisms are presented at the modeling level using as examples two basic models of hardware schedulers and a model for memory systems. After that the implementation of the fork-join mechanism is described at the runtime level of the simulator. Finally the mapping of the simulation model to execution hardware is overviewed and problems arising from different parallelization strategies are studied.

5.1 Modeling hardware accelerated scheduling

Scheduling is about assigning tasks to system resources. The scheduler should at the same time maximize the throughput – the total number of tasks completing execution in a given time, and minimize latency – the time between task submission and completion. Usually these goals collide and a compromise must be sought.

Latency increases as the granularity of tasks get smaller. In MPSoCs a central software scheduler is too slow in traversing the data structures containing the tasks and scheduling information. Hardware schedulers can see the system state as whole and thus are well suited to speed up the scheduling. The type of the system dictates which scheduling algorithm and semantics (push or pull scheduling) work best for system load balancing.

PSE can be used to model different types of schedulers and scheduling algorithms. In the following subchapters two hardware schedulers, a push mode and a pull mode scheduler, are modeled using the PSE primitives. A small scale experiment using the push mode scheduler is also presented. After that the internal implementation of a scheduler is presented using a load balancing scheduler as an example. Finally an example of how memory systems can be modeled with PSE is presented.

5.1.1 Pull mode scheduler

In Figure 13 model of a simple HW pull mode scheduler is presented. The scheduler uses two hardware queues and a core lock to provide access to the resources. Once a core lock is acquired the task is able to use the core and the resources of the memory and bus submodels. After the task releases the core and queue locks a new task can be selected for execution (the release part of the HW model is omitted from Figure 13).

In Figure 14 a small part of a low-level task graph model representing the resource usage is shown. The graph uses the resources described in Figure 13. Tasks entering the graph make requests for the atomic queue token which is used to provide sequential access to the cores. In turn some other tasks with no inter-task dependencies could use the parallel queue for better scalability (simultaneous access to all cores).

After having acquired the queue token, the task proceeds to request for a core lock. The core lock queue uses a *highest priority served first* policy, that can give

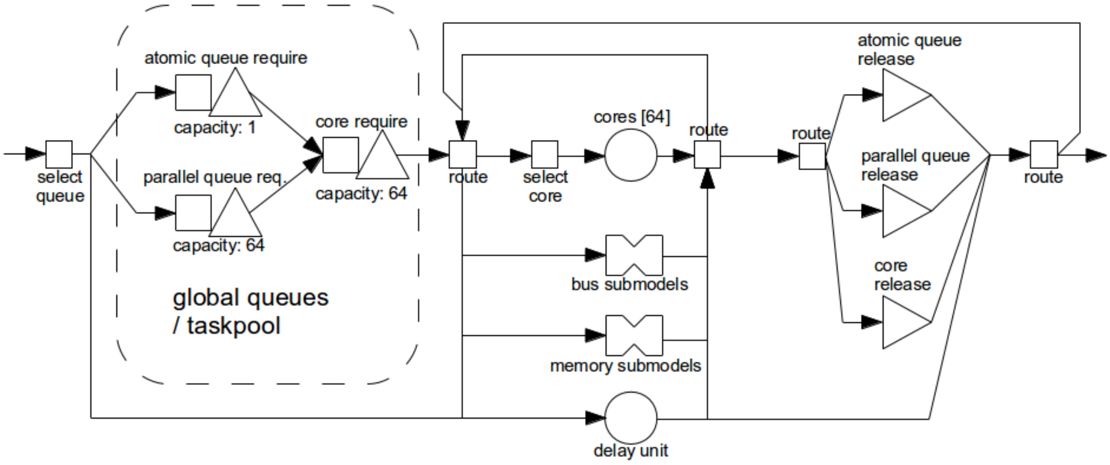


Figure 13: An example of a system providing its resources using pull mode scheduling and two hardware queues for atomic and parallel access to the cores.

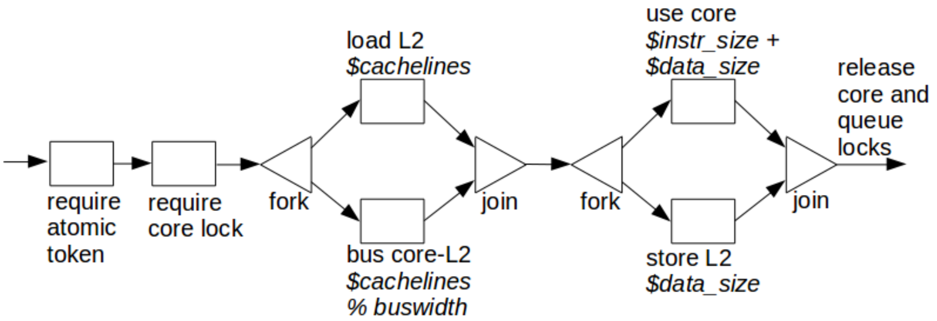

require atomic token
require core lock
fork
load L2 $cachelines
bus core-L2 $cachelines % buswidth
join
fork
use core $instr_size + $data_size
store L2 $data_size
join
release core and queue locks

Figure 14: An example of a low-level task graph describing resource usage of the HW model in figure 13.

priority for e.g. tasks in the atomic queue to prevent their starvation. After acquiring the core lock a task can use the core the bus and memory submodels. The HW layer induces delay to the task proportional to the amount of service requested. For example, in the Figure 14 the *use core* request utilizes the core for the amount specified by the value of the expression: *$instr_size + $data_size*. These variables are defined for each individual task by the workload model.

5.1.2 Push mode scheduler

Figure 15 represents a model for a push mode scheduler. Push mode scheduler distributes tasks upon their arrival into local queues close to cores. The scheduling is done in the *select core* node, where custom code can be inserted to model the scheduling algorithm. In a push mode scheduler the scheduling algorithm is in central role for balancing load across the cores.

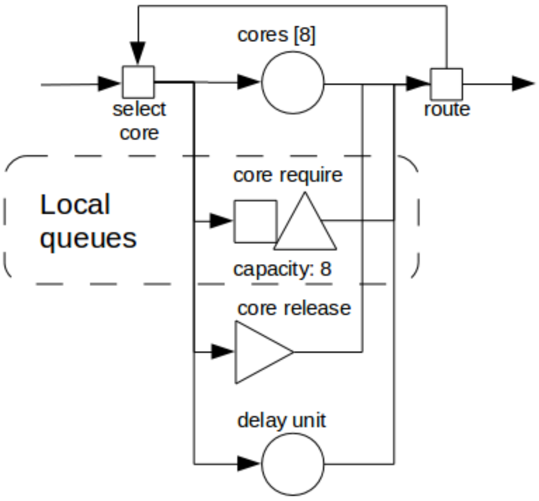
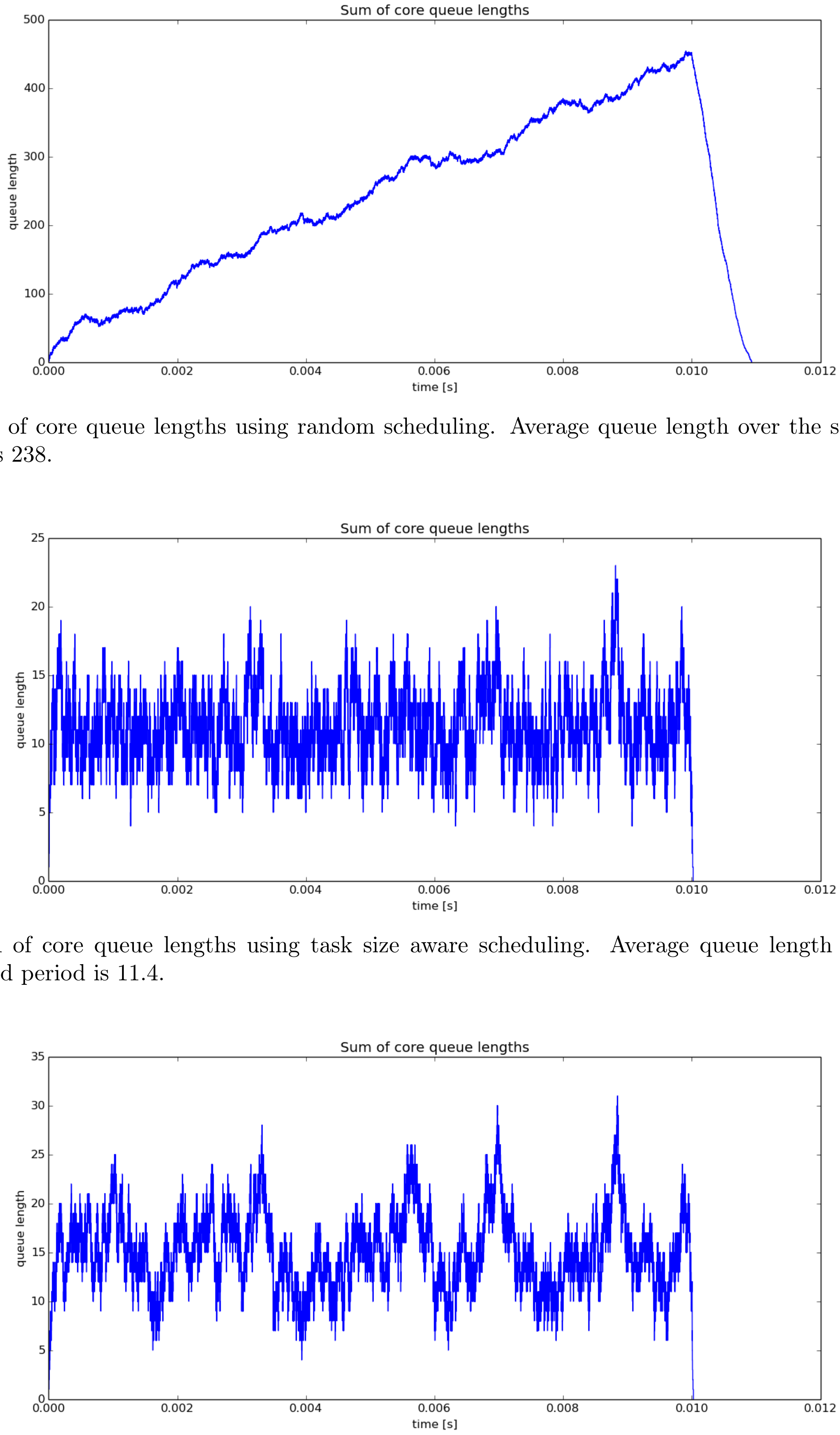


Figure 15: An example of a system providing access to its cores using push mode scheduling. Queues are local to each core and the scheduling decision is done at the select core node, after which the task is pushed to the designated local queue. An additional *delay unit* node is used to provide a means to model the HW delays of scheduling decisions, queue accesses, etc.

PSE can be used to estimate the performance of different scheduling algorithms. In Figure 5.1.2 results of a small scale experiment are presented. In the experiment a push mode scheduler is used to balance load using three different scheduling algorithms. These algorithms are a random scheduler, a single stepping scheduler and a task size aware scheduler. The random scheduler assigns tasks to cores randomly, the load balancing scheduler assigns tasks evenly among cores according to their size and the single stepping scheduler assigns tasks sequentially always to the next core in turn.

Overhead related to the actual implementation of the scheduler was not modeled. The timing overhead could be modeled by including an additional service delay of desired amount in the resource request model.



(a) Sum of core queue lengths using random scheduling. Average queue length over the simulated period is 238.

(b) Sum of core queue lengths using task size aware scheduling. Average queue length over the simulated period is 11.4.

