Mälardalen University Press Doctoral Theses
No. 270

UTILIZING HARDWARE MONITORING TO IMPROVE THE QUALITY OF SERVICE AND PERFORMANCE OF INDUSTRIAL SYSTEMS

Marcus Jägemar

October 2018

MÄLARDALEN UNIVERSITY
SWEDEN

ISBN 978-91-7485-395-7
ISSN 1651-4238
Printed by E-Print AB, Stockholm, Sweden

Äntligen![1]

My own translation:

Finally!

— Gert Fylking, 2000 [107]

[1] The debater Gert Fylking attended the Nobel literature prize announcement several consecutive years (2000–2002) and exclamated "finally" when the winner was announced. His comment implied that the prize winner was unknown for the people that didn't belonging to the cultural elite. In this thesis we interpret the quote explicitly, that the thesis is finished at last!

Abstract

THE drastically increased use of information and communications technology has resulted in a growing demand for telecommunication network capacity. The demand for radically increased network capacity coincides with industrial cost-reductions due to an increasingly competitive telecommunication market. In this thesis, we have addressed the capacity and cost-reduction problems in three ways.

Our first contribution is a method to support shorter development cycles for new functionality and more powerful hardware. We reduce the development time by replicating the hardware utilization of production systems in our test environment. Having a realistic test environment allows us to run performance tests at early design phases and therefore reducing the overall system development time.

Our second contribution is a method to improve the communication performance through selective and automatic message compression. The message compression functionality monitors transmissions continuously and selects the most efficient compression algorithm. The message compression functionality evaluates several parameters such as network congestion level, CPU usage, and message content. Our implementation extends the communication capacity of a legacy communication API running on Linux where it emulates a legacy real-time operating system.

Our third and final contribution is a framework for process allocation and scheduling to allow higher system performance and quality of service. The framework continuously monitors selected processes and correlate their performance to hardware usage such as caches, floating point unit and similar. The framework uses the performance-hardware correlation to allocate processes on multi-core CPUs for minimizing shared hardware resource congestion. We have also designed a shared hardware resource aware process scheduler that allows multiple processes to co-exist on a CPU without suffering from performance degradation through hardware resource congestion. The allocation and scheduling techniques can be used to consolidate several functions on shared hardware

thus reducing the system cost. We have implemented our process scheduler as a new scheduling class in Linux and evaluated it extensively.

We have conducted several case studies in an industrial environment and verified all contributions in the scope of a large telecommunication system manufactured by Ericsson. We have deployed all techniques in a complicated industrial legacy system with minimal impact. We have shown that we can provide a cost-effective solution, which is an essential requirement for industrial systems.

Sammanfattning

TELEKOMMUNIKATIONSBRANCHEN står just nu inför en stor utmaning där kommunikationsprestanda och snabba leveranstider blir allt mer viktiga för att positionera sig i den ökande konkurrensen. I denna avhandling har vi addresserat detta problem på tre sätt. Det första sättet är att reducera utvecklingstiden genom att replikera hårdvarulast från produktionssystem på testnoder. Den andra genom att förbättra kommunikationsprestandan genom automatisk meddelandekomprimering. Den tredje, och sista, genom att implementera allokerings- och schemaläggningstekniker som möjliggör konsolidering av mjukvara på delad hårdvara.

Våra tekniker reducerar utvecklingstiden genom att flytta en del av prestandaverifikationen från slutet av utvecklingscykeln till den mycket tidigare programmeringsfasen. Vi metod börjar med att mäta resursanvändande och prestandan för ett produktionssystem som kör hos en kund. Från dessa mätningar skapar vi en modell som vi sedan använder för att återskapa hårdvarulasten på en testnod. Att köra funktionstester på ett testsystem som har liknande hårdvarulast ger ett tillförlitligt resultat. Genom att använda vår metod kan vi flytta vissa tester från prestandaverifikationen i slutet av utvecklingscykeln till programmeringsfasen och därmed spara utvecklingstid.

Under våra tester märkte vi att kommunikationssystemet var överlastat och att processorn inte användes fullt ut. För att öka kommunikationsprestandan implementerade vi en metod som automatiskt komprimerar meddelanden när det finns processorkapacitet att använda. Vi implementerade ett reglersystem som väljer den bästa ur en mängd av kompressionsalgoritmer. Vår mekanism utvärderar automatiskt alla algoritmer och reagerar på förändringar i processorlast, nätverkslast eller meddelandeinnehåll.

Av ekomiska skäl vill företaget konsolidera flera mjukvarufunktioner till en hårdvara. När vi testade prestandan före och efter konsolidering märkte vi en synbar prestandaförsämring. Orsaken till prestandaförsämringen var att programmen som förut körde själva nu skall dela på resurser såsom cache och liknande. Vi har också utvecklat en teknik för att automatiskt allokera processer

på ett kluster av kärnor för att maximera prestandan. Vi utvecklade också en teknik för att låta flera processer dela på en kärna utan att för den skull påverka quality of service för varandra. I vår implementation använder vi performance monitoring unit (PMU) för att mäta resursanvändning. Vi programmerar också PMU så att den genererar ett avbrott när en process har uttömt sin tilldelade mängd av resurser.

All programvaruutveckling och test har genomförts på ett industriellt telekommunikationssystem tillverkat av Ericsson. Alla tekniker är implementerade för bruk i produktionsmiljö och monitorerings- och modelleringsfunktionaliteten används kontinuerligt i felsökningsysfte av produktionssystemet. De tekniker vi presenterar i denna avhandling ger också en kostnadseffektiv lösning, vilket är en viktigt krav för industriella system.

I dedicate this thesis to my beloved wife Karolinn and my lovely daughters Amelie, Lovisa and Elise.
[Art by L. Jägemar]

Acknowledgements

I could not imagine the amazing personal journey it is when studying for a Ph.D. I utterly and completely underestimated the level of commitment and the massive amount of work involved in finishing a Ph.D., as stated by Paulsen [227]: *"The Ph.D. education is the highest education available - no wonder that it is demanding and difficult. Accept this."*. I have come to many insights during my studies. The most important personal insight is that this thesis is not a product of a gifted genius that shuffled through the studies and quickly finished it. It is something completely different. Writing a Ph.D. thesis, has for me, been much more of a constant challenge requiring long-term goals that can bridge short-term ups and downs. Having at least a tiny fraction of what is nowadays called grit [72, 73] is something that I discovered in the very final stages of writing this thesis. This determination saw me through seven years of part-time research interrupted with industrial work and several parental leaves. Working on this thesis is one of the most rewarding achievements in my professional life. I wish that all people would get the opportunity to fulfill their goals in the same way as I have.

I could not have done this without help from many people. The following people have in various ways been involved in my Ph.D. project.

First of all, I would like to thank my supervisors and co-authors, Björn Lisper, Sigrid Eldh, Andreas Ermedahl and Moris Behnam for your knowledge, support, patience and constructive discussions during my studies. We have in many ways acted as a team with diverse competencies and achieved many exciting results. I would also like to express gratitude towards my manager at Ericsson, Magnus Schlyter, who supported me from the first day until completing the thesis.

The work presented in this Ph.D. thesis has been funded by Ericsson and the Swedish Knowledge Foundation (KK stiftelsen) through the ITS-EASY [279] industrial Ph.D. school at Mälardalen University.

Furthermore, thanks to all students in the ITS-EASY research group, we all share the ups and downs of studying for a PhD; Apala Ray, Daniel Hallmans, Daniel Kade, David Rylander, Eduard Paul Eniou, Fredrik Ekstrand, Gaetana Sapienza, Kristian Wiklund, Markus Wallmyr, Mehrdad Saadatmand, Melika Hozhabri, Sara Dersten, Stephan Baumgart, and Tomas Olsson.

I would also like to thank my additional co-authors: Jakob Danielsson, Gordana Dodig-Crnkovic, Rafia Inam, Mikael Sjödin, Daniel Hallmans, Stig Larsson and Thomas Nolte. I enjoyed working with you.

I have the greatest gratitude to my parents; my mother and father who always wanted me to study hard to become something they never could.

Finally and foremost, I want to express my endless love for my wife Karolinn and our three daughters, Amelie, Lovisa, and Elise. I would not have been able to write this thesis without your support and encouragement. I am also grateful for all support and understanding when being away on conference trips and writing papers late at night.

Marcus Jägemar

September 2018, Sigtuna, Sweden

“ “

— The unsuccessful self-treatment of a case of “writer’s block”,
Dennis Upper [289][2]

[2]Dennis Upper pinpoints the writer’s block problem in his empty article. The comedy continues as the reviewer sarcastically states that he cannot find any faults, although he has used both lemon juice and X-ray when reviewing the article. The problem is real, as many writers can vouch for, even though the article is a joke.

List of Publications

THIS thesis is a monograph based on multiple contributing conference papers, technical reports, patents, and journal article. The following list shows the publications most closely related to the topics presented in the thesis, followed by other publications by the author.

Related Publications

A Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl and Björn Lisper. *Towards Feedback-Based Generation of Hardware Characteristics.* In Proceedings of the International Workshop on Feedback Computing, 2012 [150].

B Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl and Björn Lisper. *Automatic Multi-Core Cache Characteristics Modelling.* In Proceedings of the Swedish Workshop on Multicore Computing, Halmstad, 2013 [151].

C Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl and Björn Lisper. *Automatic Message Compression with Overload Protection.* Journal of Systems and Software, 2016 [153].

This journal article is an extension of the already published paper H [152].

D Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl and Moris Behnam. *A Scheduling Architecture for Enforcing Quality of Service in Multi-Process Systems.* In Proceedings of Emerging Technologies and Factory Automation (ETFA), Limasol, Cyprus, 2017 [157].

This paper is an extension of patent O [155].

E Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl, Moris Behnam and Björn Lisper. *Enforcing Quality of Service Through Hardware Resource Aware Process Scheduling.* In Proceedings of Emerging Technologies and Factory Automation (ETFA), Torino, Italy, 2018 [147].

This paper is an extension of patent P [156].

Other Publications

F Rafia Inam, Mikael Sjödin and Marcus Jägemar. *Bandwidth Measurement using Performance Counters for Predictable Multicore Software.* Proceedings of the International Conference on Emerging Technologies and Factory Automation (ETFA), 2012. [136]

G Daniel Hallmans, Marcus Jägemar, Stig Larsson and Thomas Nolte. *Identifying Evolution Problems for Large Long Term Industrial Evolution Systems.* In Proceedings of IEEE International Workshop on Industrial Experience in Embedded Systems Design, Västerås, 2014. [122]

H Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl and Björn Lisper. *Adaptive Online Feedback Controlled Message Compression.* In Proceedings of Computers, Software and Applications Conference (COMPSAC), Västerås, 2014. [152]

I Marcus Jägemar and Gordana Dodig-Crnkovic *Cognitively Sustainable ICT with Ubiquitous Mobile Services - Challenges and Opportunities.* In Proceedings of the International Conference on Software Engineering (ICSE), Firenze, Italy, 2015. [146]

J Jakob Danielsson, Marcus Jägemar, Moris Behnam and Mikael Sjödin. *Investigating Execution-Characteristics of Feature-Detection Algorithms.* In Proceedings of Emerging Technologies and Factory Automation (ETFA), Limasol, Cyprus, 2017. [59]

K Jakob Danielsson, Marcus Jägemar, Moris Behnam, Mikael Sjödin, Tiberiu Seceleanu. *Measurement-based evaluation of data-parallelism for OpenCV feature-detection algorithms.* In Proceedings of Computers, Software and Applications Conference (COMPSAC), 2018. [60]

L Marcus Jägemar. *Mallocpool: Improving Memory Performance Through Contiguously TLB Mapped Memory*, In Proceedings of Emerging Technologies and Factory Automation (ETFA), 2018. [145].

Technical Reports

M Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl and Björn Lisper. *Technical Report: Feedback-Based Generation of Hardware Characteristics*, 2012. [149].

N Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl, Björn Lisper and Gabor Andai. *Automatic Load Synthesis for Performance Verification in Early Design Phases.* Technical Report, 2016. [154].

This technical report is an extension of the already published papers A [150], B [151] and the technical report M [149].

Patents

O Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl. *Decision support for OS process scheduling based on HW-, OS- and system-level performance counters*, Pat. Pending 62/400353, United States, 2016. [155].

P Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl. *Process scheduling in a processing system having at least one processor and shared hardware resources*, PCT/SE2016/050317, United States, 2016. [156].

Licentiate Thesis

Q Marcus Jägemar. *Utilizing Hardware Monitoring to Improve the Performance of Industrial Systems*, Licentiate Thesis[3], Mälardalen University, 2016. [144].

[3] A licentiate is an intermediate postgraduate degree academically merited between an M.Sc. and a Ph.D.

Never, for the sake of peace and quiet, deny your own experience or convictions.

— Dag Hammarskjöld[4]

[4]Secretary-General of the United Nations 1955-61, Nobel prize winner 1961.

Contents

I

Thesis

Talk is cheap. Show me the code.

— Linus Torvalds [288]

1 Introduction

In this thesis we share the result from our investigation on how to make a large-scale [122] telecommunication system [23] more competitive by improving its software. The system we investigated has a significant market share [293] and is used as an infrastructure system throughout the world. We start this Chapter in Section 1.1 by outlining our research goals and process.

Our contributions comes from four areas. First, we investigate how to monitor the performance of the system with the goal to identify and fix performance-related software bugs, Section 1.2. We use our performance monitor throughout the rest of our work. Secondly, we utilize the performance monitor to model the execution behavior of the production system, Section 1.3. We use the model to replicate the load and mimicking the execution behavior of the production system on test nodes in the lab. Mimicking the execution behavior is useful for finding performance-related bugs in the early phases of the development process. Thirdly, we use the performance monitor to identify a performance problem in the communication subsystem of our target system, Section 1.4. To improve the performance we devised an automatic compression mechanism that trades CPU capacity when there is limited bandwidth. In our fourth and final contribution, Section 1.5, we devised a method to efficiently allocate processes over multiple CPUs while enforcing Quality of Service (QoS) through process scheduling. The driving force is the increased demand to consolidate several system functions on fewer multi-core CPUs without them affecting the QoS of each other. We finish this chapter in Section 1.6 by giving a short overview of the chapters in this thesis.

1.1 Researching Uncharted Territories

The primary goal of our research has always been to improve the performance of our target system. Defining the outcome as well as the way to reach it was one of the tasks in our assignment.

Researching the topics presented in this thesis has been like sailing through uncharted territories. We started off with a clear goal to investigate the execution characteristics, i.e., hardware usage and system performance. In doing so, we implemented a tool called Charmon, see Section 1.2 that helped us to monitor computers, denoted nodes, in a computer system for days and weeks continuously. The low resource usage of Charmon enabled us to run it on production systems, revealing information on how our monitored system performed in real-world situations.

The next step was to utilize the execution characteristics data to create a model of the production node hardware resource usage. We implemented a tool called Loadgen, see Section 1.3, which automatically mimics the previously monitored production system on a test node. The Loadgen program implements a feedback controller that controls several memory access loops so that the test node reaches the same cache utilization as the production node.

We noticed that the communication performance was not efficient during the implementation and verification phase of the Loadgen tool. More specifically, the performance dropped when the network was congested. By investigating the Charmon logs further, we deduced that the CPU load was low in many of the network congestion situations. Our idea was to implement a mechanism, see Section 1.4, that automatically, and transparently, compress messages when there is available CPU capacity, and the network congestion level is high.

To reduce the manufacturing costs the product department decided to consolidate several system functions on fewer CPUs. We utilized Charmon to investigate the effects of simultaneously running multiple system functions on one CPU. The effects were quickly detected, and the memory subsystem suffered heavily when several IO-intensive applications competed for resources. The discoveries triggered us to look at current process allocation and scheduling algorithms. We could not find any suitable algorithms in the literature, so we devised a new algorithm, see Section 1.5. Our algorithm automatically evaluates the resource usage of processes and allocated them appropriately over a set of cores on a multi-core CPU. We also developed a scheduling algorithm to enforce QoS so that the system functions do not affect the execution performance of each other. Both these algorithms have resulted in patents [155, 156].

In short, the Charmon tool was an eye-opener for us. The tool provides much information on system behavior. It became easy to evaluate and develop improvements intelligently by utilizing the Charmon tool. It provided information that was invaluable when motivating the need for improvements. Particularly, when providing execution characteristics from production environments.

1.2 Monitoring a Production System

We implemented a characteristics monitoring tool aimed for running at customer sites. Our goal with the monitoring tool was to get a better understanding of real-world systems by sampling the hardware usage. Our monitor samples hardware events from the CPU or any other low-level hardware components. We grouped these events into sets that represent a certain type of behavior, for example, cache-usage, translation lookaside buffer (TLB) usage, cycles per instruction. Running a monitoring tool in a production environment pose special restrictions and requirements such as:

- It must be possible to run the monitor on a production system.
- The monitor must have a low probe-effect [99] since it is not allowed to affect the behavior and performance of production system.
- The monitor must be able to capture long time intervals because the system behavior changes slowly depending on end-customer usage.

We addressed the production environment constraints by being very restrictive when implementing the monitoring application. First, we followed the company development process when implementing our monitoring application. Several experiences system engineers reviewed the system design and we verified the application in our test environment. It is vital that no undesired behavior or faults occur when running in a sensitive environment. Secondly, we have chosen a low hardware event sample frequency (1Hz) to reduce the probe effect. A low sampling frequency also reduce the memory requirement for hardware usage samples. The sampling frequency is sufficient for the slowly changing behavior of our target system.

1.3 Modeling a Production System

We devised a method that automatically synthesizes a hardware characteristics model from data obtained by the monitoring tool, see Section 1.2. The model can replicate the hardware usage of the production system.

Our goal was to create an improved test suite consisting of a hardware characteristics model together with a functional test suite. We assumed that such a test suite should improve testing and make it possible to discover, primarily performance related bugs, in the early stages of system development. Finding bugs in the early design phases adheres well to the desire of reducing the total system development time since bug-fixing becomes much more difficult and time-consuming further from the introduction of the bug [29].

Our method uses a Proportional Integrative Derivative (PID) controller [22] to synthesize the model automatically from the hardware characteristics data obtained through our monitoring tool. No manual intervention is needed. The overall method is generic and supports any type of hardware characteristics. The system we investigated is memory-bound and mostly limited by cache and memory bandwidth. We have implemented one PID-control loop per characteristics entity. In our model, we have used L_1I-cache, L_1D-cache and L_2D-cache hardware usage to represent the behavior of the system.

Definition 1 The *cache* acts as a small intermediate memory that is substantially faster than the RAM. The subscript index determine the *cache level*, starting with 1 for the first cache-level. The capital letter "I" denotes the *instruction cache* and "D" denotes the *data cache*.

Definition 2 The *translation lookaside buffers* (TLB) temporarily store memory mappings between the virtual, which is visible to a process, and the physical address space. The capital letter "I" denotes *instruction* and "D" denotes *data*.

We have evaluated our monitoring and modeling method by synthesizing a model for L_1I-cache, L_1D-cache, and L_2D-cache misses according to the hardware characteristics extracted from a running production system. We have successfully tested our load synthesize model by detecting a bug that was not possible to find in the original test suite. The message RTT degradation was 0.75% when we tested a new version of the production system on the original test suite. Such small performance degradation is not possible to detect with the automated test suite because it is within the limits of performance variation of the system. We detected a performance degradation of 10.8% when running the

test suite together with Loadgen, which clearly signals a performance problem and it is readily detectable by the automated test suite.

1.4 Improving the Communication System

We contrived and implemented a mechanism to automatically find and use a compression algorithm that provides the shortest message Round-Trip Time (RTT) between two nodes in a communication system.

Our goal, when performing this work, was to improve the communication performance of our target system. We had already implemented the monitoring tool, Section 1.2, and the characteristics model, Section 1.3 and could use these tools for performance measurements.

We added a software metric to our monitoring tool, measuring message RTT. We could deduce that 1) The message RTT varied depending on the network congestion levels and 2) The hardware usage varied but was relatively low in certain conditions. We assumed that we could trade computational capacity for an increased messaging capacity by using message compression. We defined some critical considerations such as:

- The compression algorithm must be selected automatically because the message content can change over time and depend on the location of system deployment.
- Our mechanism should only use message compression if there are computational resources to spare since other co-located services should not starve.
- Our mechanism must handle overload situations with grace and message compression can be resumed when the system has returned to normal operation.

Our implementation automatically selects the most efficient compression algorithm depending on the current message content, CPU-load and network congestion level. We evaluated our implementation by using production system communication data gathered at customer sites and replayed it in a lab (with explicit customer concent). Our experiment shows that the automatic compression mechanism produces a 9.6% reduction in RTT and that it is resilient to manually induced overload situations.

1.5 Improving Performance while Enforcing Quality of Service

We designed and implemented a Shared Resource Aware (SRA) process scheduler. SRA monitors both performance and hardware resource usage of individual processes in a system. We measure the performance in high-level metrics such as message turnaround time or the number of operations per second. The SRA algorithm measures the hardware resource usage by utilizing the performance monitoring unit (PMU) to quantify the number of accesses to hardware resources. SRA measures, interprets and acts on processes' hardware resource usage and their applicaiton performance to efficiently allocate (*where to run*) and schedule (*how and when*) different processes. The key properties of SRA are:

- SRA continuously monitors the hardware resource usage and continuously calculate the correlation towards the process performance. Having a good understanding of the correlation between hardware resource usage and performance is vital when reducing the effects of shared hardware resource.
- SRA uses the hardware resource-performance correlation to allocate processes over the set of available CPU cores thus improving the system performance by reducing shared hardware resource congestion.
- SRA uses performance counters to detect when a process overuse its stipulated hardware resource quota. SRA may decide to context switch the process when an overflow occurs to minimize the effects on other processes co-executing on the common hardware. Enforcing a strict shared resource quota makes it possible to provide a QoS simultaneously as improving the system-level performance.

We implemented the process allocation part of SRA as a core affinity selector in Linux. Our initial experiments indicate that it is possible to gain up to 30% performance increase compared to the standard Linux CFS process scheduler by allocating cache-bound processes in a way that is shared resource-aware. We designed the process scheduling part of SRA as a new Linux scheduling policy and implemented it as a new scheduling class in Linux.

1.6 Outline

The thesis continues in Chapter 2 (background) with further explanations of our target system. We describe standards and functionality supported by the telecommunication system we investigated. We also describe system setup, design, and structure. Chapter 3 (research method) define the research questions we addressed in this thesis. We also delimit our research and describe the methodology we used. We conclude the chapter by describing validity issues. We list our contributions in Chapter 4 and illustrate how the publications relate to each other and to the research areas. The four following chapters describe our contributions in detail. Each chapter follows a similar structure, starting with an introduction to each research area closely followed by the system model and definitions. We continue by describing our implementation, experiments and conclude each chapter with related/future work. The chapters are Chapter 5 (measuring execution characteristics), Chapter 6 (load replication), Chapter 7 (automatic message compression), and Chapter 8 (resource aware process allocation and scheduling). We present a summary of our contributions in Chapter 4 together with each publication. Chapter 9 concludes the thesis by describing our main findings and directions of our future work.

More and better collaboration between academia and the software industry is an important means of achieving the goals of more studies with high quality and relevance and better transfer of research results.

— *D. Sjøberg, T. Dybå , M. Jørgensen* [272]

2 Background

We believe that it is vital to understand the context of industrial settings within which we have worked on this thesis. This background chapter describes some of the most fundamental components and behaviors of our target system.

We start by listing telecommunication standards, Section 2.1, and how they relate to current and future telecommunication services, Section 2.2. The platform we have worked with supports various standards spanning from 2G (GSM) via 3G (UMTS, WCDMA) and 4G (LTE) and further towards the current 5G standard. The primary driver for new communication standards is the growing demand for higher communication bandwidth. We continue, in Section 2.3, by defining our view of large-scale industrial systems [122]. Such systems have common attributes such as strong system uptime requirements, many simultaneously deployed software and hardware generations and considerable size and complexity. We also describe various deployment scenarios, in Section 2.4, for our target system. We continue in Section 2.5 by describing implementation details, development process, and other detailed system-specific information. We conclude the chapter with Section 2.6 giving a detailed description of the OS:es used in our target system and our experiments.

