Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Kristian Hartikainen

Performance Analysis of Packet Processing Systems

Master's Thesis
Espoo, May 26, 2016

Supervisor: Prof. Heikki Saikkonen
Advisor: Vesa Hirvisalo D.Sc. (Tech.)

Aalto University
School of Science

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author: Kristian Hartikainen

Title:
Performance Analysis of Packet Processing Systems

Date: May 26, 2016 **Pages:** 105
Major: Software systems **Code:** T-106
Supervisor: Prof. Heikki Saikkonen
Advisor: Vesa Hirvisalo D.Sc. (Tech.)

This thesis investigates the use of measurement, simulation, and modeling methods for the performance analysis of packet processing systems, and more precisely hardware accelerated multiprocessor system-on-chip (MPSoC) devices running task-parallel applications. To guarantee the tight latency and throughput requirements, the devices often incorporate complex hardware accelerated packet scheduling mechanisms. At the same time, due to the complexity of these systems, different software abstractions, such as task-based programming models, are used to develop packet processing applications. These challenges, together with dynamic characteristics of the packet streams makes the performance analysis of packet processing systems non-trivial.

We demonstrate that, with extended queue disciplines and support for modeling parallelism, resource network methodology is a viable approach for modeling complex MPSoC based systems running task-based parallel applications on dynamic workloads. The main contributions of our work are three-fold. First, we have extended the toolset of an existing in-house modeling and simulation software, Performance Simulation Environment. The extensions enable modeling of user-definable queue disciplines, which further enable flexible modeling of complex hardware interactions of MPSoCs and the parallelism of task-based programming models. Secondly, we have studied, instrumented, and measured the characteristics of a packet processing system. Finally we have modeled a multi-blade packet processing system with customizable workload and task-parallel application models, and run simulation experiments.

In both experiments, the model acts as expected. According to the experiment results, the resource network concept seems to be a viable tool for the performance analysis of packet processing systems. The chosen abstraction level provides desired balance between the functionality, ease of use, and simulation performance.

Keywords: performance analysis, modeling, simulation, resource networks, packet processing, network processing unit, data plane, hardware accelerated scheduling
Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Kristian Hartikainen		
Työn nimi: Pakettiprosessointijärjestelmien Suorituskykyanalyysi			
Päiväys:	26. toukokuuta 2016	**Sivumäärä:**	105
Pääaine:	Ohjelmistotekniikka	**Koodi:**	T-106
Valvoja:	Professori Heikki Saikkonen		
Ohjaaja:	Tekniikan tohtori Vesa Hirvisalo		

Tässä työssä tutkitaan mittaus-, mallinnus-, ja simulaatiometodien käyttöä pakettiprosessisysteemien, tarkemmin ottaen tehtävärinnakkaisia sovelluksia ajavien laitteistokiihdytettyjen moniydinjärjestelmien, suorityskykyanalyysiin. Tiukoista viive- ja läpivirtausvaatimuksista johtue pakettiprosessointilaitteistot sisältävät usein monimutkaisia laitteistokiihdytettyjä pakettiajoitusmekanismeja. Laittestojen monimutkaisuudesta johtuen pakettiprosessointisovellusten kehittämiseen käytetään usein erilaisia ohjelmointiabstraktioita, kuten tehtävärinnakkaisia ohjelmointimalleja. Laitteston ja ohjelmiston asettamat haasteet yhdessä pakettivirtojen dynaamisen luonteen kanssa tekevät pakettiprosessointijärjestelmien suorituskykyanalyysista epätriviaalia.

Työssä havainnollistamme, että laajennettujen jonokurien ja rinnakkaismallinnustuen avulla resurssiverkkometodologia on toimiva lähestymistapa tehtävärinnakkaisia rinnakkaisohjelmointisovelluksia ajavien monimutkaisten laitteistokiihdytettyjen moniydinjärjestelmien suorituskykyanalyysiin dynaamisilla työkuormilla. Työmme päätulokset ovat kolmiosaiset. Ensinnäkin, olemme laajentaneet olemassaolevan mallinnus- ja simulaatioohjelmiston, Performance Simulation Environmentin, ohjelmointityökaluja. Laajennukset mahdollistavat käyttäjän määriteltävien jonokurien mallintamisen, mikä edelleen mahdollistaa tehtävärinnakkaisia sovelluksia ajavien laittestokiihdytettyjen moniydinjärjestelmien laittestovuorovaikutusten joustavan mallinnuksen. Toiseksi, olemme tutkineet ja mitanneet erään pakettiprosessointijärjestelmän ominaisuuksia. Viimeiseksi, olemme mallintaneet pakettiprosessointijärjestelmän muunnettavilla työkuormilla ja tehtävärinnakkaisilla sovellusmalleilla, sekä suorittaneet näitä simulaatiokokein.

Molempien kokeiden mallit käyttäytyvät odotetulla tavalla. Koetulosten perusteella resurssiverkkokonsepti vaikuttaa toimivalta työkalulta kompleksien pakettiprosessointijärjestelmien suorituskykyanalyysiin. Valittu abstraktiotaso tarjoaa toivotun tasapainon simulaation suorituskyvyn, toiminnallisuuden ja helppokäyttöisyyden välillä.

Asiasanat:	suorituskykyanalyysi, mallinnus, simulointi, resurssiverkot, pakettiprosessointi, laitteistokiihdytetty vuoronnus
Kieli:	Englanti

Acknowledgements

I would like to thank my supervisor, Prof. Heikki Saikkonen, for his invaluable feedback during the final stages of writing this thesis.

Besides my supervisor, I would like to thank my instructor D.Sc. (Tech) Vesa Hirvisalo, who provided me with support and feedback throughout the project, and without whom this thesis would not have been possible.

I express my sincere gratitude to Jussi Hanhirova, who worked with me in solving many technical difficulties, provided me with insightful comments, and guided me throughout the project.

I would also like to thank my good friend Risto Vuorio, whose company, both at school and spare time, over the last few years has been invaluable for me.

Finally, I thank my family for their endless support.

Espoo, May 26, 2016

Kristian Hartikainen

Contents

1 Introduction **8**
1.1 Problem statement . . . 9
1.2 Contributions . . . 10
1.3 Structure . . . 12

2 Computing Trends **14**
2.1 Modern Computing . . . 14
2.1.1 The End of Free Lunch . . . 14
2.1.2 Parallel Computing . . . 15
2.2 Big Data . . . 18
2.3 Virtualization . . . 18
2.3.1 Platform Virtualization . . . 19
2.3.2 Operating System Level Virtualization . . . 20
2.4 Cloud Computing . . . 20
2.4.1 Energy Consumption . . . 23
2.4.2 Datacenter Networks . . . 23
2.5 Fog Computing . . . 24

3 Packet Processing **26**
3.1 Packet Switched Networks . . . 26
3.1.1 Network Components . . . 27
3.1.2 Traffic Characteristics . . . 28
3.2 General Packet Processing Framework . . . 28
3.2.1 Ingress and Egress . . . 28
3.2.2 Processing Paths . . . 30
3.2.3 Packet Processing Functions . . . 31
3.3 Processing Hardware . . . 33
3.3.1 Processing Elements . . . 34
3.3.2 Parallel and Pipelined Architectures . . . 35
3.4 Programming Models . . . 36
3.4.1 Intel Data Plane Development Kit . . . 37

3.4.2 Open Event-Machine 37
3.5 Example Network Processing System 38
3.6 Characteristic Behavior 39
3.6.1 Communication Latencies 40
3.6.2 Memory Characteristics 43

4 System Performance Analysis 47
4.1 Performance Analysis 47
4.1.1 Evaluation Techniques 47
4.1.2 Performance Metrics 48
4.1.3 System Components and Environment 49
4.2 System Modeling 50
4.2.1 Queuing Networks 51
4.2.2 Resource Networks 51
4.3 Simulation 52
4.3.1 Monitoring 53
4.4 Modeling and Simulation Software 53

5 Performance Simulation Environment 55
5.1 Toolset Overview 55
5.2 PSE Model 57
5.3 Monitoring 59
5.4 Resource Network Simulator 60
5.4.1 Simulator Engine 61

6 Mechanism For Extended Queue Disciplines 62
6.1 Service Routines 63
6.2 Runtime Structures 65
6.2.1 RNS Resource 65
6.2.2 RNS Client 66
6.3 Reserve and Select Functions 67
6.4 Discipline Examples 68

7 Modeling a Packet Processing System 71
7.1 Hardware Model 71
7.2 Modeling the Task Scheduler 74
7.2.1 Application Models 74
7.2.2 Global Hardware Scheduler 75

8 Demonstrative Experiment and Discussion 80
8.1 Experiment Setup 80
8.2 Experiment 1: Global Queue Interrelations 81
8.2.1 Simulation Measurements 83
8.3 Experiment 2: Queue Coremasks 84
8.3.1 Simulation Measurements 87
8.4 Experiment Analysis 87
8.5 Discussion 91
8.5.1 Challenges 91
8.5.2 Discoveries 92
8.5.3 Future Work 92

9 Conclusions 94

Chapter 1

Introduction

The proliferation of Internet devices and sensors have led to a situation where data are produced faster than can be transmitted, stored, or processed. Majority of the data is transmitted as packet streams over the packet switched networks. Along the way, various network devices, such as switches, routers, adapters, are used to forward and process the streams.

Packet processing devices face an enormous performance challenge. While the amount of data being transmitted is increasing, at the same time, the packet processing tasks, which should be done on-the-fly, have become more and more complex. In addition to the usual forwarding, the packet processing systems are responsible of various other functions, such as traffic management (shaping, timing, scheduling), security processing, and quality of service. At the same time, technology and customer requirements are rapidly changing, forcing the vendors to seek programmable solutions to achieve shorter development cycles and more revisions.

Network processing equipment can generally be divided into three categories: easily programmable general Central Processing Units (CPU), well-performing but hardwired Application-Specific Integrated Circuits (ASIC), and middle-ground Network Processing Units (NPU). The focus of this thesis is on the middle-ground NPU devices, typically built as a multiprocessor system-on-chip (MPSoC). MPSoCs integrate multiple heterogeneous or homogeneous functional units, such as processors, memory, circuits, and peripherals, on a single chip. The promise of MPSoC devices are their better performance, functionality, and energy usage, due to the multiple interconnected specialized processors.

Network processing systems are, in essence, queuing systems. Different components of the MPSoC devices, such as network interface cards, CPUs, and the global system scheduler, queue the packets in memory between the processing steps. Each component fetches the packets from the memory, or

queues, based on certain rules that can be seen as queue disciplines.

The treatment of the packets flowing through the network processing equipment is called packet processing. Packet processing consists of series of functions performed, typically in parallel, on the packets. The functions are often implemented as separate software applications. Different parallel programming models, such as task-parallelism, are used to abstract the complexity of these devices, enabling efficient application development and portability. In the packet processing context, the task parallelism can be seen as each task consisting of an operation (code) done on a packet (data). Again, the task parallel programming frameworks can be seen as a set of queues abstracting the underlying hardware.

Packet processing systems are complex. To keep up with the ever increasing performance requirements, the behavior of the systems needs to be understood. This thesis investigates the use of measurement, modeling, and simulation methods for the use of packet processing performance analysis. We present a way to model complex hardware and software interactions of a modern MPSoC network processing systems and task-based parallel programming frameworks, using an extended resource networks concept. Resource networks are based on the queuing networks concept, often used for packet processing problems and other computing system modeling, thus being a natural way of describing MPSoC systems and task-based parallel programming applications.

The tool chosen for the modeling and simulation is an in-house simulator and modeling software, Performance Simulation Environment (PSE). We will instrument and measure the characteristics of a modern packet processing system, and build a resource network model using PSE's built-in editors. We extend PSE to support user definable queue disciplines, enabling more detailed models of the systems global packet scheduler. We present two demonstrative experiments, showing that the model and PSE extensions work as expected. The models are simulated using PSE's discrete event simulator engine.

Our experiments demonstrate that, with extended custom queue disciplines and support for modeling task parallelism, resource network methodology is a viable approach for performance analysis of such packet processing systems.

1.1 Problem statement

Packet processing system performance analysis is non-trivial for several reasons. Modern MPSoC based packet processing systems are parallel pro-

cessing systems with complex hardware interactions. Understanding these systems is difficult, due to the architectural heterogeneity, complexity, and non-deterministic behavior. The non-deterministic behavior of the MPSoCs is a result of parallelism, communication delays, complex memory systems, and hardware and software scheduling mechanisms.

The nature of packet switched network data introduces strict performance requirements for the packet processing devices: the manipulation of the packet streams passing through the system must often be done on the on-the-fly, while at the same time, the data volumes are often huge and unpredictable.

The performance of the task parallel applications is heavily dependent on the scheduling mechanism of the system. To guarantee the performance of task parallel programming models, the scheduler has to efficiently map the functions and minimize the context switch overhead. MPSoC devices are inherently parallel, and the scheduling is often done globally at multiple points of the system.

Queuing and resource networks are widely studied concepts, often used for packet processing problems and other computing system modeling. However, the complex hardware level interactions between the MPSoC components, dynamic workload, and task-based parallelism methods, make the traditional queue concepts insufficient. One way to address this problem, is to extend resource networks with user-definable queue disciplines allowing global view of the system, and methods for modeling parallelism.

The research problem of this thesis is: How to analyze the performance of packet processing systems. Or more precisely: how to extend the resource network concept to support more accurate models of hardware accelerated many-core systems, and on the other hand, how to support modeling of task-based parallel programming applications on dynamic workloads.

1.2 Contributions

We will present a method for modeling and simulation of the hardware accelerated many-core packet processing systems. We extend the in-house discrete event simulator, Performance Simulation Environment (PSE), to enable use of customized queue discipline algorithms. We will also model a modern network processing system using the tools provided by PSE and simulate the model by PSE's discrete event simulation engine. The major contributions of our work can be summarized as:

- Instrumenting and measuring the characteristics of a modern network processing system

- Extending an existing in-house modeling and simulation framework, Performance Simulation Environment (PSE), to support user-definable queue disciplines through a plug-in interface
- Designing and building a PSE model of a MPSoC based packet processing system
- Proof of concept experiment demonstrating performance analysis of the implemented hardware model and task-based parallel application models

Our first contribution includes setting up an environment for measuring the memory and communication delays of a packet processing system. The measurements are used to gain in-depth understanding of the system interaction of such systems, and to further determine the correct abstraction level for our performance analysis models. The measurements consisted of executing several micro-benchmarks using low level hardware application programming interfaces. The automatization scripts and test programs are documented and available to be used in further experiments on similar systems.

The extension of PSE includes a new plug-in code mechanism, which allows customized queue discipline algorithms to be defined by external C-code. The design and implementation affected nearly all parts of the underlying simulation framework. In addition, PSE's memory usage has been improved to enable simulation of larger systems and workloads. Together the extension and other improvements consisted of roughly 700 edited and 600 new lines of code. These features are documented and available to be further utilized by PSE users.

Our third major contribution is the implementation of simulation model of a multi-blade network processing system. The models are based on our insights gained from the measurement and studies of a reference system and packet processing applications written on task-based programming frameworks. They consists of decoupled workload, software, and hardware models, highlighting the ability to decouple the different functional parts of the system, and enabling modularization and further reuse. The packet processing software applications are modeled using event based programming paradigms. The global packet scheduler, which is the key for modeling task-based applications, is modeled using the implemented plug-in code mechanism.

Finally, we have built a proof-of-concept experiment to validate the models and implemented PSE extensions. Further, it demonstrates how the implemented models can be used for performance analysis of a network pro-

cessing system. With the decoupled models, we are able to easily adjust the software parameters, affecting the sought performance metrics, such as latency and throughput, of the system.

1.3 Structure

Chapter 2 presents an overview to the context of this thesis. It motivates the performance analysis of packet processing systems. We will describe the reasons that have led the IT industry to widely adopt paradigms called cloud and fog computing. Further, we describe the cloud and fog computing together with the relevant technologies, such as virtualization and software defined networking, enabling these paradigms.

Next, in Chapter 3, we explain the concepts needed to understand the functionality of modern packet processing hardware and software, and their relation to queuing theory. We will present a more detailed view on task-based programming framework, called Open Event-Machine, and example of pipelined hardware architecture of network processing systems. The chapter finishes with characteristic measurements of a network processing system to gather in-depth understanding of such systems system.

In Chapter 4, we will present the basic concepts of performance analysis, modeling, and simulation. We will begin by presenting different evaluation techniques and performance metrics, further defining the components of system under a performance study. Then, we present the basic concepts of modeling, and queuing and resource networks, to underline their usage in traditional packet processing systems performance analysis. Finally, we will describe the simulation model and monitoring, with a short survey of the existing simulation software.

Chapter 5 presents the simulation tool, Performance Simulation Environment (PSE), used in our work. The chapter begins with an overview of the PSE's toolset. After that, we will describe the three main components of a PSE model. Finally, we will present the built-in discrete event simulator of PSE.

Chapter 6 presents the implemented plug-in code mechanism for Performance Simulation Environment. The extensions enable modeling of user-defined queue disciplines written in C-code, and is our attempt to address the lack of global queue scheduling, which is required to use PSE for more detailed modeling of hardware scheduled many-core systems.

The example model of a packet processing system is presented in Chapter 7. We will first describe the main characteristics of the model, and further describe a more detailed view of the global packet scheduler functionality.

The demonstrative experiments are presented in Chapter 8. We will describe the two experiment setups used, and their measurement results. We will analyze the experiment results and further discuss the challenges, discoveries, and the future work within the topic of the thesis.

The conclusion of our work are presented in Chapter 9.

Chapter 2

Computing Trends

This chapter presents the background of this thesis and motivates the need for, and the context of, efficient packet processing systems. We begin with an overview of today's challenges in hardware development, which have resulted in the transition from traditional single threaded computing to concurrent, multi-threaded and parallelized programming. Then, the dimensions of the required data processing capacity are emphasized, by explaining the concept named Big Data. Further, the characteristics of packet switched networks' data, and further the context of packet processing, i.e. cloud and fog computing, are explained.

2.1 Modern Computing

The terminology around the parallelized computation is somewhat ambiguous. Throughout this chapter, we will be using the term *parallel computing* as a general term referring to the opposite of *single-threaded* or *sequential* computing, unless explicitly stated to mean the act of *parallel computing*, in which multiple calculations are carried out *simultaneously*. The general term comprises many different types of computing, such as *concurrent*- and *distributed computing*, as well as the type of *parallel computing* itself.

2.1.1 The End of Free Lunch

Until the turn of the millennium, the main drivers for the speedups in computing were the increasing CPU clock frequency and cache sizes, as well as the execution optimizations. Faster clock speeds meant that more cycles could be run in the same time, whereas the execution optimization enabled more work to be done with the same amount of cycles. Further, the increas-

ing cache sizes kept larger parts of the computation close to the CPU cores, away from slow shared memory. [79]

A notable fact about all these drivers is that, while they lead to speedups in parallel computing, they also provide direct speedups for sequential programs. Partly for this reason, until the end of 1990s, the majority of the software applications were single-threaded and sequential. [79]

However, in the mid 2000s, the CPU performance gains hit the wall. Despite the fact that the transistor densities continued following the Moore's Law [61], several physical problems limited the development of higher clock speeds. Dennard scaling, a law proposed by in 1974 stating that the power density of transistors stays constant while the transistor themselves get smaller, seems to be broken down around 2005. As the transistors got even smaller, new current leakage and chip heating problems arose and the silicon industry hit the so-called *powerwall.* [28, 71, 79]

These chip manufacturing challenges have led into an era of *dark silicon*, meaning that we cannot afford to switch all the transistors on and off at each clock cycle due to the power constraints. Before the power wall, the forefront of the silicon industry was to maximize the chip speeds. However, in the era of dark silicon, the power consumption has become perhaps the most important performance indicator, especially in the datacenter scale computing. [7, 71, 79, 94]

And while the silicon industry are facing various hardware development challenges, the software developers are also forced to adopt new programming paradigms to keep up with the ever-increasing performance requirements. [79]

2.1.2 Parallel Computing

The parallelization of computing has been researched for several decades. Before the sequential execution speedup gains hit the wall, it remained a minor but important field of research, mainly in high-performance computing. Parallel-, concurrent-, and distributed computing quickly became the dominant answers for mainstream computing, to continue enjoying the benefits of the exponential computing speedups.

The *parallel computing type* refers to computing in which subparts of a program are solved at the same time. *Concurrent computing*, on the other hand, executes multiple computations on overlapping time periods. In *distributed computing*, the overlapping or simultaneous executions are carried out on multiple computers connected to each other.

There exist many different forms of parallel computing, often divided into four high level categories: bit-level, instruction-level, task, and data

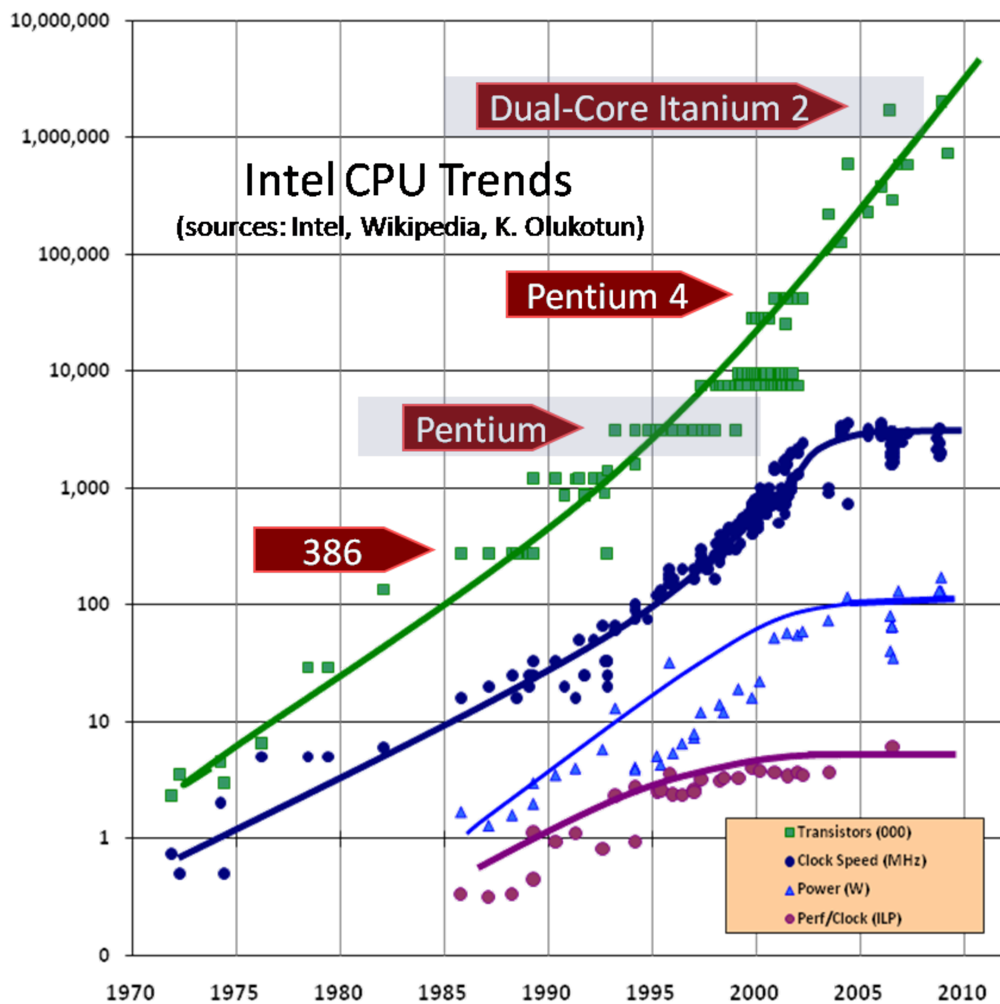


Figure 2.1: Intel CPU trends. The transistor density follows the Moore's Law, while the growth of CPU speed have stopped. [79]

parallelism. Parallelism can be implemented in both software and hardware levels. [20]

Bit-level parallelism refers to increasing the processor word size, thus reducing the needed number of instructions to be executed. In instruction level parallelism, the processor executes multiple instructions at the same time. Instruction level parallelism can happen in both hardware or software level. [20]

The data parallelism refers to the simultaneous execution of the same function, on multiple cores, across the elements of the input data. Modern graphics processing units (GPU's), often implemented as wide single instruction, multiple data (SIMD) principle, are examples of data parallelism. [20]

In contrast to data parallelism, task parallelism refers to execution of multiple cores of (possibly) completely different functions, across the same or different input data. In a typical multi-core processor, task parallelism is achieved by executing different threads or processes, on each core. [20]

Another form of parallelism, specific to the context of network processing, is called *Packet-level parallelism.* Network packets are typically processed on packet-by-packet basis. Most network packets are independent of each other, except for the ordering constraint for packets belonging to the same flow, thus enabling different forms of parallelism to be implemented in network processing units. [55]

Introducing parallelism into the computing brings various new challenges in the software development. For example, the lack of a global clock, and possible failure of components demands extensive care from the developers, on top of the already challenging software development. Different frameworks and methods have been implemented, to ease the software development of efficient parallel applications. The focus of this thesis is on task level parallelism, especially in the context of packet processing applications. We will present a view of a task-based programming framework, called Open Event-Machine, more comprehensively in Subsection 3.4. [7]

While implementing parallel systems is challenging for software developers, there are also clear limitations in the speedup gains that these methods can achieve. Amdahl's Law states the rather obvious maxima in the speedup that the program can achieve by scaling the computation over N processors. Suppose that the parallelizable proportion of a program is P (and thus the non-parallelizable proportion being P-1), then the maximum speedup, as denoted by Amdahl's Law, is: [6]

$$\frac{1}{(1-P)+\frac{P}{N}}$$

The implication of Amdahl's Law is that the speedup of a computer

program is always limited by the program's non-parallelizable proportion, thus bringing new challenges to the utilization of the available computing resources.