(c) Sum of core queue lengths using stepping scheduling. Average queue length over the simulated period is 15.3.

Figure 16: Example results from a push scheduler system using different scheduling algorithms. The system has one push mode scheduler assigning tasks for an eight core system. The computation time of one task depends directly on the size of the task. The first scheduler (a) assigns tasks to cores randomly, the second (b) balances tasks evenly amoung cores according to their size and (c) assigns tasks incrementally to the next core in turn.

5.1.3 Dynamic scheduling

Dynamic scheduling in PSE means routing events according to the internal state of the models. In practice this requires inserting custom code into the code generated by the PSE compilers.

In the models the basic routing of events is static as the flow of events is defined at event creation by the workload parameters. Dynamic routing, or scheduling, can be implemented by plugging custom code into the routing nodes of the resource provision model, or into the branch nodes of the resource usage models.

The workflow requires compilation of model files first into C code, then manual editing of desired parts in the files, and finally compilation into a final simulation program.

Manual editing of the generated files could target for example the code of the route node. The *RN_index_selection_HARDWARE* function in the hardware model code assigns tasks entering the route node to a target node specified with a node index. Listing 9 shows a fragment of code of a scheduler that keeps track of assigned task sizes per CPU. The scheduler balances load by assigning work always to the CPU that has received least work so far (determined by the size of the task).

Listing 9: *Example code for the load balancing scheduler presented in Figure 15. The scheduler maintains a table with assigned task sizes and always pushed a new task to the core with least received load so far. The plugin code has been emphasized using red text color.*

```
int cpu_loads[8] = {0};   // table used to keep track of CPU loads
int min_ind = 0;             // index the CPU with min load
int min_load = INT_MAX; // minimum load value

void RN_index_selection_HARDWARE1(RN_Process *p,
                                  char *port_name,
                                  int port_number) {
    int i; /* selection to CPU */

    // find the CPU with minimum load
    for (i = 0; i <= 7; i++) {
        if (cpu_loads[i] <= min_load) {
            min_ind = i;
        }
    }
    min_load = cpu_loads[min_ind];

    // assign work to the CPU with min load and update CPU loads table
    p->next.state.node_index = min_ind;
    p->next.port_name = 0;
    p->next.port_number = 0;
    cpu_loads[min_ind] = cpu_loads[min_ind] + (int)RN_value(p, "size");
}
```

After manual editing of the model code the model files need to be recompiled to an executable simulator program.

5.1.4 Modeling memory

Maximizing the utilization of a MPSoC is in general about hiding the latencies related to the usage of the memory systems by doing something else while waiting the memory.[55] As the memory system is crucial to the performance of a MPSoC it is in a key position when modeling the systems.

In PSE one possible approach to model memory systems is to divide the memory into reservable blocks and induce a memory access delay proportional to the internal state of the memory model. Figure 17 represents the general idea of modeling memories using reservable blocks. In the example of Figure 17 *core 4* makes a memory load request and *core 3* and *core 2* are being served at the same time. The returned delay for core 4 can be modeled to take longer than it would have taken were there less simultaneous accesses.

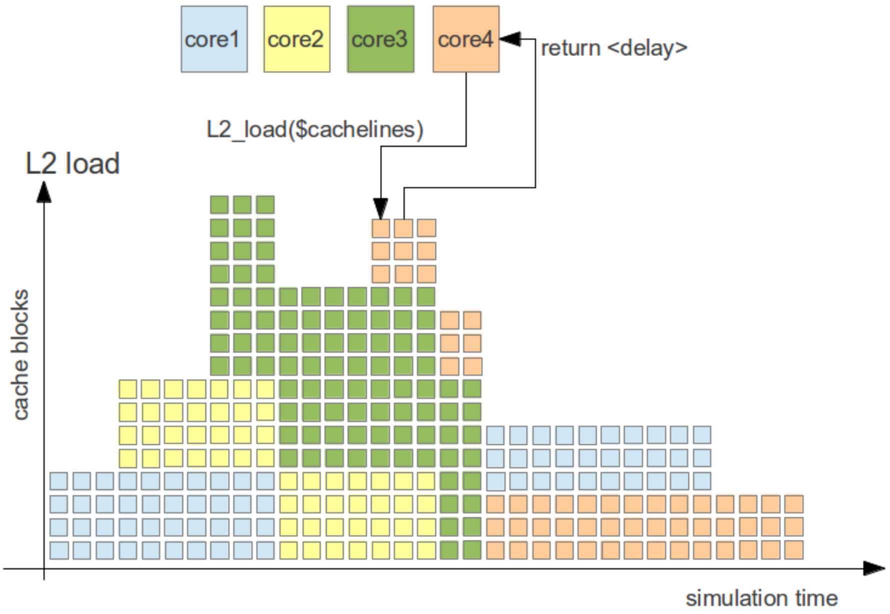


Figure 17: In PSE the memory system can be modeled by dividing the memory components into reservable blocks. Depending on the amount of total reserved blocks at a certain point in time, the service time of the memory access can be varied.

An example implementation for the memory model of Figure 17 is presented in Figure 18, Figure 19, Listing 10 and Listing 11.

Figure 18 presents the hardware model for one type of reservable memory blocks (here cache blocks). The size of the memory is defined using the *capacity* parameter of passive resources. A delay unit node is used in the model to induce a delay to a cache access.

Figure 19 presents the application model for memory access. An event loops to the *reserve blocks* node until it has acquired all required blocks, then it enters the *delay access* node and makes a resource request to the delay unit of the hardware

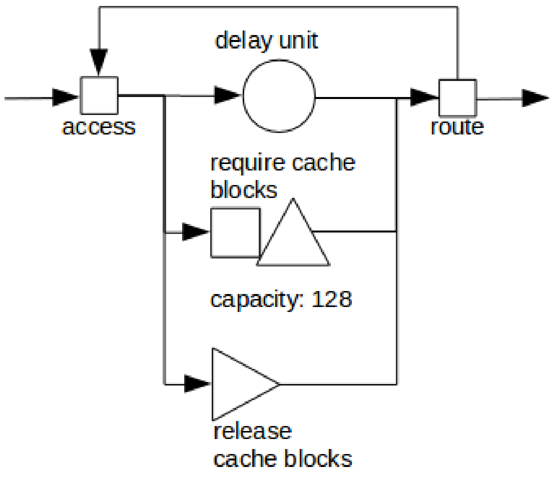


Figure 18: An example of a hardware model for cache memory. Memory is reserved and released in blocks and a delay unit is used to induce a delay for the memory access.

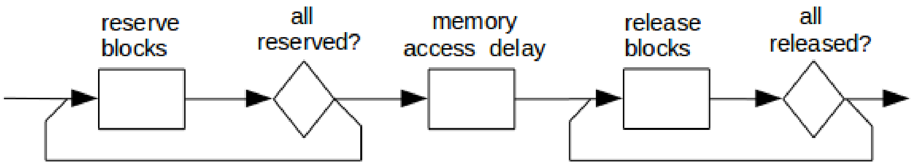


Figure 19: An example of a application layer model for memory accesses. Amount of cache blocks to be reserved is defined in the workload parameters. After the task has acquired all cache blocks it needs it accesses the *delay access* node and makes a resource request. Depending on the internal state of the memory model a delay is applied to the task.

model.

Listing 10 shows an example of how the required workload parameters can be defined in the workload model. The parameter *reserved_ cacheblocks* is used to dynamically store the amount of cache blocks an event holds during the execution of the simulation.

Listing 10: *Example of required workload parameters that are used to determine the number of required cache blocks and the amount of acquired cache blocks.*

```
...
required_cacheblocks:10 + RNS_random_int(0,8)
reserved_cacheblocks:0
```

Listing 11 presents a fragment of the code generated from the hardware model presented in Figure 19.

Listing 11: *Example implementation for reserving a defined amount of memory blocks. The original code is generated from the model presented in Figure 19 and the implementation of dynamic memory block reservation and requested delay is achieved by using plugin code. The plugin code is highlighted using red text color. This code fragment presents the implementation of the three first nodes of model in Figure 19 labeled as: reserve blocks, all reserved? and delay access.*

```
#define N_CACHEBLOCKS 128
...
int free_cacheblocks = N_CACHEBLOCKS;
void SC1_SOFTWARE() {
 ...
 // 'reserve blocks' node
 SCL32:
  RN_resource_request_HARDWARE(p, "cacheblock");
  RNS_reserve_resource(((RN_model_HARDWARE*)p->fp->node)->
          cacheblock[p->state.node_index], 0, 0);
  free_cacheblocks–;
  RN_add_binding(p, "reserved_cacheblocks", RN_value(p, "reserved_cacheblocks") + 1);
  goto SCL34;

 // 'all reserved?' node
 SCL34:
  if (RN_value(p,"reserved_cacheblocks") < RN_value(p, "required_cacheblocks")) {
    goto SCL32;
  else {
    goto SCL31;
  }
  fprintf(stderr, "No␣branch␣matched\n");
  exit(1);

 // 'delay access node'
 SCL31:
  RN_resource_request_HARDWARE(p, "delay_unit");
  RNS_demand_device(((RN_model_HARDWARE*)p->fp->node)->
          delay_unit[p->state.node_index],
          RN_value(p, "size")*(calc_delay(free_cacheblocks)), 0, 0);
  goto SCL33;
 ...
}
float calc_delay(int free_cacheblocks) {
  return ((N_CACHEBLOCKS - free_cacheblocks )/N_CACHEBLOCKS) + 1.0;
}
```

The statement *#define N_CACHEBLOCKS 128* in Listing 11 defines the size of the memory model, and the variable *free_cacheblocks* is used to store the number of currently free blocks. The resource network function *RN_value* is used to return the value of a defined workload parameter and *RN_add_binding* is used to write a new value to a workload parameter. With these two functions events can carry information of how many cache blocks they need and how many they have acquired. In the *delay access* node the internal state of the memory affects the amount of service requested from the delay unit. In this example the implementation of how much the internal state of the memory affects the delay is specified by multiplying the requested service amount *size* by the value returned by the user defined function *calc_ delay*. *calc_ delay* can be used to estimate the memory behavior at a desired level of accuracy. In this example *calc_ delay* is kept simple for demonstration purposes and it just multiplies the requested amount with a scaling factor that varies between 1.0 and 2.0 depending on the state of the memory.

5.1.5 Fork and join

Fork and join allows dynamic creation of processes, which can be used for example to make simultaneous request for different resources. This can be used e.g. to model situations where a core is kept busy until a memory access has been fulfilled.

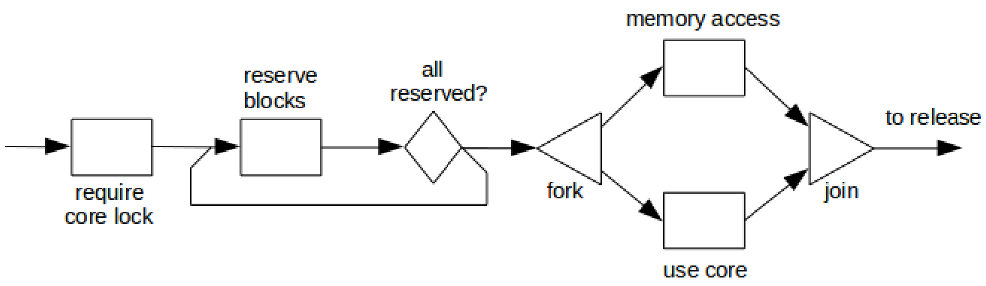


Figure 20: An example usage of the fork-join method. Both the core lock and the memory blocks are held as long as both the memory access and use core requests have been fulfilled.