Telecom. Standard	Max Down Link Speed	First Introd.	Main Features
1G (NMT, C-Nets, AMPS, TACS)	-	1980	Several different analog standards for mobile voice telephony.
2G (GSM)	14.4kbit/sec circuit switched, 22.8kbit/sec packet data [106]	1991	The first mobile phone network using digital radio. Introduced services such as SMS.
→ GPRS	30–100kbit/sec	2000	Increased bandwidth over GSM.
→ EDGE	236,8kbit/sec	2003	Increased bandwidth over GSM and GPRS.
3G (UMTS, WCDMA)	384kbit/sec	2001	Mobile music and other types of apps started to be used through more advanced smartphones. The phones changed awareness and increased the demand for higher communication bandwidth.
→ HSPA	14.4–672Mbit/sec [219]	2010	Increased bandwidth over 3G.
4G (LTE)	100Mbit/sec–1Gbit/	2009	Mobile video.
5G	1Gbit/sec to many users simultaneously	2018	Massive deployment of high bandwidth to mobile users, smart homes, high definition video transmission. Focus on low response time.

Table 2.1: The most important telecommunication standards and their communication bandwidth linked to the main features introduced by the standard.

2.1 Telecommunication Standards

Telecommunication systems are complex because they implement several communication standards. Standards define how systems should interact and is a

fundamental tool when connecting different manufacturer's systems. The standards continuously evolve to reflect customer demands, which drive equipment manufacturer to continually develop new features and system improvements. Several standards execute concurrently for efficiency reasons. See Table 2.1 for a list of telecommunication standards and their main features.

Groupe Spécial Mobile (GSM) [287] (2G) was introduced in 1991 and provided the second generation of mobile communication. It was the first commercial and widely available mobile communication system that supported digital communication [235]. Needless to say, the GSM system was an astonishing commercial success with 1 billion subscribers in 2002 [63] and 3.5 billion [118] in 2009. The introduction of GSM changed the way people communicate by allowing a significant portion of the population in industrialized countries to use mobile phones. Several extensions to the GSM standard, GPRS, and EDGE, further increased the communication bandwidth, thus allowing the implementation of even more complex services.

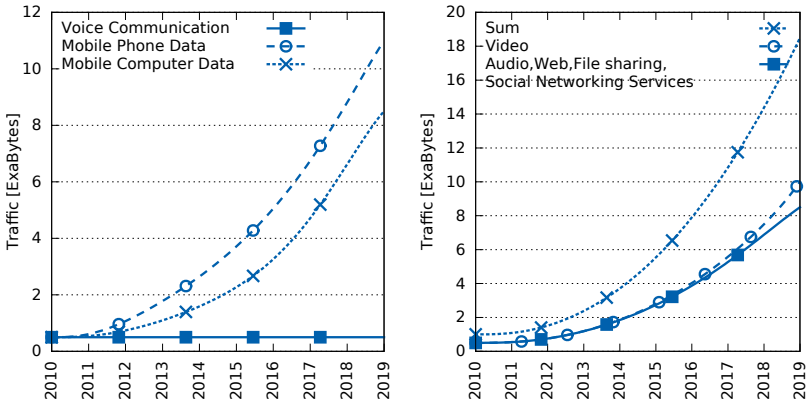
In 2001, the third generation (3G) standard was introduced as a response to customer demands for further increased bandwidth. The 3G standard is also known as Universal Mobile Telecommunication System (UMTS).

A fourth increment (4G) of the telecommunication standard, also called Long Term Evolution (LTE) [142], was introduced to the market in 2009. At this point, a large part of the industrialized world had adapted the "always-online" paradigm. The society, as a whole, looks favorably on mobile broadband and social networking services [146] demanding higher capacity in the telecommunication infrastructure.

Today, in 2018, we are standing on the brink of the next telecommunication standard to be implemented (5G). It is estimated to be released to the market in 2020 with substantial improvements compared to LTE [24]. The first improvement is a massive increase in bandwidth when there are many simultaneous users. A drastically reduced latency (below 1ms) is needed to support traffic safety and industrial infrastructure processes [87]. There is also an increasing demand for a reduction of energy consumption [42] so that it is environmentally friendly [88], while also making it possible to install network nodes in remote places [86] with scarce power supply.

2.2 Telecommunication Services

The introduction of mobile phones quicky made voice communication the most important service. It was the natural way to extend the already existing wire



(a) Voice and data traffic.

(b) Mobile application traffic.

Figure 2.1: The graph [146] shows world-wide market outlook for mobile traffic 2010 – 2019 [84].

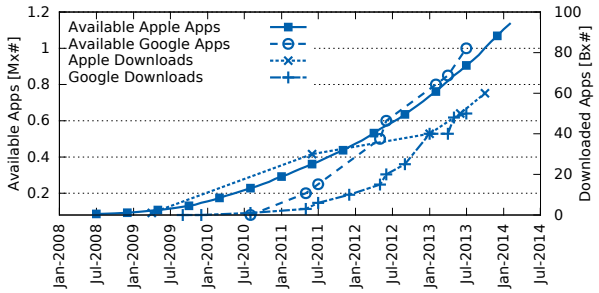


Figure 2.2: The graph [146] shows download-statistics for mobile phone application gathered from several online sources [12, 13, 45, 138, 285].

bound voice service into the mobile era. Voice services have now reached its peak from a capacity perspective [84], see Figure 2.1a. It is also apparent that data communication is rapidly increasing for both mobile phones and mobile computers. A report [85] by Ericsson Consumer Lab attributes the increased data usage to five main usage areas:

- *Streaming services* are quickly gaining acceptance among the population and include on-demand services such as music, pay-per-view TV and movies. Ericsson estimates that mobile video will be one of the most requested services in the coming years (2010–2019), see Figure 2.1b.
- *Home appliance monitoring* is increasing rapidly. For example water flood monitoring, heat and light control, refrigerator warning systems, coffee-machine refill sensors, entry and leave detection and much more.
- Data usage are expected to increase further at a rapid pace with the use of *Information Communication Technology (ICT)* devices such as mobile phones, watches, tablets and laptops. There is a common acceptance to use ICT devices for a large portion of daily activities [90] such as bank transactions, purchases, navigation, etc. The use of devices is expected to further increase the utilization of telecommunication networks [312]. The extraordinary increase in download rate of mobile apps indicates the acceptance of mobile usage among people, see Figure 2.2.
- *Vehicle communication* to support self-driving cars [87] and automated vehicle fleet management [88].
- Reduced network latency is needed to implement *Industrial infrastructure* [87] operations over wireless networks.

The overall increase in geographical and population coverage paired with new services, such as the ones described above, will contribute to an enormous growth in mobile data traffic. The geographical coverage was in 2014 mainly focused on Europe and USA with Asia, mainly India and China, quickly catching up and surpassing [88]. In 2015 there were approx. 7.4(3.4)[1] billion mobile subscribers world-wide and it is estimated that there will be 9.1(6.4) billion subscriptions by 2021 [88]. Increasing both geographical and population coverage causes an unprecedented change in global mobile data usage, which is currently one of the biggest challenges for network operators.

2.3 Industrial Systems

The system we have targeted and also performed our experiments upon is an execution platform handling several generations of telecommunication standards. The platform has been developed by Ericsson for several decades and is called Cello or Connectivity Packet Platform [4, 168] (CPP). The platform is generic and supports many existing communication standards [75], including 3G and LTE.

[1] The number of advanced smartphone subscriptions in parenthesis.

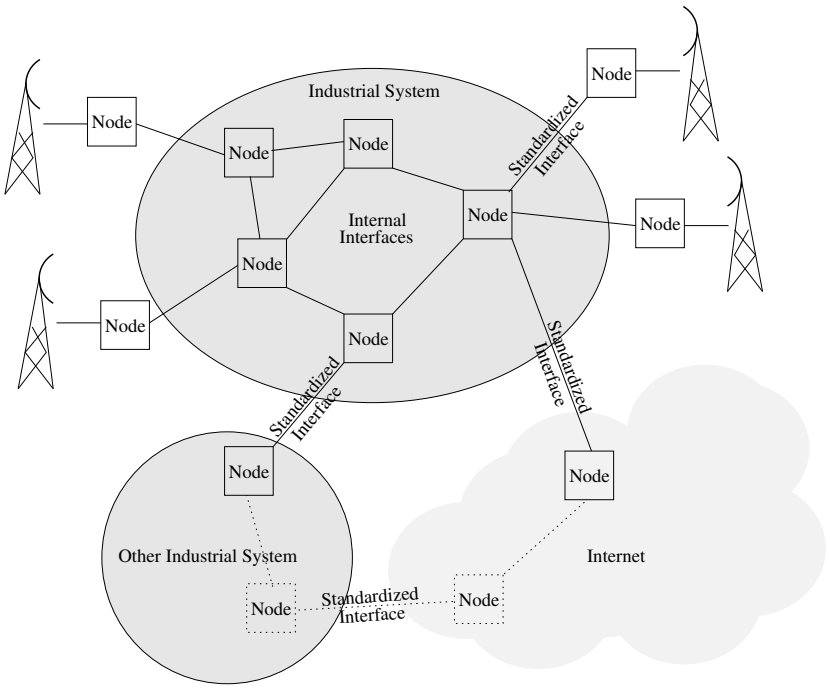


Figure 2.3: Industrial systems interacts with surrounding systems using standardized interfaces. We have concentrated on node-internal characteristics and performance improvements for internal interfaces.

The telecommunication system we have investigated in this thesis shares similar properties with other large-scale industrial systems. We believe that other systems also can use our research results since they share a similar system structure and behavior. We show a simplified overview of the telecommunication system investigated by us in Figure 2.3. The system distributes over many computers, denoted *nodes*. Internal nodes that implement a subset of the system functionality do not necessarily use standardized communication protocols. Performance improvements can be achieved using proprietary protocols over internal interfaces. Standardized communication is necessary for external communication such as the interoperability between equipment manufacturers. We have defined behavioral patterns that are common to industrial and telecommunication systems [122] . Some examples are:

- There is a low acceptance for system downtime.
- There are multiple concurrent hardware and software generations.
- The lifetime spans over several decades.
- The size and system complexity causes long lead-times when developing new functionality.
- Substantial internal communication between nodes inside the industrial system. External connections require the use of standardized protocols, for example 3GPP for telecommunication systems, Figure 2.3.

We have tried to generalize our research as far as possible. We believe that our research results should be applicable for many other systems sharing the same structure and behavior as the type of telecommunication system we have investigated. Some industrial systems are located in large server facilities, providing easy access for engineers and scientists. Other industrial systems are located in "friendly" places where a support engineer can access them and extract any information needed. Telecommunication systems are typically deployed in a different type of environment. Most network operators have their own infrastructure where the telecommunication nodes are located. Support and maintenance personnel is often employed by the operator. In the rare cases when the operator receives support help from the equipment manufacturer, they are not given full access to the nodes. Such restrictions makes it difficult to monitor hardware characteristics for production nodes. Operators are traditionally very restrictive towards running diagnostics, test programs or monitoring tools that are not verified as production level software.

Physical access restrictions also make it vital to have adequate error handling that gathers enough information when a fault occurs. It is not possible to retrieve additional troubleshooting information at a later time meaning that all necessary information must be generate automatically and packaged together with the trouble report. The scenario of restricted node access is one aspect we have addressed in this thesis work. System developers have always demanded hardware characteristics measurements for production nodes, but it has been hard to obtain such information.

Figure 2.4: Many circuit boards (to the left) are interconnected to form a cabinet (to the right). Courtesy of Ericsson 2016.

Figure 2.5: Several interconnected cabinets construct a large-scale telecommunication system. One node in Figure 2.3 can vary in size from a single circuit board up to several cabinets. Courtesy of Ericsson 2016.

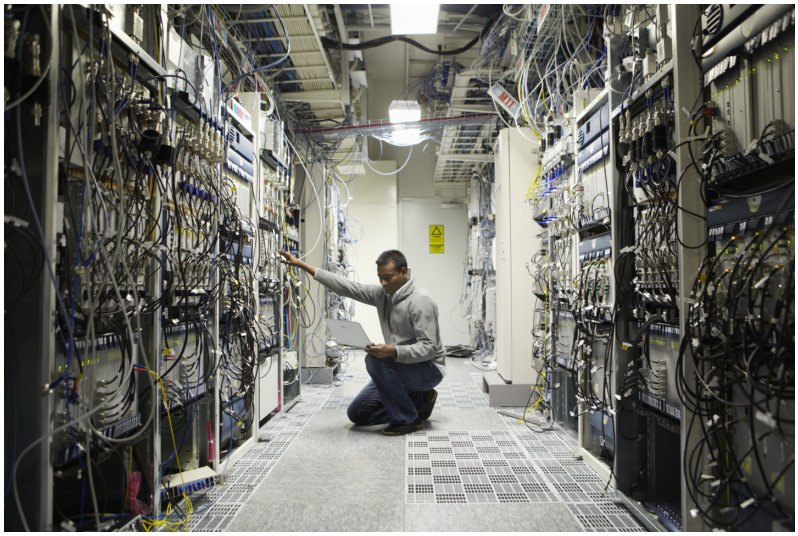


Figure 2.6: Complex lab test environment. Courtesy of Ericsson 2016.