2.2 Big Data

The Internet traffic volumes are growing rapidly. For the first time, in 2010, the number of devices connected to the Internet (12.5 billion) passed the world's human population (6.8 billion). This growth is explained by the number of mobile devices (mobile phones and tablets, especially in developing countries), and the proliferation Internet of Things (IoT) devices and sensors. Cisco projects the number of mobile devices connected to the Internet, to grow up to 50 billion through 2020. IoT devices and sensors included, this number will double to over 100 billion. [29]

According to EMC corporation sponsored IDC study [82], the size of the digital universe is estimated to grow ten-fold, from 4.4 zettabytes to 44 zettabytes, between 2013 and 2020. The data comes from several different sources and in many different forms. At the same time the processing requirements are growing. Everyday end users are expecting services to be delivered in higher quality near real-time, and growing number of industry, medical, and other performance critical applications are depended on the processed data.

With the enormous data masses, data diversity, and with complex processing requirements, often referred as Big Data, the traditional data processing methods become inadequate and the new performance analysis solutions become more important.

2.3 Virtualization

Virtualization is an act of dividing a common set of computing resources into a multiple isolated execution environments. It enables multiple operating systems to be run, in parallel, on a single processing unit, thus alleviating the efficiency problems of parallel computing.

Before the existence of multi-user operating systems and the rapid drop in hardware cost around 1980s, the virtualization was used to allow multiple users to share the same mainframe hardware. Until the end of 1990s, it remained mainly a practice of computing industry and academic research, often requiring special hardware with explicit support for virtualization. VMWare's introduction of virtualization to the x86 architecture [89] and

the personal computer industry introduced the benefits of virtualization to the wider masses. [10, 14]

2.3.1 Platform Virtualization

In the traditional, non-virtualized system, the hardware platform resources are controlled by a single operating system. Virtualization introduces a new layer, the *hypervisor* (also referred as virtual machine monitor), to the software stack. The hypervisor lies underneath the existing software components, abstracting the hardware inputs, outputs, and the behavior for the use of multiple operating systems. The virtualized system is called a *virtual machine*. [10, 14, 83]

The benefits of the virtualization are numerous, mainly due to the software implementation of the virtual machines. It enables scalability and flexible migration of computation loads across different infrastructures, leading to improved hardware utilization, dynamic resource allocation and management, isolation, security, and automation. [66]

The virtual machines can be *live migrated* over to another physical machine. While the non-virtualized systems are often under utilized, the flexibility of software enables the machines often to be run on optimal usage level. Maintenance costs can be reduced by software automation and security can be increased by additional software services, such as workload isolation. These benefits have encouraged companies to seek savings, by adopting virtualization in various contexts, ranging from the desktops and datacenters to network switching. [66, 83]

To enable fully functional virtualized environments, the virtualization of different system components, such as CPU, memory, I/O, and storage, need to be considered. CPU virtualization techniques are often divided into three categories: binary translation (full virtualization), paravirtualization, and hardware-assisted virtualization. [14, 41, 66]

Full virtualization refers to a method where the communication between the virtualized operating system (guest) and the underlying host operating system is (nearly) completely emulated, meaning that the virtual hardware is functionally identical to the underlying machine. Unmodified guest operating systems can be run similarly as on native hardware, as the virtualization layer completely decouples them from the underlying hardware. On the other hand, the complete emulation of the hardware instructions induces performance overhead to the full virtualization. [10, 14, 41]

In paravirtualization, instead of emulating the hardware environment, the guest operating systems are executed in isolated environments. It enables the communication between the guest operating system and the hypervisor, and

thus improving the performance and efficiency of the virtualization. [10, 41]

However, paravirtualization requires modifying the guest operating system kernel to enable direct communication with the virtualization layer monitor. For this reason, paravirtualization compatibility and portability is limited, and it can cause significant support and maintainability costs. [10, 41]

Hardware assisted virtualization introduces new hardware features to ease the virtualization. A common method is to provide a new privilege level below the ring 0, to allow the guest operating system to intercept and emulate privileged operations of the underlying hardware. Hardware assisted virtualization removes the need for binary translation and paravirtualization, thus, at least theoretically, solving many of the current virtualization problems. [41, 66]

2.3.2 Operating System Level Virtualization

Running multiple operating systems on single hardware, as done with hypervisor-based virtualization, comes with the cost of efficiency. In many scenarios, such as high performance computing, game hosting, or MapReduce [21], the overhead from running multiple kernels on the same machine becomes a problem. Another common method for server virtualization is to run the virtualization layer within the operating system. [77, 92]

Operating system level virtualization is based on the kernel's ability to support multiple isolated user-space instances, or software *containers*. Instead of emulation, programs running in containers consume the host operating system's standard system call interface, resulting in negligible virtualization overhead compared to hypervisor-based virtualization. The drawback of container-based virtualization is its inflexibility; each guest must be running on the same operating system kernel as the host machine. [77, 92]

Examples of operating system level virtualization implementations include for example Linux Containers (LXC) [19], Linux-VServer [24], OpenVZ [63], and Docker [60]. While most of the container implementations seem to achieve near-native performance, the management capabilities, such as, performance isolation, vary between them.

2.4 Cloud Computing

Cloud computing paradigm provides a shared pool of easily configurable and flexible computing resources via convenient, on-demand network access. One of the driving forces for cloud computing has been its promise of economics of scale. Virtualization techniques enable cloud computing centers, compared

to traditional on-premise solutions, to be built using cheaper hardware, cooling, electricity, network capacity and smaller number of administrators per computer. At the same time, it alleviates the problem of inefficient resource usage, caused by the challenges discussed in Section 2.1.1. [58]

According to the definition of The National Institute of Standards and Technology, cloud computing is composed of five essential characteristics (on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service), three service models (software as a service, platform as a service, and infrastructure as a service), and four deployment models (private cloud, public cloud, community cloud, and hybrid cloud). [58]

The *on-demand self-service* characteristic means that the consumer of the service can provision computing resources, such as server time and network storage, automatically without requiring human interaction with each service provider. *Broad network access* requires the service to be available over the network and accessible using standard mechanisms, which promote use by heterogeneous platforms such as mobile phones, tablets, laptops, or personal computers. *Resource pooling* refers to virtualization techniques discussed in Section 2.3. *Rapid elasticity* means that the consumer is able to elastically and automatically, provision and release the seemingly unlimited resources on demand. The last characteristic, *measured service*, refers to the automatic control and optimization of the resources by metering the usage of the services. [58]

The characteristics of cloud computing make it tempting for both customers and service providers. From the customer perspective, it provides advantages of high computing power, cheap service costs, high performance, flexible scalability and accessibility as well as high availability. It reduces the up-front infrastructure costs for companies, allowing them to focus on their core businesses instead of on infrastructure, and getting their applications running faster, with improved manageability and less maintenance. On the other hand, it provides completely new business models for the service providers. [25, 58]

The three service models divide the cloud services into logical levels based on the computing layers provided by the cloud service. Figure 2.2 presents the comparison these levels together with the on-premise solution. On-premise solution refers to model where the customer manages the complete computation stack.

The lowest of the three levels, *Infrastructure as a service (IaaS)* model, provides the customer the services, which abstract the details of typical hardware resources and operating systems. The computing resources are typically abstracted using a hypervisor, such as VirtualBox, KVM, Hyper-V that runs the virtual machines visible to customers as guests. Examples of

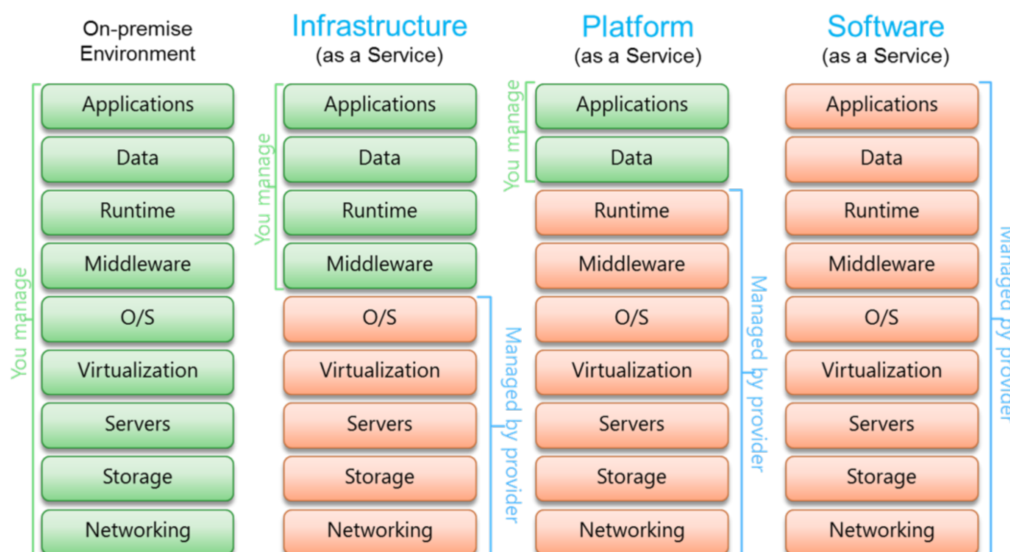


Figure 2.2: A comparison of the three cloud service models and on-premise solution. Figure from [54]

IaaS providers are Amazon Web Services [62] and Rackspace [69]. [58]

In the next service level, *Platform as a service (PaaS)* model, the service provider takes care of the platform-level middleware and runtime components. These components typically include programming-language execution environment, databases, and web servers. Google AppEngine [74] and Windows Azure Platform [70] are examples of PaaS providers. [58]

Software as a service (SaaS) model provides, on top of the IaaS and PaaS, the management of the application and data level. The customers access the software from cloud clients, typically web browsers via personal computers, laptops, tablets or smartphones. SaaS has become a common model for delivering wide range of different applications, such as Facebook [30], Salesforce [73], and Google Gmail [80]. [58]

Private Cloud infrastructure is hosted internally by a company or organization, targeting a specific set of customers. Private clouds are typically more affordable to setup and offer more flexibility, while also providing the companies full privacy on their data and applications. *Public Cloud*, on the other hand, is exposed to any customer over a public network. The data, applications and other resources are stored over the service provider's data center, and thus the security has to be considered. The underlying infrastructure is often similar between these two models. [58]

The cloud service can also be a combination of the public and private

clouds. This kind of deployment model is referred to as a *hybrid cloud*. Hybrid cloud allows the utilization of the flexibility and cost of the public cloud solutions, while at the same time maintaining the security sensitive data in their own infrastructure. *Community cloud* refers to cloud infrastructure that is shared between multiple organizations, typically with common security, jurisdiction, and compliance concerns in mind. Community clouds can be hosted and managed either internally or externally from the sharing organizations. [58]

2.4.1 Energy Consumption

The power consumption has become the major concerns in the design of today's warehouse scale datacenters. In 2005, the world's server power demand (including cooling and auxiliary infrastructure) was about 14,000MW, resulting in about $7.2B electricity costs. Majority of these costs come from cooling, which is the consequence of the heat generated by the computing resources such as CPU, memory, storage, and networking. [31, 52]

Virtualized datacenters can greatly increase the usage rates of computing resources, decreasing the total server energy consumption. However, the benefits of cloud computing are tightly coupled to the scale of the individual datacenters. While the total energy consumption is reduced, the energy usage of individual datacenters is often huge. [40]

The datacenter costs can be optimized by more efficient hardware implementations and increased usage rates, but companies also have to consider different non-technical solutions. The design and construction of warehouse scale datacenters should be treated similarly to traditional factories. Size, location, and physical design of the datacenter can have considerable effect on the costs. [40]

Google's Summa datacenter is a good example of the scale of the cloud datacenter. Google, a company residing in California, invested over $200M to build a datacenter in Hamina, Finland. The benefits from Finland's energy infrastructure, cold climate, developable land and available work force, outweighed all the negative qualities of building the datacenter on the other continent, nearly 9000km away from the company's head quarters. [35]

2.4.2 Datacenter Networks

For compatibility and cost reasons, datacenter networks are often built from commodity Ethernet switches and routers (scaling out), rather than from expensive high-end hardware (scaling in). Due to the limits in the switch port

densities even in the high-end hardware, the traditional single rooted topologies are replaced with multi-rooted tree topologies such as fat-tree or leaf-spine. Multi-rooted network design allows scaling the data center networks' bisection bandwidth by adding new spine switches into the network. [3, 84]

Cloud computing datacenters can consist of hundreds of thousands of servers, supporting large variety of services, such as high performance computing, MapReduce [21], and web services. Many of these applications have several inter-dependent components divided across the servers. Thus, the inter-node network bandwidth has become the major bottleneck in data centers. [3]

To efficiently utilize datacenter resources, and to answer the required performance guarantees, efficient network traffic load balancing mechanisms are needed. Several different proposals, such as CONGA [5], Presto [39], and Hedera [4], have been made to overcome the issues of traditional hash based schemes.

2.5 Fog Computing

While the benefits of Cloud Computing provide efficient alternative to the traditional on-premise solutions, there are certain issues, which have to be addressed to enable a new breed of ever demanding applications and services. Characteristics, such as mobility, geo-distribution, location awareness and low latency are intractable to achieve due to the centralized nature of cloud computing. The proliferation of devices and sensors has led to a situation where the data are produced faster than can be transmitted, stored, or processed. [12, 85]

Fog computing is an extension to the traditional cloud computing architecture, where parts of the cloud computing services, mainly computation, storage, and networking, are carried out in the edge of the communication network. It is defined to provide the following characteristics on top of the existing cloud computing architectures: low latency and location awareness; wide-spread geographical distribution; mobility; very large number of nodes, predominant role of wireless access, strong presence of streaming and real time applications, heterogeneity. [12]

High virtualization and efficient stream processing are the key elements for successful fog computing.

Until recently, the typical network architectures have been implemented on proprietary or special purpose hardware. The growing networking traffic and competition in communication services have led to the development of software based solutions. Software based solutions enable flexible and

extensible networking, and measure up the tight standards for stability, protocol adherence, and quality, previously achieved only with the hardware solutions. [50]

The main enabling technologies of fog computing are the software-defined networking (SDN), and further the network functions virtualization (NFV). Software defined networking separates the data plane and the control plane functionalities of the network, making the data plane switches simple packet forwarding devices, while leaving the routing decision control logic for the control plane. Thus the network control becomes directly programmable, centrally managed, programmatically configurable, and vendor agnostic, amongst many other benefits from softwareization. [50]

SDN is often complemented with network functions virtualization. In NFV, the network devices are virtualized (using similar techniques as discussed in Section 2.3) and the network functionalities of the devices are implemented in software packages. [23]

Long Term Evolution (LTE) [75] and Evolved Packet Core (EPC) work as a natural platforms for the fog's edge data centers. Small cloud stations can be deployed to the EPCs, and the routers can be utilized as the virtualization infrastructure. This also enables the application services to be co-located where needed. [85]

Chapter 3

Packet Processing

Packet processing refers to the methods and concepts applied to transport data packets through the various network elements in a communication network. The network routers, switches, and other devices, such as computers and smartphones, all have their own packet processing subsystems to manage the packet traversal between the network elements. This thesis focuses on the packet routing equipment in the middle of Internet backbone.

This chapter presents the general methodology of packet processing and network processing systems. We begin by describing the general packet processing framework used in network processors. After that, we present the packet flow handling methods, emphasizing the importance of packet buffering and queuing theory in the packet processing. Further, we present an overview of the hardware architecture and task-based programming models used in packet processing systems.

Finally we will have a closer look on a specific network processing unit, and instrument the system to obtain the required understanding to choose the abstraction level for the performance analysis model. Two different measurements will be done: one to measure the communication latencies, and another to measure the memory latencies and throughput.

3.1 Packet Switched Networks

Packet switching refers to a method of transmitting data in separate, suitable sized blocks, called packets, between the links of a telecommunications networks. The importance of packet switched networks is increasing in every part of the communication networks, which is explained partly by the Internet itself being an enormous packet switched network.

3.1.1 Network Components

The Open Systems Interconnection (OSI) model is a standard conceptual model, which characterizes the communication functions involved in a general network communications of computer system. The model defines seven hierarchical layers of functional elements, between the physical interconnections (layer 1) and the software applications (layer 7). [45]

The Internet Protocol Suite, or Transmission Control Protocol / Internet Protocol (TCP/IP), is a set of core protocols for the Internet and similar packet switched networks. It provides end-to-end data communication and specifies data packetizing, addressing, transmission, routing, and receiving in the packet network. In addition to TCP and IP protocols, it provides several other network and transmission layer functionalities. The functionalities are divided into four abstraction layers: application layer, transport layer, internet layer, and link layer. [13]

Internet Protocol (IP) is an essential Internet layer protocol of the Internet Protocol Suite. It provides the necessary addresses and (unreliable) delivery mechanisms for datagrams between the transmission endpoints. It is a connectionless, best-effort protocol with no error control, sequencing, or end-to-end reliability, but rather contains only the functions for packet fragmentation and delivery through the network. The packets in the IP protocol can be forwarded independent of other packets, forwarding them on-the-fly by routers. [22]

A packet refers to the unit of data transmitted across the network. Packets consist of two parts: the header information and the payload data. Network routers use the header information to direct the packet to destination system, whereas the payload is extracted and used by the application software. All the higher layer protocols in the TCP/IP stack are encapsulated in the IP-packet's payload section. [22]

In conventional IP-based network architecture, the packet processing equipment and functionality is partitioned into three components: management plane, control plane, and data plane. [16]

The control plane refers to the parts of the routing architecture, which carries and consumes the control packets needed to describe the network topology and correctly route the actual data packets. The control packets originate from and are destined for a router. [16, 93]

The data plane, often referred as forwarding plane, deals with the actual data forwarding in the network. It defines the part of the routing architecture, which decides destination and takes action for the data arriving to it. The decision is typically determined by a look-up table, which the incoming packet is compared to. The management plane carries the traffic used for

the administrative tasks of the network. [16, 93]

3.1.2 Traffic Characteristics

The nature of the high-speed packet processing entails new types of constraints to the computing; advanced stream processing systems are needed to process the data on-the-fly in the vicinity of the sources. [12]

In the traditional von Neumann model of computing [87], the computation is carried out by modifying the data stored in memory. However, the growing volumes of data and strict latency requirements, require not only distributing the computation, but also changes the nature of it more towards stream processing. The manipulation of the data streams passing through the system must often be done on the on the fly. [12, 81]

In packet switched networks, the data streams are called *flows*. Flows represent the data in specific time periods between specific network end-points. A single flow contains a large number of packets. The Internet Protocol allows each individual packet of a stream to be processed in any order, leaving the ordering of the packets to the end devices. This characteristic makes packet processing applications look like ideal for a parallel computing. However, protocols such as TCP/IP (accounting over 80% of the Internet traffic [32]), have significant performance reductions due to reordering of packets in a single stream. For this reason, the packet processing systems often have built-in schemes, such as specialized queue management and buffering, to efficiently keep flow specific packets traversing through the queues in order. [36, 53]

3.2 General Packet Processing Framework

A general packet processing framework consists of three primary aspects: packet processing functions, packet direction (ingress, egress, or combination of both), and packet processing paths. These three aspects are depicted in Figure 3.1. The packets enter the system from the left, from the physical line interface (ingress direction), and follow either the slow or the fast path. The packets are then forwarded either to the switch fabric or back to the physical line interface (egress direction). [34]

3.2.1 Ingress and Egress

Ingress and egress refer to the packet processing done for the packets entering the network processor from the network, or packets leaving the network processor to the network, respectively. Ingress and egress processing, although

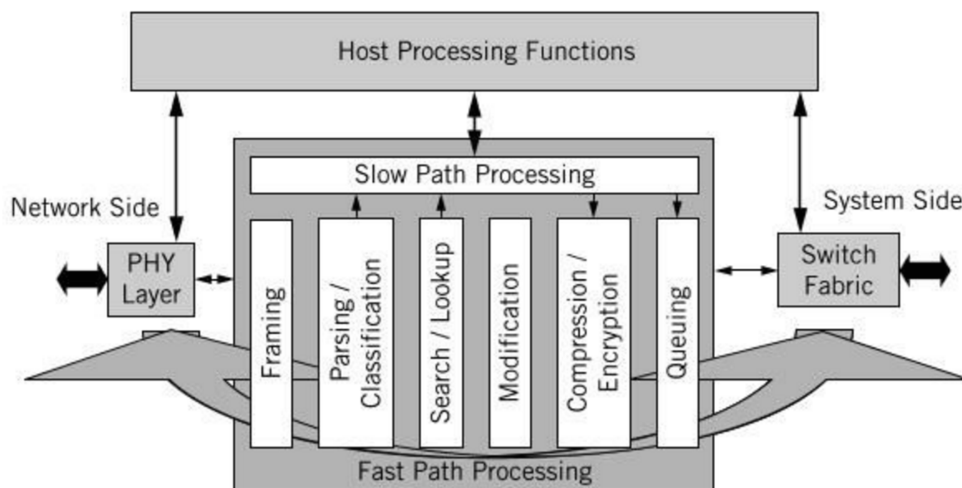


Figure 3.1: A general packet processing framework. Figure from [34]

not necessarily being clearly separated in modern systems, play an important role in the network processing units. The unclear separation of these phases is due to the various different implementations in today's network processing units. Some processors have one processing direction, from packet input to its output, possibly happening through the same interface. Other implementations might not have distinguishable elements that target ingress or egress processing. [34]

Figure 3.2 outlines the basic two-part equipment implementation scheme, called half-duplex, for packet ingress-egress processing. The first part of the picture depicts the line cards for receiving/transmitting the packets from/to the network. The second part consists of switching fabric, service cards, and other processing functions and mechanisms that packets go through internally.