The fork-join queue is considered a typical model for parallel processing systems. The fork and join of RNS follow the semantics of fork-join queue presented in [58]. In PSE fork and join is defined so that on fork an event is split into two execution paths by creating a new process and the related new event. Both the original and the forked process take their own execution paths in the resource usage network. Later the paths are joined so that the neither of the events is allowed to proceed before both the events have reached the join node.

The RNS implementation of fork and join uses a waiting list where waiting processes are placed. A process is put to the waiting list if it arrives to the join node before the other forked process. As the later arriving process reaches the join node it uses signaling to retrieve the first process from the waiting list.

Listing 12 is a fragment from code generated by the *tgc* compiler from an application model made by *tge*. The example shows the code and runtime calls that a *fork* node translates into.

Listing 12: *The RNS runtime requests of a fork node.*

```
cp = RN_copy_process(p);
RN_push_id(p, RNS_next_process_id());
RN_push_id(cp, RNS_current_process_id());
RNS_push_int((int)(intptr_t)cp);
RNS_process("SC2_SOFTWARE", SC2_SOFTWARE, NULL);
```

First *fork* allocates memory for the forked process using RN_copy_process. Then it pushes identifiers of the two processes (the parent and the fork) into each others id lists. Using RNS_push_int arguments are passed to the forked process. Call to RNS_process launches execution of the forked process. Both the forked and the parent process can make an arbitrary number of resource request during the execution, but their execution has to meet at a join node.

Listing 13: *The RNS runtime requests of a join node.*

```
if (RNS_is_waiting(RN_peek_id(p))) {
  RNS_signal(RN_peek_id(p));
} else {
  RNS_wait();
}
```

Join (presented in Listing 13) checks RNS_is_waiting list if the other process is already waiting. If yes, it signals the process, otherwise it calls RNS_wait.

Listing 14: *Runtime requests of RNS_wait call.*

```
void RNS_wait() {
    RNS_waiting[RNS_waiting_count++] = RNS_current_process;
    RNS_yield();
}
```

Upon a call to RNS_wait (presented in Listing 14) the caller is put to the waiting list (RNS_waiting) and it's execution is yielded. RNS_yield calls RNS_suspend which returns control to the RNS scheduler allowing other events to be scheduled.

Once the later arriving process reaches the join node it uses RNS_signal to retrieve the waiting process from the waiting list.

Listing 15: *The RNS runtime requests of signaling.*

```
void RNS_signal(uint64_t process_id) {
    ...
    for (i = 0; i < RNS_waiting_count; i++) {
        if (RNS_waiting[i]->id == process_id) {
            event.process = RNS_waiting[i];
            break;
        }
    }
    ...
    event.trigger_time = RNS_simulated_time;
    RNS_Heap_insert(event);
    RNS_waiting[i] = RNS_waiting[RNS_waiting_count - 1];
    RNS_waiting_count--;
```

Signaling (presented in Listing 15) searches the RNS_waiting table for the process defined by process_id. If the process is found it is removed from the waiting list and put on top of the event list.

PSE does not pose any limits or make sanity checks on how fork and join are used. It is easy to generate models that lead into deadlocks by making cyclic resource requests in the forked branches. It is on modeler responsibility to keep the models simple to avoid the deadlocks.

5.2 Mapping PSE to hardware

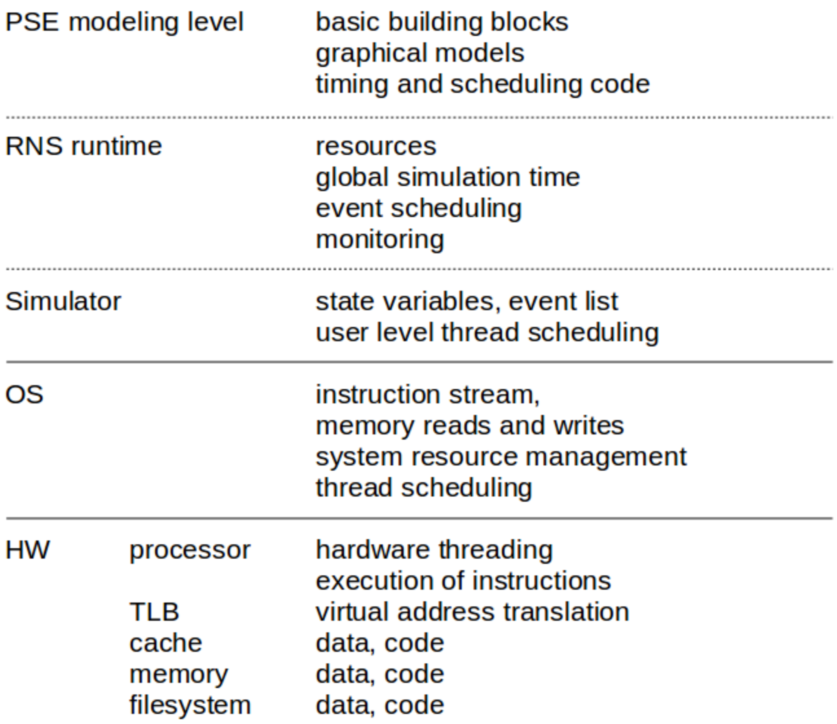
PSE models are abstract representations of systems, which are used to generate a simulator program. The simulator program is executed on a computer where it's behavior can be observed. In this chapter the mapping of PSE models to an execution hardware is overviewed.

At the highest level PSE models are represented graphically using the basic building blocks described in chapter 4.4. On the PSE modeling level scheduling algorithms and timing details can be specified using plug-in code.The graphical models are transformed into C code using the PSE compiler tools. The compiled model code uses the interface of RNS runtime.

At the RNS runtime level the simulation models are represented using the resource reservation based mechanisms. The RNS runtime handles event scheduling and timing of the simulation execution. RNS runtime also implements the monitoring routines. RNS is implemented using the C language with standard libraries and the GNU Pth [87] threading library.

The simulation model files and the RNS libraries are compiled into a simulator program. At this level all models are represented using state variables and events translate into processes that manipulate the state variables. The simulator uses user level thread scheduling to maintain ordering of the event execution.

The simulation program is executed on top of an operating system. The operating system sees the simulator program as a stream of instructions accessing the system resources.



PSE modeling level		basic building blocks graphical models timing and scheduling code
RNS runtime		resources global simulation time event scheduling monitoring
Simulator		state variables, event list user level thread scheduling
OS		instruction stream, memory reads and writes system resource management thread scheduling
HW	processor	hardware threading execution of instructions
	TLB	virtual address translation
	cache	data, code
	memory	data, code
	filesystem	data, code

Figure 21: A conceptual view to PSE simulator mapping from abstract models to execution hardware.

The state variables and the simulator program code are stored in physical memory. The physical memory addresses are translated to virtual address using TLBs. During the execution of the simulator state variables can reside on several different levels of the cache subsystem. As the simulation is monitored traces are written to the filesystem.

The conceptual view of the simulation mapping in Figure 21 shows how simulation execution involves scheduling at multiple levels. At the PSE modeling level schedulers are represented using plugin code. This translates to dynamic routing rules at the RNS runtime level. At the RNS runtime level the RNS scheduler performs event scheduling. The operating system schedules the thread executing the simulator. The scheduler might for example suspend the thread if the simulator hangs waiting for memory or filesystem operations.

5.3 Parallelizing discrete event simulators

There is a rich research tradition on parallelizing discrete event simulators [40], but traditionally the approach has been in modeling distributed systems characterized by relatively small communication overhead when compared to computation. The heterogeneous manycore systems, such as MPSoCs, have a much more dominating communication overhead when compared to distributed systems. The very nature of MPSoCs (shared resources, tight interconnection, fine granularity of tasks) force the events that model their behavior into constant interaction. In addition, the use of scheduling methods that dynamically affect the model state make the lookahead possibilities very narrow.

Despite the challenges PDES research is facing progress is constantly made. Some recent PDES and manycore related research papers include e.g. How PDES scales on manycore platforms [111, 104], implementation of SystemC simulation kernel using GPUs [77] parallel simulation of tightly-coupled SystemC models of MPSoCs using an asynchronous approach [92] and an conservative synchronous approach [94]. Although the previous papers deal with PDES and manycores, their abstraction level is low and intended to accelerate cycle accurate simulations.

Because of the complex interactions between the elements of MPSoCs the abstraction level of their models must be raised if full system simulations are to be performed. The general approach for modeling these systems is to use different approximation and estimation methods to suppress excess communication in order to make simulation execution feasible. Simulators such as Graphite [72] and TaskSim [90] use several different levels of abstraction to model and simulate both the hardware and software of MPSoC based systems.

5.3.1 PSE replicated trials

The RNS simulator engine executes in a single thread and schedules the events sequentially always maintaining the global time. A straight forward way to utilize parallel hardware with PSE is to execute separate instances of the simulator in different threads. This replicated trials approach can be used to reduce variability in the results or it can be used to explore a larger set of different parameter settings simultaneously. A drawback of this approach is that each thread must hold the entire simulation in memory.[40]

In order to measure how PSE scales to the replicated trials approach a simple experiment was conducted. The execution of a simulation model was repeated multiple times using different random number generator seed values for each instance. The set of different simulator instances were executed first sequentially using only one thread and then by distributing the instances for multiple threads. The same scalability experiment was performed by executing three versions of the same simulation model: using no probing, using metric probing and using trace probing. The *no probing* version of the simulation produced no output and was used only as a reference for the other versions. Execution of the *metric probing* version of the simulation produced 1kB of data, while the *tracing version* produced 1,2GB of data.The total execution time of the simulation instances was measured by varying the thread count from one to eight on a Intel Core i7 920 computer with 18GB RAM. The execution times using different number of threads are presented using the scaling efficiency defined in equation 1.

$$E_N = \frac{t_1}{N * t_N} \tag{1}$$

E_N is the scaling efficiency using N threads

t_1 is the execution time using one thread
N is the number of threads
t_N is the execution time using N threads

Figure 22 represents the scaling efficiency of the three simulation versions.

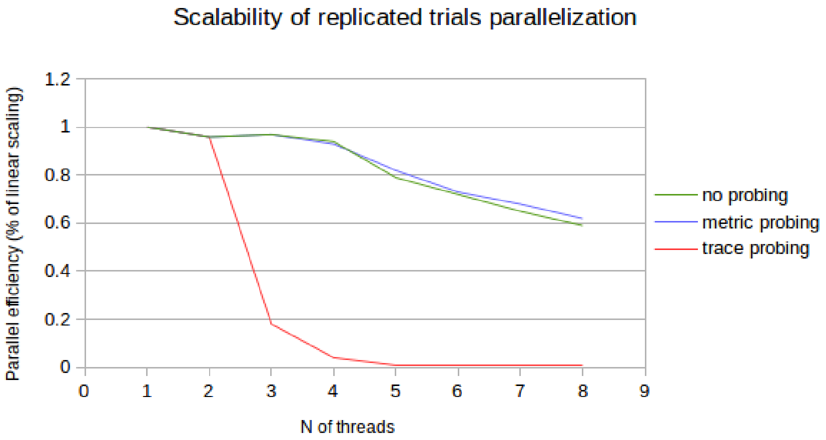


Figure 22: Scaling efficiency of independent simulation instances was studied by executing a simulation with no probing at all, with metric probing and trace probing. Simulation was repeated by varying the number of threads from 1 to 8. Results show that models using metric probing benefit from running multiple instances in parallel, whereas trace probing easily saturates the system and prevents scalability.