2.4 Deploying Our Target System

A node is a system entity that can be implemented with different physical components. The physical layout of a telecommunication system is governed by strict rules. One cabinet, to the right in Figure 2.4, consists of three vertically mounted sub-racks. Each sub-rack holds up to 20 circuit boards, illustrated to the left in Figure 2.4. In total, a cabinet sums up to approximately $20 * 3 = 60$ circuit boards, depending on the desired configuration. Several cabinets can be connected to form a large-scale node, see Figure 2.5. Each circuit board can have several CPUs with multiples of 10's of cores each. In total the largest systems can consists of thousands of CPU's.

It is possible to deploy the system in several different levels, which is particularly useful for testing purposes. Running one board by itself provides the most basic level of system used for low-level testing. A slightly bigger system is achieved when at least two boards are interconnected to form a small cluster. This level of system is useful for verifying cluster functionality. Much more complex testing scenarios can be formed by configuring larger nodes, such as Figure 2.6. These type of nodes are seldom available for software design

purposes since they are very costly. Large-scale nodes are mainly used when testing complex traffic scenarios and for performance related verification.

A fully operational telecommunication system needs additional equipment such as various antennas, cabling, GPS, and operator interaction computers. The system also requires many mechanical parts to house nodes and towers to mount antennas. We do not consider those types of equipment and have only focused on the part of system related to message communication and traffic handling.

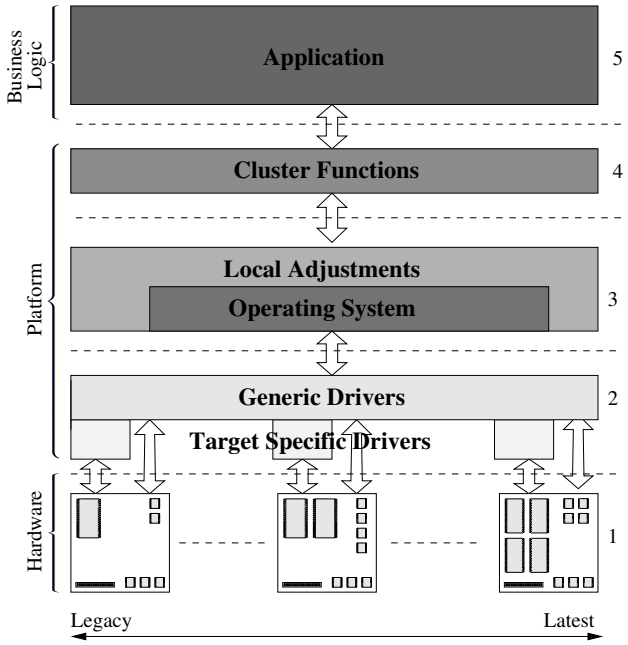


Figure 2.7: There are five abstraction levels (right) implementing the complete system spanning from hardware to business logic (left). There are multiple hardware implementations (bottom) spanning from legacy single-core processors (1–A) to advanced multi-core processors (1–C). The same platform (2-4) and application (5) supports all hardware implementations.

2.5 System Details

We followed the guidelines presented by Petersen [231] to contextualize our investigated system. We investigated a large telecommunication system [23, 293] where each node in the system overview, Figure 2.3, is described internally as in Figure 2.7. From a high-level perspective there are five abstraction levels (to the right in figure) that are structured in three functional parts (to the left in figure).

The hardware (level 1) is implemented with custom made circuit boards with varying performance capabilities depending on desired functionality and year of manufacture. The performance spans from older single-core boards up to several CPU's, each utilizing 10's of cores. Memory capacity is varying from a few MB's up to many GB's per CPU.

Hardware variations put great emphasis on designing drivers (level 2) that must be generic as well as provide support target specific functionality. The drivers must maintain a stable legacy interface towards the OS. Application programming interface stability is vital in large scale system development.

Third party vendors deliver the OS (level 3) and depending on the use-case it is either a specifically tailored proprietary real-time OS or Linux. The API-functionality supplied by the OS must be both backward and forward compatible regardless of changes to the OS and the hardware. Changing low-level functionality should not be propagated upwards to higher levels.

Cluster functionality is implemented (level 4) to support board interoperability, communication mechanisms, initial configuration, error management, error recovery and much more. The majority of the platform source code is implemented at this level. It is a complex part of the platform (levels 2–4) with complicated system functionality to maintain high-availability. Sharing the platform between multiple hardware platforms is vital for the maintainability of the complete system.

The application runs on the uppermost level of the system (level 5). It is by far the largest portion of all layers when comparing computational capacity, memory footprint and any functional metric. There are several applications that each implement a complete telecommunication standard, such as GSM [287], WCDMA [130, p 1–10] or LTE [142]. Several high-level modeling languages have been used to model these applications in combination to low-level native code. The model is, in some cases, used to generate low-level programming code that is natively compiled for a specific target. The resulting code is complex to debug, especially from a performance perspective. One issue is the sheer size of the application, which footprint is many Gigabytes. Furthermore,

it sometimes runs inside an interpreting/compiling virtual machine shadowing internal functionality. We have mainly worked with the platform parts in our studies (levels 2–4).

Maturity and Quality

The CPP telecommunication platform is a very mature product, and Ericsson deployed the first test system in 1998 [294] and released the first commercial system in 2001. The system is deployed worldwide and had a market share of 40% [247] in 2015. Nokia-Alcatel-Lucent (35%) and Huawei (20%) share most of the remaining market share. Being competitive is a key factor, and one of the most critical success factors for the resulting products is to keep development times as short as possible [110, 251, 284, 286, 293]. There are, in general, new hardware releases every 12-24 months to improve performance and consolidate functionality on fewer boards. Constant development activities using an agile [68] development process results in continuous customer releases of new software versions.

There are strict quality requirements on telecommunication systems, similar to other large infrastructure systems. In particular, there is little acceptance for downtime. Typically, a system is required to supply a 99.999% [176] uptime (five nines) meaning a maximum of roughly 5 minutes system downtime per year. Such high availability is difficult to reach because regular system updates may result in system restarts lowering the uptime. Intelligent traffic handling allows nodes to process traffic when a particular node is updated. There are many simultaneously running generations of software and hardware in an interconnected telecommunication system [122]. Multiple software and hardware revisions increase the complexity, especially when designing new functionality and debugging legacy problems.

It is also difficult to develop telecommunication systems because of the strict system level agreements (SLA) [305]. There are several levels of SLA, varying from customer demands of certain uptime, such as 99,999% [176] (five nines) uptime, to the quality of service (QoS) for the OS. This thesis address QoS for the process scheduler in Chapter 8.

Size and Type of System

To give an idea of the system size we present the number of source lines (SLOC) [216]. The OS is either a legacy third party real-time OS (many million

lines)[2] or Linux (15 million lines [189]). Running on top of the OS is a management layer providing cluster awareness and robustness. This layer consists of several million lines of code. The business logic is implemented using a model-based approach with large and complex models. It implements the complete communication standard for terminating traffic and handling call-setup. This part of the system has cost several thousands of man-year to develop, and the execution footprint is many GB.

The system is an extensive embedded distributed system [276]. Each execution unit (board) runs an OS that supports soft real-time applications. The boards are interconnected to form a large distributed system. Processes executing on one board can easily connect to processes executing on another. Interconnect poses many practical difficulties for standard OS:s, for example, the vast number of concurrently running processes. Furthermore, the system is designed to be both robust and scalable [113]. Customizing a telecommunication platform is a significant and challenging task. There is an operational and maintenance interface containing literary thousands of possible customization options. To further add to the overall complexity, it is also possible to make individual choices on how to connect each physical node in the network, see Figure 2.3.

Programming Languages

The system is built using many different programming paradigms. Drivers, abstraction level 2 in Figure 2.7, are implemented in either assembler or C. The OS, level 3, is also implemented in C and assembler where high performance is needed. The rationale for selecting C as the main programming language is historical but knowledge (at the time) and execution efficiency was the main reasons for the decision. The OS, level 3, is supplied by a third party company. For maintainability reasons, the surrounding code implements local OS adjustments. During our research, we have mainly implemented functionality in level 3.

Moving the abstraction further from the hardware changes the programming paradigm to support higher level programming languages. For cluster functionality, level 4, several programming languages are used, such as C and C++ for legacy code. Depending on requirements, recent functional additions may be implemented in either Java or Erlang.

Various model-based approaches have been used when implementing the application layer, level 5. There are several applications implementing differ-

[2] Business aspects prohibits us from disclosing the exact number of lines of code.

ent parts of the telecommunication standards described in Section 2.2. The applications share the common execution environment provided by lower levels (1–4).

Hardware

Message processing system usually consists of two parts [262, p1], the control system and the data plane. The control system implements functionality for configuring and maintaining the system throughout its life span. The data plane is mainly concerned with payload handling, i.e. routing messages towards their destination. In our system, the control system hardware is different from the data plane hardware. The former is partially implemented with common off-the-shelf hardware while the latter uses tailored CPU's with specialized hardware support for packet handling. We have investigated the control system, which has a communication rate in the range of Gbit/sec. The traffic terminates at the destination node where the CPU performs some message processing. We have not investigated the data plane.

The CPP system runs on more than 20 [23] different hardware platforms depending on the required performance. Low-power boards may be using ARM CPUs while high-end circuit boards aimed towards heavier calculations may use powerful PowerPC® or x86 CPUs. Using multiple hardware architectures is a challenging task. Platform code from level 4 and upwards, see Figure 2.7, must be hardware agnostic to be easily portable and efficiently maintained. The same applies to the application software, level 5, executing on top of the platform.

Development Process

Developing a large infrastructure system [122] puts great effort into development tools and the applied development process. Individual tracking of each code change is a requirement. Customers require continuous improvements with little or no consideration to the age or version of the software and hardware. It is hard to support systems with mixed hardware generations, and each software release must support several simultaneously running hardware generations. As an indication of the system size, thousands of skilled engineers [122] have spent decades implementing the system. The design organization is distributed over many geographic locations, requiring intense coordination.

When we started our research, the development process consisted of many sequential steps of different complexity and size. We have since then started to use agile development processes.

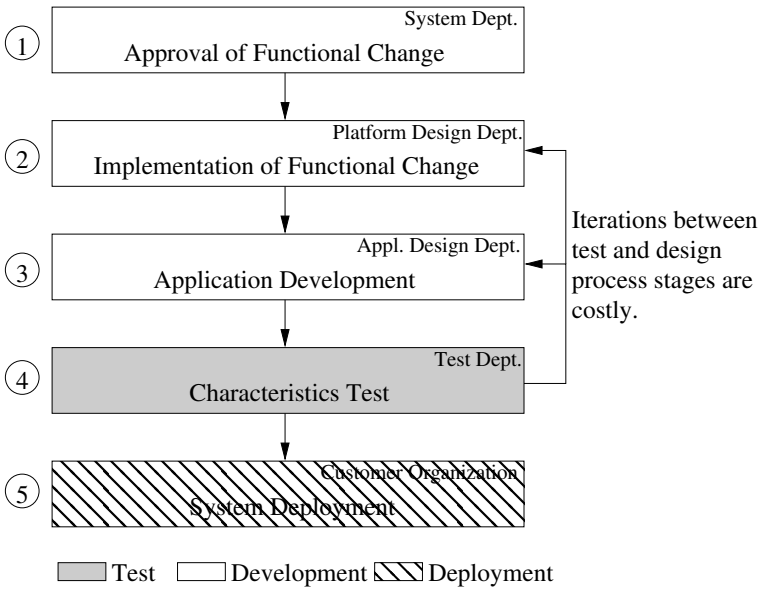


Figure 2.8: System development waterfall model.