Half-duplex processing consists of two dedicated processors for each direction. The functions executed on the ingress packets and egress packets can be distinguished, in which case the network processing unit often consists of separated processing paths for the two ways. Typical ingress processing tasks consists of, for example, error checking, classification, traffic management, header manipulations, and prioritization and queuing. Packet egress tasks, on the other hand, include checksum calculation, address lookup, packet forwarding, segmentation and fragmentation, traffic management, and packet prioritization and queuing. [34]

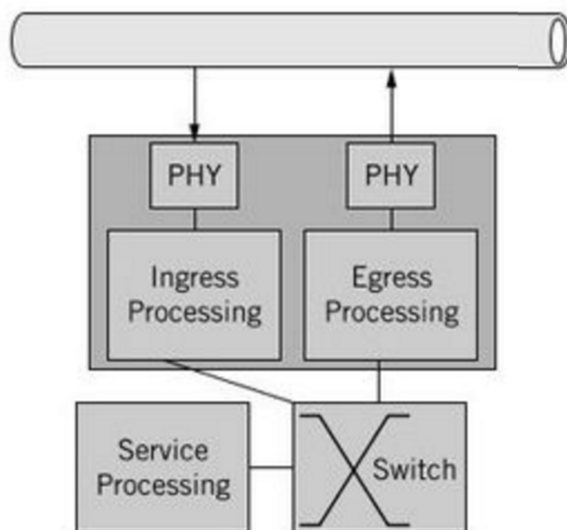


Figure 3.2: Two part ingres-egress scheme. Figure from [34]

3.2.2 Processing Paths

In many cases, the packets flowing through the network processing unit need to be processed at wire speed, meaning that the number of packet leaving and arriving the processing unit must be equal at certain time interval. The introduction of increasing Ethernet speeds imposes strict constraints for the network processors. While the control- and management plane traffic can often be processed with fairly non-trivial solutions, the main bottleneck, to keep up with the scalability of optical transmission technology, is the data plane processing. [34]

The comparison of the speeds of Ethernet based transport networks and the processor speeds gives a perspective of the packet processing requirements. A typical Ethernet link in the network backbone, with 100Gbps speed and minimum frame size of 64 bytes, leaves the packet processing system 6.7ns to process each packet. [34]

Even in the most trivial packet processing applications, this processing requires several different processing steps. For each packet, the network processor must execute complex packet parsing, to parse destination address and port. Then, based on this information, the processor executes several searches to retrieve the destination details from the memory, possibly containing hundreds or thousands addresses. It is evident that completely software based slow path solutions are insufficient to answer these requirements. [34]

Fast path architecture refers to a path through a computer program, which incorporates smaller number of instructions or other optimization methods, compared to the normal path. In packet processing systems, the vast majority of packets require very little processing. Thus, the data processing is often split into two, referring to the data plane and control plane processing: fast path and slow path. [1, 34]

The typical slow path of the packet processor is run on top of an operating system stack. The fast path layer processes packet outside the operating system environment, often with hardware acceleration, thus avoiding the overheads occurring from the thick software stack. This leaves only a small number of packets that require special processing, to be forwarded to the slow path able to do more complex processing. Typical examples of slow path packets in IP network are IP options and ARP packets. In MPSoC based packet processing systems, the processing cores can often be dynamically configured to run fast path or slow path. [1, 34]

3.2.3 Packet Processing Functions

Packet processing functions are separate tasks, following each other, in the packet processing path of packet processor. These functions can be categorized, for example, as framing, parsing and classification, search/lookup/forwarding, packet modification, compression and encryption, and queuing and traffic management. [34]

The packet processing starts with the packet entering the network processor through the network interface controller's ingress port, immediately followed by the packet *framing*. Framing assures that packets or datagrams can be correctly extracted from the incoming data frames. The incoming data frames are tested for correctness, to make sure all the bits in the frames are received correctly. If necessary, the framing functions attempt to fix the bits of the incorrect frames. Finally the integrity of the frames is validated, to make sure that all the packets' content arrived correctly. [34]

Framing is done similarly in the egress direction, in order to assure correctness of out going packets. In the outgoing framing phase, the needed headers are attached or modified, proper terminals and trailers are added to the packet data, and error detection and correction information is appended. [34]

After framing, the packet is *parsed and classified*, meaning that the network processing unit inspects the packet data in order to understand its type, and then classifies it according to the application requirements. Parsing and classification are two combined subtasks, sometimes carried out either separately or together, depending on the system. [34]

Packet parsing can be very simple and trivial, or it can be a complex task requiring unique language to describe and dedicated hardware to execute the process. Parsing means of identifying the relevant fields in the incoming data packet. These field's values are then used for either further packet parsing or classification, which is why these two are sometimes tied together. [34]

Classification refers to the functions that categorize the packets into separate flows, by the rules defined in the system. Each packet belonging to a specific flow, then take the same processing steps. Packet classification can be stateful or stateless, static or dynamic, and the fields to be classified have variable of fixed lengths and offsets. In stateless classification, the decision is made solely based on the packet content, independent of other packets. In stateful classification, the system state changes based on the processed packets, and affect the classification decisions. In static classification all the classification criteria are predefined, and the rules are fixed. In dynamic classification, on the other hand, the classification criteria and rules are computed based on, for example, incoming packet or system's state. Packet classification can be implemented in hardware or software, depending of the required complexity. [34]

Search and *lookup* functions are not packet processing phases themselves, like the other discussed packet processing functions. Rather, they are atomic operations used during the packet processing, such as classification, *forwarding*, or any other phase. Almost every packet processing activity starts with an IP-lookup, thus making the search and lookup functions are probably the most important packet processing operations, in terms of effect to processing speed, in packet processing. Various different solutions, often referred to as *search engines*, have been developed to enable searches at wire speeds. Search engines are software processes or hardware solutions, often packed in separate co-processors. [34]

In the *packet modification* phase, the network processor drops or modifies the packet being processed, or possibly generates new packets as specified by the application. Packet modification is one of the key operations in many packet processing applications, especially in the ones other packet forwarding. Packet modification phase is also used for some traffic analysis, management and statistics collection tasks. [34]

Some packet processing systems *compress and encrypt* the packets before they leave the system. Compression and encryption are often done in the access network processors, whereas in the trunks of the network core, the packets just flow through. Compression is typically used in networks and applications where the bandwidth is limited, and security is used for privacy, data integrity, and authentication purposes. [34]

Finally, in the packet transmission phase, *queuing, prioritization, and*

traffic management are used to make sure the traffic patterns are as expected. The traffic management process forwards the packets to appropriate output queues and schedules them for transmission, according to the line and receiver conditions, and the parameters, such as priorities, of the packets. Traffic management is also used to meter the packets, and transmit the packets on a desired rate and burstiness. Due to the complexity of traffic management, it is often implemented by a dedicated traffic manager co-processor. In some cases, packets incoming to the system also go through a traffic management phase. [34]

3.3 Processing Hardware

Traditionally, the focus in the development of network equipment – switches, routers, and various other middleboxes – has been to achieve high performance with very limited packet processing functionalities. The network appliances and middleboxes have typically been deployed on special purpose Application-Specific Integrated Circuits (ASIC) hardware, mainly due to high performance requirements compared to the hardware development, and the absence of extensibility and flexibility requirements. [26]

However, the required network functionality is becoming increasingly sophisticated. In addition to the traditional layer 2 and layer 3 routing and switching tasks, the network elements are required to handle more demanding tasks (e.g. application acceleration, encryption, and intrusion detection) all the way to layer 7. These extensions often require modified per-packet processing on the router data plane, and as the specialized network processors are difficult to extend and program, both industry and research are seeking new, more flexible networking solutions. [26, 27]

On the other extreme of the design spectrum, are the software routers built from general-purpose operating systems and x86 hardware platforms. The main promise of the software routers is their extensibility in the software and hardware development: the system's network functionalities, both in the data and control plane, can be modified fully by the software, thus mitigating the hardware design and development burden of the network developers. [26]

The software routers also enable several other properties of the general purpose computer ecosystem. Huge manufacturing volumes and widespread supply and support chain allow much cheaper hardware prices. The rapid advances in commodity semiconductor technology, such as the state-of-the-art power management features, can provide significant advantages over the slow hardware upgrade cycles of the traditional ASIC based systems. [26]

The challenge with the full software routers is the scaling the approach

to high-speed networks. Between these two extremes exist design solutions, Network Processing Units (NPU), which offer much of the programmability of commodity hardware and the performance of specialized ASIC systems. [26]

The focus of this thesis is on the middle-ground network processors, between the specialized ASIC solutions and general purpose x86 hardware, which are used for mid- and high-level used for communication, datacenter, and higher OSI-level applications, and in some cases for the Internet core networking. Thus, this section's point of view is on these elements.

In high-speed networking environments, a single network processing unit is not sufficient for executing all the different processing functions, but instead, different parallel processing and multiprocessor architectures are often considered. Nearly all network processing units today are multiprocessor system-on-chip architectures, meaning that they are built from several small processors working in parallel, co-operating, on the same silicon chip. The programmable processor of the network processing unit is referred to as *processing element*. [34, 64]

3.3.1 Processing Elements

Various architectural design choices can be made at different system levels. The programmable processing elements themselves can be designed in several ways. The basic requirements for processing element are a basic instruction set, memory and a data path suitable for multi-element environment operation. [34]

A processing element can implemented in many forms, and the architecture depends much of the intended environment. They can be reduced Instruction Set Computing (RISC) CPUs (typically programmable), or dedicated hardware units (typically non-programmable) for specific packet processing tasks, such as classification, per-flow queuing, buffer and traffic management, which require wire-speed processing. [34]

Processing elements can operate independently, or be grouped into functional blocks. Similarly as in the design of network processors, there is often a trade-off between the functionality and flexibility of processing element, and the speed. Whether the processing elements are complex multi-threaded units, or small and simple, in typical network processor architecture, the goal is to achieve high performance by maximizing the utilization of the on-chip elements. Thus the elements should be configured to correspond the needs of the network processor. [34]

Hardware accelerators and co-processors refer to state machines that are specialized to a certain packet processing tasks, such as cyclic redundancy check, search and lookups, or security. Hardware accelerators operate inde-

pendently of the network processor's processing elements, and are called as a functional unit from other elements. Co-processors are hardware accelerators that are themselves programmable. [34]

3.3.2 Parallel and Pipelined Architectures

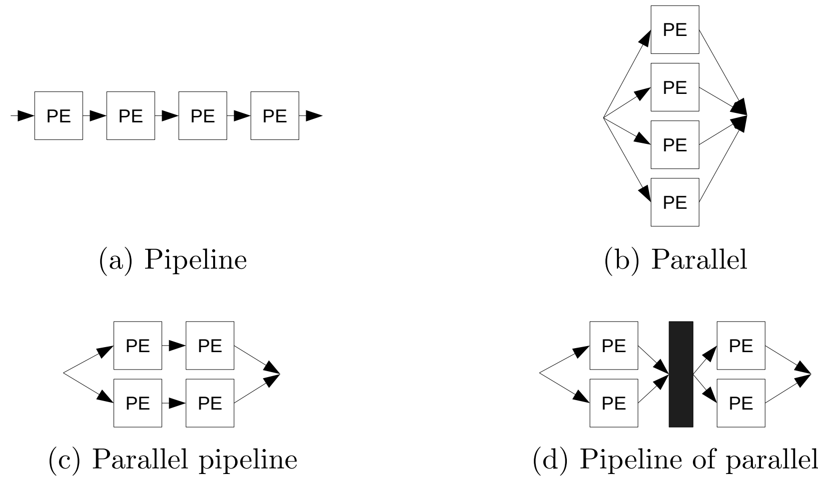
The arrangement, or topology, of processing elements can be divided in two fundamentally different approaches: parallel and pipeline architectures. Depending on the application, the topology can consists of multiple homogeneous or heterogeneous processing elements, organized in parallel, pipeline, or a combination of these two. Figure 3.3 depicts the four different combinations of the architectures. [34]

In pipelined architecture (Figure 3.3a), several (homogeneous or heterogeneous) processors are organized in one after each other, forming a number of processing steps that every packet goes through. The pipeline is often implemented by heterogeneous processing elements, each of which do part of the total packet processing work. The advantage of the pipelined architecture is to have a dedicated, specialized processing element for packet processing tasks. Also, a fully pipelined architecture forces the packets to be ordered in the same order as they enter, making the behavior of the packet stream more predictable. In an ideal pipeline, each stage should require the same time to perform. The packet processing tasks, however, are often complex and dynamic by their nature, and in heterogeneous implementations different elements require separate programming models. Thus the pipeline implementations are often more difficult than a typical parallel architecture. [34]

The parallel processing (in this context) refers to a topology where several processing elements work in parallel on the packets dedicated to them. In the extreme case (Figure 3.3b), the entire packet processing is done by the same processing element. The extreme case is typically implemented on homogeneous, "general purpose" packet processors capable of executing all the required packet processing tasks. [34]

It is typical to combine these to approaches into parallel pipeline of processors (Figure 3.3c) or pipeline of parallel processors (Figure 3.3d). In a pool of parallel pipeline, each incoming packet is directed to one of the pipelines, which processes in steps, until the processing is finished. This configuration is a compromise between strict parallel processors and strict pipeline of processors, allowing each pipeline to execute different subtasks. [34]

In a pipeline of parallel processors, pipeline consists multiple processing steps carried out with a multiple parallel elements. Typically, the processing elements within each parallel step are homogeneous, whereas the elements between different steps can be heterogeneous. The time taken for processing



(a) Pipeline (b) Parallel

(c) Parallel pipeline (d) Pipeline of parallel

Figure 3.3: Different processing element architectures

can vary between the stages and between the processors of each stage, while reducing the waiting times for the processing, and the need of synchronization and buffering. [34]

The parallel configuration is typically used for higher layer networking applications, where as the pipeline configurations are used in line cards that require high-speed processing at low layer networking applications.

3.4 Programming Models

Programming of the high-speed packet processing systems is challenging, due to the parallel nature of the hardware, and the dynamicity of the packet streams. Generic software frameworks are often used to abstract the underlying hardware complexity, and ease the parallel application development. Abstractions try to hide the details of hardware implementation from applications developers, providing ease of development and portability of code through different hardware implementations.

We will further present two different software frameworks, Intel Data Plane Development Kit (DPDK) and Open Event-Machine (OpenEM), used for packet processing development. Both of these frameworks share the same fundamental idea of abstraction: they provide software queues, which are mapped to the hardware components of the system. Software queues provide flexible way of scheduling and ordering the packets between the processing cores between the packet processing tasks.

The difference between the frameworks is their level of abstraction and the method of describing parallelism. Intel DPDK provides lower level abstrac-

tions targeted for Intel hardware. OpenEM, on the other hand, is a higher-level abstraction, which implements more strict queue types and higher-level scheduler on top of the lower queue abstraction frameworks, such as DPDK for Intel hardware. In this thesis, we are mainly interested in the OpenEM type task-based parallel programming models.

3.4.1 Intel Data Plane Development Kit

Intel Data Plane Development Kit (DPDK) provides a clean application programming interface and a set of coherent libraries and drivers, for Intel x86 processors. It has a generic support for many CPU's and network interface controllers ranging from Intel Atom processors to Intel Xeon processors. It supports system with and without non-uniform memory access (NUMA), and any number of processing cores. [44]

DPDK runs as a Linux user-space application, utilizing the pthread library. Similarly to many other packet processing frameworks, DPDK implements a run-to-completion model, to minimize the process-switching overhead. The model removes the typical operating system scheduler, mitigating the context switch overhead. All the devices must be accessed by polling, thus eliminating the interrupt overhead. The packets can be passed between the cores, enabling more efficient and flexible core usage. [44]

DPDK's Environment Abstraction Layer (EAL) abstracts the hardware environment from applications and libraries, enabling hardware agnostic implementation of packet processing applications. EAL provides services such as core affinity and assignment procedures, memory management, atomic and lock operations, and bus accesses, and interrupt handling. These features are exposed as programming libraries. [44]

DPDK has an active ecosystem around it, with wide vendor support. It is also well documented and includes several software examples demonstrating the best practices for data plane architectures, application profiling, and performance tuning. [44]

3.4.2 Open Event-Machine

Open Event Machine (OpenEM) is an event-driven programming framework for multi-core dataplane applications, developed by Nokia Solutions and Networks (NSN). It has been designed to ease the implementation of event and packet processing applications for different MPSoC devices. One of main drivers for the development of OpenEM has been easy integration with modern hardware accelerators. [88]

The key concepts of OpenEM framework are execution objects, events, queues, and the scheduler. *Execution objects* are the main building blocks the OpenEM application. They are the run-to-completion functions, describing the processing logic of the application. Each execution object has one or more *queues* attached to it. The *scheduler* selects the events from the queues, based on the global interrelations of the queues. When the event is selected from the queue, the corresponding execution object is attached to the processing core, and the event is passed to it as a parameter. Once the execution object finishes its run, the scheduler chooses a new event from the queues similarly. [88]

The OpenEM framework itself is easily portable across different hardware. As the OpenEM specification is based on used lower level queue abstractions, such as DPDK, it can easily be implemented on platforms supporting queue abstraction frameworks. Some hardware vendors, such as Texas Instruments [43], offer ready-made OpenEM support on their hardware. OpenEM's reference implementation is based the Intel DPDK framework. [88]

From the programmer's perspective, OpenEM's event-driven programming model relates closely to actor based programming models such as Erlang [86] and Akka framework [2]. While the use cases for these frameworks are completely different, and thus cannot directly be compared with OpenEM, it is worth mentioning their message passing support. Support for combined inter-node and intra-node parallelism has potential benefits for efficient scaling in the future. HCMPI [17] is an example of experimental framework that combines task-parallelism with message passing. OpenEM's current specification does support inter-node messaging, however it is not clearly defined.

3.5 Example Network Processing System

Typical network processing units are optimized for high-performance, high-bandwidth, and low power consumption software-defined control-plane and data-plane applications.

The ingress and egress processing are handled by separate processing elements, having input processors that work together to manage the received packets, and to perform required processing before scheduling the packets to application cores. Once the required ingress computation is done, an input processor sends the packet's work entry to the scheduling unit to be scheduled for processing.

The egress functions are handled similarly by specific packet transmission units. When a core finishes a packet processing, it notifies the output

processor that the packet is ready for transmission. The output processor then directly copies the packet data from the shared memory into its internal memory, optionally computes checksums for the packet header, transmits the packet, and optionally frees the packet data from the memory.

Our example system consists of multiple application cores, with several hardware acceleration units for enhanced packet processing and minimized software development complexity. The packet management accelerators offload the actual packet processing cores from many general packet receive, buffering, buffer management, flow classification, quality of service, and transmit processing. The accelerator functions can be customized using software, and accessing the configuration registers. Together with the hardware acceleration units, the processing cores can handle most of the processing requirements of all the way through layer 2 to layer 7 in the standard OSI model [45].

One of the key features of the unit is its global packet scheduling unit, which is responsible or the packet scheduling and synchronization. It frees the actual packet processing applications, running on the application cores, from the complex packet scheduling and ordering tasks. The cores execute a loop, and when a core is ready for the next packet, it requests work from the scheduler, which then schedules the next work based on the quality of service priority and work group.

The scheduler has efficient locking mechanisms for protecting the critical regions without explicit software locking, and allows packet processing to be done in parallel or atomically, while still maintaining the packet flow order. The processing cores can also be dedicated for specific flows. One of our goals is to be able to model the scheduling functionality with PSE, as it is one of the key elements in the packet latency and throughput when processing several flows at the time.

The network processing unit provides several memory policies for optimized multi-core packet processing. The cores often have dedicated L1 data and L1 instruction cache, and a shared L2 cache. Different cache features and policies are offered, for example to avoid unnecessary data writes after the packet transmission, and to automatically send the received packet header to L2 cache and the packet data to main memory, bypassing the L2 cache.

3.6 Characteristic Behavior

This section presents characteristic behavior of the system, providing in-depth knowledge required choosing the correct modeling abstraction level. Some of the system characteristics are simple enough to be obtained directly

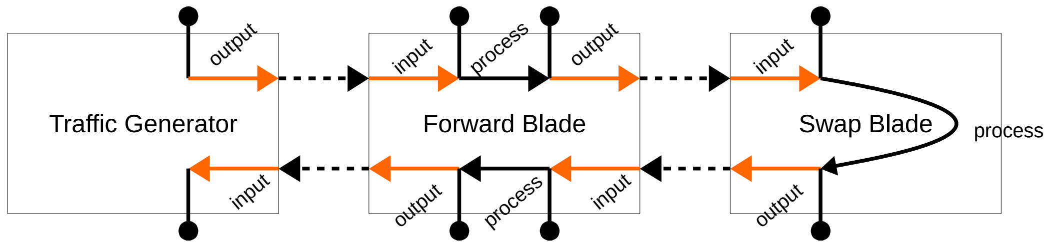


Figure 3.4: The setup used to measure the communication latencies. The communication between the nodes, packet processing on the main cores, and the input and output processing, are marked with the dashed arrows, solid black arrows, and orange arrows respectively. The probes present the points of measurement.

from the literature, while others are either unavailable, or are presented in an unusable form to be used in the simulation model. The missing communication and memory related parameters are obtained by measuring the example hardware. These are discussed in the following sections, respectively.

3.6.1 Communication Latencies

By communication latencies, we refer to the time in the input and output phase of the packet processing, between physical receive/transmit ports and the actual core processing, as described in Section 3.5. We will include the times spent in the scheduler unit for both of these metrics, as due to our resource constraints, we were unable to do the measurements with the required detail to break down these delays. Also, for our modeling purposes, it is accurate enough to assume that the input and output phases consume equal amount of processing time.

The input and output phase latencies are measured by generating traffic from external machine, and passing it through two identical network processing units back to the generator itself. The measurements were done at two independent points in the processing path, to validate the accuracy of the measurements.

In Figure 3.4, the rectangles represent three different computing units (traffic generator, forward unit, and swap unit), and the probes present the points of time measurements. The traffic generator is a typical desktop computer running Ubuntu operating system, and the traffic was generated by Mausezahn [37]. Both of the network processing units are running Linux operating systems.

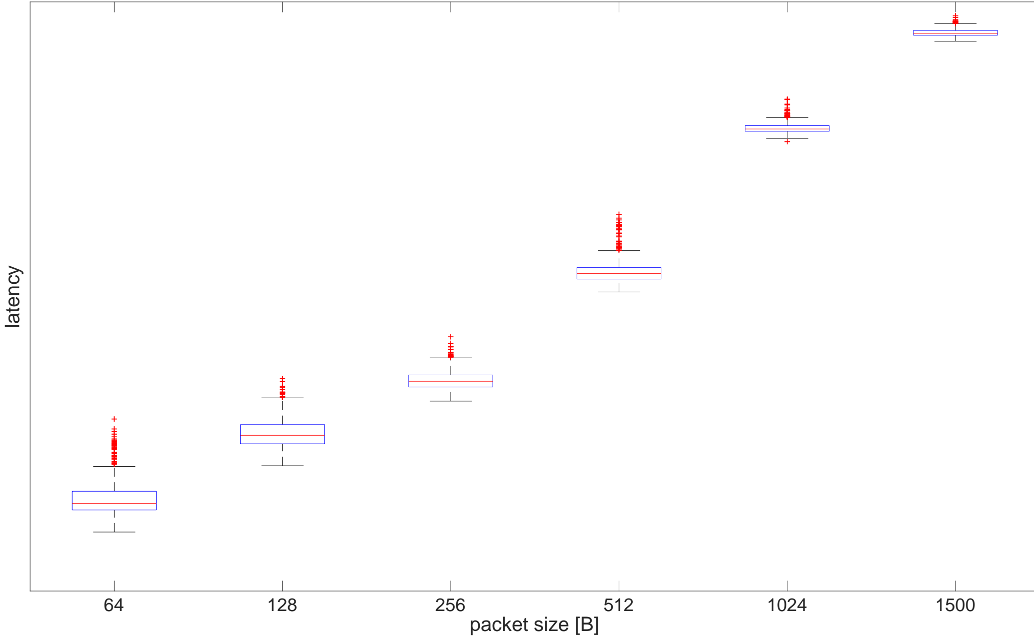


Figure 3.5: Abstract latency of the input and output phase of the example unit. On each box, the central mark is the median, the edges of the box are the 25th and 75th percentiles. Both of the axes are on logarithmic scale.