Figure 22 shows that the *no probing* and *metric probing* versions of the simulation scaled well up to four threads. E.g. using one thread the *metric probing* version of the simulation took 10 minutes and 20 seconds and with four threads the execution time was 2 minutes and 45 seconds. Increasing the number of threads beyond four gave only marginal execution time improvements. The *trace probing* version of the simulation was able to benefit from simultaneous instances, but then the system saturated. Linux monitoring tools revealed that the system was running out of memory and was forced to constantly swap simulation instances in and out from memory.

Probing has a obvious effect on the scalability of the replicated trials approach, but also the size and complexity of the model and the total execution time of the simulation affects the scalability. Currently there is no other way than trial and error in determining an optimal amount of threads for the simultaneous execution of a particular set of simulation instances. An automated way for determining optimal thread count would be needed. To achieve this the program should be aware of execution hardware resources, of the complexity of the simulated model and of the resource requirements of the simulator program.

5.3.2 Parallel PSE

Overall four main problems related to parallelizing PSE simulator engine RNS can be pointed out:

- Simulator functions are limited
- Locking is not easy
- Parallel I/O is not easy
- Automatic parallelization is not easy

PSE could be parallelized by decomposing the simulator engine into multiple worker threads. But as pointed out in Chapter 3.3, this approach will not scale as the functions of the engine are limited. In order to make use of the future manycore platforms the simulator should be able to scale to hundreds or thousands of cores. A possible path for parallelizing a simulator using the functional decomposition approach could be through implementation efficient parallel data structures that could benefit from multiple worker threads. But it is hard to say anything concluding on this approach without actual implementations and performance evaluations.

The decomposition of the simulator could be done also by decomposing the simulation models into disjoint logical processes. This approach means in practice that the simulators state variables are partitioned into disjoint sets. Similarly as the functional decomposition approach requires synchronization, the logical processes need synchronized communication. Basic approaches are to use message passing or shared variables.[40] Problems related to message passing were discussed in Chapter 3.3, the approach causes severe overhead when simulating tightly connected systems such as MPSoCs. The shared variables approach seems to be a slightly better approach for the purpose of simulating MPSoCs. Shared variables need locking which is hard, as discussed in[47, 33]. Transactional memories attempt to hide the complex details from programmers, but there are still many problems to solve.[63, 62]

Collecting metrics from simulation execution is not a straightforward task. System input and output (I/O) usually the bottleneck in write intensive applications. In parallel write intensive applications I/O becomes the bottleneck much easier. The problems related to parallel I/O are as old as high-performance computing (HPC).[71] And the problems still persist with the new high-performance platforms.[18] I/O is considered one of the major challenges for current and upcoming high-end systems. There is huge potential for performance improvements and better algorithms are needed.[70]

The extraction of parallelism should be automated to fit the tool chain of PSE. Automatic parallelization has been studied but found hard. Parallel hardware is heterogeneous and requires fine tuned applications to benefit from the peculiarities. Parallelizing compilers must have knowledge of the platform.[57]

There is no single recipe that could be followed when parallelizing PSE. The amount of parallelism that can be extracted depends not only on the simulation model, which can vary from small and simple models to highly interacting large and complex models, but also on the execution platform. Most likely some form of a hybrid method should be applied for parallelizing the simulator. Choosing the right way to combine the different ways of parallelization could speed up the computations, increase scalability, and allow efficient utilization of the underlying hardware.

.

6 Demonstrative experiment

In this chapter a small scale demonstrative experiment with PSE is presented. The presented experiment is adapted from the paper presented at the ESM'2014 conference.[37]

The goal of the presented experiment is to evaluate the viability of the PSE modeling approach for early design space exploration.

6.1 Experiment setup

In the experiment PSE is used to evaluate different configurations on how a NPU (Network Processing Unit) and a GPU (Graphical Processing Unit) can be used to accelerate processing of network packet video streams. In the simulation model, the NPU and the GPU are both connected to a host CPU via a PCIe bus. Overview of the setup is illustrated in the Figure 23.

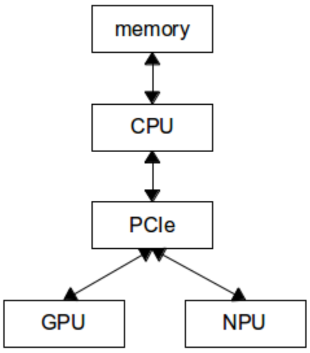


Figure 23: Simulated HW setup consists of a host computer where a NPU and a GPU are attached to the host CPU via a PCIe bus.

The NPU is used to identify network packets containing slices of video keyframes and to write into memory those keyframes that belong to a video stream we are interested on. When a complete video keyframe has been encountered, the network packets that hold it are forwarded to the host computer via the PCIe link. The host computer extracts frames from the network packet payloads and prepares a kernel for GPU execution.[79] The host is configured to buffer a selectable amount of keyframes into one batch before sending the batch to the GPU for further processing. In the GPU, an image recognition algorithm is applied to the provided keyframes. Once the GPU returns the results the host CPU is used to post-processes them and to send them onwards using the NPU.

In this demonstration, simulation is used to investigate the general applicability of the above defined system design. In practise the simulation model is used to evaluate alternative HW/SW architectures and the effect of different scheduling decisions to model performance.

The goal is to evaluate in general the sanity of such a system, and to more precisely estimate the amount of video streams that can be processed using the

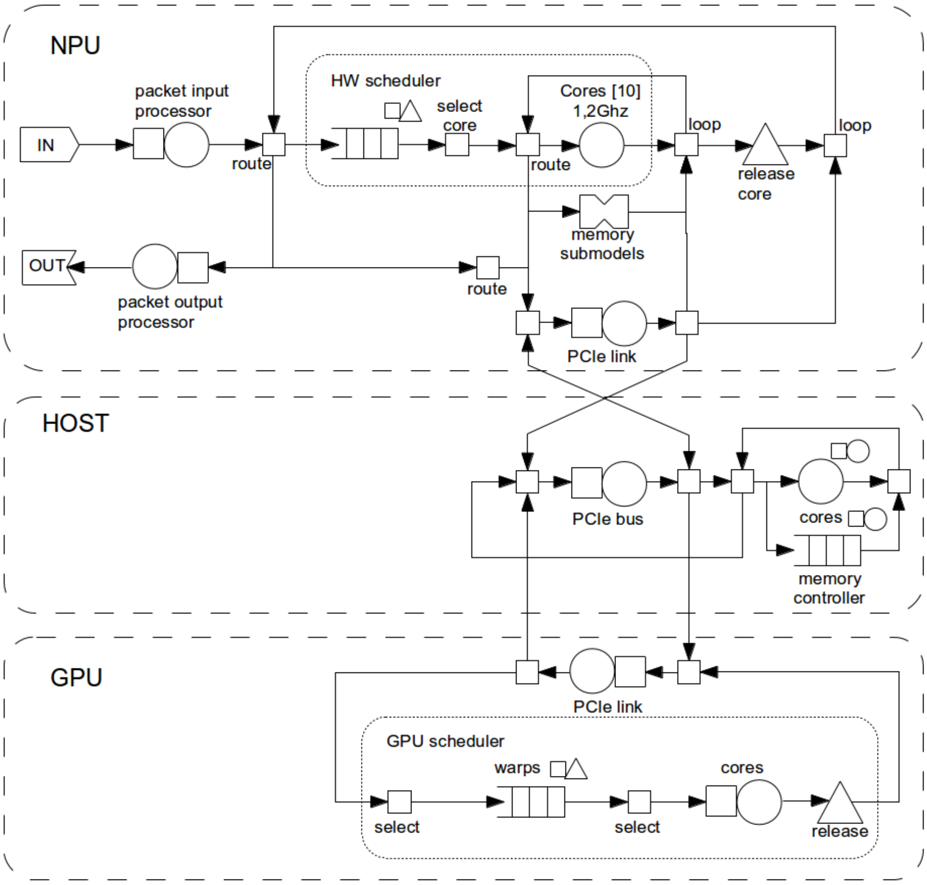


Figure 24: The PSE model of the simulated HW. The NPU, the host and the GPU each contain a set of cores that are used to process requests made by the software model. The PCIe bus is modeled as DMA devices (PCIe link) on the NPU and on the GPU, and as a bus device on the host. Packet input and output processors define the maximum amount of ingress and egress traffic for the system.

planned system. This information can be used to decide the overall direction of the further development of such a system.

Simulation workload characteristics are based on an estimate of a typical HD video stream. We have estimated the required processing time on the NPU by using evaluation board measurements. CPU and GPU processing times have been estimated by measuring the compilation and execution times of a Cuda-based image recognition algorithm. PCIe bus transfer capacity and speeds have been deduced from measurements in [13].

To estimate the viability of the system under study we use the simulation model of the HW as a base and measure the system performance using different workloads and application configurations. The application models are based on average execution time estimates of the SW operations performed in the system. These are measured separately from the different system components.

To minimize the effect of speculated parameter values we induce stochastic variability to HW, SW and workload parameters and repeat the simulation multiple times to acquire a mean estimate with standard deviation bounds.

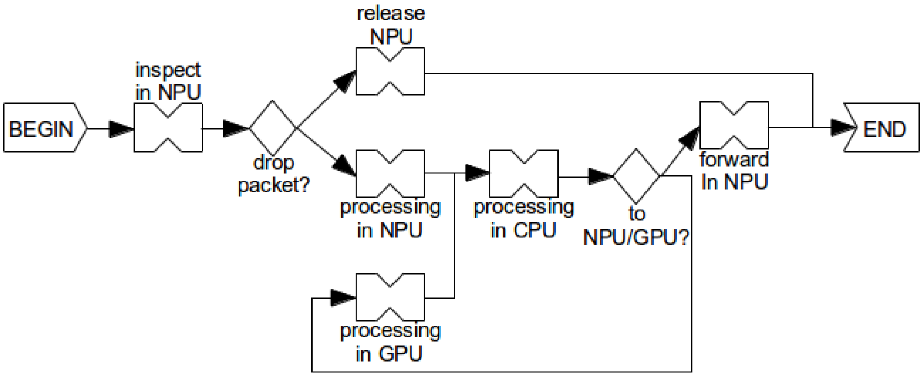


Figure 25: Overview of the simulated SW. Each arriving packet is first examined and classified in the NPU. If a packet contains slices of a keyframe that is being collected, it is further processed. Otherwise the NPU resources are released and the packet is dropped. After processing the packet in the NPU it is sent to the host CPU, where it is decided whether to sent the collected frames to the GPU or not. Work coming back from the GPU is post-processed in the CPU and sent to the NPU which forwards it into the network.

The video being modeled in the simulation is a high definition 1080p stream, which has a frame resolution of 1920x1080 pixels, a color depth of 24 bits per pixel and a frame rate of 30 frames per second. The uncompressed size of a keyframe in such a video stream is about 6.2 MBytes.

We estimate that a HEVC [102] encoding for the video produces a stream that has a bitrate of about 0.9 MBytes/s. On a network with a MTU (maximum transmission unit) of 1500 bytes the payload size is about 1400 bytes. This translates to approximately 640 network packets/s. We estimate that a single stream has in general 6 keyframes and 24 interframes. We further estimate that half of the data is encoded into keyframes and the other half in interframes, thus the keyframe size is about 75000 bytes. On average a keyframe packs into 50 network packets.

In Figure 26 the contents of the *inspect in NPU* submodel are represented. An arriving event enters the *require core lock* node which is a request for a passive resource, namely for a NPU core in the HW model presented in Figure 24. If a core is free then the event is able to proceed forward, otherwise it is queued waiting to be scheduled. In the *fork* node the event is split into two active resource requests, *load L2* and *bus core-L2*. The parameter value *$stream_id* defines the amount that the active resources are requested for.