Requirement phase The requirement phase is the first step in the development process. This is the place where the system department specify function requirements and decide when a system function should be implemented, ① in Figure 2.8. Requirements for the system department may originate from customers, market trends or internally.

Design phase The second step in the development process is the design phase. The design phase consists of a chain of activities that each depend on the successful completion of earlier activities, similar to the waterfall model [252]. We use agile methods [68] within each development substage allowing parallel development of system functions. The first activity in the design phase is platform development, denoted ②. The primary requirement on the platform is to provide an adequate execution environment for subsequent application development. Such an execution environment contains an OS and drivers together with low-level APIs and a cluster-aware middleware. Finally, multiple application development departments build the applications, ③, that implements the business logic, i.e., the real customer-demanded functionality.

System test phase The third major development process stage contains system-testing, ④, and product-release, ⑤. Although software unit testing is performed throughout the development phase, no full-scale performance test can be done before all parts of the system are completed. Testing departments measure the application execution characteristics (hardware resource usage) and performance when both the platform and the application have been finalized. Usability analysis and application performance is measured at the end of the development cycle [101] because it usually requires both a fully working system and a suitable test environment. The system can be released to customers when it meets both functional and performance requirements.

2.6 Operating Systems

Our target system has used several different OS:es over time as various system implementations had different requirements. It was common to use tailored OS:es during the 90's. The burden of maintaining inhouse developed OS:es prompted a more economically viable solution. Enea OSE, Section 2.6.1 was introduced at first as a consultancy project and later as a full product. Using a third-party OS resolved many of the problematic issues that troubled the design organization. For example keeping up with the latest software technologies, migrating the OS to new targets and many other obstacle. A similar reasoning later prompted the switch from OSE to Linux. It is difficult to pinpoint the exact year but early tests were made around 2010. We briefly describe Linux in Section 2.6.2.

2.6.1 Enea OSE

The Enea OSE is a general-purpose real-time [39, p430] OS [273]. OSE was originally developed for use in many generations of Ericsson telecommunication systems. The idea for OSE sprung from the need of a general-purpose real-time OS that was both simple to handle and had high performance. We describe the most important OSE services [111] in the following paragraphs:

Process scheduling OSE implements various type of processes. The most common process type is the prioritized process, which is handled by a fixed priority preemptive scheduler. There are 32 priority levels [81]. A running prioritized process can only be interrupted by another prioritized process if the latter one has higher priority or if the running process yields. It is also possible to use interrupt-, timer- and background processes. An interrupt process is

typically triggered by an external event, such as an arriving Ethernet packet. A timer process has a recurring execution pattern and runs at specific intervals. Background processes have the lowest priorities and will only execute when no other process demands the CPU [80].

Memory management Each application, denoted load-module in OSE, has its own memory domain that may be shared with other applications by forming process blocks [41, p4]. Applications can, in general, not access common memory unless explicitly configured. Such memory protection improves the system stability because stray memory accesses can be avoided. OSE also tries to locate corrupted memory buffers by implementing various buffer endmark checks when making system calls [81, p39].

Centralized error handling Error handling [81, p39] in OSE is generally system-wide. It is generally not necessary to handle possible error return codes when calling system functions. The kernel will detect that an error has occurred and call either an error handler connected to the process or a system-wide error handler. The main benefit of this mechanism is that it makes it possible to centralize error handling and not scatter it all over the system code.

Message passing Processes in OSE communicate through a signalling interface. The signalling interface sets up an inbox where received messages are tagged. When the recipient polls the inbox the message is copied from the sender to the receiver. The message interface is very efficient since it minimize the number of process context switches, thus allowing extensive message passing while maintaining a high performance.

There are many other convenience services supported by OSE [82, p39]. For example heap managing, program loading, persistant storage, command-line interface and many more.

Our target system has evaluated and used various type of system setups [115] ranging from standalone OSE systems to hybrid approaches. For legacy systems, the most common setup is although the standalone and pure OSE-based system. Around 2010 the market trend showed a relentless drive of moving to open source software such as Linux. Some reports [91] showed that the performance impact of such a move would not be too great. Much work was spent in trying to bridge the gap between OSE and Linux [214] although the move to the Linux OS was unavoidable. The desire to move large parts of the target system to Linux has also triggered numerous internal and external [255] investigations. Most investigations state that it is feasible to move from OSE to Linux but it

requires further investigations. We describe Linux in the next section of this thesis.

2.6.2 Linux

Linux is a vast OS. It has more than 15 million [189] lines of code (SLOC) [216] and supports many different architectures and a wide array of drivers. The generability comes with a performance cost making it difficult for Linux to compete with tailor-made real-time OSes. Linux has, on the other hand, some advantages such as its free availability of source code and a huge installed base. Among others, these two factors have resulted in a vast Linux development community. A company can, to some extent, expect that drivers will be automatically developed for new hardware without themselves doing the job. The Linux community has, of course, also addressed the performance penalty of generability. There has been much work related to performance optimizations, such as the early adoption of Symmetric Multiprocessing Processors (SMP). There is also an extensive array of tools suitable for performance analysis, such as Perf [188], Oprofile [184], Valgrind [215], PAPI [109] and many others.

Linux was at the beginning mostly suitable for desktop computers, and subsequently, it made its way into the server market. Quite recently [249] Linux has started to support real-time behavior making it suitable for use in embedded industrial products, such as telecommunication systems. The lack of licensing fee for Linux has been a dominant driving force to migrate legacy functionality from tailored OS:es to Linux.

History Linux was first released in 1991 [270] named version 0.01. It was a basic OS with no networking support, and it ran only on Intel® 386 hardware. The first official version of Linux was delivered 1994 and supported only i386-based computers. After many structural changes and major redesigns, the Linux kernel of today does not much resemble the initial version, see Table 2.2. The Linux community added support for additional architectures in the 1.2 Linux kernel in [270] and the process scheduler was still simple and designed to be fast when adding and removing processes [163]. The most important change in the 2.0 release included rudimentary Symmetric Multiprocessor Support (SMP). The 2.2 kernel release added support for scheduling classes making it possible to use several scheduling policies for different processes, such as real-time and fair scheduling. The release also improved on the previous SMP support [163].

The 2.4 kernel added $O(N)$ the process scheduler, which divided the execution time into epochs. All processes belong to the ready-queue when an epoch

Version	Year	Major Design Changes
1.0	1994	The initial (official) Linux delivery only targeting i386-based computers.
1.2	1995	Added a modular architecture and support for several additional architectures such as Alpha, SPARC and MIPS. The process scheduler was implemented with a circular run-queue utilizing a round-robin scheduling policy [27]. The main design goal was to be simple and fast when adding and removing processes [163, 270].
2.0	1996	Added multiple architectures and SMP support.
2.2	1999	Added support for scheduling classes, which made it possible to use several scheduling policies for different processes. This release also added more advanced support for SMP system [163].
2.4	2001	The $O(N)$ scheduler was added. It divided the execution time into epochs and iterated over all processes in the system selecting basing the selection mechanism on a metric function. Parts of the execution quanta left-over from one epoch could be passed on to the next epoch [163, 194].
2.6	2003	The $O(1)$ scheduler is released to reduce scalability issues related to the earlier scheduler [1, 163, 173, 174].
2.6.23	2007	The CFS scheduler [96] was created to improve the responsiveness of desktop applications [172].
3.14	2014	The EDF scheduler [97, 181] is merged into Linux main track.

Table 2.2: A brief history of Linux kernel key releases.

starts. As processes are assigned to CPUs thus exhausting their execution quota, the scheduler moves them to the wait-queue. The epoch ends when all processes are in the wait-queue, which starts a new epoch by swapping the wait-queue and ready-queue [172]. The scheduler selects the next task by iterating over all processes in the system and using a metric selection function. A new epoch can inherit the execution quanta left-over from a previous epoch [163, 194].

The community quickly discovered that the $O(N)$ scheduler has scalability problems with large systems having a massive number of concurrent processes. The 2.6 release introduced the $O(1)$ scheduler, which provided a constant time for each process scheduling selection [1, 163, 173, 174]. The $O(1)$

scheduler solved many of the scalability problems but showed latency problems for user-interactive applications. There were heated debates in kernel discussion groups how to solve the issue of responsiveness. One suggestion was to introduce the Staircase scheduler [173] and later the Rotating Staircase DeadLine (RSDL) [174] scheduler into the OS. Both schedulers aimed to reduce latency and provide better desktop support for Linux. These two schedulers acted as a starting point for the later development of the Completely Fair Scheduler (CFS) [172] which replaced the $O(1)$ scheduler in the Linux kernel 2.6.23 [96]. Improving the responsiveness [172] for desktop applications triggered the development of the CFS scheduler [96].

I believe that many events in my work and life have been a matter of luck or accident. But I am also aware of several occasions on which I explicitly made choices to step off the obvious path, and do something that others thought odd or worse. . . I have come to think of these events as 'detours' from the obvious career paths stretching before me. Frequently these detours have become the main road for me. There are obvious costs to such detours. Other choices might have made me richer, more influential, more famous, more productive, and so on. But I like what I am doing, even though the path has involved a lot of wandering through uncharted territory.

— L. David Brown[3]

[3] Quoted from the book by M. Brydon-Miller, D. Greenwood and P. Maguire [37]

3 Research Method

We We have had a large-scale telecommunication system [23, 293] at our disposal during the work on this thesis. This is a unique opportunity for real-world research on a large scale system. Instead of using a theoretical example that is well suited, we had to let the particular system influence us when we formulated the hypothesis, see Section 3.1, and the research questions in Section 3.2. We have expressed our research questions generically to ensure that they can address issues that are problematic for many other large-scale communication systems. We have also clarified some essential requirements related to each research question.

The research delimitations are closely related to the research questions because they limit the scope of our research. We have listed the most significant delimitations in Section 3.3.

We have performed several case studies during the monitoring and modeling phases to explore and describe our environment. We adopted a hands-on approach in the performance improvement phases to solve the particular problems found by using the techniques devised by earlier phases. We list in detail the the research methods used in each study in in Section 3.4. The chapter is concluded in Section 3.5 by listing the threats to validity.

3.1 The Hypothesis

We have investigated several ways to improve the performance of large telecommunication systems. We have formulated the hypothesis as follows:

> **By monitoring relevant hardware and software characteristics of a large industrial telecommunication system, we can find performance improvement areas.**

3.2 Research Questions

The goal of our research is a systematic collection of characteristics data that can be used to model the hardware usage of the system and to find performance improvement areas. The research questions below mirrors the requirements obtained from the industrial environment where we have worked.

3.2.1 System Monitoring

The telecommunication system we have focused on is well understood and thoroughly tested from a functional perspective. The system has not reached the same level of maturity concerning characteristics testing. New functionality is well defined and implemented according to detailed specifications by engineers with long experience in system development. However, the system complexity and the difficulty to monitor the system behavior and the hardware usage makes it difficult to understand what impact software changes will have on the system behavior. This reasoning leads to our first research question:

> **Q1 How to monitor the hardware usage and software performance of a production system without noticeable side-effects on the monitored system?**

We refine the research question, Q1 with additional constraints making it compliant with general industrial requirements:

- The probe-effect [108] of the resource monitor must be negligible to admitting the tool in a customer production environment.

- The resource monitor must support sustained monitoring times, several days or weeks, as well as high-frequency sampling.
- The resource monitor must be easily adaptable to different systems, architectures, and scenarios.

3.2.2 System Modeling

We continued to work with characteristics monitoring, see Section 3.2.1, and quickly understood that our monitoring mechanisms could be useful for other purposes than only characteristics monitoring. The design organisation where we performed our tests had for a long time struggled with the problem of having long lead times between platform development and characteristics testing. According to system architects the long lead-time results in difficult and time-consuming bug fixes. Early error detection is very difficult [6], but when successful it allows software errors to be corrected sooner than previously possible [282] leading to a reduction in development cost [29, 30]. This reasoning resulted in our second research question:

Q2 How to automate modeling and replication of hardware usage for a production system?

We refine the research question, Q2 with additional constraints making it compliant with industrial requirements:

- The mechanism should be fully automatic because we want to include it in an automated test framework.
- The mechanism should be generic for most types of industrial systems to make it applicable over the complete product suite with varying hardware and software implementations.