The packet is first generated at the packet generator and sent to the forward unit at time t_0^d. Forwarding unit receives the packet at time t_{10}^r, does the required processing and forwards the packet to the swap unit at time t_{10}^d. The swap unit receives the packet at time t_2^r, does the same processing as the forward unit (except with different destination address), and forwards the packet back to the forward unit at time t_2^d. Finally the forward unit receives the packet at time t_{11}^r and forwards it to the traffic generator at time t_{11}^d, which marks it received at time t_0^r. The time t_f spent in the input and output phase of one unit is then

$$t_f \approx \frac{t_{11}^r - t_{10}^d - (t_2^d - t_2^r)}{2}. \tag{3.1}$$

We measured the times for packet sizes of 64B, 128B, 256B, 512B, 1024B, and 1500B, repeating the measurement for each packet 10000 times. Figures 3.5 and 3.6 present the statistics of the resulting times t_f for the different packet sizes.

As shown in the Figure 3.5, the time spent in the input and output phase of the unit is roughly linear regarding to the packet size. The variation of the

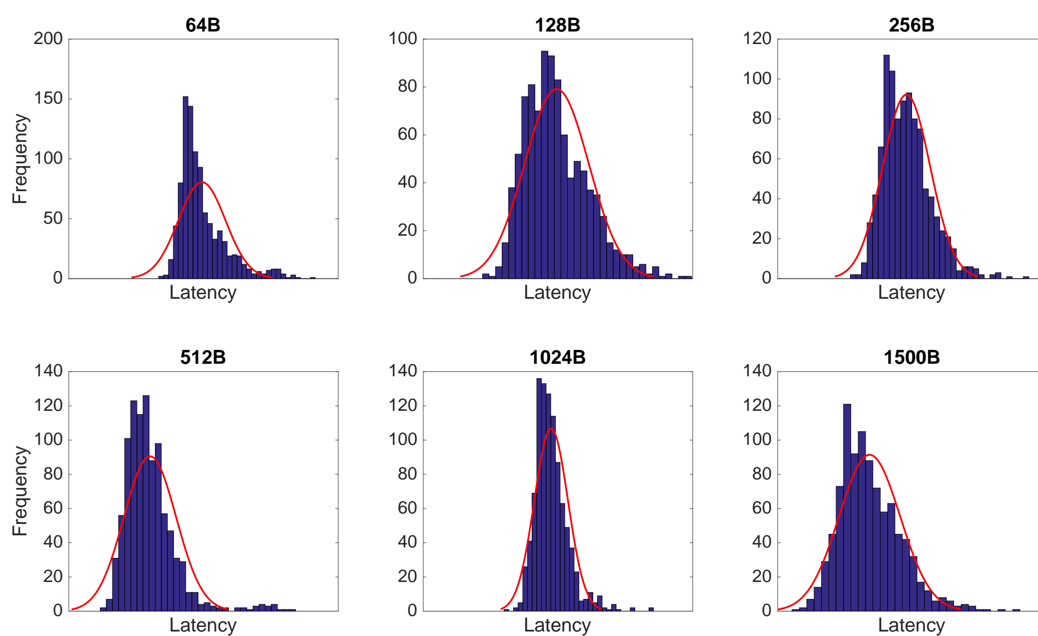


Figure 3.6: Abstract latency frequency histograms for each packet size, along with a normal density function with parameters, estimated with maximum likelihood method from the data.

data is relatively small on all packet sizes. The trend of the latency with respect to the packet size corresponds to the trend of the values measured with at the external traffic-generator (which causes constant overhead regardless of the packet size). The corresponding plots for the external traffic-generator measurements are omitted for clarity.

As seen from both of the Figures 3.5 and 3.6, there exist points with unexpectedly large deviation from the rest of the group. These deviations seem to be independent of the packet size, and thus we assume that the scheduling unit causes them. This behavior is also statistically incorporated in the simulation model.

The only packet size dependent operations in the input and output phases are the memory transfers done for the actual packet data between the memory (L2/RAM) and IN or OUT. All the other operations are done based on the packet header, thus requiring constant amount of time regardless of the packet size.

Since we cannot make a distinction between the input and output phase, in the simulation model presented further, we will adjust the input/output phase amount so that they consume the IN and OUT units for the amount that corresponds the constant term of equation 3.2. The variable (non-constant) term is caused by the memory copies in the input/output phases, and is proportional to the packet size.

The input and output delays used in the simulation model are estimated by fitting a linear model to the data, using least square estimate. The determination coefficient of the estimate $R^2 = 0.997$. The delay for the input and output phase are divided evenly, resulting in

$$t_{in} = t_{out} = \frac{1}{2}(0.0018\frac{1}{B} * packet_size + 1.036). \quad (3.2)$$

3.6.2 Memory Characteristics

Memory delays were measured using Multi-core Processor Architecture and Communication (MPAC) benchmarking library [48]. Both, latency and throughput were measured using different dataset sizes and number of threads. Each test was run with 200,000 repetitions.

Figure 3.7 presents the latency characteristics for different packet sizes and thread counts. Notice that the y-axis the graph is logarithmic. As expected, the memory latencies grow together with the size of the write. The transition between the memory levels can be seen as the jumps in the latency graph. With 128B - 1KB write sizes both the read and write arrays fit in the L1 cache (32KB), and thus the latency per repetition is independent

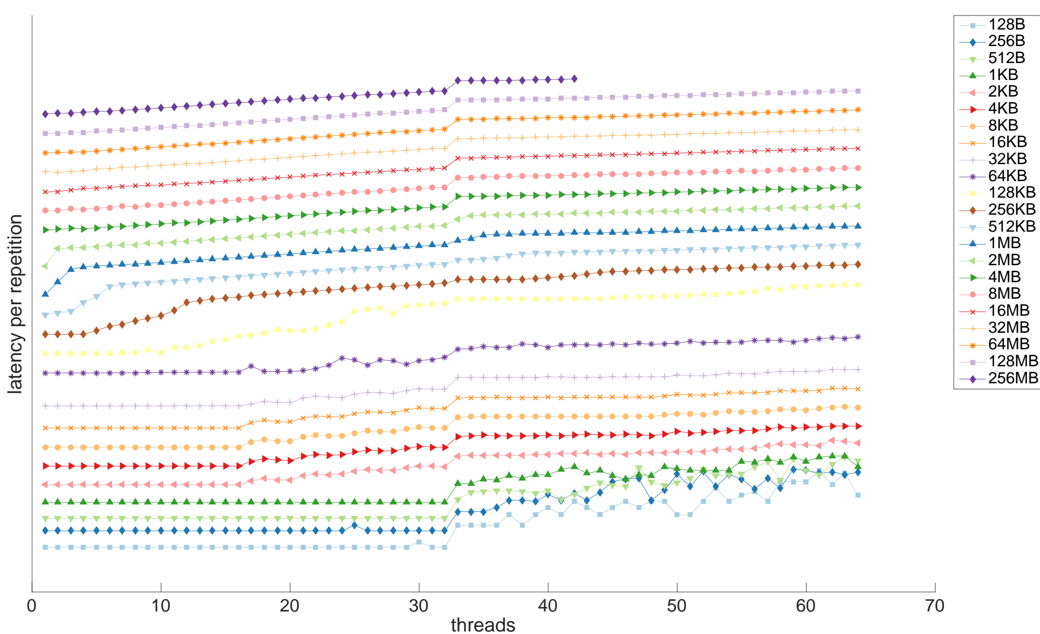


Figure 3.7: Abstract memory latency of the example unit across number of threads for integer data type (32bit), measured by MPAC. The latencies for different packet sizes are marked with colors as shown in the legend (in bytes). The y-axis is presented in logarithmic scale.

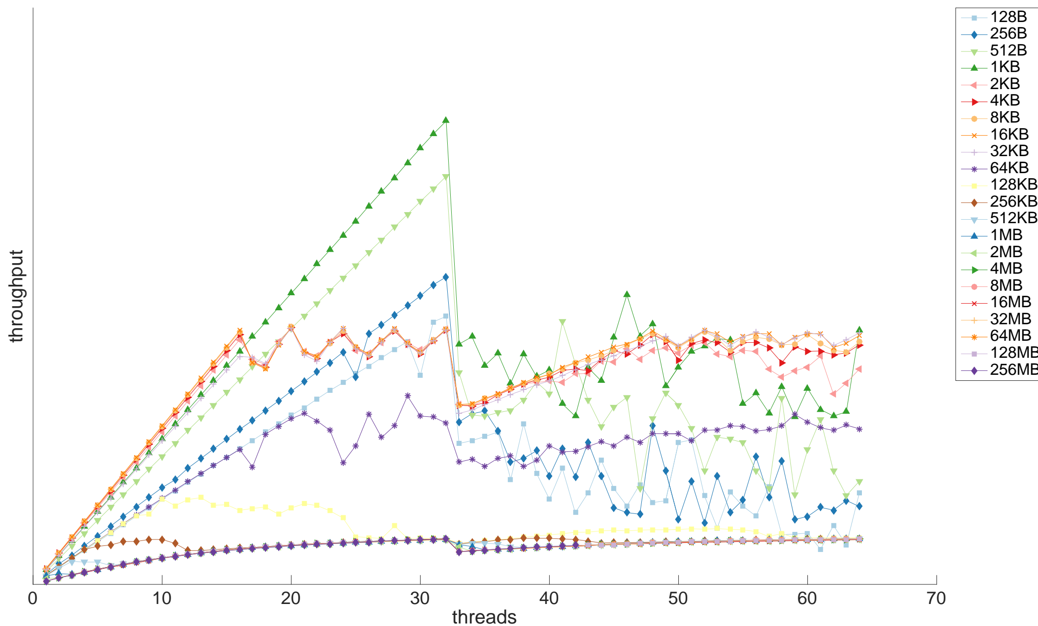


Figure 3.8: Abstract memory throughput of the example unit across number of threads for integer data type (32bit), measured by MPAC. The throughput for different packet sizes are marked with colors as shown in the legend (in bytes).

of the thread count. With write sizes above 2KB, some of the writes hit L2 cache, increasing the latency as the thread count increases. Similarly, the step from L2 cache to RAM can be seen 128KB, 256KB, 512KB, 1MB, and 2MB write sizes, where both write and read arrays completely fit in the L2 cache with 8, 4, 2, 1 and 1 threads, and move to RAM beyond that.

The write latencies also thrash beyond 32 threads, especially for the cache sizes. This does not affect the main core memory accesses in the model, as only 32 threads are used for packet processing. However, these numbers work as a reference when modeling the memory communication of other units such as the scheduler.

Figure 3.8 presents the throughput characteristics for different packet sizes and thread counts. Again, as expected, the maximum throughput is achieved with 1KB write lengths and 32 threads, when both the write and read arrays fit in the caches. The write throughput scales linearly with 128B - 1KB write sizes for up to 32 threads, with 2KB - 64KB up to 16 threads, and for 256KB, 512KB, and 1MB, write sizes up to 8, 4, and 2 threads respectively. The transitions between the memory levels are similar as in the latency graph. Again, a clear thrashing can be seen with more than 32

threads.

Chapter 4

System Performance Analysis

In this thesis, we study and evaluate the performance analysis methods of a modern packet processing system. This chapter introduces the modeling, measurement, and simulation methods used in our work.

We will start by introducing the common performance analysis methods and metrics, and then further examine the modeling and simulation techniques. Further, we present the queuing and resource networks, which are fundamental concepts for modeling packet processing systems. We will describe the basic concepts of simulation, the primary method for solving more complex networks needed to model modern packet processing systems. We end the chapter with a short survey of the existing simulation software.

4.1 Performance Analysis

For almost every computer system – whether it is a high performance application on the cloud [46] or an army fuel-supply system [72] – the performance is one of the most sought-after criteria. To achieve the highest performance for the lowest cost, different performance evaluation techniques are required at different system life cycle stages. The choice of evaluation criteria and techniques used to evaluate the system performance vary between systems. These two choices are discussed in the following subsections. [47]

4.1.1 Evaluation Techniques

Performance evaluation can be done using various techniques. These techniques are generally divided into three categories: analytical modeling, simulation, and measurement. The former two techniques are based on symbolic models of the real-life system, whereas the measurements are done on the

system itself. Analytical approaches use mathematical methods to solve the model, and simulation imitates the operation of the system by executing the model on a simulator [9]. Measurements are done by instrumenting the real system with various hardware and software tools. [47]

No strict programmatic rules can be given to select the right technique. However, there are some considerations that can be used to guide the decisions: system life-cycle stage; available resources, such as time, money and tools; required level of accuracy; trade-off evaluation; and salability. [47]

The life-cycle stage of the system is usually the first to consider. In early design stage, the evaluation is often done using analytical methods or simulation, as it is impossible to measure a yet non-existing system. Measurements are, thus, often used for improving existing systems. [47]

Available resources also dictate the technique selection. Running the measurements and simulations are often more time consuming [33] than the analytic approach, and the required time can be difficult to predict. They both also require special equipment and tools, which are expensive and need special skills to operate. The analytical methods are generally considered less time consuming and less expensive than measurement and simulation. [47]

An important thing to note about the simulation software is their parallelization. Managing the events of, and monitoring during, the simulation, in addition to the challenging in parallel computing itself, make the scaling of simulation software extremely complex. [33]

The required level of accuracy should also be considered. For analytical models to be solvable, they often have to be very simple abstractions of the original system. Thus, the results of the analytical methods are often approximate and less accurate than simulation or measurement. Similarly to analytical methods, simulations are abstract, but often much closer to the real system. Even measurements, despite being most accurate of these methods, can produce results that do not agree with the actual system behavior. The accuracy of simulations and measurements can often be enhanced by spending more time and money on them. [47]

Different evaluation techniques are often used together. Taking advantage of two or more methods simultaneously can be used to validate and verify the analysis results. On the other hand, different methods can be used to complement each other to enhance the analysis process. [47]

4.1.2 Performance Metrics

Every performance study needs a set of performance criteria or metrics, which vary with the service provided by the system. Service requests made to the system produce different outcomes: the system either performs the service –

correctly or incorrectly – or refuses to perform it. The metrics associated with these outcomes are called speed, reliability, and availability, respectively. [47]

When the service result is correct, the performance metrics are used to measure the responsiveness, productivity and utilization of the system. For example, in a network packet processing system, responsiveness could be measured as the packet response time, productivity as the throughput, and utilization as the percentage of time the cores are busy. [47]

If the service result is incorrect, the metrics describe the probabilities of the error, for example, how probably an unintentional packet drops or out-of-orderings occur. When the system fails to perform requested service, it is helpful to classify the different causes of failure, and determine the probability and the duration for each. [47]

It should be noted that many systems provide multiple services, and the number of metrics can be large. Also, different evaluation techniques provide different metrics at different times of the service. For example, some simulators allow white-box-like view of the system states during the simulation, whereas, with analytical methods, the details of the system are often unavailable. [47]

4.1.3 System Components and Environment

In performance analysis, a *system* can be defined to be a set of objects that work together, in regular interaction or interdependence, to accomplish some goal or purpose. Every system is a subsystem of broader *system environment*, whose changes can affect the system. For every performance analysis study, a *boundary* between the system and its environment must be set. [9]

For example, computer systems are often enormously complex. Designing them as a hierarchy of smaller subsystems, and combining them with compatible interfaces help manage their complexity. In a study of a network processing system, the higher-level system can be viewed to consist of several processors, auxiliary memory, memory controllers, and other smaller subsystems. These subsystems can further be viewed as a set of smaller subsystems of subsystems: the processor has several processing cores, each core consists of different functional units, and each functional unit consists of logical circuitry. [9]

The objects of interest in a system are called *entities*, which are associated with a set of *attributes*. An *activity* is a specified length time period. A system *state* completely describes the system at any given time of a specific study. The state might be changed by immediate occurrences called *events*. The events affecting the system are divided into two groups by their

source: *endogenous* events occur within the system under study, and *exogenous* events occur in the system environment. [9]

Continuing with the above example, each of the mentioned components can be seen as the entities of the system. There are several activities and events at different levels of the system. At the higher level, these can be seen as the packets flowing through the system: writes and reads to the memory, execution on different processors, and queuing for the processor time. At a lower level, these could be the computation done by the logical units or the flipping of the transistors' state.

Systems can be categorized into discrete and continuous systems, as per the type of their state change. In a *discrete system*, the state changes only at a discrete time points, and in a *continuous system*, the state change is continuous over time. In practice, almost every system is a combination of both continuous and discrete changes. These systems are often classified by their dominant type. [9]

4.2 System Modeling

Model is a representation of either hypothetical or real-life system under study. By the definition a model should be a simplified representation of the original system. It should represent the studied system with enough detail to provide relevant conclusions and, at the same time, only consider those details that affect the investigated problem. The decision between the level of accuracy and abstraction usually requires knowledge of the system under modeling. [9]

Like with the system representation, the basic building blocks of the model can be defined as entities, attributes, activities and events. The model does not necessarily contain the exact replica of the components of the system, but rather simplified components that represent the system with enough detail. [9]

In the example study of network processing unit, the likely goals would be to determine the packet throughput and latency of the system. In that case, the system could be modeled with sufficient accuracy by omitting all the lower level details of the CPU. On the other hand, a detailed performance study of a specific CPU might require even the minute details of the functional units or logical circuitry. [42, 57]

The models may be categorized as being static or dynamic, and deterministic or stochastic. Static models represent steady-state time-invariant systems, whereas the dynamic models represent systems as time-variant. Deterministic models contain no random variables, meaning that for the known

set of inputs the result, of solving the model, will result in a unique set of outputs. Stochastic models on the other hand include random variables, thus leading to a random set of outputs. [9]

Models can further be divided into discrete and continuous, analogously with the discrete and continuous systems described in Section 4.1.3. However, it is possible to model continuous systems with discrete models, and vice versa. Just as the real-life systems, the models can be a mix of both continuous and discrete models. [9]

4.2.1 Queuing Networks

Network structures are often used for describing the models. For example petri nets [68], markov chains [11], queuing networks [11], and resource networks [59] are widely researched model definitions with a well-developed theory for analyzing the system behavior.

Queue based modeling schemes are inherent in packet processing systems performance analysis. As presented earlier, the software running on top of complex MPSoC hardware are in essence queue abstractions, used to hide the complex hardware interactions of the system. Queuing networks is a specific modeling concept, in which the system under study is represented with a restricted set of building blocks: tasks, queues, resources, and routes. Queuing theory models the behavior of the tasks arriving the system at random inter-arrival periods. [11]

The resource queues can have various scheduling policies, defining the order in which tasks are delivered to the resources. The policies are referred to as *queue disciplines*. The queue disciplines are the main components to models different behaviors of a queuing system. A model consists of simple queuing disciplines, such as first in first out (FIFO), last in first out (LIFO), priority, or shortest task first, can often be solved analytically to find the equilibrium or transient state of the system. [11]

However, models of systems such as packet processors often require more complex policies, to correctly mitigate the system under study. Thus, solving the queuing networks often require approximate methods, such as simulation, to be solved. Queuing networks can be used to describe various systems, and many different problems have well studied, efficient queuing networks solutions for them. [11]

4.2.2 Resource Networks

Unfortunately, the lack of simulation time task routing logic and detailed monitoring, and confined resource descriptions, make queuing networks un-

suitable for more complex computing system studies. To overcome these challenges, for the studies and experiments of this thesis, we have chosen a modeling tool, which incorporates the resource networks modeling concept. [11]

Resource networks extend the queuing networks concept by decoupling the network into resource usage network and resource provision network, and introducing a passive resource type. The resource usage network is a directed graph, describing the possible paths and resource usage requests for the tasks. Resource provision networks are similar to the traditional queuing networks, describing the available resources and their interconnections in the system. Resource networks also enable new task control mechanisms such as fork-join and branching. [59]

Thus, resource networks support modeling resource requests, dynamic scheduling and different load balancing schemes. These techniques enable modeling of more complex interactions of parallel systems with dynamic workload and resource usage. Due to the dynamicity introduced by the flexibility, resource networks are also often solved by simulation. [59]

4.3 Simulation

This section describes the basic principles related to simulation. Despite most of the examples in this chapter being about computing, simulation is widely studied and used method also in several other contexts. The concepts described below, e.g. entities, attributes, or activities, have different realizations from system to system.

Simulation is an artificial imitation of the operation of a real-life system over time. The system behavior is studied by developing a simulation model, based on a set of assumptions concerning the characteristics and functions of the system. The assumptions are presented in mathematical, logical, and symbolic relationships between the objects of interest of the system. An artificial operation history is generated by executing the simulation model, generally on a simulator program, with respect to system input and time. Data are collected from the simulation similarly as if the real system was being measured.

Three different simulation advancement designs are presented in [67]: event-advance, unit-time advance and activity based. In event-advance design the system state changes only when the event occurs. Thus the system state advances from snapshot to snapshot, meaning that the state is unchanged between two successive events. In unit-time advance design, the master clock is advanced in fixed time increments. Activity based design is

a continuous design method, which models the system as a set of conditions that determine when the activities start or stop. [67]

4.3.1 Monitoring

To make conclusions about simulation, the information about the simulation system needs to be gathered. Similarly to the measurement-based techniques, the system is instrumented and the data are saved during the execution. The constructed simulation model is simulated, and the execution is monitored to gather metrics of interest. [47, 67]

There are two generally used approaches for gathering simulation metrics: trace-based and on the fly. The tracing approaches produce raw data from the execution, which are then often post-processed in suitable way. In on-the-fly approaches, the simulator program aggregates the data during the simulation, thus reducing the amount of output. [47, 67]

4.4 Modeling and Simulation Software

In [8], Austin et al. present three drivers for measuring simulator software model: performance, flexibility, and detail. Performance refers to the amount of simulation the model per resources. Flexibility describes how customizable the simulator software is, in terms of modeling. Lastly, detail refers to the level of abstraction used in the simulator. In theory, simulator software could exist, that fulfills all these characteristics at the same time. However, in practice, at least one of them is missing. [8]

Partly because of this, no single general-purpose simulator exists. On the contrary, there is a multitude of different simulators design with different goals in mind. On a high-level, the simulator software can be categorized as cycle accurate, functional, and high-level simulators.

The lowest level cycle accurate simulators are mainly used in low-level hardware designs, and micro-architecture optimizations. At this level, the hardware execution can be modeled precisely, at the cost of simulation speed. As the simulation is carried out on software, it is typically orders of magnitudes slower than the execution on real hardware. They are also difficult to maintain, and thus, in many applications this level simulators are too fine detailed. The hardware models at these levels are typically described with special purpose hardware design languages such as VHDL or Verilog. [65, 90]

One abstraction level higher, functional simulators are used to simulate the system architecture together with execution of program binaries. They are typically used to measure how applications behave on certain hardware.

For example, GPGPU-Sim and Barra, used to simulate NVIDIA graphics processing units, are functional simulations. They both can execute CUDA code, and thus can be used to evaluate the performance of different GPU software implementations. Most of the simulation related to existing MPSoC devices are done on the functional level. [56]

While cycle accurate and functional simulators are important part of design and optimization of hardware and software, in many situations, higher-level simulation is needed. On higher level, the applications and hardware are of presented with coarse abstractions of the underlying system details, to enable early design space exploration of non-existing systems.