Figure 25 represents an overview of the simulated software. The *inspect in NPU* submodel contains the computation required to determine the type of the incoming packet. Details of this submodel are represented in the Figure 26. Whether the packet does not contain keyframe contents it is dropped. Branching to further

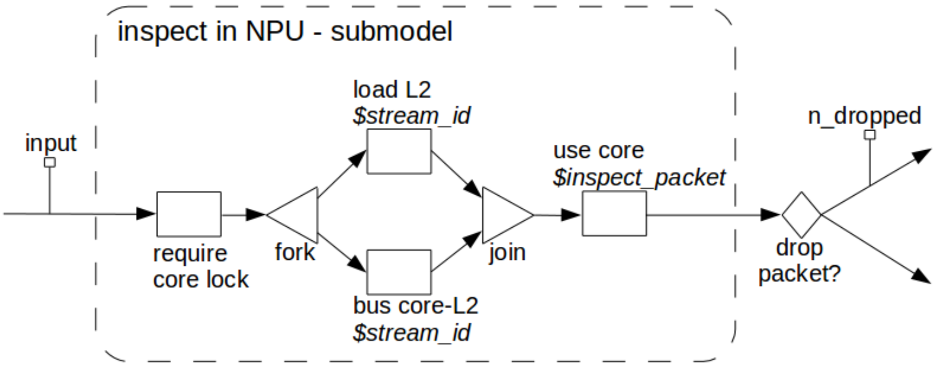


Figure 26: SW model for the initial inspection of an arriving packet in the NPU. (This model is the detailed view of the *inspect in NPU* submodel in the Figure 25.)

processing or releasing the NPU is done on the *drop packet?* node. The actual packet dropping is performed in the *release NPU* submodel. If the network packet contains a slice of a keyframe from a stream being tracked, the network packet is further processed in the *processing in NPU* submodel. In the *processing in NPU* submodel the packet contents are saved into memory and the keyframe slice status is updated. If all the slices of a keyframe have been encountered the keyframe data is forwarded to the host. *processing in CPU* submodel includes the details of the processing that is done upon receiving a keyframe from the NPU. The node *to NPU/GPU?* is used to route the keyframe batch to the GPU and results from the host CPU to the NPU. The submodel *processing in GPU* contains the resource requests related to GPU processing. The submodel *forward in NPU* contains the steps taken to forward the results from the host onto the network.

The system model uses several scheduling mechanisms at the different steps during the video stream processing. A tag based HW scheduler is used on the NPU for fast distribution of work among the NPU cores. On the host a SW scheduler is used to divide work among the host threads. A GPU scheduler divides work to the GPU cores from a warp queue.

6.2 Simulation results

In the experiment, one goal was to find out if the system is able to track 100 video streams and pick 2 keyframes from each stream per second for further processing. Another goal was to determine whether there exists a batch size of keyframes to be sent for the GPU that minimizes the overall latency of getting the results.

In the following the results of a 10 second simulation of 100 video streams are presented.

In the experiment we measured the latency in processing the video keyframes and the load on the NPU, the host and the GPU cores, as well as on the PCIe bus. We parameterized the amount of complete frames to be collected on the host computer before sending a batch to the GPU. The load on the PCIe bus was monitored on the NPU and the GPU links and on the host PCIe bus.

We measured the effect on the system performance by simulating with varying the video keyframe batch size from 1 to 100 keyframes. Each simulation was repeated 30 times.

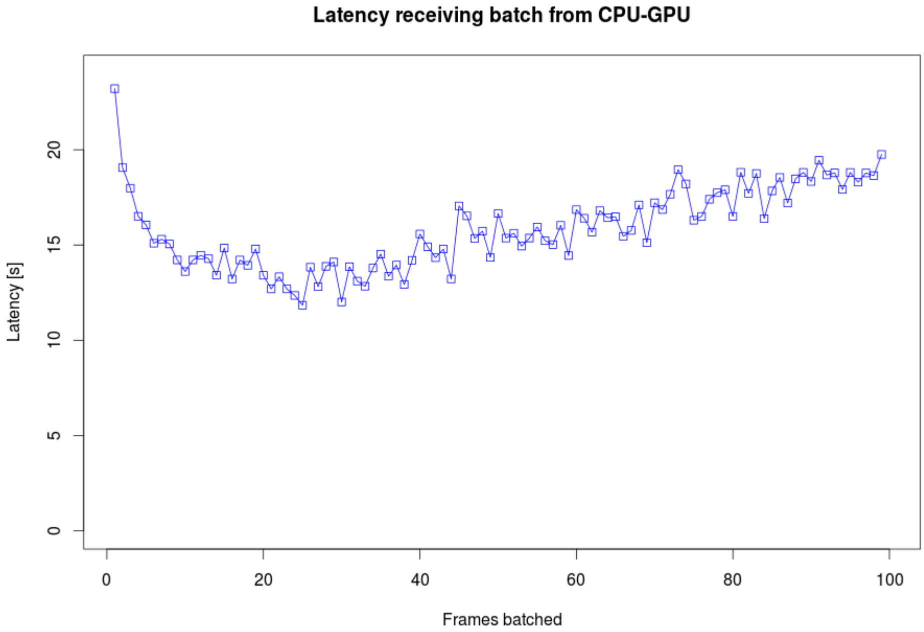


Figure 27: Latency between sending and receiving work from NPU to the CPU-GPU system.

In Figure 27 the latency of getting back the result to the NPU from the host-CPU-GPU subsystem is presented. From Figure 27 it can be seen that the latency reaches its minimum value of approximately 12 seconds with a batch size of about 25 keyframes.

Figure 28 shows the average pending transfers on the PCIe bus. The NPU PCIe link is able to handle all transfers. The host and GPU PCIe links on the other hand have transfers waiting in the buffers. Most saturated is the host PCIe bus. This can be understood by looking at the Figure 23 and by considering the complete path that a work takes during the processing. Once a packet is received in the NPU it is forwarded to the host CPU via the PCIe bus, then from the host the gathered data is written to GPU memory via the PCIe bus. Once the GPU completes the kernel the results are copied back to host memory again via the PCIe bus. Finally the post-processed data is returned back to the NPU, again over the bus. Thus each data item is being transferred four times in some form over the bus. This apparently forms a bottleneck for the system performance.

In Figure 29 the overall queued requests to the cores on the NPU, the host CPU and the GPU are presented. With a very small batch size for the keyframes (about 1 to 5 keyframes) the GPU access forms a bottleneck. The larger the bathes are less there is queuing work for the GPU. On the other hand the latency of receiving work back from the GPU increases at the same time. The smallest latency of getting work

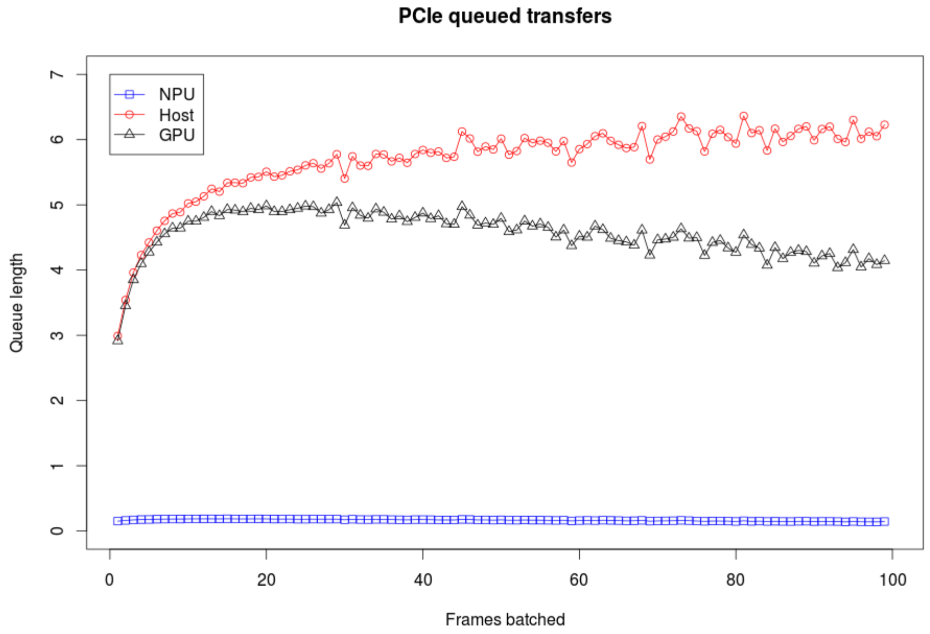


Figure 28: The average number of transfers queued on the PCIe.

back from GPU is achieved with a batch size of about 25 frames. From Figure 29 it can be seen that with a batch size of 25 frames there is on average 200 queued tasks waiting for access to the GPU.

The simulation results show that a optimal keyframe batch size that minimizes the latency of GPU processing can be found. The results further indicate that the PCIe link is most likely to saturate and form a bottleneck on the overall system performance. Further investigation should be done whether it is possible to do direct data transfer between the NPU and the GPU over the PCIe bus so that all the transfers would not need to go through the host memory - and generate surplus traffic on the PCIe bus.

Time to execute the one simulation instance takes about 5 seconds. Execution time of the whole experiment, that is repeating the simulation 30 x 100 = 3000 times, by varying the batch size parameter from 1 to 100 and executing each run 30 times with different seeds, took 4 hours on an Intel Core i5-3320M CPU running at 2.60GHz with 8Gb of memory.

6.3 Conclusions on experiment results

The demonstrative experiment shows how accelerated processing can be modeled. In the experiment model, we connected a GPU into a NPU for video stream processing. Simulating the model, were able to point bottleneck formation and attainable performance with throughput and latency information based on the simulation. According to the results obtained from the presented experiment, abstract simulation seems to be a viable approach in early phase design space exploration. Results

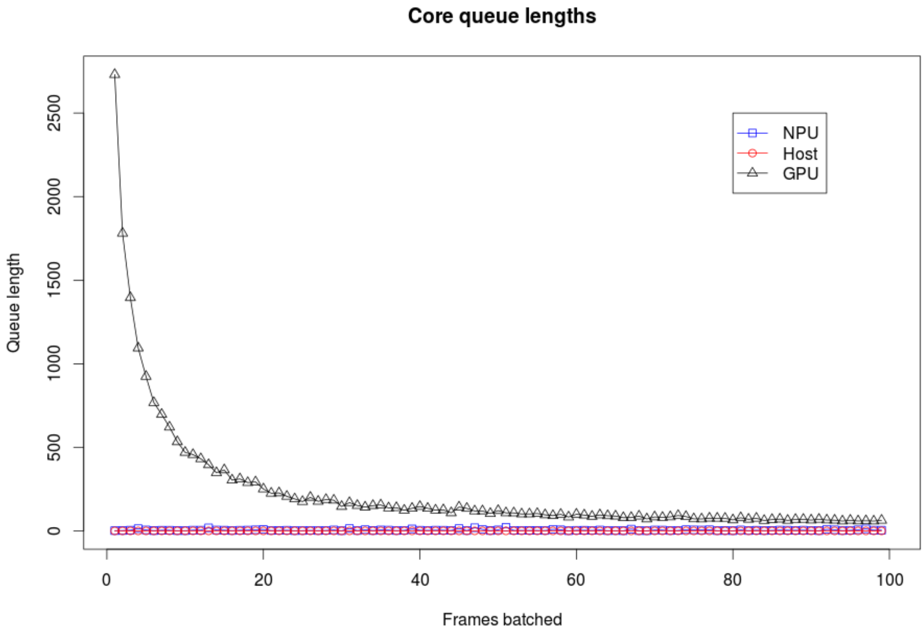


Figure 29: The average queued events waiting processing on the NPU, the host and the GPU.

suggest that even when there is no physical prototype available, it is possible to estimate individual system parameters from existing individual system components. A simulation model can be constructed, and the model can then be used to estimate the performance of different system configurations.