3.2.3 Improving System Performance

Our first two research questions targeted characteristics monitoring of a production system and performance bottlenecks detection. The natural next step was to target performance improvements for the system. How to use the extracted execution characteristics information to identify areas where the performance of our target system can be improved? This reasoning resulted in our third research question.

Q3 How can the communication performance of a telecommunication system be improved through message compression while retaining the system load within pre-defined limits?

We refine the research question, Q3 with additional constraints so that it complies with general requirements for our industrial system:

- The communication performance improvements must be fully automatic since network operators do not allow access to the system after deployment.
- The network congestion level and CPU utilization are different for various deployment scenarios and also changes over time due to alternating usage patterns. Any communication improvement method must automatically adapt to a changing environment, and it is therefore not possible to optimize it for a specific scenario.
- The system must be able to handle and improve the communication performance for multiple concurrent communication streams.
- Other co-located services, such as databases, JAVA machines, SFTP, SSH- and Telnet servers, should not be negatively affected by the communication improvements.
- Robustness and automaticity have higher priority than performance.

Improving the performance of our investigated system is the overall goal of this thesis. The target is to design an automatic mechanism that is robust and works well in an industrial environment.

3.2.4 Process Allocation and Scheduling to Efficiently Enforce Quality of Service

We have continued to study the performance impact of shared hardware resource congestion. We have investigated process allocation and scheduling. We realized that it is difficult to estimate the hardware usage of a computer system. It is even more challenging to map the hardware-usage to the user-experienced performance automatically.

There are many efforts to enforce QoS by limiting the CPU time-quota available for processes in a system [44]. Time-quota approaches do not necessarily tackle the performance impact of shared hardware resource congestion. We can

exemplify this by a scenario where processes p_0 and p_1 execute on adjacent cores having a shared cache. Accesses from p_0 will affect the cache availability for p_1, even if the OS process scheduler assigns sufficient CPU quota to each process. In this context, we define *allocation* as the way to distribute processes over a set of cores, and *scheduling* as the way to control the execution of processes on the same core.

Q4 How can an operating system process scheduler provide high performance and enforce shared resource quality of service by allocating and scheduling processes on a multicore CPU?

As with the previous research questions, we refine the research question, Q4 with additional constraints so that it complies with general requirements for our industrial system:

- Processes should be allocated so that they achieve high performance.
- It must be possible to ensure that a pre-defined shared resource quota is available for QoS sensitive processes.
- It must be easy to deploy the scheduler in an industrial environment.
- The scheduler must simultaneously support legacy scheduling policies such as real-time/deadline, time-sharing, etc.

3.3 Delimitations

We have chosen to limit the scope of our investigation to one particular system, which is the telecommunication system to which we have had privileged access. It is hard to gain access to other industrial systems since we need to modify parts of the system to perform our research. We have performed our experiments on one type of system, but our opinion is that our methods apply to many other large-scale industrial systems. We believe that the general methods are applicable for many other systems, although the specific experimental results are unique for our target system. Some delimitations specific to our research are:

- We have not yet explicitly verified that characteristics testing in early design phases reduce the total system development time, but earlier research [29, 30, 282] strongly implies that.

- The telecommunication system we have investigated is IO-bound, and we have therefore mostly focused on modeling the low-level cache usage.
- For the system modeling parts of this thesis, we have opted to use a low sample frequency (1Hz) that may be insufficient in some cases. We think that it is sufficient for our static model synthesis procedure. The characteristics of our target system are relatively static where the resource usage slowly changes depending on end-user behavior. The reason for this was that operator requirements forced us to guarantee that the production environment would not experience any probe effect. See Section 3.2.1.

Most limitations stem from the fact that it is challenging to get customer consent to access production nodes. Customers are very concerned that any system change may affect stability, security or performance, and it is therefore usually difficult to run any monitoring tool at a customer site. One of the goals in future system development is to migrate from the current development process to a DevOps development process, which depends on making runtime monitoring data available smoothly between Development and Operations and vice versa of a system (DevOps) [245].

3.4 Research Methodology

We have used two qualitative methods [263] to obtain the research results presented in this thesis. The first is case studies [182, 253, 254] to explore and describe the investigated object. The second method is action research [185] when iteratively implementing improvements in an industrial environment.

We base the papers A, B and the technical reports M and N on case studies of the telecommunication system at our disposal. We opted to use the case study method to get a better understanding of the system characteristics of production system. We also wanted to describe the system behavior.

We were active participants of the design organization [230] during the research for paper C, extending paper H. Changing the position from an observatory view (as in the case study) to a participatory role allowed us to switch method towards action research and a more improvement-centric view. We continued to use the same method during our work on paper D, E and patents O, P. Table 3.1 relates each research question to publication and research method.

RQ	Sect.	Publication	Question Type	Research Method
Q1	3.2.1	A (M, N)	Exploratory/ Descriptive	Case study
Q2	3.2.2	B (M, N)	Exploratory/ Descriptive	Case study
Q3	3.2.3	C (H)	Problem Solving/ Improvement	Action research
Q4	3.2.4	D (O) and E (P)	Problem Solving/ Improvement	Action research

Table 3.1: Mapping the research questions to methods [244, 254].

3.5 Threats to Validity

We have performed all our research within the scope of an industrial environment. One of the apparent benefits of investigating an industrial system is that it provides excellent insight into a production system with real-life customers and user scenarios. For example, we have gathered the data we used in Papers A, B and C at customer sites running production systems with real traffic.

We have also based the work for papers D and E on the same production system but we have used the test framework for generating realistic use cases. The requirements for all papers stem from the production environment.

However, performing research in the scope of an industrial system introduces some difficulties commonly not seen in pure academic environments. It is hard to obtain the scientific rigor needed for academic publications, and it is challenging to publish raw data or implementation details due to corporate secrecy. It is also challenging to get unrestricted access to a production system for research-oriented testing purposes. We have often been allowed a very limited time-frame for running our implementations on production nodes and with far-reaching limitations on capacity usage. Such limitations contrast to the academic interest of having a well-isolated system where it is possible to determine all execution conditions completely.

We have followed the guidelines by Runeson [254] and Wohlin [302] to categorize and describe how we have performed our experiments. We divide the validity discussion into subcategories described in the following sections.

3.5.1 Construct Validity

The construct validity [302, p108] describes the relationship between theory and observation, for example, if our test design has captured the topics we wanted to investigate.

Our test design for Papers A and B was to 1) Extract characteristics data from a production system running at a customer site and 2) Synthesize a model using a production test system. 3) Test the model using a customer bug fix. We duplicated the real development process in our test design, which indicates that our early-stage performance benchmarking approach works in a real-world application. We also assumed, according to earlier research by Boehm [29, 30], and Tassey [282], that it is economically beneficial to catch bugs in the initial phases of the development process. They state that the cost of fixing a bug increases with the distance between where a bug was introducing to where it is corrected. We accept this claim because it is widely accepted in both industry and academia, and we have not verified it by ourselves. We wanted to evaluate if it was possible to synthesize the hardware usage of a production system and the mimic the load on a test system. Our experiments show that we have addressed the construction bias.

For the tests in Paper C we modified an existing test system to replay previously sampled communication data from a production system. We also added synthetic data to force the test system into corner-cases where our method automatically selects other compression algorithms than the one used for production system messages. We also introduced synthetic load generators to mimic overload scenarios. We wanted to investigate if we could construct an automatic method that selectively compresses messages using the best of several compression algorithms. Our tests show that our implementation fulfills the desired functionality.

We set up the test environment for Papers D and E after detailed discussions with engineers handling our target system. During our discussions, we concluded that we could run the standard test system for our initial allocation and scheduling tests. The test system captures the principal behavior of the production system by mimicking packet processing and high cache and memory utilization. We desired to construct a system that automatically collects the hardware resource usage of processes and then uses the derived information for maximizing their performance by allocating them over a CPU core cluster to maximize the performance. Additionally, the system should support the possibility to restrict the usage of specified hardware resources for individual

processes, thereby not affecting the performance of other processes. We have therefore shown our studies has thoroughly addressed the construct bias.

3.5.2 Internal Validity

The internal validity [162] reflects the quality of the data analysis. In other words, that have we described and investigated the cause and the correct causal effect between the things we investigate.

Before starting our research, several senior system architects stated that the system we are investigating is IO-bound and memory-bound, which in effect are the system bottlenecks. We empirically verified their statement by using our characteristics monitor to investigate the system characteristics. We also verified the claim that our system is IO-bound later in our work when we had developed the resource-performance correlation method. We have run our tests on one telecommunication system that is similar to other large-scale systems, see Section 2.3.

For Papers A and B we synthesized a model for L_1I-cache, L_1D-cache, and L_2D-cache miss ratio. The model was then used to clone the production system hardware-usage on a test node. We believe that the model is sufficiently accurate by verifying that the performance impact of a real bug fix is similar in the model environment and the production environment. Our tests with the production system shows that we discovered a real performance related bug when using the load synthesize mechanism.

For Paper C we have sampled production system message data for use with the automatic compression mechanism.

We have implemented and tested the techniques presented by Papers D and E in the scope of the telecommunication system we are investigating. We have implemented the hardware monitoring part by using the Perf API in Linux kernel-space, which is easily reusable in other settings and environments. The system-level monitoring is application specific and needs to be implemented by the application developer. The analysis part is implemented in a high-level language and is completely portable. Our correlation engine automatically selects the highest correlated resource with the performance and it is system agnostic since it can use any resource usage metric and application performance metric. Our tests show that the resource-performance correlation model correctly captures the performance bottleneck for our type of system.

3.5.3 Conclusion Validity

The conclusion validity describes the relationship between the treatment and the outcome [302, p104]. Have we drawn the correct conclusions from the available data?

We have identified some threats to the conclusion validity for Papers A and B. We have tested our monitoring and modeling method on one production system. The test set is too small for far-reaching generalizations, which forces us to limit our conclusions to the particular type of system we have investigated. Our target system has the highest market share (40% [247]) among telecommunication systems, which strengthen our belief that the research is representable for this particular system type. We can reason that many other systems are similar to our from the resource and performance perspective. The similarity implies that our techniques should be widely applicable with only minor corrections to certain hardware dependent parts. However, we cannot be certain since we have not tested and evaluated our method on other industrial systems. It is challenging to get operator consent to verify our modeling mechanism on other manufacturers equipment. However, we believe that the generic mechanism is highly usable for other types of systems with minor modifications. Adapting our modeling method requires the cache generator functions to be adapted to different cache structures.

We have implemented our automatic message compression mechanism, described in Paper C, on the same system used for Papers A and B. We extracted our test data from a running production system. We believe our automatic compression mechanism is sufficiently generic and can be utilized by many communication system. Migrating the mechanism to another system require minor modifications such as modifying the set of compression algorithms suitable for the new system.

We have validated our techniques presented in Papers D and E on a test system that replicates applications running on our target system. We have to the best of our knowledge taken threats of validity into considerations and systematically attempted to address them. There is always a risk that the proposed solutions are not as general as we attempt.

3.5.4 Method Applicability

We argue that the findings in Papers A and B gives us excellent insight into the behavior, resource usage, and performance of our investigated telecommunication system. Understanding the relationship between resource usage and

performance is vital when developing performance critical systems. Our tools can help the system designer to understand where to look for system-wide bottlenecks.

The implementation described in Paper C has improved the messaging performance by applying selective compression when there are CPU-cycles to spare. The most significant benefit is that the compression mechanism automatically finds the algorithm providing the lowest message round-trip time. We do not need to make any manual analysis which it makes a suitable mechanism for a large-scale system with changing execution patterns.

We have had many discussions at both academic conferences and with industrial partners. The conclusion has often been that current process schedulers lack hardware resource awareness, as the one presented in Papers D and E. We argue that a process scheduler may not need to be resource-aware in all scenarios. Many end-user desktops will be better off with low latency process scheduling to support user responsiveness. Other scenarios favor high throughput, such as server farms handling large volumes of work requests. There is still a need for resource-aware schedulers when running systems that should provide a cost-effective (efficient) QoS environment. Our telecommunication system is such a system, and many other systems share this use-case. All of those systems can benefit from our resource aware process scheduler.

We have implemented our ideas in a telecommunication system but also published the results in the academic community. Our contributions are now part of our company's product portfolio, which further indicates that our research is needed and valuable. We believe that our target system is representative of other large-scale systems and especially systems with extensive communication. The test of time will show if our results impacts other systems.