This thesis provides an example of a discrete event simulator, PSE, used for modeling packet processing applications in fog computing context. However, the breadth of different systems in fog and cloud computing field is enormous, and PSE is only one example amongst many others. Again, these simulators are build on different abstraction levels, ranging from more detailed (functional) simulators for specific computing units in datacenters, to higher level simulators capable of simulating interrelations of multiple full scale datacenters.

There are several simulators targeted for simulation of cloud computing datacenters on different level. CloudSim [15] is an extensible toolset for modeling and simulation of cloud computing infrastructure and application services, originally released and developed by the University of Melbourne. CloudSim can be used for simulating large-scale datacenters, virtualized server hosts with customizable provisioning policies, energy-aware computational resources, data center network topologies and message-passing applications. Companies such as Hewlett Packard use CloudSim for simulating resource provisioning, energy efficiency, optimization, and further research.

Various different projects extend the CloudSim API for advanced cloud computing simulations. For example, CloudSimEx [18] enables MapReduce [21] simulation capabilities and parallel simulations, Cloud2Sim [49] enables simulation execution on multiple distributed servers, and CloudAnalyst [91] enables simulation of large-scale cloud applications in terms of geographic distribution of both servers and workloads.

Some simulator software, such as Greencloud [51], are designed for the measuring the energy and power consumption of different datacenter setups. GreenCloud provides tools for simulating CPI, memory, storage, and networking resources, with Independent energy models for each resource. It also supports virtualization and virtual machine migration, and full TCP/IP implementation.

The next chapter present a more detailed view of the modeling and simulation framework, Performance Simulation Environment, used in this thesis.

Chapter 5

Performance Simulation Environment

This chapter describes a more detailed view of modeling and simulation framework, Performance Simulation Environment (PSE), and its usage for performance analysis of a packet processing system. We begin with an overview of PSE concepts and tools, and continue by describing the three components of a PSE model. After that, we go through the simulation and monitoring of PSE applications.

5.1 Toolset Overview

PSE is a toolset and simulation environment for dynamic performance analysis of, initially designed but not limited to, parallel computing systems. The tools consist of graphical model editors, compiler tools, and discrete event simulator runtime. Figure 5.1 depicts the organization of the PSE tools.

The graphical model editors – workload editor, wle; task graph editor, tge; sequence chart editor, sce; and resource network editor, rne – are used to build and edit the PSE model representation of a system. Each model editor has a corresponding compiler (wlc, tgc, scc, and rnc, respectively), which is used to compile the textual model representations into C-code.

Figure 5.2 presents the workflow from the modeling into an executable simulation model. First, the graphical editors are used to describe the model components. These components are then compiled into C-code using PSE compilers. Finally, the compiled C-files and the built-in simulator runtime libraries are compiled into an executable simulator program, using generic C compiler such as GCC [78]. The resulting program can be run on top of Linux operating system on commodity hardware.

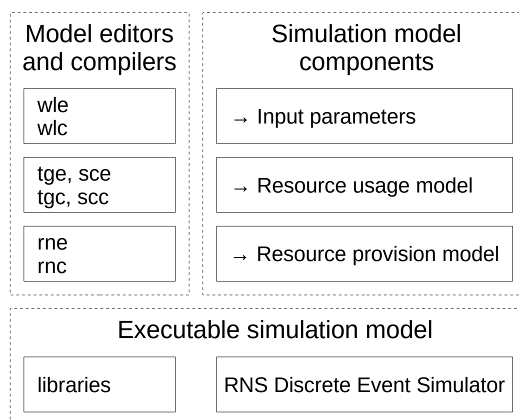


Figure 5.1: PSE toolset includes the model editors, compiler tools and the simulator runtime libraries.

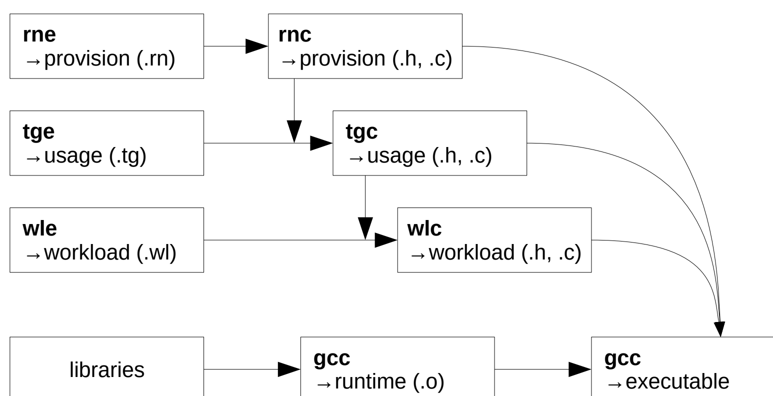


Figure 5.2: PSE compilation workflow.

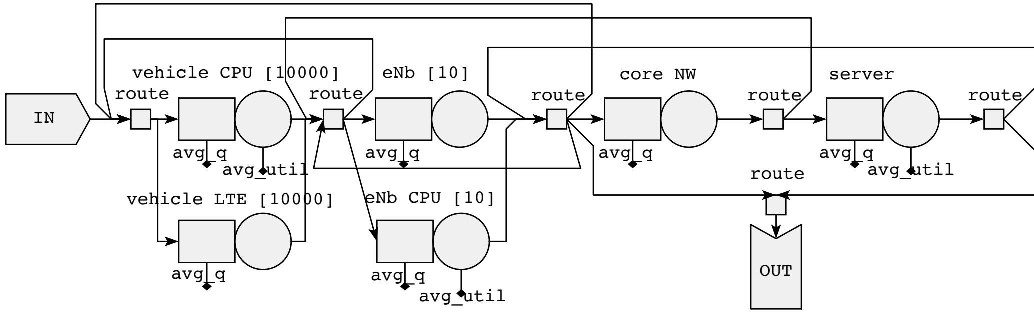


Figure 5.3: An example of resource provision model from a vehicle communication experiment. The model contains six active resource nodes presenting the resources needed for the communication between the vehicles and the cloud datacenter. The probes are used to gather data from the simulation. Figure from [38]

5.2 PSE Model

PSE models the system under study as a resource network. The complete model consists of three main components: resource provision model, resource usage model and workload model. Each of the components is presented as directed graphs, where the nodes represent model entities and the arcs represent the possible flow directions of the tasks.

The resource provision model represents the available system resources, for example computer hardware. The graph nodes represent resource entities, and arcs represent the possible usage order of the resources. The resources are consumed by the tasks, generated by the workload model and guided by the resource usage model.

Figure 5.3 presents an example of a resource provision model. The model is from PSE's example experiment [38] setup, where vehicles communicate with a cloud datacenter through LTE base stations and core network. There are six resource nodes, representing the vehicle CPU and LTE resources, the LTE basestation's evolved node B [75] (eNb) CPU and communication resources, and the core network and server resources.

A resource can be either active or passive. Active resources provide service and introduce service delay to the tasks using them. An example of active resource could be a processor core, which can serve certain amount of processing cycles per unit time. Passive resources do not induce direct delay to the jobs, but their possession is required to access certain other resources. Memory partitions could be an example of passive resource. The example in

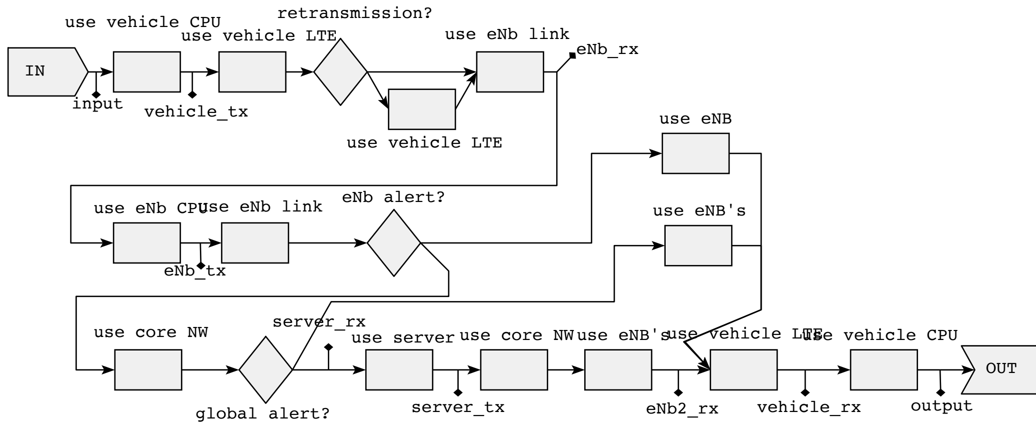


Figure 5.4: An example of resource usage model from a vehicle communication experiment. The rectangles present resource usage nodes. Figure from [38]

the Figure 5.3 contains only active resources.

The resource usage models can be presented as message sequence charts or tasks graphs. We omit the discussion of the sequence chart in this thesis. Task graph is a presentation of the resource usage of the tasks arriving to the system. The nodes in the task graph can be divided into three categories: execution nodes describe the resource usage events and activities, branching nodes conditionally guide the tasks through the graph, and fork/join nodes present task subdivision. The arcs present the flow of control in the system.

Figure 5.4 presents the resource usage model of the vehicle example presented above. The rectangle nodes consume the resources from the resource provision model presented in the Figure 5.3. The branching nodes are presented as the parallelograms. Note that only one of the paths is taken after each parallelogram.

The workload model generates tasks, which traverse through the system according to the rules defined in the resource usage model, consuming the resources defined in the resource provision model. The nodes in the workload graph describe the task generating processes, and the arcs define the relationships between them. The graph representing the workload model must be acyclic.

The event spawn rate can be constant or random (specified for example with probability distribution). When an event is spawned, it progresses through the resource provision model triggering the resource usages, thus getting delayed.

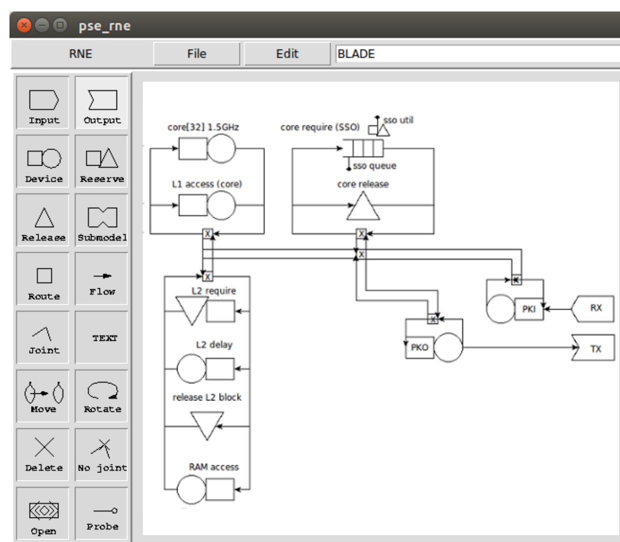


Figure 5.5: The graphical user interface of the resource network editor. The actual resource network model is presented in the middle, and the toolbar on the left.

5.3 Monitoring

PSE has flexible, built-in, monitoring support, which offers both trace-based and on-the-fly monitoring. The monitoring is controlled by attaching probes to the nodes and vertices of the simulation model nodes. It can be done on all the three model levels and practically every simulation system state change can be captured. There are essentially two different types of probes in PSE: the trace probes for trace-based measurements, and the metric probes for on-the-fly measurements.

In the resource provision model, the probes can be attached to two different parts of the resource, to measure either the resource utilization, or its queue size. The trace probes capture every change in the resource utilization or queue size, whereas the metric trace produce aggregate only the descriptive statistics. The descriptive statistics currently include mean, standard deviation, minimum, maximum, sum, and total number of tasks that passed through.

In the resource usage model, the probes can be attached to the model edges, producing a time stamped trace whenever a task travels the edge. The timing can be either absolute time with respect to the global system time, or relative to the process start time. The metric probes can be used to capture the average times of all tasks relative to the start of the process.

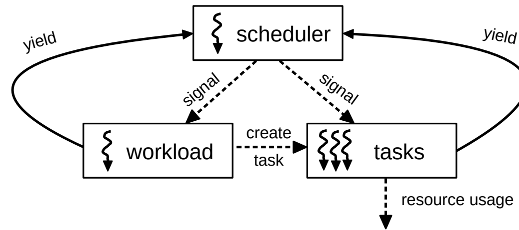


Figure 5.6: The thread cycle of performance simulation environment's resource network simulator engine.

Probes in the workload model are used to control the grouping of the resource usage and resource provision probes.

The probe output is written in a text file defined in the probe node attributes. The trace probes write the output in Comma-Separated Values [76] format, and the metrics traces write a standard descriptive statistics output. Using the trace based probing can substantially slow down the simulation, as the output files easily grow very large, slowing the writing. Thus, whenever the complete trace log is not needed, it is recommended to use the metric based probing. Figure 7.1 in Section 7.1 presents examples of probes attached to core-, memory- and scheduler units in the hardware model as well as the in- and out- probes in the software model.

5.4 Resource Network Simulator

Performance Simulation Environment provides a discrete event simulator engine, named resource network simulator (RNS). The final simulator program is created by compiling the RNS runtime libraries together with the generated simulation model code. The simulator engine manages the simulation execution, i.e. tracks the global simulation time, schedules tasks and manages the system monitoring.

The simulator inputs are generated by the workload model, which spawns a new system thread for each generated input. The input can be either a control input or an actual workload task. The former of these are used for the simulation control, for example changing or resetting the simulation time or monitoring metrics. The latter are the actual task entities presented in Section 7.1.

5.4.1 Simulator Engine

Figure 5.6 represents the model of execution in the RNS engine. The RNS scheduler, running in an infinite loop in its own thread, signals the workload threads or the task threads, based on the trigger time. RNS advances in the event-advance manner, meaning that the thread with the smallest trigger time, based on the scheduler's internal bookkeeping, gets always executed first.

The workload threads spawn new tasks to the actual task-pool, according to the code generated from the workload model. The execution of the task-pool threads advance the actual simulator, consuming the resources based on the values defined in the resource usage model. The actual code that is consumed by the threads is generated by compiling the application models with references to the workload model, hardware models and the runtime libraries. Each time a thread's task encounters an event that is dependent on the other thread's execution, it yields the execution back to the scheduler thread, which then signals the thread, again, with the smallest trigger time.

Chapter 6

Mechanism For Extended Queue Disciplines

This chapter presents the implemented plug-in code extensions for Performance Simulation Environment. The extensions enable modeling of customized queue disciplines written in C-code, and is our attempt to address PSE's lack of global queue scheduling.

The implementation is based on our observations, presented in the previous chapters, of the implementations of networks processing systems and the task-parallel packet processing applications: the systems are essentially controlled through queue abstractions, while the queues are also inherent in the PSE's resource network models. However, what PSE lacks, is the ability to control the execution time behavior of the resource provision nodes (hardware) through the interdependent resource usage nodes' (application) attributes.

The implemented mechanism enables investigation of task-based parallel programming applications, linking the described software and hardware functionality with each other. This again enables the performance analysis of task-parallel applications running on hardware scheduled MPSoC devices.

We begin this chapter by explaining the PSE's service routines, which guide the resource usage between the resource usage and resource provision models. Then we present the runtime structures, RNS_Resource and RNS_Client, which are relevant to understand the implementation of the custom queuing functions. Then we present the requirements for the select and reserve functions used to implement the queuing policies. Finally, we describe example implementations of two simple disciplines: first-come-first-serve and highest-priority-served-first.

6.1 Service Routines

The main interface between the threads executing the resource usage code, and the resource provision models, are the five RNS service routines: RNS_demand_device, RNS_use_device, RNS_reserve_resource, RNS_delay_process, and RNS_release_resource. The functions are used to implement both the active and passive resource usage at the runtime.

Listing 6.1: RNS_demand_device

```
1 void RNS_demand_device(RNS_Device *d, double service_amount,
2                        char *taskname, char *group, uint64_t pc) {
3   ...
4   RNS_use_device(d, service_amount/d->speed, taskname, group, pc);
5   ...
6 }
```

RNS_demand_device routine in Listing 6.1 is a simple wrapper routine, which converts the demanded service amount (service_amount) into corresponding service time, based on the device entity speed (d->speed). It then calls the RNS_use_device routine with the resulting service time.

Listing 6.2: RNS_use_device

```
1 void RNS_use_device(RNS_Device *d, double service_time,
2                     char *taskname, char *group, uint64_t pc) {
3   ...
4   position = RNS_reserve_resource(d->resource, taskname,
5                                   group, pc, &DEFAULT_QUEUE_ATTRS);
6   RNS_delay_process(d->resource->name, service_time);
7   RNS_release_resource(d->resource, taskname, group, position);
8   ...
9 }
```

RNS_use_device in Listing 6.2 reserves the resource, delays the process (i.e. the task) and immediately releases the resource for other processes.

Listing 6.3: RNS_reserve_resource

```
1 uint64_t RNS_reserve_resource(RNS_Resource *r, char *taskname,
2                               char *group, uint64_t pc,
3                               RNS_Queue_Attribute *attrs) {
4   RNS_Client *queue = NULL;
5   uint64_t position, processing_position;
6   ...
7   r->reserve(r, pc, &queue, &position, attrs);
```

```
8     ...
9     set_client(queue, position, RNS_current_process, usage_group, pc, attrs);
10    ...
11    if (queue == r->waiting_queue) {
12      ...
13      RNS_yield();
14      ...
15      processing_position = RNS_current_process->scheduled_to;
16    } else if (queue == r->processing_queue) {
17      ...
18      processing_position = position;
19    }
20    r->processing_queue[processing_position].processing = 1;
21    ...
22    return processing_position;
23  }
```

Listing 6.3 summarizes the RNS_reserve_resource routine. Whenever the process executing the resource usage code consumes a resource (passive or active), the RNS_reserve_resource -function gets called. In the case of passive resource, the call happens directly in the generated resource usage code. If the requested resource is active, the RNS_reserve_resource call happens through the RNS_use_device or RNS_demand_device functions.

RNS_reserve_resource calls the reserve function bound to the resource entity as explained further. The reserve function assigns the task either to the resource's processing queue or waiting queue. If the reserve function assigns the task to the waiting queue, the thread yields the execution back to the scheduler.

Listing 6.4: RNS_delay_process

```
1  void RNS_delay_process(char *name, double seconds) {
2    RNS_Event event;
3    ...
4    event.trigger_time = RNS_simulated_time + seconds;
5    event.process = RNS_current_process;
6    ...
7    RNS_Heap_insert(event);
8    RNS_yield();
9    ...
10 }
```

RNS_delay_process function, presented in Listing 6.3, delays the given process for the time defined by the *seconds* parameter and generates an event

that will be triggered when the requested service has ended.

Listing 6.5: RNS_release_resource

```
void RNS_release_resource(RNS_Resource *r, char *taskname,
                          char *group, uint64_t release_index) {
  ...
  new_index = r->select(r, release_index);

  if (r->waiting_count == 0 || new_index == RNS_LARGE) {
    // None of the waiting clients satisfy the select constrains, e.g. core group
    unset_processing(r, release_index);
  } else {
    move_to_processing(r, release_index, new_index);

    event.trigger_time = RNS_simulated_time;
    event.process = r->processing_queue[release_index].process;

    RNS_Heap_insert(event);
  }
  ...
}
```

RNS_release_resource function, shown in 6.3, selects the next process to be executed from the resource queue, based on the resource's queue discipline function. The selected process is inserted in to the heap of schedulable processes, and thus gets immediately scheduled.

6.2 Runtime Structures

6.2.1 RNS Resource

Each resource entity in the PSE resource provision model definition corresponds to a RNS_Resource runtime variable. The runtime resources are initialized in the beginning of the simulation, based on the code generated from the resource provision entity. Listing 6.6 presents the RNS_Resource struct.

Listing 6.6: RNS_Resource struct.

```
struct RNS_Resource {
  uint64_t id;
  char *name;
  RNS_Probe *probes[RNS_SMALL];
```

```
5     uint64_t probe_count;
6     char *group_name;
7
8     uint64_t (*select)(struct RNS_Resource*, uint64_t);
9     uint64_t (*reserve)(struct RNS_Resource*, RNS_Client**,
10                        uint64_t*, RNS_Queue_Attribute*);
11    uint64_t capacity;
12    RNS_Client *processing_queue;
13    uint64_t processing_count;
14    RNS_Client waiting_queue[RNS_LARGE];
15    uint64_t waiting_count;
16 };
```

The *id* is a unique identifier for the resource, determined by the RNS runtime. The *name*, *capacity*, and *group_name* are defined by the attributes in the model entity. The *probes* array contains the references to the probe structs attached to the resource entity.

The *processing_queue*, and the *waiting_queue* present the data structures used to store the references to the clients that are being processed by and waiting for the resource, respectively. The size of the processing queue is determined, at runtime, by the *capacity* parameter defined in the resource provision model. The size of the waiting queue is fixed, defined by the built-in *RNS_LARGE* constant. *processing_count* and *waiting_count* are initialized to 0, and present the number of clients in the respective queues.

The two function pointers, *select* and *reserve*, are the actual functions used to implement the queue disciplines for the resource. The values of these pointers are determined by the *discipline*, *file*, *select_function*, and *reserve_function* in the resource entity's attributes.

The possible values for *discipline* attribute are: *CUSTOM*, to use custom disciplines determined in the external plug-in files; or one of *FCFS*, *LCFS*, *HPSF*, *LRSF*, to use the built-in disciplines. If the discipline attribute is set to use the built-in disciplines, the model entity's *file*, *select_function*, and *reserve_function* can be left blank, and the correct pointers for the *select* and *reserve* functions are set automatically.

If the discipline is set to CUSTOM, then the pointers to the discipline functions are set to the functions named by the values of *select_function* and *reserve_function* parameters found in the file named by the *file* parameter.

6.2.2 RNS Client

RNS_Client is the runtime representation of the task consuming a resource. The RNS_Client struct is shown in the Listing 6.7. The *process* field is a

pointer to the RNS_process that reserved the resource. *usage_group* is used for trace probe grouping, and *pc* is the old priority attribute left here for the backwards compatibility. In the current version of PSE, the priority should be specified in the *attrs* field, together with all the other attributes to be used in the select and reserve functions. The *processing* parameter specifies whether the client is currently being processed or not.

Listing 6.7: RNS_Client struct.

```
1 struct RNS_Client {
2   RNS_Process *process;
3   RNS_Probe *usage_group;
4   uint64_t pc;
5   RNS_Queue_Attribute *attrs;
6   int processing; // 1 processing, 0 not processing
7 };
```

Each time a process reserves resource, the corresponding RNS_Client entity's *attrs* field, at the RNS_Resource's processing or waiting queue, is set to the values described by the node in the resource usage model. The *RNS_Queue_Attribute* fields are determined in the compile time, based on the attributes determined in the resource usage model's nodes.

6.3 Reserve and Select Functions

As explained above, the queue discipline functionality is determined by the two functions: the reserve function and the selection function. These two functions must follow certain rules and definitions to guarantee the proper scheduling and simulation behavior. The definition of the reserve and select functions are shown in the Listings 6.8 and 6.9, respectively.

Listing 6.8: The definition of the reserve function.

```
1 uint64_t reserve(RNS_Resource *r,
2                  RNS_Client **queue,
3                  uint64_t *position,
4                  RNS_Queue_Attribute *new_client);
```

The reserve function is called from the RNS_reserve_resource function when the task requests the resource for the first time. The parameter *r* represents the requested resource entity, and the *new_client* is the client requesting for the resource. The reserve function needs to perform the following tasks:

- Set the *queue* to point to either the processing queue or the waiting queue of the resource r
- Assign an integer to the *position*, presenting an empty position in the queue
- (Optionally) reorganize the waiting queue
- Return 0 in case of success, else return -1

All the necessary information required to implement the queuing decision can be accessed through the resource r, the *new_client*, as described above.