7 Discussion

In this chapter the results of this thesis are discussed. First, the challenges encountered within the topic of the thesis are presented. Next, related work on the field of study is summarized. Then, the findings made during the process are discussed. Finally, some directions for possible future work are proposed.

7.1 Challenge

Performance analysis of heterogeneous MPSoCs is difficult for several reasons. MPSoCs are tightly-connected systems with lots of shared resources. Their operation is based on efficient distribution of the workload among the computing resources. Besides the cost of computation, it is also the cost of communication that play a significant role in MPSoCs. MPSoCs are generally used to process dynamic data streams. Dynamic system input, parallel memory accesses and the shared nature of MPSoC interconnects and other resources lead to non-deterministic delays in the system. The resource interaction is further mixed with scheduling, which is usually performed at multiple points in the system. As pointed out in Chapter 2.3, hardware accelerated scheduling is usually the only viable approach for managing the MPSoC resources. The non-deterministic behavior of MPSoCs, and the scheduling that is performed at multiple points in the system, makes the performance analysis a hard problem.

When analysis of MPSoCs at an abstract level is enough, there exists different analytical modeling methods that can be used (see Chapter 3.5). But what these methods lack, is a way for understanding how the systems behave during execution. In order to understand the behavior of MPSoCs, simulation methods and tracing of the simulation execution is required.

To enable tracing, the simulation needs to be executed and the execution needs to be monitored. While this can be done with many simulation tools, such as SystemC or SpecC, integration of monitoring in a way that the simulation scales on parallel execution platforms is very hard, especially when simulating close to the ISA level.

The performance of simulation execution on parallel platforms is a question of special importance. The future manycore platforms are constantly evolving into a direction where they soon contain thousands, if not millions of cores. With regard to parallel simulation, two main problems arise. First, how to model and simulate such manycore systems? And second, how to parallelize the execution of simulations to benefit from the future manycore platforms?

7.2 Related work

Despite the challenges, parallel simulation is a widely used method in performance analysis of MPSoCs. However, there exists no single method that could be used in an universal manner to model and simulate all parallel systems, nor MPSoCs. Instead, the field of parallel simulation contains many different methods, tools and approaches that attempt to balance between competing goals, and cope with a specific subset of the problem space.

The traditional problem of balancing between the accuracy and speed of the simulation, is increasingly challenging with growing parallelism. The problem has been addressed using different techniques to suppress the amount of required synchronization, and by dynamically adapting to different levels of abstraction to speed up parts of the simulation. For example, the Graphite simulator [72] uses analytical estimation techniques to limit the amount of required synchronization, and also relaxes the event timing so that the events are not necessarily executed in timestamped order. Another approach is implemented in TaskSim [90], which uses four levels of abstraction, and allows dynamic switching between the abstraction levels. Sniper simulator [21] uses a hybrid simulation approch which is based on the usage of an analytical interval core model with a micro-architecture structure simulation. The level of abstraction used in PSE corresponds closely to the second highest abstraction level of TaskSim [90]. There applications are presented as required computation and synchronizations.

In general, more the simulator makes use of estimation techniques to limit the amount of required communication, the more inaccurate the output monitored from the simulation execution will be. PSE monitoring produces exact traces of the state changes in the models. This allows reasoning about the cause and effect relationships of the execution.

Design of PSE has been affected by simulators like RESQME [26]. But where RESQME does queuing network simulation that can be solved using any basic DES, the layered resource network concept of PSE requires custom simulator kernel to operate because of the detailed probing mechanisms. The modeling concepts of RESQME and PSE have similarities, but through the workload modeling PSE corresponds more to other simulators such as the JMT package [12]. Both PSE and JMT are extendable simulation tool sets with different tools to support different modeling methods. More tools and techniques have been discussed in Chapter 3.4.

Parallelization of simulations faces most of the same problems that parallelization of computation in general does. A survey of parallel programming models and tools is done in [54]. Chapter 5.3 contains more detailed references to work done with parallelizing simulators.

Parallelization of simulators is hard because simulation execution is monitored. Simulations could be executed fast, but because probe measurements need to be collected, the simulator needs to write output. This easily becomes a problem as the system I/O (input and output) is a typical bottleneck.[18] An example of system I/O saturation is presented using PSE in Chapter 5.3.1.

A possible approach is to supress the amount of output by computing average metrics and key values during the simulation execution. But as with the analytical modeling methods, this approach does not enable the observation of the behavior of the simulation model during the execution. Detailed monitoring requires writing of usually very large traces. With parallel simulations, and simulations of parallel systems, this quickly becomes a problem.

The answer to the probe I/O problem is parallel I/O. An overview to parallel

I/O is presented in [54]. Another view to parallel I/O is in the real-time hardware monitoring approaches used in MPSoCs.[59] Examples of parallel I/O include the Haskell multicore I/O manager *Mio* [101] and the MPI-IO Parallel I/O Interface [29]. The problems the HPC community are facing with I/O are fundamentally to those encountered with detailed monitoring of parallel simulators. Examples of HPC applications employing parallel I/O include e.g. the Blue Gene supercomputer [18], and the data acquisition system of the CERN Large Hadron Collider [20].

7.3 Discoveries

The abstraction level of MPSoC simulation must be raised in order to make the simulations executable but general statements on the right levels of abstractions are hard to make. As pointed out by authors of TaskSim, the different levels of abstraction should complement each other, and be used for different purposes.[90]

Different abstraction levels and the modeling mechanisms have limitations in their expressive power. In addition, models are meaningful only when the users understand these models. The problem of suitable abstraction level is not just about finding a balance between accuracy and speed but also a balance between model detail and model clarity.

PSE modeling concepts were taught to participants of the ParallaX education day [86]. According to the feedback received, the toolset has potential to be used for teaching performance analysis of parallel systems, and in their understanding in general. But in order to deduce any comparable metrics on the clarity, or expressiveness of different modeling mechanisms, one quickly finds himself in the middle of interdisciplinary, mainly cognitive science related, challenges.

Research in high-level simulation techniques is motivated by the fact that complex systems, such as IoT, are growing in complexity, and require new application development tools. Efficient modeling and simulation tools are needed to support the design process.

Efficient simulators need to take advantage of the available parallel execution platforms. This is especially important, as core counts are constantly growing. Efficient usage of parallel platforms is not only required for the purposes of reaching faster simulation execution speeds but for the sake of the parallelization problem itself. For example, the compiler toolchain and the runtime system of PSE, or any other simulation toolset, can be seen as abstractions of a general purpose computer system. Parallel execution of a simulator program faces the same problems, as all software on parallel hardware. The PDES problem links PSE to a wider research problem context. PSE can be seen as a research platform for parallelism in general.

7.4 Future work

This thesis is part of ongoing research. Several directions for future research directions exist. As shown in this thesis, the resource network methodology, and the dynamic scheduling models can be used for modeling parallel systems. But in order for the PSE toolset to meet future challenges of growing core counts of both the modeled systems, and the simulation execution platforms, the PSE tools need to evolve.

Future challenges include the need for ultra-large-scale simulations, such as IoT applications, or MPSoCs with millions of cores. Simulation of such models require lots of memory and computing resources. This calls for new modeling methods and simulation parallelization techniques. E.g. simulator runtime should be able to reuse the model description by decoupling model state from description. Advanced memory management schemes, such as shared events, application memory regulation, and collective file I/O should be used. And finally, the simulator should make use of parallel hardware, and be able to scale with increased core counts.

Future directions for research which continue the theme of parallel simulation, and performance analysis presented in this thesis, can be categorized into three branches:

- new modeling and simulation features
- parallelization of simulations
- research in related topics

Additional features for modeling and simulation include better support for modeling accelerator devices. PSE needs for example a multifork-join method that could be used to model GPU-style threading, and related scheduling and divergence issues. This would need a new lightweight way to present threads in the simulation kernel, and group them for execution.

As energy efficiency is a key metric in modern MPSoCs, a way to model the power consumption should be included into PSE. The power model should also be accessible from the RNS runtime to include support for energy aware scheduling. Power modeling has been recently incorporated to e.g. the Graphite[61] and Sniper[21] simulators.

Parallelization of PSE could be done essentially in two ways, either porting the old sequential code to support parallelism, or rewriting the simulator engine from scratch.

The first way would be easier by using the conservative simulation approach, as there is no need to save model state history which simplifies the porting of sequential code, and related data structures. The conservative approach would mainly require proper synchronization in order to make use of parallel hardware.

The second way would require rewriting not only the simulation kernel but most of the other code too. Rewriting most of the code from scratch would offer better possibilities to incorporate more complex parallelization strategies. Likely, the most effective way would be to use a form of adaptive approach that uses variations of optimistic and conservative methods. Parallelism extraction should be incorporated also into the compiler tools, where the most effective parallel execution strategy could be estimated. The compiler tools should use graph decomposition methods to extract parallelism in models, and automate the partitioning process. Problems relate to the adaptiveness. It would be easier to parallelize a simulator tuned to execute only certain types of models but PSE can be used in many different ways e.g. producing models of varying complexity and interactions. Parallelizing a more general purpose simulator is always harder. Parallelization of PSE faces also the problem of execution monitoring the related parallel I/O. This problem certainly requires synchronization but also different coordination mechanisms (such as collective I/O), to minize the negative impact to performance.

Future possible topics related to PDES are numerous. They include e.g. research in parallel data structures, algorithms for parallel I/O, and monitoring of parallel execution. Parallel simulation research is tightly connected to research in parallelizing compilers, parallel languages and runtime systems. Therefore simulation and modeling requires understanding of the systems under study. Building a simulator program requires understanding of the software development methods and tools, as well as of the underlying execution hardware, and execution mechanisms. Parallel simulation of MPSoCs integrates most computer system related research fields into an entity, through which these can be explored.

Parallelism is currently the greatest challenge the field of computer science faces. The problems parallel simulation tackles with, are the very problems the entire field is facing. Therefore, more research efforts are strongly needed.

8 Conclusions

The goal of this thesis was to investigate the use of simulation, measurement and modeling methods for analyzing the performance of parallel accelerator rich platforms. The motivation behind this work was that as the new heterogeneous parallel hardware is complex and traditional programming methods are not working as such, a higher level of abstraction is needed in order to analyze the performance of different scheduling, application partitioning and runtime configuration decisions.

This thesis presented a way how to model, simulate and analyze the performance of MPSoCs with accelerators using the resource reservation based mechanism. Focus of this thesis is in scheduling, how to model and analyze hardware, software and hybrid scheduled systems. Performance analysis was done by constructing executable models of systems with adjustable monitoring mechanisms. Modeling in PSE is based on the use of graphical editor tools which allow model description using basic building blocks. PSE models are simulated using discrete event based approach.

Concrete contributions of this thesis include updating an existing simulation framework to support parallelism. Main contribution is on one hand that modeling concepts of PSE have been widened and on the other hand that the supporting mechanisms have been implemented. The implemented fork-join mechanism allows modeling of parallel resource requests.

This thesis has shown that the resource network methodology augmented with dynamic scheduling is a viable approach in modeling heterogeneous MPSoCs with accelerators. With the use of concrete example models this thesis showed how dynamic scheduling can be modeled and simulated using the reseource reservation based methodology. Similarly a way to model memory systems was presented. A small scale demontrative experiment was also done. The demonstrative experiment showed that PSE suits well for early design space exploration.