1. *Du skall inte tro att du är något.*
2. *Du skall inte tro att du är lika god som vi.*
3. *Du skall inte tro att du är klokare än vi.*
4. *Du skall inte inbilla dig att du är bättre än vi.*
5. *Du skall inte tro att du vet mer än vi.*
6. *Du skall inte tro att du är förmer än vi.*
7. *Du skall inte tro att du duger till något.*
8. *Du skall inte skratta åt oss.*
9. *Du skall inte tro att någon bryr sig om dig.*
10. *Du skall inte tro att du kan lära oss något.*

— Aksel Sandemose[1] [259]

[1] Sandemose formulated these statements in his book “En flykting korsar sitt spår”. He expresses the opinion that no person should believe that they can achieve something, become something better, or even should strive to achieve something in life. This is often referred to as the ”Jante law”.

4 Contributions

THE contributions presented in this thesis spans over several technical areas where. Each contribution has been published within various academic communities. We start this chapter in Section 4.1 by mapping each publication to a research area and display the publication order. We have mapped the research questions (Q) to sections of publications in the following ways:

- A low-intrusive characteristics long-term monitoring application for industrial use (Q1), Section 4.2.
- An automatic load replication mechanism (Q2), Section 4.3.
- An automatic message compression mechanism that reduce the message round-trip time by content-aware compression (Q3), Section 4.4.
- A resource aware process allocation mechanism that allocates processes to increase performance (Q4), Section 4.5.
- A resource aware process scheduling mechanism that enforce QoS between processes sharing a common hardware (Q4), Section 4.6.

The two last contributions each address a subset of a common research question. Each of the sections above contains the publication abstract and a brief outline of our contributions.

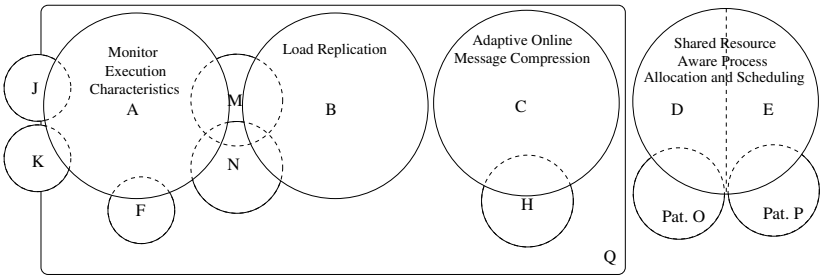


Figure 4.1: Our four main research areas.

4.1 Publication Mapping, Hierarchy and Timeline

This thesis consists of four major research areas as depicted in Figure 4.1. The first area relates to execution characteristics measurement and is mainly based on

Paper A [150]. Our research in the area is also connected to our other Papers F [136], J [59] and K [60]. The second area, load replication, is mainly based on Paper B [151]. The technical report N [154] further expands and adds contributions to both Paper A and B. Adaptive online message compression is the third technical area. This area is based on the journal Paper C [153], which is an extension of Paper H [152]. The fourth and final research area is published in Papers D [157] and E [147]. Paper D describes resource efficient process allocation by extending our work in Patent O [155]. Paper E presents a QoS aware process scheduler extending Patent P [156]. The Licentiate thesis Q [144] was published as a collection-of-papers and contains the first three research areas.

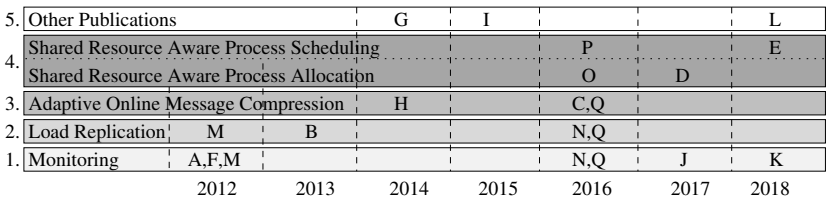


Figure 4.2: Publication order.

We depict the paper publication order in Figure 4.2. Some papers spans multiple lines since they cover several research areas.

4.2 Paper A

We have addressed research questions Q1 (Section 3.2.1) in:

Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl and Björn Lisper. *Towards Feedback-Based Generation of Hardware Characteristics.* In Proceedings of the International Workshop on Feedback Computing, 2012. [150]

The paper is further expanded in the technical report N [154].

Paper abstract

In large complex server-like computer systems it is difficult to characterise hardware usage in early stages of system development. Many times the applications running on the platform are not ready at the time of platform deployment leading to postponed metrics measurement. In our study we seek answers to the questions: (1) Can we use a feedback-based control system to create a characteristics model of a real production system? (2) Can such a model be sufficiently accurate to detect characteristics changes instead of executing the production application? The model we have created runs a signalling application, similar to the production application, together with a PID- regulator generating L_1 and L_2 cache misses to the same ex- tent as the production system. Our measurements indicate that we have managed to mimic a similar environment regarding cache characteristics. Additionally we have applied the model on a software update for a production system and detected characteristics changes using the model. This has later been verified on the complete production system, which in this study is a large scale telecommunication system with a substantial market share.

Contribution

Thesis writer (me) is the main and first author of Paper A and the extended technical report N [154]. We implemented a method for long-term monitoring of large-scale systems. The first implementation supported Enea OSE running PowerPC®, and it was later ported to Linux and other architectures such as Intel® x86. We also evaluated the method on a deployed telecommunication systems to get realistic hardware usage information.

4.3 Paper B

We have addressed research questions Q2 (Section 3.2.2) in:

> Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl and Björn Lisper. *Automatic Multi-Core Cache Characteristics Modelling.* In Proceedings of the Swedish Workshop on Multicore Computing, Halmstad, 2013. [151]
>
> The paper is further expanded in the technical report N [154].

Paper abstract

When updating low-level software for large computer systems it is difficult to verify whether performance requirements are met or not. Common practice is to measure the performance only when the new software is fully developed and has reached system verification. Since this gives long lead-times it becomes costly to remedy performance problems. Our contribution is that we have deployed a new method to synthesise production workload. We have, using this method, created a multi-core cache characteristics model. We have validated our method by deploying it in a production system as a case study. The result shows that the method is sufficiently accurate to detect changes and mimic cache characteristics and performance, and thus giving early characteristics feedback to engineers.We have also applied the model to a real software update detecting changes in performance characteristics similar to the real system.

Contributions

Thesis writer is the main and first author of Paper B and the extended technical report N [154]. We devised a method to automatically replicate the load of a production system on a test system making it possible early detection of performance bottlenecks. The replication method utilizes a feedback controller to minimize the manual work required. Running performance tests early in the development process reduces the total system development time, which is a clear advantage in an industrial environment with a competitive market.

4.4 Paper C (Based on Paper H)

We have addressed research question Q3 (Section 3.2.3) in:

Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl, and Björn Lisper. *Online Message Compression with Overload Protection.*, Journal of Systems and Software, 2016. [153]

This journal article extends Paper H.

Paper abstract

In this paper, we show that it is possible to increase the message throughput of a large-scale industrial system by selectively compress messages. The demand for new high-performance message processing systems conflicts with the cost effectiveness of legacy systems. The result is often a mixed environment with several concurrent system generations. Such a mixed environment does not allow a complete replacement of the communication backbone to provide the increased messaging performance. Thus, performance-enhancing software solutions are highly attractive. Our contribution is 1) an online compression mechanism that automatically selects the most appropriate compression algorithm to minimize the message round trip time; 2) a compression overload mechanism that ensures ample resources for other processes sharing the same CPU. We have integrated 11 well-known compression algorithms/configurations and tested them with production node traffic. In our target system, automatic message compression results is a 9.6% reduction of message round trip time. The selection procedure is fully automatic and does not require any manual intervention. The automatic behavior makes it particularly suitable for large systems where it is difficult to predict future system behavior.

Contributions

Thesis writer is the main and first author of Papers C and H [152]. Our main contribution is the idea to compress selectively messages depending on network congestion level, message content, and current CPU usage. We have also implemented and evaluated the complete message compression selection mechanism in a telecommunication system. This journal article is an extension of conference Paper H [152]. We have extended Paper H by adding additional compression algorithms and a thorough rework of the paper structure. We have also elaborated on a scenario where the content of a message-stream changes.

4.5 Paper D (Based on Patent O)

We have addressed research question Q4 (Section 3.2.4) in:

> Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl and Moris Behnam. *A Scheduling Architecture for Enforcing Quality of Service in Multi-Process Systems.* In Proceedings of Emerging Technologies and Factory Automation (ETFA), Limasol, Cyprus, 2017. [157]
>
> This paper is an extension of patent O [155].

Paper abstract

There is a massive deployment of multi-core CPUs. It requires a significant drive to consolidate multiple services while still achieving high performance on these off-the-shelf CPUs. Each function had earlier an own execution environment, which guaranteed a certain Quality of Service (QoS). Consolidating multiple services can give rise to shared resource congestions, resulting in lower and non-deterministic QoS. We describe a method to increase the overall system performance by assisting the operating system process scheduler to utilize shared resources more efficiently. Our method utilizes hardware and system-level performance counters to profile the shared resource usage of each process. We also use a big-data approach to analyzing statistics from many nodes. The outcome of the analysis is a decision support model that is utilized by the process scheduler when allocating and scheduling process. Our scheduler can efficiently distribute processes compared to traditional CPU-load based process schedulers by considering the hardware capacity and previous scheduling and allocation decisions.

Contributions

Thesis writer is the main and first author of Paper D, which is based on Patent O. Our first contribution is methods for process resource monitoring, which connects to my earlier research presented in Publication N. We also contributed the with the idea to correlate hardware resource usage with application performance and use the correlation for making process allocation decisions to maximize the system performance. We have also implemented the functionality in the framework of a large telecommunication system. The system can automatically monitor the resource usage of processes and efficiently distribute them over the CPU core cluster depending on their hardware resource usage.

4.6 Paper E (Based on Patents P)

We have addressed research question Q4 (Section 3.2.4) in:

> Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl, Moris Behnam and Björn Lisper. *Quality of Service Process Scheduling by Shared Resource Supervision.* In Proceedings of Emerging Technologies and Factory Automation (ETFA), Torino, Italy, 2018 [147]
>
> This paper is an extension of patent P [156].

Paper abstract

The demand for more advanced and computationally demanding system functions drives hardware manufacturers to improve system performance continuously. More powerful hardware is not always possible due to the increased cost. Many companies try to improve system performance through function consolidations where multiple functions share a common hardware. In legacy systems, each function had individual execution environment that guaranteed hardware resource isolation and therefore the Quality of Service (QoS). Consolidation of multiple functions increases the risk of shared resource congestion. Current process schedulers focus on time quanta and do not consider shared resources. We present a novel process scheduler that complements current process schedulers by enforcing QoS though Shared Resource Aware (SRA) process scheduling. SRA programs the PMU so that it generates an overflow interrupt when reaching the assigned process resource quota. The scheduler swaps out the process when receiving the interrupt making it possible to enforce QoS. We have implemented our scheduling policy as a new scheduling class in Linux. Our experiments show that it efficiently enforces QoS without seriously affect the shared resource usage of other processes executing on the same shared resource cluster.

Contributions

Thesis writer is the main and first author of Paper E which extends Patent P. Our first contribution is to reduce the performance impact for processes sharing hardware resources. We enforce QoS by letting the PMU overflow interrupt trigger a context switch. We implemented the scheduling method in the framework of a telecommunication system with complex and high hardware resource usage. We also implemented the new scheduling class in Linux and evaluated it by using a test application that mimics the target telecommunication system.

Du ska alltid tänka: Jag är här på jorden denna enda gång! Jag kan aldrig komma hit igen! Och detsamma sa Sigfrid till sig själv: Tag vara på ditt liv! Akta det väl! Slarva inte bort det! För nu är det din stund på jorden!.

My own translation:

You should always think: I am here on earth only once! I can never get back here again! Sigfrid said the same thing to himself: Take care of your life! Take care of it! Don't waste it! For this is your moment on earth!

— *Vilhelm Moberg* [213]

5 Measuring Execution Characteristics

This section corresponds to research question Q1 (Section 3.2.1), which we have addressed in papers N (A, B, M).

> *How to monitor the hardware usage and software performance of a production system without noticeable side-effects on the monitored system?*

MONITORING has always been an important issue when optimizing the user-experienced application and system performance [10]. Having trustworthy measurements is vital for making well-founded optimization decisions. It is of similar importance to measure the execution characteristics, i.e. hardware resource usage, of a system to find performance bottlenecks.