The select function, in the Listing 6.9, is called each time the resource is released by a client currently holding it. The parameters of the select function determine the resource that is being released (r), and the index at the processing queue that the previous client was placed at ($release_index$). Based on the resource parameter r and the *release_index*, the selection function needs to return an index of the client to be moved to the processing queue. The waiting queue items with index greater than the returned index will be automatically shifted to fill the empty element in the queue.

Listing 6.9: The definition of the select function.

```
1 uint64_t select(RNS_Resource *r, uint64_t release_index);
```

6.4 Discipline Examples

The following subsections present examples of two queue discipline implementations: first-come-first-serve (FCFS), and highest-priority-served-first (HPFS). Both of these functions are simple, and built-in the PSE runtime. They are used here to exemplify the use of reserve and select functions.

Both of the disciplines use the same reserve function, represented in Listing 6.10. The function first checks if the number of clients being processed by the resource is smaller than its capacity.

If so, the function continues by iterating over the resource's processing queue elements until it encounters an empty processing slot. It then assigns the *queue* variable to point to the resource r's processing queue, and the *position* variable to point to the element index i.

If all the processing slots are reserved the function assign the *queue* variable to point to the resource's waiting queue, and the position variable to point to the first empty index.

Listing 6.10: The reserve function used for FCFS and HPSF disciplines.

```
uint64_t reserve(RNS_Resource *r,
                 RNS_Client **queue,
                 uint64_t *position,
                 RNS_Queue_Attribute *new_attrs) {
  uint64_t i;

  if (r->processing_count < r->capacity) {
    for (i=0; i<r->capacity; i++) {
      if (!(r->processing_queue[i].processing)) break;
    }
    *queue = r->processing_queue;
    *position = i;
  } else {
    *queue = r->waiting_queue;
    *position = r->waiting_count;
  }
  return 0;
}
```

Listing 6.11 represents the select functions used for the FCFS and HPSF disciplines. The select function for the FCFS is simple, and does not use any resource or queue parameters for its decision. It always returns 0, representing the index for the first element in the waiting queue.

The HPSF_select function uses the client priorities, defined in the resource usage model nodes, to make the scheduling decision. It iterates through all the clients in the resource's waiting queue, and keeps track of the client with highest priority. After the iteration is over, it returns the index of the client with highest priority.

Again, in both cases, the RNS_release_resource automatically shifts the clients, with index larger than the selected, to fill the empty slot.

Listing 6.11: The select functions for FCFS and HPSF disciplines.

```
uint64_t FCFS_select(RNS_Resource* r, uint64_t release_idx) {
  return 0;
}

uint64_t HPSF_select(RNS_Resource* r, uint64_t release_idx) {
  RNS_Client client;
  uint64_t best = INT_MIN;
  uint64_t current;
  uint64_t index, i;

```

```
11   for (i = 0; i < r->waiting_count; i++) {
12     client = r->waiting_queue[i];
13     current = client.attrs->queue_priority;
14     if (current > best) {
15       best = current;
16       index = i;
17     }
18   }
19   return index;
20 }
```

Chapter 7

Modeling a Packet Processing System

In this chapter, we present an example model of a pipelined network processing system running OpenEM based task-parallel applications. First, we will introduce the high-level model components and entities. The reference values, gathered from the measurements presented in Section 3.6, are then plugged in to the model and the relevant details are discussed. Finally, we describe the implementation of the scheduler unit using the PSE plug-in interface.

7.1 Hardware Model

We created a high-level simulation model of a network processing system with Performance Simulation Environment. As our interests are mainly in the applications' and scheduler unit's effect on the packet throughput and latency, we will not model the specific details of all the hardware components. Some of the components, such as the input and output phases, and the memory models, are modeled statistically. Figure 7.1 shows a layered representation of the main components of the final model: workload, hardware, and software. The workload model and software model's application steps vary between different applications, and are not fixed part of the hardware model per se, but rather presented here to give a full example of the model.

The workload model, at the top of the picture, consists of two packet streams. The *TRAFFIC GENERATOR* node activates the two stream nodes, which again generate packets that enter the software layer. The workload nodes contain parameters, such as the application id's, that are used to con-

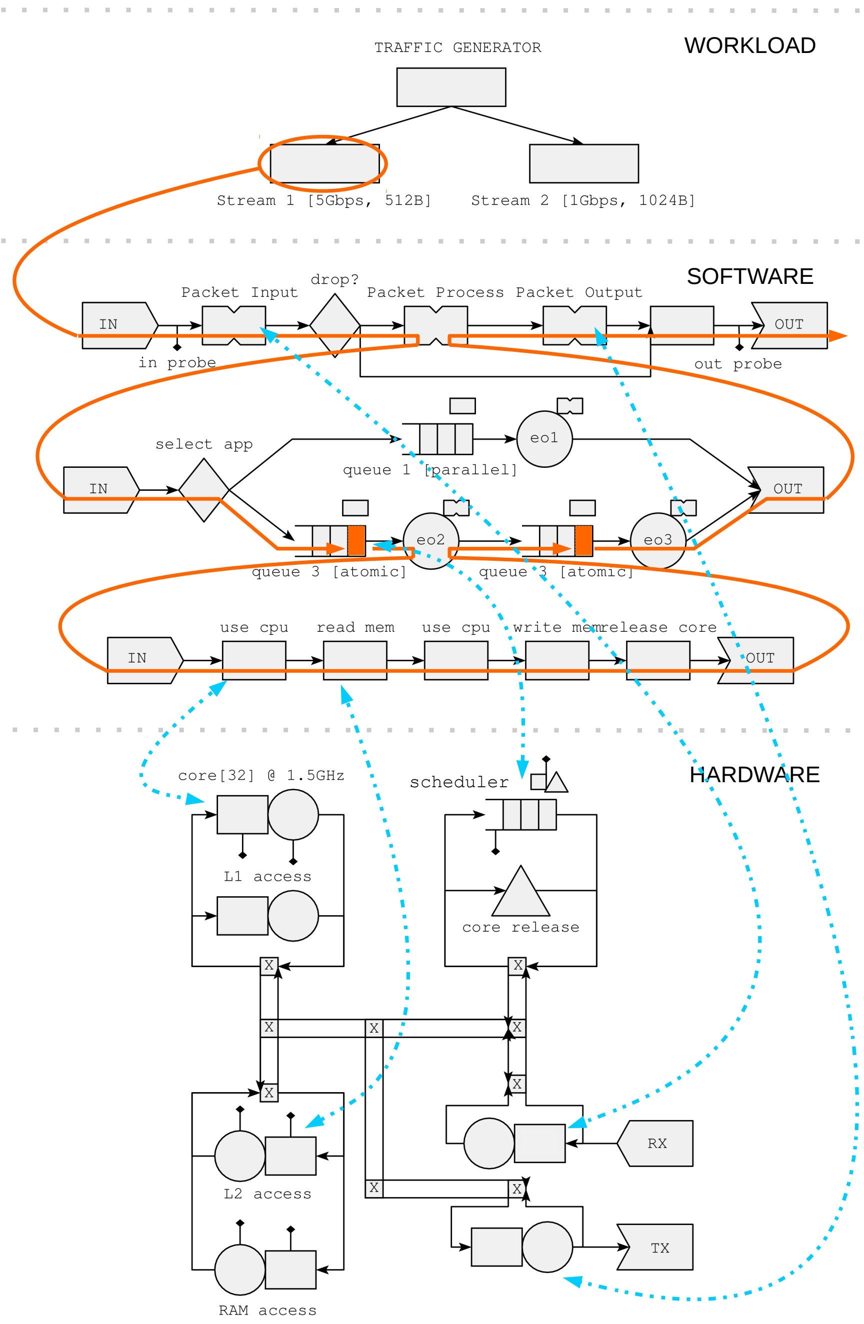


Figure 7.1: Graphical presentation of the PSE model of a network processing system. The workload model (top) generates network packets, which then flow through the software model (middle), consuming the hardware resources (bottom). The orange arrows represent an example of a packet's path through the software model, and the blue arrows the resource usage at each software model node.

trol the packet flows in the software model.

The software model is divided into several submodels. The top software level model consists of packet input, packet processing and packet output submodels. The submodel view of the packet input and packet output are omitted from the picture for the sake of simplicity, as in both of these phases, the core and memory usage is linearly dependent on the packet size with additional Gaussian noise. In the input phase, the packet consumes specific amount of core cycles for the header processing, and copies the packet header and the packet data to the memory. Packet output node copies the packet from the memory, and consumes certain amount of clock cycles for the packet checksum calculations.

The packet processing submodel is presented in the middle software layer. The select app node forwards each of the packets to one of the two packet processing application, based on the application id attribute defined in the workload model. The queue nodes represent the core scheduling done by the scheduler hardware unit. The packets arriving in the upper application have priority of 1, and they can be processed in parallel. In the application below, there are 2 atomic queues with priority 3. When the packet receives the passive resource from scheduler, it can enter the actual processing application, called execution object (eo). The execution objects are submodels that consume core cycles and memory similarly to the packet input and packet output submodels presented above. The scheduler/core passive resource is released inside each execution object.

The hardware model is a simple one level model containing no submodels. In the bottom left hand corner, there are IN and OUT devices providing processor cycles for the packet input and packet output phases. The only passive resource node, scheduler, provides core access resources, which can be released with the core release node. The application cores are shown in the top left corner of the hardware layer. There are 32 cores, and a specific L1 cache access for each of them. The L2 and RAM memory resources provide the delay for reading and writing to memory.

The probes, attached to the scheduler unit, the cores, and the memory nodes, are used to gather statistics from the execution. Each of the units has two probes, one to measure the resource usage, and the other to measures the queue for the corresponding resource. In this hardware model, the routing nodes (squares) and the edges connecting the resources do not have any functional meaning.

7.2 Modeling the Task Scheduler

The scheduler unit has an important effect on the tasks' throughput times, as it controls the order in which the tasks get service from the application cores. This section first describes the application nodes used to model the scheduler unit, and then the actual plug-in functions used to implement the scheduling functionality on the hardware model level.

When modeling the applications on PSE, it is helpful to consider the flow from the task's perspective. When a task has passed the input processing, it is put into the scheduler queue to wait to be scheduled on the application cores. Each time a core finishes its previous task, it requests for a new work from the scheduler-unit, which then schedules a task based on the flow quality of service priority and work group.

7.2.1 Application Models

Each packet processing application consists of two parts: the scheduler queue node, and the actual processing. The software layer in Figure 7.1 presents an example of two main applications. The second application is divided into two sub-applications. The parameters for the first application's queue and execution object are shown in the Listings 7.1 and 7.2, respectively.

Listing 7.1: The attributes of the scheduler queue.

```
1 queue 2 [atomic]
2 name:core_require
3 type:reserve
4
5 queue_type:atomic
6 queue_id:eo2queue
7 queue_priority:3
```

The first line in Listing 7.1 specifies the display title for the node, as shown in the Figure 7.1. The *name* and *type* attributes specify that the resource usage is passive, and the required resource provision entity is *core_require*, i.e. the scheduler. The parameters on lines 5-7 define the parameters that are used in the custom scheduling code on the hardware level. The *queue_type* value atomic specifies that two nodes entering the scheduler from the same resource usage node, cannot be processed simultaneously. The *queue_id* is used to keep track of the tasks being processed, and *queue_priority* is used to globally prioritize tasks between the queues.

Listing 7.2: The attributes of execution object.

```
1 eo2
2 name:APP2EO2
3 type:submodel
4 file:app2/eo2.tg
```

The execution object node parameters are shown in the Listing 7.2. It is a simple submodel node. It specifies the display title, and the name of the submodel to be used. The *file* attribute specifies the file that defines the submodel. Note that, each execution object submodel needs to release the scheduler/core resource, as shown in the Figure 7.1.

7.2.2 Global Hardware Scheduler

To model the processing cores' run-to-completion behavior, the scheduler unit is modeled as a passive resource. Each time a task is entering a processing application, it first acquires the passive scheduler resource. The amount of available passive scheduler resources is equal to the processing cores in the system, and a task cannot use processing core without holding the scheduler resource.

The scheduler unit model requires the use of custom scheduling functions. These are modeled using PSE's custom plug-in interface, presented in Chapter 6. The functionality is implemented in two custom functions, written in C-code: the selection function, and the reserve function. The resource node parameters are changed to use these functions. The parameters of the scheduler node used in the example model are presented in the Listing 7.3.

Listing 7.3: The attributes of the scheduler node to determine the custom scheduler.

```
1 scheduler
2 name:core_require
3 capacity:32
4
5 discipline:CUSTOM
6 file:custom/scheduler.h
7 select_function:CUSTOM_select
8 reserve_function:CUSTOM_reserve
```

The first three lines specify the node title, name, and capacity. The capacity is set to the amount of available processing cores, meaning that no more than 32 tasks can be processed simultaneously. The *discipline* parameter CUSTOM, on line 5, specifies that a custom scheduler is used instead of the built-in scheduling functions. The *file* parameter specifies the path of the

C header file that declares the scheduling functions. The *select_function* and the *reserve_function* parameters specify the two functions that are required to implement the scheduling logic.

The reserve function is called each time a task enters a resource usage node in the resource usage model, and it determines whether the task can immediately be served, or if it has to wait for the service. If the task can be processed immediately, it is inserted to the processing queue of the resource, and to the waiting queue otherwise. If the task gets put to the waiting queue, the reserve function also needs to reorder the queue.

Listing 7.4 describes the reserve function used to model the scheduler. The function takes four input arguments: r contains the data of the resource being reserved; *queue* is a pointer whose value is assigned either to the processing queue or the waiting queue; *position* is a pointer, whose value is assigned to the position in the queue; *new_client* holds the parameters, defined in the workload and resource usage models, of the new task.

Listing 7.4: The *CUSTOM_reserve* function for scheduler.

```
1  uint64_t CUSTOM_reserve(RNS_Resource *r, RNS_Client **queue,
2                          uint64_t *position, RNS_Queue_Attribute *new_client) {
3    uint64_t i, j;
4    char *queue_id = new_client->queue_id;
5    int atomic, flow_already_processing;
6    RNS_Client client;
7
8    atomic = (!strcmp(new_client->queue_type, "atomic"));
9
10   // first try to place the client in the processing queue
11   if (r->processing_count < r->capacity) {
12
13     /*
14      * Find the correct place to position the client
15      * 1) check if the resource in position i is in use
16      * 2) check that two clients from the same atomic queue wont get processed
17      * 3) check if the client's coremask allows processing on this core
18      */
19     j = r->capacity + 1;
20     flow_already_processing = 0;
21     for (i=0; i<r->capacity; i++) {
22       client = r->processing_queue[i];
23
24       if (client.processing) {
25         if (atomic &&
```

```
26                !strncmp(client.attrs->queue_id, queue_id, strlen(queue_id))) {
27              flow_already_processing = 1;
28              break;
29            } else continue;
30          }
31          if (CHECK_COREMASK(new_client->queue_coremask, i)) j = i;
32        }
33
34        if (j <= r->capacity && !flow_already_processing) {
35          *queue = r->processing_queue;
36          *position = j;
37          return 0;
38        }
39      }
40
41      /* no suitable cores (available or accepted by this client's coremask),
42         place the client to waiting queue */
43      *queue = r->waiting_queue;
44
45      /* find the first waiting client with higher priority,
46         place the client right before it */
47      for (i=0; i<r->waiting_count; i++) {
48        client = r->waiting_queue[i];
49        if (client.attrs->queue_priority > new_client->queue_priority) break;
50      }
51
52      // move the clients
53      for (j=r->waiting_count; j>i; j--) {
54        r->waiting_queue[j] = r->waiting_queue[j-1];
55      }
56
57      *position = i;
58
59      return 0;
60    }
```

On the lines 11-39, the reserve function attempts to place the task in the processing queue. The if-statement, on line 11, checks if the resource capacity is full. If there are available cores, then the processing conditions are checked, by going through all the cores, as shown in the for-loop on lines 21-32. If the new task's flow is marked atomic (in the reserve node in resource usage graph), and another task from the same flow is being processed on one of the cores (if-condition on lines 25-26), then the for-loop breaks, and the

task gets set to the waiting queue. If a core is not processing, and the flow's coremask permits processing on the core (if-condition on line 31), we assign core's index to variable j. Finally, if there is available core (j is smaller or equal than the resource capacity) and none of the tasks from the same atomic flow are being processed, we set the queue to point to the resource's processing queue, and the position to the variable j, and return.

If the processing conditions are not met, i.e. the execution goes past the if-block, then the task is set into the waiting queue. Line 43 assigns the *queue* to point to the waiting queue of the resource. The for-loop on lines 46-48 finds the first task with larger priority, at index i, and the for-loop on lines 51-53 moves all the higher priority tasks one step further on the queue. Finally the index i is assigned to *position*, and the function returns.

Each time a core ends a processing of a task, a new task is selected for the processing, using the custom select function. Listing 7.5 shows the code used for the select function to model the scheduler.

Listing 7.5: The *CUSTOM_select* function for scheduler.

```
1  uint64_t CUSTOM_select(RNS_Resource *r, uint64_t release_index) {
2    uint64_t i, j;
3    RNS_Client waiting;
4    char *qid;
5    int already_processing, atomic;
6
7    for (i=0; i<r->waiting_count; i++) {
8      waiting = r->waiting_queue[i];
9
10     if (!CHECK_COREMASK(waiting.attrs->queue_coremask, release_index)) continue;
11
12     atomic = (!strcmp(waiting.attrs->queue_type, "atomic"));
13
14     if (atomic) {
15       qid = waiting.attrs->queue_id;
16
17       already_processing = 0;
18       for (j=0; j<r->capacity; j++) {
19         client = r->processing_queue[j];
20         if (!client.processing) continue;
21         if (j == release_index) continue;
22         if (strncmp(client.attrs->queue_id, qid, strlen(qid))) continue;
23         already_processing = 1;
24         break;
25       }
```

```
26
27        if (already_processing) continue;
28      }
29
30      return i;
31    }
32    return RNS_LARGE;
33  }
```

CUSTOM_select takes the resource *r*, and the index of the released core *release_index* as an input. The outer for-loop, starting at line 7, goes through all the tasks in the waiting queue, and finds the first task that satisfies the processing constraints, similarly as the reserve function. Line 10 checks if the waiting task's coremask allows the task to be processed on the core. If the waiting task's flow is atomic (line 12), we need to go through all the processing cores to check that there is no task being processed from the same flow (lines 18-24). If the task was not atomic, or no tasks from the same flow were being processed, the function returns the index of the task in the waiting queue. Otherwise we move to the next waiting task and repeat. If no task from the waiting queue can be scheduled, the function returns *RNS_LARGE*. The RNS automatically moves the clients when it removes the task from the waiting queue.

Chapter 8

Demonstrative Experiment and Discussion

This chapter presents two demonstrative simulation experiments done on the NPU model presented in Chapter 7. The goal of the experiments is to demonstrate how the parallel mechanisms of a task based-programming model, i.e. queue types, priorities, and coremasking, affect the scheduler, and hence the packet throughput and latency. At the same time, it validates our NPU model and the PSE's implemented plug-in-code functionality.

In the first experiment, we will use the global queue information to control the resource provision based on global queue priorities and disciplines defined in the software queues. The second experiment presents the use of queue coremasks, to control the use of specific hardware resources, through the software model.

8.1 Experiment Setup

Both experiments are run on the hardware model represented in Figure 8.1. The model consists of six active resources. The IN and OUT units are consumed by the input and output phases of the packet flow, respectively. Each 32 processing cores have a L1 cache associated with it, and the L2 cache and RAM are shared between the cores. The cores are served as first come first serve basis, as the scheduling logic is taken care of by the scheduler unit.

The scheduler unit uses the custom scheduling functions presented in Section 7.2.2, enabling the use of atomicity, queue priorities, and coremasks. The core release is a typical release node referring to the scheduler unit.

The same high level software model is used for the both experiments, as seen in the Figures 8.2 and 8.7. The *Packet Input* and *Packet Output* nodes

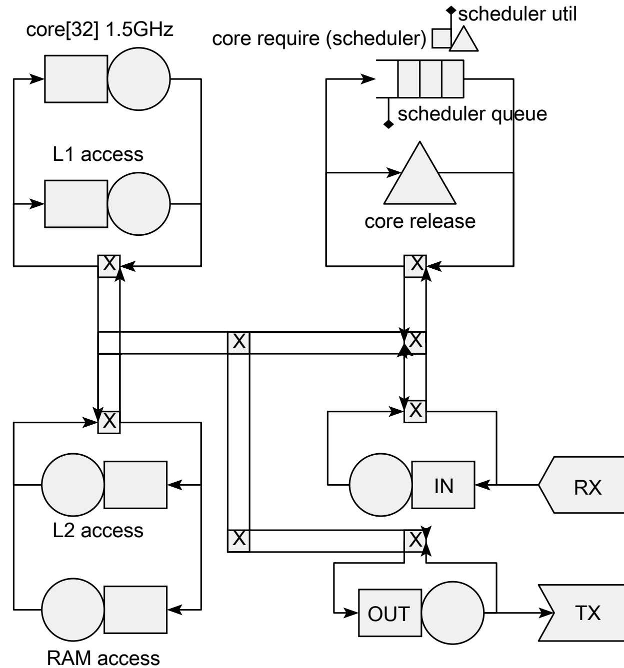


Figure 8.1: Resource provision model (hardware) used in the experiments.

consume the IN and OUT units as discussed in Section 3.6, delaying the packets relative to their size, according to the equation 3.2. The workload and packet process submodels are unique for both experiments.

We gather two different metrics of the system. First, we are interested in the core utilization and queue lengths for each processing step. These are measured by the probes attached to the scheduler unit as shown in hardware model in Figure 8.1. Secondly, we are interested in the packet latencies. They are measured by calculating the time difference between the *out probe* and the *in probe* (Figures 8.2 and 8.7) for each packet. All the probes write absolute traces of the packets.

8.2 Experiment 1: Global Queue Interrelations

The first experiment consists of two different simulations and measurements. We will first demonstrate a packet processing application, whose throughput is limited due to the bottleneck occurring from a slow atomic processing. The application is then modified, extracting part of the atomic execution object

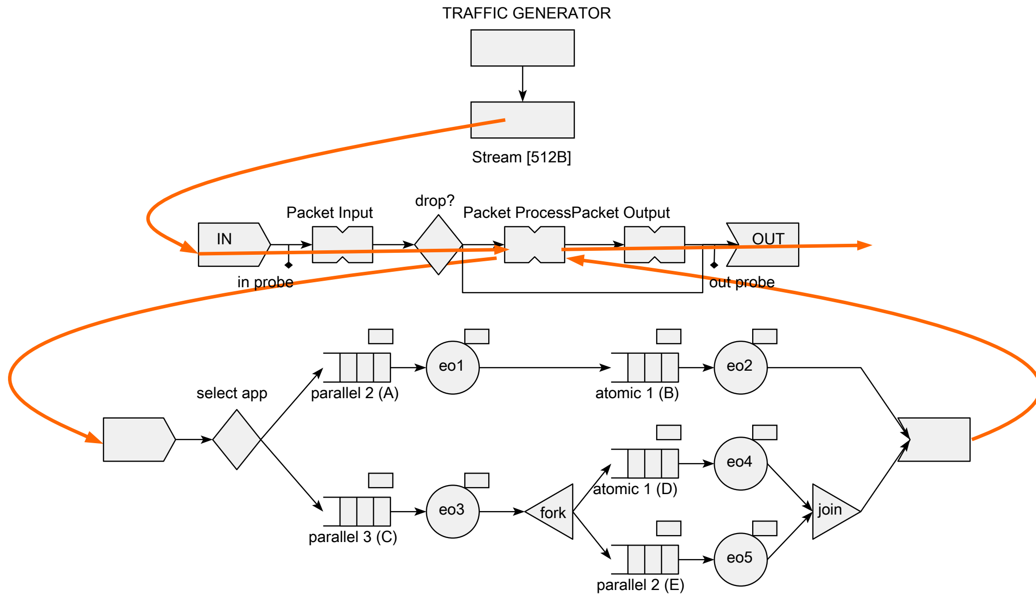


Figure 8.2: Workload and resource usage (software) models used in the first experiment. The atomic queue in the upper application (application 1) produces a bottleneck to the system. The application 2 below removes this bottleneck by extracting part of the processing into parallel.

into parallel, thus breaking the bottleneck.