The PSE runtime can be used to collect a comprehensive set of simulation metrics and traces. The detailed monitoring capabilities of PSE allow exact tracing of events which makes causal reasoning of event sequences possible.

The heterogeneous multi- and manycore systems-on-chip are increasingly complex devices with lots of interacting components. The future platforms will have even greater number of cores and accelerators. MPSoCs are used in embedded systems with high-performance real time requirements, and systems of systems are built using these components. Research in high-level simulation techniques is motivated by the fact that complex systems, such as IoT, are growing in complexity and require new application development tools. Efficient modeling and simulation tools are needed to support the design process.

Similarly new higher level modeling methods and novel parallelization strategies are needed in order to model, simulate and analyze the performance of such systems using as execution platform the available manycore platforms.

Parallelization of PSE and simulators in general is a research question that needs addressing more and more as the core counts on the execution platforms are constantly growing. This is not only for the sake of speeding simulation execution, but also because of PDES research itself. Research made within the field of parallel simulation can lead to novel discoveries that have applications in the general domain of parallel computing.

References

[1] 3PMCES. DATE 2014. http://www.ecsi.org/workshop2014/date/3pmces-proceedings [Accessed: 2014-11-1].

[2] Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H. Moin, Jan Reineke, Bernhard Schommer, and Reinhard Wilhelm. Impact of Resource Sharing on Performance and Performance Prediction: A Survey. In *Proceedings of the 24th International Conference on Concurrency Theory*, CONCUR'13, pages 25–43, Berlin, Heidelberg, 2013. Springer-Verlag. DOI: 10.1007/978-3-642-40184-8_3.

[3] Sarita V. Adve and Hans-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Commun. ACM*, 53(8):90–101, August 2010. DOI: 10.1145/1787234.1787255.

[4] AMD. `http://www.amd.com/`.

[5] AMD. SimNow. http://developer.amd.com/tools-and-sdks/cpu-development/simnow-simulator/ [Accessed: 2014-09-09].

[6] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. COTSon: Infrastructure for Full System Simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, January 2009. DOI: 10.1145/1496909.1496921.

[7] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A View of the Parallel Computing Landscape. *Commun. ACM*, 52(10):56–67, October 2009. DOI: 10.1145/1562764.1562783.

[8] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Comput. Netw.*, 54(15):2787–2805, October 2010. DOI: 10.1016/j.comnet.2010.05.010.

[9] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, February 2002. DOI: 10.1109/2.982917.

[10] M. Bamakhrama and T. Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 195–204, Oct 2011.

[11] Twan Basten, Emiel Van Benthum, Marc Geilen, Martijn Hendriks, Fred Houben, Georgeta Igna, Frans Reckers, Sebastian De Smet, Lou Somers, Egbert Teeselink, Nikola Trčka, Frits Vaandrager, Jacques Verriet, Marc Voorhoeve, and Yang Yang. Model-driven Design-space Exploration for Embedded

Systems: The Octopus Toolset. In *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I*, ISoLA'10, pages 90–105, Berlin, Heidelberg, 2010. Springer-Verlag. http://dl.acm.org/citation.cfm?id=1939281.1939293.

[12] M Bertoli, G Casale, and G Serazzi. The JMT simulator for performance evaluation of non-product-form queueing networks. *IEEE Computer Society*, pages 3–10, 2007. DOI: 10.1109/ANSS.2007.41.

[13] Ray Bittner, Erik Ruf, and Alessandro Forin. Direct GPU/FPGA Communication Via PCI Express. *Cluster Computing*, 17(2):339–348, June 2014. DOI: 10.1007/s10586-013-0280-9.

[14] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Trivedi S. Kishor. *Queuing Networks and Markov Chains.* John Wiley & Sons, 1998.

[15] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. PROTEUS: A High-performance Parallel-architecture Simulator. In *Proceedings of the 1992 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '92/PERFORMANCE '92, pages 247–248, New York, NY, USA, 1992. ACM. DOI: 10.1145/133057.133146.

[16] R. E. Bryant. SIMULATION OF PACKET COMMUNICATION ARCHITECTURE COMPUTER SYSTEMS. Technical report, Cambridge, MA, USA, 1977.

[17] Rainer Buchty, Vincent Heuveline, Wolfgang Karl, and Jan-Philipp Weiss. A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. *Concurrency and Computation: Practice and Experience*, 24(7):663–675, 2012. DOI: 10.1002/cpe.1904.

[18] Huy Bui, H. Finkel, V. Vishwanath, S. Habib, K. Heitmann, J. Leigh, M. Papka, and K. Harms. Scalable Parallel I/O on a Blue Gene/Q Supercomputer Using Compression, Topology-Aware Data Aggregation, and Subfiling. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 107–111, Feb 2014. DOI: 10.1109/PDP.2014.60.

[19] Lukai Cai, Shireesh Verma, and Daniel D. Gajski. Comparison of SpecC and SystemC Languages for System Design, 2003. Technical Report CECS-03-11. Available at: http://users.ece.utexas.edu/~gerstl/ee382v-ics_f09/soc/tutorials/Comparison of SpecC and SystemC Languages for System Design.pdf [Accessed: 2014-08-08].

[20] F. Carena, W. Carena, S. Chapeland, V. Chibante Barroso, F. Costa, E. Dénes, R. Divià, U. Fuchs, A. Grigore, T. Kiss, G. Simonetti, C. Soós, A. Telesca, P. Vande Vyvre, and B. von Haller. The ALICE data acquisition

system. *Nuclear Instruments and Methods in Physics Research A*, 741:130–162, March 2014. DOI: 10.1016/j.nima.2013.12.015.

[21] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sampled Simulation of Multi-Threaded Applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, 2013.

[22] Cavium. `http://www.cavium.com`.

[23] Cavium. OCTEON Programmers Guide - The Fundamentals. `http://university.caviumnetworks.com/textbooks.html` [Accessed: 2014-03-26].

[24] Pew Research Center. *The Internet of Things Will Thrive by 2025*. Pew Research Center, 2014. `http://www.pewinternet.org/2014/05/14/internet-of-things/` [Accessed: 2014-08-12].

[25] K.M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *Software Engineering, IEEE Transactions on*, SE-5(5):440–452, Sept 1979. DOI: 10.1109/TSE.1979.230182.

[26] Kow C. Chang, Robert F. Gordon, Paul G. Loewner, and Edward A. MacNair. The Research Queuing Package Modeling Environment (RESQME). In *Proceedings of the 25th Conference on Winter Simulation*, WSC '93, pages 294–302, New York, NY, USA, 1993. ACM. DOI: 10.1145/256563.256654.

[27] S. Chattopadhyay, C.L. Kee, A Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A Unified WCET Analysis Framework for Multi-core Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 99–108, April 2012. DOI: 10.1109/RTAS.2012.26.

[28] Jianwei Chen, Murali Annavaram, and Michel Dubois. SlackSim: A Platform for Parallel Simulations of CMPs on CMPs. *SIGARCH Comput. Archit. News*, 37(2):20–29, July 2009. DOI: 10.1145/1577129.1577134.

[29] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snirt, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO Parallel I/O Interface. In Ravi Jain, John Werth, and JamesC. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, volume 362 of *The Kluwer International Series in Engineering and Computer Science*, pages 127–146. Springer US, 1996. DOI: 10.1007/978-1-4613-1401-1_5.

[30] Menascé Daniel, Almeida Virgilio, and Dowdy Larry. *Capacity planning and performance modeling.* Prentice Hall, 1994.

[31] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011. DOI: 10.1145/1978802.1978814.

[32] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V. Leo Rideovt, Ernest Bassous, and Andre R. Leblanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits Society Newsletter, IEEE*, 12(1):38–50, Winter 2007. DOI: 10.1109/N-SSC.2007.4785543.

[33] Michel Dubois, Murali Annavaram, and Per Stenström. *Parallel Computer Organization and Design.* Cambridge Univ. Press, 2012.

[34] R. Dömer. SpecC. `http://www.cecs.uci.edu/~specc/` [Accessed: 2014-08-08].

[35] Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. In Giovanni De Micheli, Rolf Ernst, and Wayne Wolf, editors, *Readings in Hardware/Software Co-design*, pages 86–107. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[36] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Task Superscalar: An Out-of-Order Task Pipeline. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 89–100, Washington, DC, USA, 2010. IEEE Computer Society. DOI: 10.1109/MICRO.2010.13.

[37] Eurosis. ESM'2014. `http://eurosis.org/cms/?q=node/2852` [Accessed: 2014-11-1].

[38] D. Evans. *The Internet of Things - How the Next Evolution of the Internet Is Changing Everything.* Cisco IBSG, 2011. White Paper.

[39] Daquan Feng, Chenzi Jiang, Gubong Lim, Jr. Cimini, L.J., Gang Feng, and G.Y. Li. A survey of energy-efficient wireless communications. *Communications Surveys Tutorials, IEEE*, 15(1):167–178, First 2013. DOI: 10.1109/SURV.2012.020212.00049.

[40] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems.* Wiley Interscience, 2000.

[41] gem5. The gem5 simulator system. `http://www.m5sim.org`.

[42] Alan D. George, Ryan B. Fogarty, Jeff S. Markwell, and Michael D. Miars. An Integrated Simulation Environment for parallel and distributed system prototyping. *Simulation*, 72:283–294, 1999.

[43] Soonhoi Ha, Sungchan Kim, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo. PeaCE: A Hardware-software Codesign Environment for Multimedia Embedded Systems. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):24:1–24:25, May 2008. DOI: 10.1145/1255456.1255461.

[44] Blake A. Hechtman and Daniel J. Sorin. Exploring memory consistency for massively-threaded throughput-oriented processors. *SIGARCH Comput. Archit. News*, 41(3):201–212, June 2013. DOI: 10.1145/2508148.2485940.

[45] Jörg Henkel and Rolf Ernst. An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(2):273–290, April 2001. DOI: 10.1109/92.924041.

[46] Sutter Herb. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.* Dr. Dobb's Journal, 30(3), March 2005, 2005. Available online: `http://www.gotw.ca/publications/concurrency-ddj.htm[Referenced:2014-8-13]`.

[47] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessor. *SIGPLAN Not.*, 27(9):262–273, September 1992. DOI: 10.1145/143371.143537.

[48] Vesa Hirvisalo. *QNS - A Queueing Network Simulator.* Technical Report PEAK-TKK22-General, Helsinki University of Technology, 1998.

[49] Vesa Hirvisalo. On Static Timing Analysis of GPU Kernels. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 43–52, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. http://dx.doi.org/10.4230/OASIcs.WCET.2014.43.

[50] J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: Simulating shared-memory multiprocessors with ilp processors. In *Computer vol. 35, no. 2*, pages 40–49, 2002. `http://rsim.cs.illinois.edu/~sadve/Publications/computer02.pdf` [Accessed: 2014-04-14].

[51] Wen-mei Hwu, Shane Ryoo, Sain-Zee Ueng, John H. Kelm, Isaac Gelado, Sam S. Stone, Robert E. Kidd, Sara S. Baghsorkhi, Aqeel A. Mahesri, Stephanie C. Tsao, Nacho Navarro, Steve S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Implicitly Parallel Programming Models for Thousand-core Microprocessors. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 754–759, New York, NY, USA, 2007. ACM. DOI: 10.1145/1278480.1278669.

[52] ITRS. International Technology Roadmap for Semiconductors, 2011. `http://www.itrs.net/Links/2011ITRS/Home2011.htm` [Accessed: 2014-06-17].