The workload model consists of a single packet stream, which is generated from a two level workload model, presented at the top in Figure 8.2. The *TRAFFIC GENERATOR* node triggers its child node with interval $RNS_random_uniform(5*10^{-5}, 15*10^{-5})$ seconds, and lifetime of 0.05 seconds. The child node creates 512B packets with interval $5.1 * 10^{-8} * RNS_random_lognormal(-10, 0.9)$ seconds for $4*10^{-5}$ seconds.

The two different applications used in the experiment are shown in the software model after the *select app*-node. The lower and upper application are referred as *application 1* and *application 2*, respectively. The application selection is done in the workload model by the *AppId* attribute.

Application 1 consists of two processing steps, both consuming CPU and memory for the range of 5000 clock cycles. The first step consists of parallel, priority 2 queue (A), and the second one is done atomically with priority 1 queue (B). Application 2 is a modified version of the first packet processing application, where the second processing step is split into parallel and atomic steps. The release nodes are omitted from the software model for clarity.

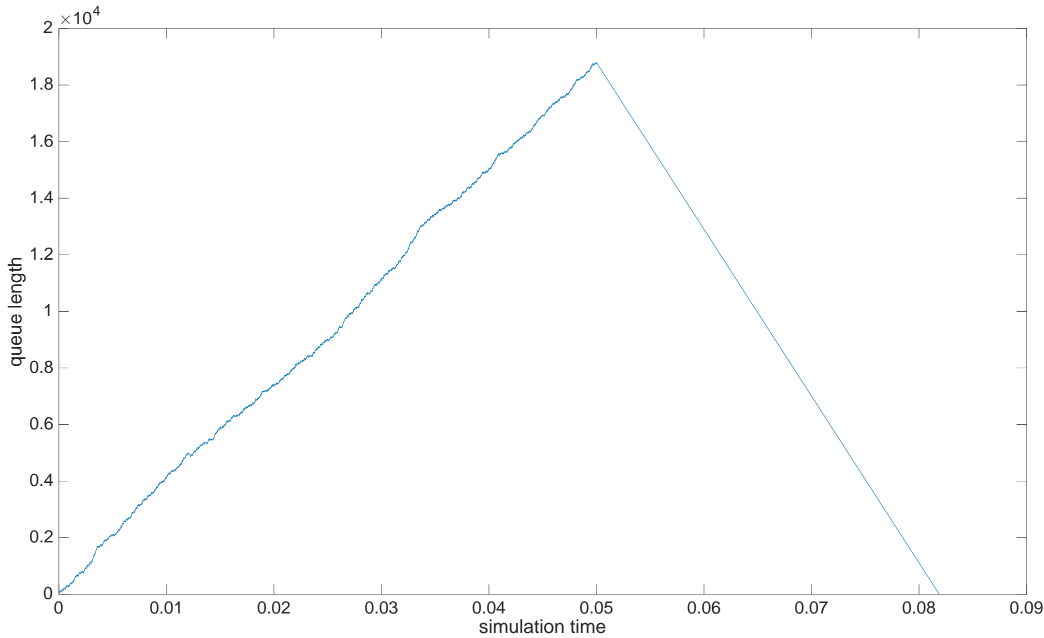


Figure 8.3: Application 1: The number of tasks in the scheduler/core queue, with respect to simulation time, from the second resource usage queue.

8.2.1 Simulation Measurements

The system was simulated using both applications separately, and the data from the probes were post-processed. We grouped the number of tasks in the scheduler/core queue, by the processing step.

Figures 8.3 and 8.4 present the data from the first simulation using the packet processing application 1. Figure 8.3 describes the number of tasks in the scheduler/core queue, which are from the atomic resource usage queue, with respect to simulation time. The corresponding graph for the first queue is omitted, as none of that tasks from it end up in the waiting queue. Figure 8.4 presents the latency of each packet through the whole system.

As shown in the Figures, the processing in the second execution object of application 1 is so heavy that the tasks accumulate into the waiting queue, and thus the packet latency keeps growing until the workload ceases.

The second application modifies the system by parallelizing part of the atomic processing. Figures 8.5 and 8.6 present the data from the second application simulation. Figure 8.5 shows the number of tasks in the scheduler/-core queue, which are from the atomic resource usage queue, with respect to simulation time. Figure 8.6 presents the latency of the packets through the whole system. Neither of the parallel queues have tasks in the waiting queue

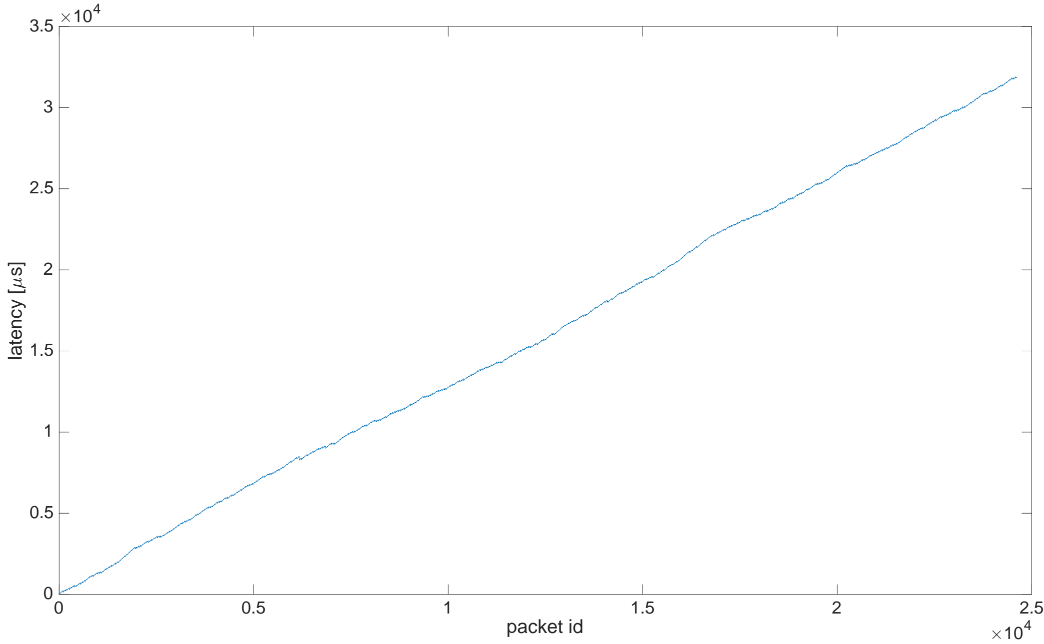


Figure 8.4: Application 1: Latency of the packets through the system.

of the scheduler/core during the simulation.

As the shown in the Figures 8.5 and 8.6, the latencies of the second application stay within 22μs, reducing the queue length below 40 over the whole simulation period.

8.3 Experiment 2: Queue Coremasks

The second experiment demonstrates the packet flow control via queue coremasks. The experiment consists of two packet flows and dedicated application processing them, as presented in Figure 8.7. The first packet stream (presented as orange), presents a higher OSI-level application data, such as video stream, which is processed on a heavy execution object. The second stream (presented as blue), on the other hand, consists of packets whose processing is done on fast processing application with strict latency requirements.

The *TRAFFIC GENERATOR* node triggers its child nodes with interval $RNS_random_uniform(5*10^{-5}, 15*10^{-5})$ seconds, and lifetime of 0.05 seconds. *Stream 1* and *Stream 2* create 512B packets with interval $6.1*10^{-7} * RNS_lognormal(-10, 0.9)$ and $10^{-6} * RNS_lognormal(5, 10)$ seconds for $4*10^{-5}$ and 10^{-5} seconds, respectively. The heavy application

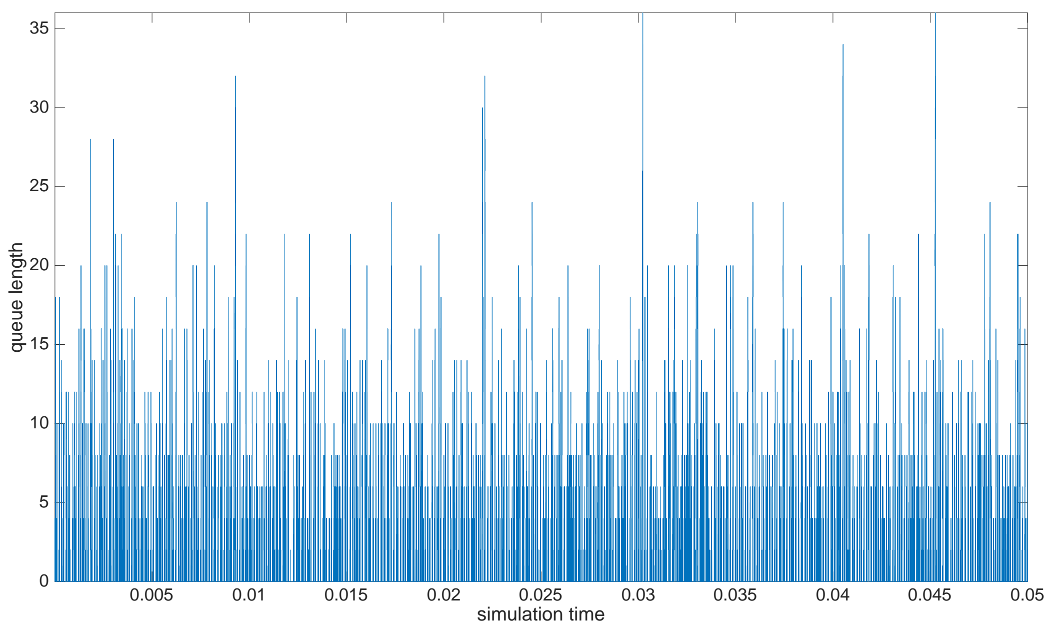


Figure 8.5: Application 2: The number of tasks in the scheduler/core queue, from the second resource usage queue.

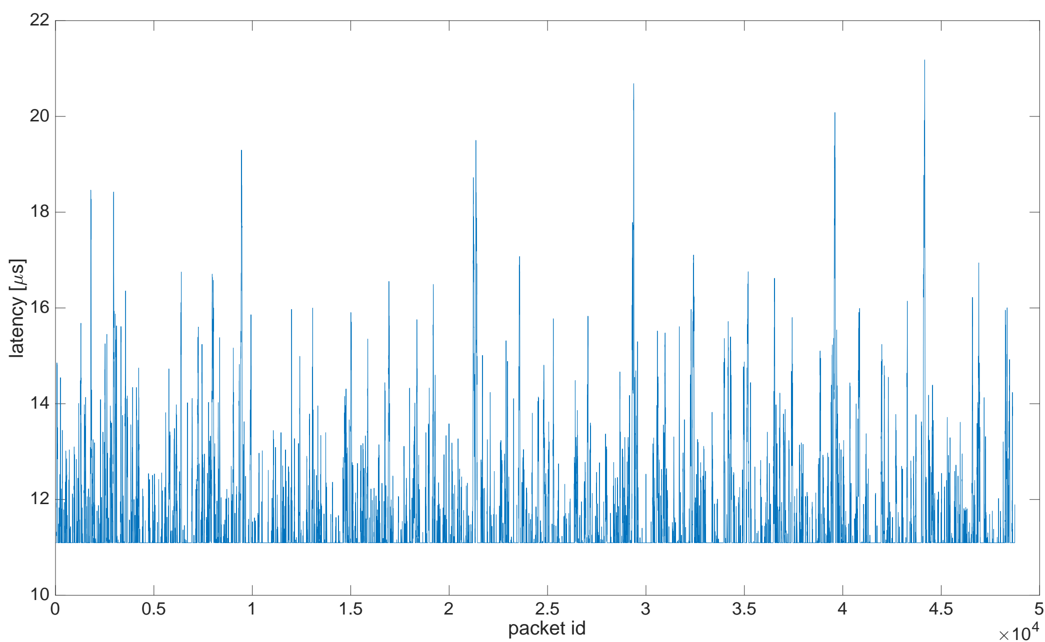


Figure 8.6: Application 2: Latency of the packets through the system.

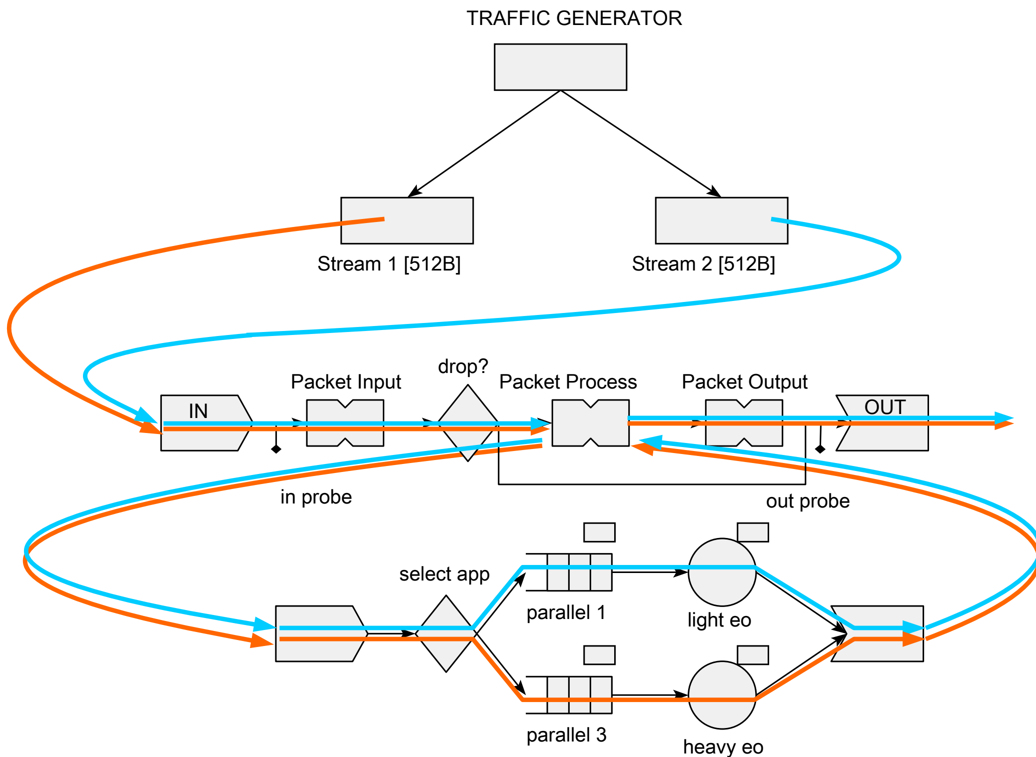


Figure 8.7: An overview of the workload and software models used in the experiment 2. The orange and blue paths present the paths of the packets through the system. Each of the flows has its own packet processing application.

consumes the CPU and memory for a total of about 50000 cycles per 512B packet, while the light application packets are processed for tenth of that, in 4500 cycles.

We will again perform two different simulations and measurements. The first one is carried out by allowing the heavy application to be processed on all the 32 processing cores. Despite the light flow being processed on higher priority queue, the processing of the heavy flow has an effect to the worst-case latency of the light flow packets. This happens due to the run-to-completion model of the cores, where light flow packets may have to queue for a core while the heavy tasks finish. In the second simulation, the heavy application processing is limited to 30 cores, dedicating two cores for the light application.

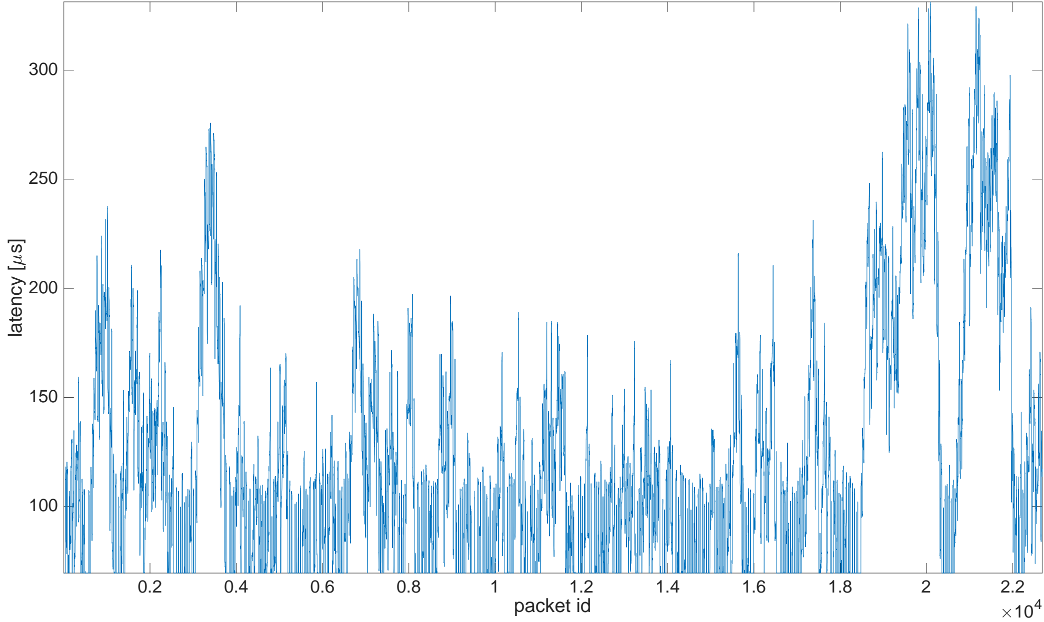


Figure 8.8: Latencies of the heavy flow packets, without coremask.

8.3.1 Simulation Measurements

The system was simulated with and without the limited coremask on the heavy application processing cores. The data from the probes were then post-processed. Figures 8.8 and 8.9 present the data from the simulation without coremask, for the heavy and light flow, respectively. The latency of the heavy flow packets is between 80μs and 340μs throughout the simulation. The throughput of the heavy application is 0.214GBps.

In the second simulation, we use a modified coremask for the heavy application, i.e. dedicating one of the cores for the light application processing. The resulting latencies of the heavy and light flow are presented in Figures 8.10 and 8.11, respectively. The light flow's worst-case packet latencies stay below 12μs. The throughput of the heavy application is 0.212GBps.

8.4 Experiment Analysis

The experiments present a proof of concept of the implemented PSE's plug-in code mechanism and the model of the reference NPU. In both experiments, the model acts as expected.

The first experiment presents the use the global queue information to

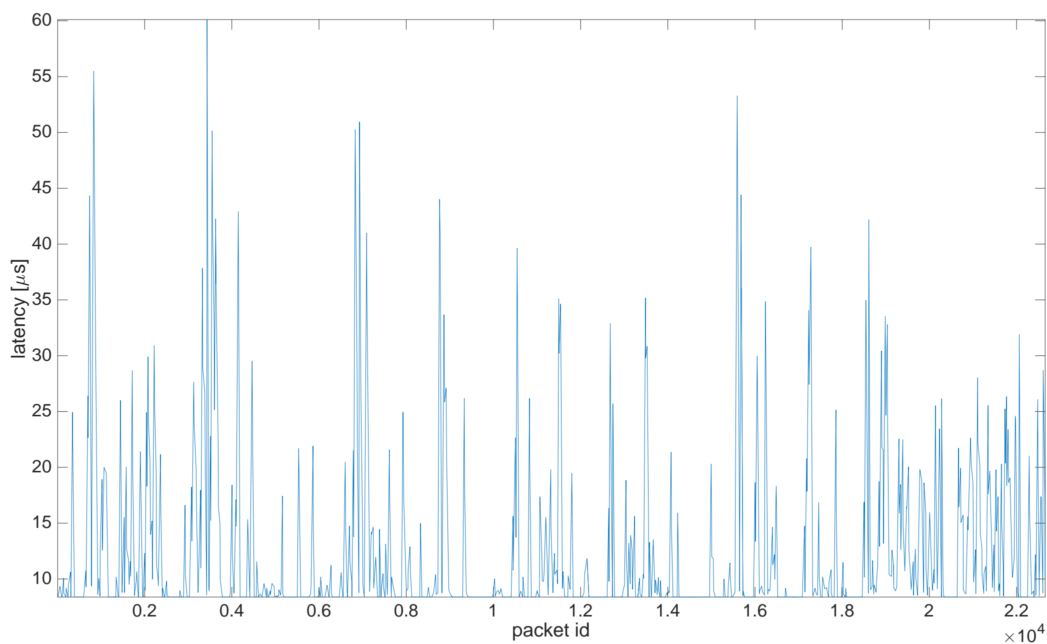


Figure 8.9: Latencies of the light flow packets, without coremask. The worst-case latencies are almost ten-fold compared to the best-case.

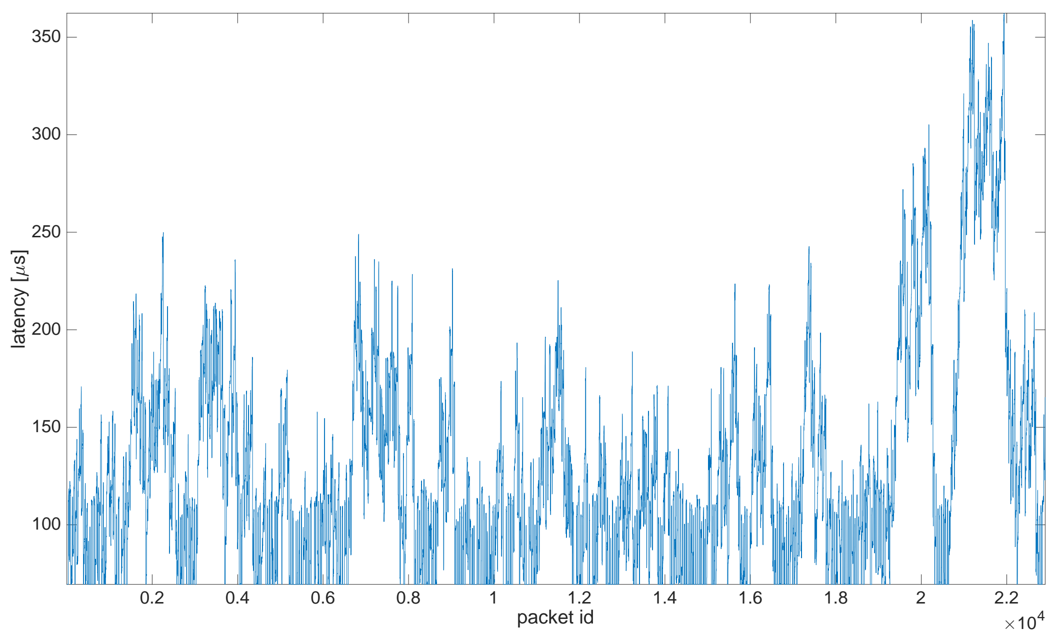


Figure 8.10: Latencies of the heavy flow packets. One of the cores is dedicated for the light flow.