[53] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley Interscience, New York, NY, 1991. ISBN:0471503361.

[54] Alfonso Nino Javier Diaz, Camelia Munoz-Caro. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. *IEEE Transactions on Parallel & Distributed Systems*, 23(8):1369–1386, 2012. DOI: 10.1109/TPDS.2011.308.

[55] L. Hennessy John and David A. Patterson. *Computer organization and design: the hardware/software interface.* Morgan Kaufmann., 2013. 5th edition.

[56] A Kalavade and E.A Lee. The extended partitioning problem: hardware/software mapping and implementation-bin selection. In *Rapid System Prototyping, 1995. Proceedings., Sixth IEEE International Workshop on*, pages 12–18, Jun 1995. DOI: 10.1109/IWRSP.1995.518565.

[57] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[58] C. Kim and AK. Agrawala. Analysis of the fork-join queue. *Computers, IEEE Transactions on*, 38(2):250–255, Feb 1989. DOI: 10.1109/12.16501.

[59] Georgios Kornaros and Dionisios Pnevmatikatos. A Survey and Taxonomy of On-chip Monitoring of Multicore Systems-on-chip. *ACM Trans. Des. Autom. Electron. Syst.*, 18(2):17:1–17:38, April 2013. DOI: 10.1145/2442087.2442088.

[60] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):162–173, June 2007. DOI: 10.1145/1273440.1250683.

[61] G. Kurian, S.M. Neuman, G. Bezerra, A Giovinazzo, S. Devadas, and J.E. Miller. Power modeling and other new features in the Graphite simulator. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 132–134, March 2014. DOI: 10.1109/ISPASS.2014.6844471.

[62] J. Larus and R. Rajwar. *Transactional Memory.* Morgan-Claypool, 2007.

[63] James R. Larus and Christos Kozyrakis. *Transactional memory.* CACM 51(7): 80-88 (2008), 2008.

[64] Edward A. Lee. *The Problem with Threads.* Computer, vol. 39, no. 5, pp. 33-42, May 2006, 2006. doi:10.1109/MC.2006.180.

[65] Junghee Lee, C. Nicopoulos, Hyung Gyu Lee, S. Panth, Sung Kyu Lim, and Jongman Kim. IsoNet: Hardware-Based Job Queue Management for Many-Core Architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 21(6):1080–1093, June 2013. DOI: 10.1109/TVLSI.2012.2202699.

[66] J. Lilius, J. Takala, and V. Hirvisalo. Strategic research agenda for ParallaX – parallel acceleration, 2012. Technical report, TiViT.

[67] Unai Lopez-Novoa, Alexander Mendiburu, and José Miguel-Alonso. A Survey of Performance Modeling and Simulation Techniques for Accelerator-based Computing. *IEEE Transactions on Parallel and Distributed Systems*, 2014. DOI: 10.1109/TPDS.2014.2308216.

[68] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33:2005, 2005.

[69] Jeronimo Castrillon Mazo and Rainer Leupers. *Programming Heterogenous MPSoCs*. Springer, 2014. DOI: 10.1007/978-3-319-00675-8.

[70] Kshitij Mehta, Edgar Gabriel, and Barbara Chapman. Specification and performance evaluation of parallel i/o interfaces for openmp. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 1–14, Berlin, Heidelberg, 2012. Springer-Verlag. DOI: 10.1007/978-3-642-30961-8_1.

[71] Paul Messina. Parallel I/O: a set of intertwined systems and applications issues. In *Parallel Processing, 1996. Proceedings of the 1996 ICPP Workshop on Challenges for*, pages 85–90, Aug 1996. DOI: 10.1109/ICPPW.1996.538593.

[72] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, Jan 2010. DOI: 10.1109/H-PCA.2010.5416635.

[73] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. *SIGPLAN Not.*, 37(7):18–27, June 2002. DOI: 10.1145/566225.513835.

[74] Gordon E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *Solid-State Circuits Society Newsletter, IEEE*, 11(5):33–35, Sept 2006. DOI: 10.1109/N-SSC.2006.4785860.

[75] Open MPI. `http://www.open-mpi.org/`.

[76] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Mark D. Hill, David A. Wood, Steven Huss-Lederman, and James R. Larus. Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator. *IEEE Concurrency*, 8(4):12–20, October 2000. DOI: 10.1109/4434.895100.

[77] Mahesh Nanjundappa, Anirudh Kaushik, Hiren D. Patel, and Sandeep K. Shukla. Accelerating SystemC simulations using GPUs. *2012 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 0:132–139, 2012. http://doi.ieeecomputersociety.org/10.1109/HLDVT.2012.6418255.

[78] NSN. Open Event Machine, 2013. `http://sourceforge.net/projects/eventmachine/`.

[79] NVIDIA. Cuda c programming guide, 2014. `http://docs.nvidia.com/cuda/cuda-c-programming-guide` [Accessed: 2014-06-12].

[80] OpenCL. `https://www.khronos.org/opencl/`.

[81] OpenMP. `http://www.openmp.org/`.

[82] Open SystemC Initiative (OSCI). SystemC. `http://sourceforge.net/projects/systemc/` [Accessed: 2014-08-08].

[83] Harry Perros. *Computer Simulation Techniques: The definitive introduction!* Computer Science Departement, NC State University, Raleigh, NC, 2009. `http://www.csc.ncsu.edu/faculty/perros//simulation.pdf` [Accessed: 2014-08-08].

[84] James Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, 1981.

[85] Artur Podobas, Mats Brorsson, and Karl-Filip Faxén. A comparison of some recent task-based parallel programming models, 2010. `http://soda.swedish-ict.se/3869/` [Accessed: 2014-08-08].

[86] Parallax project. Parallax Education Day, 2014. `http://http://www.parallax-project.fi/events/educationday/`.

[87] GNU Pth. The GNU Portable Threads. `http://www.gnu.org/software/pth/`.

[88] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *In Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, 1993.

[89] J. Ributzka. *Concurrency and synchronization in the modern many-core: challenges and opportunities.* PhD thesis, Deleware Univ., 2013.

[90] Alejandro Rico, Felipe Cabarcas, Carlos Villavieja, Milan Pavlovic, Augusto Vega, Yoav Etsion, Alex Ramirez, and Mateo Valero. On the simulation of large-scale architectures using multiple application abstraction levels. *ACM Trans. Archit. Code Optim.*, 8(4):36:1–36:20, January 2012. DOI: 10.1145/2086696.2086715.

[91] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel Distrib. Technol.*, 3(4):34–43, December 1995. DOI: 10.1109/88.473612.

[92] C. Roth, S. Reder, G. Erdogan, O. Sander, G.M. Almeida, H. Bucher, and J. Becker. Asynchronous parallel MPSoC simulation on the Single-Chip Cloud Computer. In *System on Chip (SoC), 2012 International Symposium on*, pages 1–8, Oct 2012. DOI: 10.1109/ISSoC.2012.6376364.

[93] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 311–322, New York, NY, USA, 2010. ACM. DOI: 10.1145/1736020.1736055.

[94] C. Schumacher, R. Leupers, D. Petras, and A Hoffmann. parSC: Synchronous parallel SystemC simulation on multi-core host architectures. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 241–246, Oct 2010.

[95] Graphite simulator source code. `https://github.com/mit-carbon/Graphite`.

[96] M. Sjalander, A Terechko, and M. Duranton. A Look-Ahead Task Management Unit for Embedded Multi-Core Architectures. In *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*, pages 149–157, Sept 2008. DOI: 10.1109/DSD.2008.45.

[97] M.B. Taylor. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1131–1136, June 2012.

[98] Tcl/Tk. Tcl Developer Xchange. http://www.tcl.tk/ [Accessed: 2014-08-10].

[99] ETSI TR 102 638 v1.1.1. Intelligent Transportation Systems (ITS); Vehicular Communications; Basic Set of Applications (BSA); Definitions, 2009. ETSI Technical Report.

[100] C.H. van Berkel. *Multi-core for mobile phones.* DATE'09, 2009. `http://dl.acm.org/citation.cfm?id=1874620.1874924`.

[101] Andreas Richard Voellmy, Junchang Wang, Paul Hudak, and Kazuhiko Yamamoto. Mio: A High-performance Multicore Io Manager for GHC. *SIGPLAN Not.*, 48(12):129–140, September 2013. DOI: 10.1145/2578854.2503790.

[102] Sullivan G.; Ohm J.R.; Han W.; and Wiegand T. Overview of the High Efficiency Video Coding (HEVC) Standard. *IEEE Trans. Circuits Syst. Video Techn. 1649–1668*, 2012. DOI: 10.1109/TCSVT.2012.2221191.

[103] Hao Wang, Vijay Sathish, Ripudaman Singh, Michael J. Schulte, and Nam Sung Kim. Workload and Power Budget Partitioning for Single-chip Heterogeneous Processors. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 401–410, New York, NY, USA, 2012. ACM. DOI: 10.1145/2370816.2370873.

[104] Jingjing Wang, Ketan Bahulkar, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Can PDES Scale in Environments with Heterogeneous Delays? In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '13, pages 35–46, New York, NY, USA, 2013. ACM. DOI: 10.1145/2486092.2486098.

[105] Vincent M. Weaver and Sally A. Mckee. Are cycle accurate simulations a waste of time. In *In Proc. 7th Workshop on Duplicating, Deconstructing, and Debunking*, 2008.

[106] A. Weddell. *A survey of multi-source energy harvesting systems.* IEEE, DATE, 2013. DOI: 10.7873/DATE.2013.190.

[107] A Wellig and J. Zory. Framed complexity analysis in SystemC for multi-level design space exploration. In *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, pages 416–423, Sept 2003. DOI: 10.1109/DSD.2003.1231975.

[108] T.F. Wenisch, R.E. Wunderlich, M. Ferdman, A Ailamaki, B. Falsafi, and J.C. Hoe. SimFlex: Statistical Sampling of Computer System Simulation. *Micro, IEEE*, 26(4):18–31, July 2006. DOI: 10.1109/MM.2006.79.

[109] Jonathan A. Winter, David H. Albonesi, and Christine A. Shoemaker. Scalable Thread Scheduling and Global Power Management for Heterogeneous Many-core Architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 29–40, New York, NY, USA, 2010. ACM. DOI: 10.1145/1854273.1854283.

[110] A. Wolf, W. Jerraya and G. Martin. *Multiprocessor System-on-Chip (MPSoC) Technology.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 10, October 2008., 2008.

[111] Yi Zhang, Jingjing Wang, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Exploring Many-core Architecture Design Space for Parallel Discrete Event Simulation. In *Proceedings of the 2Nd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '14, pages 95–104, New York, NY, USA, 2014. ACM. DOI: 10.1145/2601381.2601392.

[112] Gengbin Zheng, Gunavardhan Kakulapati, and L.V. Kale. BigSim: a parallel simulator for performance prediction of extremely large parallel machines. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 78–, April 2004. DOI: 10.1109/IPDPS.2004.1303013.

[113] Gongbo Zhou, Linghua Huang, Wei Li, and Zhencai Zhu. Harvesting Ambient Environmental Energy for Wireless Sensor Networks: A Survey. *Journal of Sensors*, 2014:20, 2014. DOI: 10.1155/2014/815467.

[114] S. Zhuravlev, J.C. Saez, S. Blagodurov, A Fedorova, and M. Prieto. Survey of Energy-Cognizant Scheduling Techniques. *Parallel and Distributed Systems, IEEE Transactions on*, 24(7):1447–1464, July 2013. DOI: 10.1109/TPDS.2012.20.