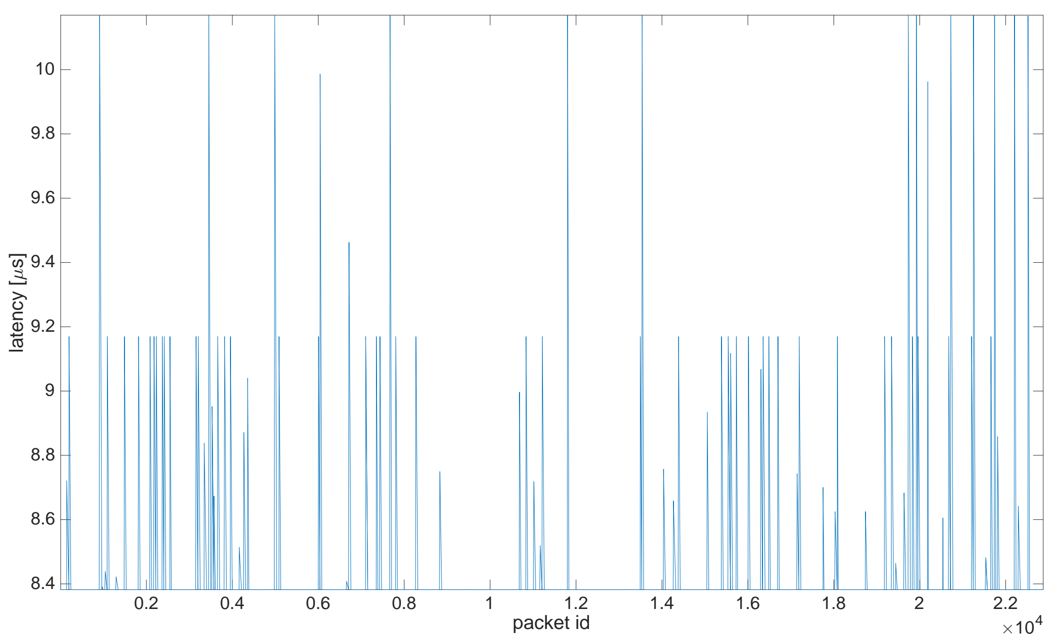


Figure 8.11: Latencies of the light flow packets. One of the cores is dedicated for the light flow. The worst-case latencies are less than 50% higher than of the best-case.

control the resource provision based on global queue priorities and disciplines. As seen in the Figures 8.3 and 8.4, the second processing step produces a bottleneck to the system. This happens due to the atomicity of the second processing queue, forcing heavy processing of the packets one at a time. By dividing the atomic processing step into a smaller atomic part and larger parallel part, the latency of the system should be reduced, as the atomic part of the processing can be done faster, and the cores can fully be utilized by the heavier parallel part of the second processing step. The measurements results of the second application, as shown in the Figures 8.5 and 8.6, validate this behavior.

The second experiment presents the use of queue coremasks, to control the use of specific hardware resources, through the software model. As shown in the Figure 8.9, without limiting the coremask, the light flow packets can be processed in 8μs at best. However, in the worst-case, when all the cores are processing heavy flow packets, the light flow packet latency is almost ten-fold compared to the best-case latency, 61μs. By dedicating a processing core for the light flow, we sacrifice the throughput of the heavy flow in order to reduce the worst-case latency of the light flow. As shown in the Figure 8.11, changing the coremask in the software model removes the bottleneck from the system. The light flow's worst-case packet latencies can be reduced to less than 50% higher than of the best-case, while the heavy flow's throughput drops only 2% compared to the non-coremask processing.

The experiments also highlight the PSE's ability to decouple the hardware and software model. All the changes in the experiment applications can be done easily through the four software model files. Once the resource provision model and the high level resource usage models are built, the application developers can prototype the system with by modifying only the software and workload parts of the model corresponding the real software applications written in task-based programming models, such as Open Event-Machine. At the same time, the hardware developers can work on the model parts corresponding to the real hardware.

According to the experiment results, the PSE's resource network concept, extended with user-defined queue disciplines, seems to be provide the desired flexibility, both in the abstraction level and the modularity, for modeling complex hardware and software co-scheduled many-core systems, such as the reference network processing unit. More importantly, the plug-in code mechanism extends to any resource network based simulation task, providing flexibility, hopefully allowing even more complex systems to be modeled.

8.5 Discussion

8.5.1 Challenges

Performance analysis of packet processing systems involves various challenges. Our work began with the instrumentation of an example packet processing system, but due to the hardware difficulties, we chose to proceed with modeling and simulation methods. The tool, Performance Simulation Environment, used for the study was chosen mainly because of our previous experience with it. The modeling work begins with the proper understanding the different components, such as the hardware, software, and workload of the system under study.

Understanding typical MPSoC based packet processing systems is difficult, due to the architectural heterogeneity, complexity, and non-deterministic behavior. Our study focused on the packet processing system's hardware packet scheduling unit, and the ability to abstract hardware on Open Event-Machine type parallel programming frameworks. We also executed various measurement experiments to gather more detailed characteristics of the hardware memory and ingress- and egress unit behavior. Instrumentation faces the same challenges as any other parallel application development.

Different parallel programming frameworks are used to abstract the hardware from the software application development. To understand the actual applications' effect on the packet processing systems, the underlying runtime frameworks must also be understood. In our work, one of the challenges was to understand the queue based Open Event-Machine implementation, and especially the global scheduler functionality and implementation on different platforms. Also, the actual packet processing functions needed to be understood, in order to create realistic application models.

Hardware system models are difficult to simulate on software; efficient simulation of inherently parallel hardware models requires non-trivial parallelization of the simulator, which has been widely studied research topic for decades. Finding appropriate model abstraction level is essential. On one hand, abstraction level affects the simulation execution time and accuracy of the results. Too detailed models are intractable and require time consuming simulations, while too high-level abstractions hide the sought characteristics of the system, resulting in inadequate results. On the other, it affects the complexity of model construction and reusability. Higher abstraction levels are used more and more often, as the systems under study are becoming more complex.

8.5.2 Discoveries

In this thesis, we created a model of an MPSoC based packet processing system, running Open Event-Machine type task-parallel packet processing applications on dynamic workloads. The goal of Open Event-Machine is to decouple the hardware and software from each other, and further enable code and performance portability between different systems.

By the implemented user-definable queue disciplines, Performance Simulation Environment (PSE), enables flexible adjustment of modeling abstraction levels. We chose to focus on the hardware packet scheduler unit of the system, and modeled it behavior on a level that enables performance analysis of Open Event-Machine applications on dynamic workloads. The chosen abstraction level hides the details of hardware level memory behavior, but still provides accurate details of Open Event-Machine's software level queues.

The abstraction level, naturally, limits the parallelization possibilities of the underlying simulator engine. When the functionality of the hardware scheduler is modeled, even on high abstraction level as done in our work, the modeling of the hardware parallelism on software eliminates the simulation performance. Also, the use of interrelated global queues increases the data dependencies between the simulation executions, resulting in difficulties in the parallel simulation execution. The problem is common when simulating hardware on software. In our experiments, the simulation performance suffered mostly due to the custom select and reserve functions. The choice to model Open Event-Machine's queue system accurately was intentional, despite its obvious effect on simulation performance.

As our experiments demonstrate, the resource network paradigm extended with custom queuing disciplines and support for modeling task parallelism, is adequate tool for performance analysis of network processing systems, running Open Event-Machine based task-parallel applications on dynamic workloads. PSE separates the workload, software, and hardware models, highlighting the ability to decouple the different functional parts of the system, and enabling modularization and further model reuse.

8.5.3 Future Work

There exist several directions for future research around the topic of this thesis. As we have shown in this work, the resource network methodology, extended with user-definable queue disciplines and support for modeling parallel systems, can be used to model more modern MPSoC based network processing units. The user-defined queue disciplines provide desired flexibility to the PSE's modeling and simulation abstraction level. However, in order

to meet the modeling and simulation requirements of larger systems, such as full scale datacenter networks or complex IoT applications, PSE needs to be further developed.

Simulation of large-scale models faces the same challenges as any large scale application. Due to the growing memory and computing resources requirements, the resource usage of PSE needs to be improved.

Despite the more efficient data structures and single threaded execution, simulator software faces the same problems as any other computer software: efficient scaling requires parallelization of the software. Parallelization of discrete-event simulator software is a wide research topic itself, and faces multiple challenges, not only the ones faced with parallelization in general, but also the simulators' nature makes the problem non-trivial. The parallelization of PSE could be done in several ways. With the current implementation of the PSE, the conservative parallel simulation (the simulation threads advance conservatively) approach seems to be most viable solution. However, to enable more advanced parallelization methods, e.g. variations of optimistic and conservative approaches, parts of the simulator core needs to be rewritten, as the current version of PSE does not support any history of the simulation.

Another future research topic around resource networks is the support for virtualization. Virtualization is an important topic in today's computing, and for example large-scale cloud- and fog datacenters are contingent on the efficient virtualization methods. Performance analysis methods need to adapt to support the needs to further understand these systems. A natural way to model virtualization with resource network concept, would be to add a new virtualization model layer between the resource utilization and resources provision models.

Chapter 9

Conclusions

This thesis investigated the use of measurement, simulation, and modeling methods for the performance analysis of MPSoC based packet processing systems running task-parallel applications. The motivation behind our work was to enable more accurate modeling of task-parallel software abstractions, such as Open Event-Machine, on the resource network concept.

We approached the problem by extending the toolset of an existing in-house modeling and simulation software, Performance Simulation Environment. The extensions enable modeling of user-definable queue disciplines, which further enable flexible modeling of complex hardware interactions of MPSoCs and the parallelism of task-based programming models.

We studied, instrumented, and measured the characteristics of a packet processing systems. Based on our findings, we modeled a multi-blade packet processing system with customizable workload and task-parallel application models, and run simulation experiments.

The experiment results suggest that resource network concept, extended with global user-definable queue disciplines, is a viable tool for the performance analysis of packet processing systems. The chosen abstraction level provides desired balance between the functionality, ease of use, and simulation performance. However, further research is required in order to scale such a method to support ultra-large-scale simulations.

Bibliography

[1] 6Wind. Fast path architecture, 2016. URL http://www.6wind.com/products/6windgate/optimized-architecture/. Accessed 6.4.2016.

[2] Akka. Akka framework. URL http://akka.io/. Accessed 6.4.2016.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-175-0. doi: 10.1145/1402958.1402967. URL http://doi.acm.org/10.1145/1402958.1402967.

[4] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855711.1855730.

[5] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 503–514, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2836-4. doi: 10.1145/2619239.2626316. URL http://doi.acm.org/10.1145/2619239.2626316.

[6] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560. URL http://doi.acm.org/10.1145/1465482.1465560.

[7] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, University of California, Berkeley, December 2006.

[8] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb 2002. ISSN 0018-9162. doi: 10.1109/2.982917.

[9] Jerry Banks. *Discrete-event system simulation*. Pearson Education, Upper Saddle River, N.J, cop. 2010. ISBN 978-0-13-815037-2.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003. ISSN 0163-5980. doi: 10.1145/1165389.945462. URL http://doi.acm.org/10.1145/1165389.945462.

[11] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006. ISBN 978-0-471-56525-3.

[12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1519-7. doi: 10.1145/2342509.2342513. URL http://doi.acm.org/10.1145/2342509.2342513.

[13] Robert Braden. Requirements for internet hosts-communication layers, 1989. URL https://tools.ietf.org/html/rfc1122. Accessed 25.4.2016.

[14] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Trans. Comput. Syst.*, 30 (4):12:1–12:51, November 2012. ISSN 0734-2071. doi: 10.1145/2382553.2382554. URL http://doi.acm.org/10.1145/2382553.2382554.

[15] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and

simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[16] H. Jonathan Chao and Bin Liu. *High Performance Switches and Routers*. Wiley-IEEE Press, 2007. ISBN 0470053674.

[17] Sanjay Chatterjee, Sagnak Tasirlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. Integrating asynchronous task parallelism with mpi. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*, pages 712–725, 2013. doi: 10.1109/IPDPS.2013.78. URL http://dx.doi.org/10.1109/IPDPS.2013.78.

[18] CloudSimEx. Cloudsimex, 2012. URL https://github.com/Cloudslab/CloudSimEx. Accessed 7.4.2016.

[19] Linux Containers. Linux containers, 2013. URL https://linuxcontainers.org/. Accessed 7.4.2016.

[20] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997. ISBN 1558603433.

[21] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[22] Stephen E Deering. Internet protocol, version 6 (ipv6) specification, 1998. URL https://tools.ietf.org/html/rfc2460. Accessed 25.4.2016.

[23] P. Demestichas, A. Georgakopoulos, D. Karvounas, K. Tsagkaris, V. Stavroulaki, J. Lu, C. Xiong, and J. Yao. 5g on the horizon: Key challenges for the radio-access network. *IEEE Vehicular Technology Magazine*, 8(3):47–53, Sept 2013. ISSN 1556-6072. doi: 10.1109/MVT.2013.2269187.

[24] Benoît Des Ligneris. Virtualization of linux based computers: the linux-vserver project. In *High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on*, pages 340–346. IEEE, 2005.

[25] Marios D Dikaiakos, Dimitrios Katsaros, Pankaj Mehra, George Pallis, and Athena Vakali. Cloud computing: Distributed internet computing for it and scientific research. *Internet Computing, IEEE*, 13(5):10–13, 2009.

[26] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629578. URL http://doi.acm.org/10.1145/1629575.1629578.

[27] N. Egi, J. Du M. Dobrescu, B.-G. Chun K. Argyraki, K. Fall, G. Iannaccone, A. Knies, M. Manesh, L. Mathy, and S. Ratnasamy. Understanding the packet processing capabilities of multi-core servers. Internet draft, EPFL, February 2009. URL http://routebricks.org/papers/rb-sosp09.pdf.

[28] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0472-6. doi: 10.1145/2000064.2000108. URL http://doi.acm.org/10.1145/2000064.2000108.

[29] Dave Evans. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 1:1–11, 2011.

[30] Inc. Facebook. Facebook, 2016. URL http://www.facebook.com. Accessed 24.03.2016.

[31] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 13–23, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. doi: 10.1145/1250662.1250665. URL http://doi.acm.org/10.1145/1250662.1250665.

[32] C Fraleigh, S Moon, C Diot, B Lyles, and F Tobagi. Packet-level traffic measurement from a tier-1 ip backbone. *Sprint ATL, Burlingame, CA Sprint ATL Technical Report TR01-ATL-110101*, 2001.

[33] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, October 1990. ISSN 0001-0782. doi: 10.1145/84537.84545. URL http://doi.acm.org/10.1145/84537.84545.

[34] Ran Giladi. *Network processors: architecture, programming, and implementation.* Morgan Kaufmann, 2008. ISBN 0080919596, 9780080919591.

[35] Google. Google summa, 2016. URL https://www.google.com/about/datacenters/inside/locations/hamina/. Accessed 6.4.2016.

[36] S. Govind, R. Govindarajan, and J. Kuri. Packet reordering in network processors. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, March 2007. doi: 10.1109/IPDPS.2007.370287.

[37] Herbert Haas. Mausezahn. URL http://www.perihel.at/sec/mz/. Accessed 25.7.2015.

[38] J. Hanhirova and V. Hirvisalo. Pse – performance simulation environment. Technical report, Aalto University, School of Science, 2014.

[39] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based load balancing for fast datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 465–478, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3542-3. doi: 10.1145/2785956.2787507. URL http://doi.acm.org/10.1145/2785956.2787507.

[40] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan and Claypool Publishers, 1st edition, 2009. ISBN 159829556X, 9781598295566.

[41] Chris Horne. Understanding full virtualization, paravirtualization and hardware assist. *White paper, VMware Inc*, 2007.

[42] C.J. Hughes, V.S. Pai, P. Ranganathan, and S.V. Adve. Rsim: simulating shared-memory multiprocessors with ilp processors. *Computer*, 35 (2):40–49, Feb 2002. ISSN 0018-9162. doi: 10.1109/2.982915.

[43] Texas Instruments. Open event machine library user guide. http://software-dl.ti.com/sdoemb/sdoemb_public_sw/bios_mcsdk/latest/index_FDS.html, 2012. Distributed with OpenEM library. Accessed 22.5.2015.

[44] Intel. Data plane development kit documentation. URL `http://dpdk.org/doc/guides/index.html`. Accessed 6.4.2016.

[45] ISO/IEC 7498-1:1994. Information Technology — Open Systems Interconnection — Basic Reference Model: The Basic Model. ISO/IEC 7498-1:1994, ISO, Geneva, Switzerland, November 1994. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=20269`.

[46] K.R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, Harvey J. Wasserman, and N.J. Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 159–168, Nov 2010. doi: 10.1109/CloudCom.2010.69.

[47] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation and modeling*. Wiley, New York, 1991. ISBN 978-0-471-50336-1.

[48] M.H. Jamal, G. Mustafa, A. Waheed, and W. Mahmood. An extensible infrastructure for benchmarking multi-core processors based systems. In *Performance Evaluation of Computer Telecommunication Systems, 2009. SPECTS 2009. International Symposium on*, volume 41, pages 13–20, July 2009.

[49] Pradeeban Kathiravelu and Luis Veiga. Concurrent and distributed cloudsim simulations. In *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, pages 490–493. IEEE, 2014.

[50] H. Kim and N. Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, February 2013. ISSN 0163-6804. doi: 10.1109/MCOM.2013.6461195.

[51] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan. Greencloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing*, 62(3):1263–1283, 2010. ISSN 1573-0484. doi: 10.1007/s11227-010-0504-1.

[52] Jonathan G. Koomey. Estimating total power consumption by servers in the U.S. and the world. Technical report, Lawrence Derkley National Laboratory, February 2007.

[53] V. P. Kumar, T. V. Lakshman, and D. Stiliadis. Beyond best effort: router architectures for the differentiated services of tomorrow's internet. *IEEE Communications Magazine*, 36(5):152–164, May 1998. ISSN 0163-6804. doi: 10.1109/35.668286.

[54] Wely Lau. A comprehensive introduction to cloud computing, 2011. URL `https://www.simple-talk.com/cloud/development/a-comprehensive-introduction-to-cloud-computing/`. Accessed 7.4.2016.

[55] Björn Liljeqvist. Visions and facts–a survey of network processors. *Electronic Engineering, Chalmers*, 2003.

[56] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso. A survey of performance modeling and simulation techniques for accelerator-based computing. *IEEE Transactions on Parallel and Distributed Systems*, 26(1):272–281, Jan 2015. ISSN 1045-9219. doi: 10.1109/TPDS.2014.2308216.

[57] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb 2002. ISSN 0018-9162. doi: 10.1109/2.982916.

[58] Peter M. Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, United States, 2011.

[59] Daniel A. Menascé, Virgílio A. F. Almeida, and Larry W. Dowdy. *Capacity Planning and Performance Modeling: From Mainframes to Client-server Systems.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994. ISBN 0-13-035494-5.

[60] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014. ISSN 1075-3583. URL `http://dl.acm.org/citation.cfm?id=2600239.2600241`.

[61] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998. ISSN 0018-9219. doi: 10.1109/JPROC.1998.658762.

[62] James Murty. *Programming Amazon Web Services.* O'Reilly, first edition, 2008. ISBN 9780596515812.

[63] OpenVZ. Openvz, 2013. URL `http://www.openvz.org`. Accessed 7.4.2016.

[64] Ioannis Papaefstathiou, Theofanis Orphanoudakis, George Kornaros, Christopher Kachris, Ioannis Mavroidis, and Aristides Nikologiannis. Queue management in network processors. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 3*, pages 112–117. IEEE Computer Society, 2005.

[65] David A Patterson and John L Hennessy. Computer organization and design. *Morgan Kaufmann*, pages 474–476, 2007.

[66] Michael Pearce, Sherali Zeadally, and Ray Hunt. Virtualization: Issues, security threats, and solutions. *ACM Comput. Surv.*, 45(2):17:1–17:39, March 2013. ISSN 0360-0300. doi: 10.1145/2431211.2431216. URL `http://doi.acm.org/10.1145/2431211.2431216`.

[67] Harry Perros. Computer simulation techniques: The definitive introduction!, 2009. URL `http://www.csc.ncsu.edu/faculty/perros//simulation.pdf`.

[68] James L Peterson. Petri net theory and the modeling of systems. 1981.

[69] US Rackspace. Inc., the rackspace cloud, 2010.

[70] Tejaswi Redkar, Tony Guidici, and Todd Meister. *Windows Azure Platform*, volume 1. Springer, 2011. ISBN 1430235632, 978-1430235637.

[71] Juergen Ributzka. *Concurrency and synchronization in the modern many-core era: Challenges and opportunities.* PhD thesis, 2013.

[72] Ihsan Sabuncuoglu and Ahmet Hatip. The turkish army uses simulation to model and optimize its fuel-supply system. *Interfaces*, 35(6):474–482, 2005. doi: 10.1287/inte.1050.0173.

[73] Salesforce. Salesforce, 2016. URL `http://www.salesforce.com`. Accessed 24.03.2016.

[74] Dan Sanderson. *Programming google app engine: build and run scalable web apps on google's infrastructure.* " O'Reilly Media, Inc.", 2009.

[75] Stefania Sesia, Issam Toufik, and Matthew Baker. *LTE - The UMTS Long Term Evolution: From Theory to Practice.* Wiley Online Library, 2nd edition, 2011. ISBN 0470660252, 978-0470660256.

[76] Y. Shafranovich. Common format and mime type for comma-separated values (csv) files, 2005. URL https://tools.ietf.org/html/rfc4180. Accessed 25.3.2016.

[77] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 275–287, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. doi: 10.1145/1272996.1273025. URL http://doi.acm.org/10.1145/1272996.1273025.

[78] Richard M Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3. 3.* 2009. ISBN 144141276X, 9781441412768.

[79] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[80] Ryan Teeter and Karl Barksdale. *Google Apps for Dummies*. John Wiley & Sons, 2011. ISBN 0470189584, 978-0470189580.

[81] William Thies, Michal Karczmarek, and Saman Amarasinghe. *Compiler Construction: 11th International Conference, CC 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings*, chapter StreamIt: A Language for Streaming Applications, pages 179–196. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-45937-8. doi: 10.1007/3-540-45937-5_14. URL http://dx.doi.org/10.1007/3-540-45937-5_14.

[82] Vernon Turner, John F Gantz, David Reinsel, and Stephen Minton. The digital universe of opportunities: Rich data and the increasing value of the internet of things. *IDC Analyze the Future*, 2014.

[83] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C.M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain K?gi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38 (5):48–56, 2005. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2005.163.

[84] A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan. Scale-out networking in the data center. *IEEE Micro*, 30(4):29–41, July 2010. ISSN 0272-1732. doi: 10.1109/MM.2010.72.

[85] Luis M. Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *SIGCOMM Comput. Commun. Rev.*, 44(5):27–32, October 2014. ISSN 0146-4833. doi: 10.1145/2677046.2677052. URL `http://doi.acm.org/10.1145/2677046.2677052`.

[86] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996. ISBN 0-13-508301-X.

[87] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, (4):27–75, 1993.

[88] Carl Wallén, Petri Savolainen, Krister Wikström, and Matias Elo. Open event machine introduction. URL `https://sourceforge.net/projects/eventmachine/files/Documents/EM_introduction_1_0.pdf/download`. Accessed 6.4.2016.

[89] Brian Walters. Vmware virtual platform. *Linux J.*, 1999(63es), July 1999. ISSN 1075-3583. URL `http://dl.acm.org/citation.cfm?id=327906.327912`.

[90] Vincent M Weaver and Sally A McKee. Are cycle accurate simulations a waste of time. In *Proc. 7th Workshop on Duplicating, Deconstructing, and Debunking*, pages 40–53, 2008.

[91] B. Wickremasinghe, R. N. Calheiros, and R. Buyya. Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 446–452, April 2010. doi: 10.1109/AINA.2010.32.

[92] M. G. Xavier, M. V. Neves, and C. A. F. D. Rose. A performance comparison of container-based virtualization systems for mapreduce clusters. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 299–306, Feb 2014. doi: 10.1109/PDP.2014.78.

[93] L. Yang, R. Dantu, T. Anderson, and R. Gopal. Forwarding and control element separation (forces) framework. Technical report, United States, 2004.

[94] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010. ISSN 1869-0238. doi: 10.1007/s13174-010-0007-6. URL `http://dx.doi.org/10.1007/s13174-010-0007-6`.