We begin this chapter in Section 5.1 by giving a short introduction to performance and execution characteristics monitoring. We discuss theoretical aspects in Section 5.2 and continue by describing our characteristics monitoring (Charmon) implementation in Section 5.3. We have performed a number of experiments to verify that Charmon works as expected, see Section 5.4. The state-of-the art related to performance and characteristics monitoring is described in Section 5.5 and we conclude the chapter by presenting our conclusions in Section 5.6.

5.1 Introduction

Performance is an important issue for most software development projects, and it can be one of the major differentiating factors in a highly competitive market. It is therefore problematic that functional aspects receive the most attention. Performance requirements may often be as vague as the phrase "The overall performance must not be worse than before". We advocate a balanced view of these two requirements and that non-functional requirement should clearly specify the required performance and system characteristics.

There is a need for performance and characteristics measurements to understand and verify where and when performance problems occur. In this thesis we use the term *performance* when we quantify the actual user-experience, i.e., to what degree does the application fulfill the available capacity. We use *execution characteristics* when discussing the resource usage of an application. Characteristics measurements can include performance measurements. The type of characteristics measurements we discuss in this thesis relates to hardware or software resource usage.

5.2 System Model and Definitions

There are several ways to measure the behavior of an application. One way to measure the performance of an application is by using a high-abstraction performance metric [7, 93] such as packets/second or operations/second. It is also possible to measure the hardware resource usage by using the Performance Monitoring Unit (PMU) inside the CPU. One reason for chosing a high-level abstraction is because the hardware resource usage may not completely capture the user-experienced performance of an application. On the other hand will the hardware resource usage explicitly describe what is needed from the hardware to reach the observed performance. Both system-level performance metrics and hardware resource usage are important when describing the performance and behavior of an application or system. The following sections describe each of the concepts.

5.2.1 Hardware Resources

A computers system executes on an hardware platform consisting of many different parts. The most important parts is the central processing unit (CPU), which contains one or several execution units (cores).

Definition 3 The CPU has a set of *cores* denoted C.

The execution performance of a computer application depends on the availability of hardware resources. The CPU is therefore constructed with multiple units that can execute in parallel to give as much performance as possible. Our target system is designed to handle large traffic volumes which put considerable strain on communication links and the memory subsystem on communication nodes. Previous work by engineers within the company has improved the execution performance through algorithm improvements and code optimizations. We will mainly focus on memory-related issues in this thesis although we can use our techniques for any hardware resource. Some of the most important hardware units for memory bound processes are TLB and caches.

Translation Lookaside Buffers (TLB)

The CPU continuously access memory when an application executes. The instruction pointer inside the CPU procedes to the next memory location. When a new memory location is accessed the Memory Management Unit (MMU) needs to translate the virtual address location, understandable by the process, to a physical address understandable by the hardware. This translation is time consuming and most hardware uses a Translation Lookaside Buffer (TLB) cache for the most used address translations. CPU manufacturers use different approaches when defining the TLB implementation. It is common to use several TLB levels and to partition it for instructions, ITLB, and data, DTLB.

Cache Memory

The cache memory in a computer system is a temporary storage of actual memory values, in contrast to the TLB that stores a temporary address translation. The cache is structured in a similar way as the TLB. The cache can be partitioned into instruction (I-cache) and data (D-cache), or being shared between both of them. A CPU is usually designed to have several cache levels, such as L_1D-cache, L_2D-cache, etc.

5.2.2 Memory Management

Memory Management [300] is one of the most important tasks for an OS. The task for the MM is to let processes allocate and free memory when they need it rather than statically allocate all memory at system startup. The MM is also responsible for programming the memory management unit (MMU) hardware

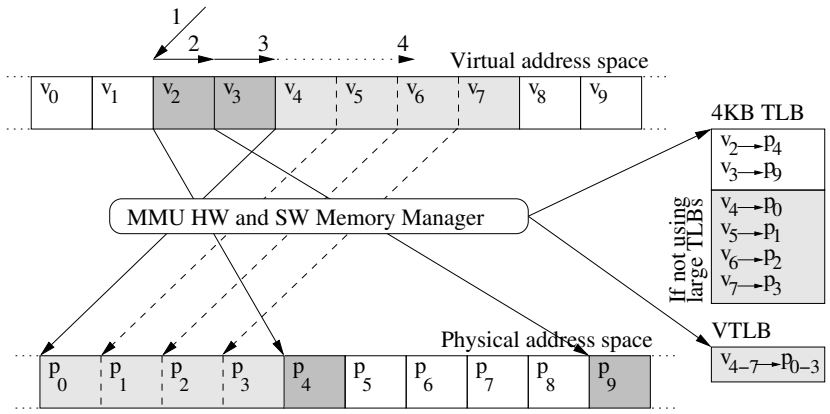


Figure 5.1: Address translation via 4 KB and variable size TLBs.

so that it protects the data of each process from being tampered with [280]. It is tempting to assume that the Linux MM source code is stable but it is still undergoing heavy development [131] due to bug fixes and the demand optimizations. There are three types [193, Ch.3.2.1] of memory allocations. The first type is used when an application explicitly allocate *static* memory through global and static variables. Secondly, the compiler uses *automatically* allocated memory when it detects local variables or function arguments. The third type is *Dynamic* memory explicitly allocated by calling a memory management API. One example of such an API is glibc [193]. The glibc library provides the `malloc()` family of functions, which implements a *heap* where memory is allocated and freed.

Address Translation

Address translation is a central part of the execution flow in modern CPUs [145]. One of the main tasks for the MMU is to translate virtual addresses, $virtaddr$, used by software to physical addresses, $physaddr$, used by the CPU and also on the bus towards the memory subsystem. The memory is divided into pages, typically 4 KB for most OS:es. The memory map contains the set of pages reserved for each process in the system. The kernel throws a segmentation fault (SEGFAULT) if a process access memory outside its memory map and the process is killed. The MMU implements fast and efficient address translation by using cached $virtaddr \rightarrow physaddr$ entries in the TLB cache because it is too slow to search the address map for each address translation [256].

The number of entries in the TLB cache is limited (in the order of 100:s of entries) and each entry in the TLB typically maps a 4 KB page. The TLB is therefore typically denoted as the 4 KB TLB. The complete TLB memory mapping capacity is $100 * 4$ KB $= 400$ KB if the TLB supports 100 mappings. Such mapping capacity is much too small to map the entire memory space for any large-scale computer system utilizing GB or TB of memory.

The TLB cache entries are replaced in a similar way as entries in the L_1-cache, L_2-cache and L_3-cache. If the total system memory working set is larger than the maximum TLB capacity. TLB cache misses are costly because the CPU must halt the program execution and search through the memory map finding the process-specific $virtaddr \rightarrow physaddr$ mapping [143].

Most CPUs also support variable size TLBs (VSP-TLB) to improve the performance. PowerPC supports large TLB with sizes spanning from 4 KB to 2 TB for every power of 2 [105]. Intel supports a similar mechanism but with fewer TLB sizes, such as 2 MB/4 MB/1 GB [140]. The VSP-TLB reduce the pressure on 4 KB TLB for applications using large contiguous memory chunks. Several contiguous 4 KB TLB pages can be replaced by one VSP-TLB mapping as depicted in Figure 5.1. An application strides through memory ① and access a virtual address located on page $virtaddr_2$, which the MMU translates to physical address $virtaddr_4$ by searching the 4 KB TLB. The stride continues through ② ($virtaddr_2 \rightarrow virtaddr_9$). It is possible to drastically improve the performance by using one VSP-TLB ($virtaddr_{4-7} \rightarrow physaddr_{0-39}$) mapping instead of 4 separate 4 KB TLB entries. The performance improvement is even greater if the contiguous memory is several MB or GB. The reasoning above applies to both instruction, ITLB, and data, DTLB, TLB for which most hardware implements separate TLBs.

5.2.3 Systems, Applications and Processes

A computer system is the execution environment consisting of both hardware and software that provides the required functionality. A system typically consists of one or several software applications. The code in an application executes in the context of one or several processes $p \in P$ where P is the set of all processes in the system. Each p has its own memory space protected by the memory management unit (MMU). It is possible to have one or multiple threads, $t \in T$, for one p. All t belonging to a p shares the same memory context provided by process p.

Definition 4 Let $sys \in SYS$ denote the *system* under investigation where the complete set of systems is SYS.

Events
CPU Pipeline Unit Stalls from the following pipeline stages:
→ Fetch (Nr. fetches, Nr. prefetches, Instruction Buffer empty/full)
→ Decode (Nr. stalls)
→ Issue (Simple/Complex integer, Load-Store, Branch, Floating point, Altivec)
→ Schedule (Simple/Complex integer, Load-Store, Branch, Floating point, Altivec)
→ Retire (Completion buffer empty/full)
CPI/IPC
Data Load/Store miss rate/ratio
Branch miss rate/ratio
L_1I-cache miss rate/ratio
L_1D-cache miss rate/ratio
L_2I-cache miss rate/ratio
L_2D-cache miss rate/ratio
L_3-cache read miss rate/ratio (System Wide)
L_3-cache write statistics (System Wide)
Cycles/Interrupts
L_1ITLB miss rate/ratio
L_1DTLB miss rate/ratio
L_2TLB miss rate/ratio
VSP-ITLB miss rate/ratio (Variable size TLBs)
VSP-DTLB miss rate/ratio (Variable size TLBs)

Table 5.1: Examples of hardware event that can be monitored by the PMU.

Definition 5 We denote the *application* under investigation as $appl \in APPL$ where the complete set of applications in sys is $APPL$.

Definition 6 Let $p \in P$ be one *process* of the complete set of processes P executing on system sys. We use a subscript, $p_i \in P$, if we need to differentiate between multiple processes.

We often compare processes belonging to different applications in our examples. We therefore define P_{appl} as the set of processes belonging to a certain application $appl$.

Definition 7 The set of processes for $appl$ in sys is denoted P_{appl}, where $P_{appl} \subseteq P$ and $appl \subseteq APPL$.

5.2.4 Hardware Resource Monitoring

Every system that runs on a computer depends on the availability of low-level resources, which makes it possible for the system to complete its task. A resource can, from a system-level perspective, relate to both software and hardware. A software resource is often a service provided by the OS or a middleware Application Programming Interface (API). A hardware resource is typically provided by the CPU or some peripheral component. Some examples of hardware resources are the Arithmetic Logic Unit (ALU), caches, MMU and many other. It is often necessary to measure the hardware resource usage for a process when debugging performance related issues. We therefore introduce the concept of hardware resource samples, $m_{r,p}$ of hardware resource r for process p.

Definition 8 The *hardware resource*, $r \in R$, is one of the total set of hardware resources, R. We use a subscript i, such that $r_i \in R$, if we need to differentiate between multiple resources.

Definition 9 Let r_{appl} denote the set of resources used by an application $appl$.

Definition 10 A bounded series of *resource usage samples* of hardware resource $r \in R$ for process $p \in P$ is denoted by $m_{r,p}$, by $m_{r,appl}$ for an application $appl \in APPL$ or by $sys \in SYS$ for a system.

Measuring resource usage There are many different tools available for measuring the r_{appl} for application $appl$. One efficient way [83] to measure r_{appl} usage is by utilizing the Performance Monitoring Unit (PMU) [226] of modern CPUs. The PMU is implemented purely in hardware and once it has been programmed it runs without any software intervention or execution cost [108]. It is therefore an efficient way to measure the resource usage. The PMU is widely available for Intel® [183] and similarly on PowerPC® where it is called Performance Monitor Counters (PMC) [104]. Engineers have used the PMU for a long time when investigating user-space performance problems. A more recent trend is to use the PMU for kernel space performance investigations [258]. Table 5.1 shows some examples of hardware resources that can be monitored by the PMU. No hardware resource monitoring tool was available for our legacy target OS when we started our investigation leading up to this thesis. At the time there were some tools for Linux, for example, Perf [62], that implemented a subset of our requirements. Because of the GPL-license, it is politically difficult to port Perf to a proprietary OS. We opted to implement a tailored monitoring tool that supported all of our industrial requirements.

Charmon currently runs on two different OS:es, Enea's OSE for our legacy system, and Linux for current and future systems. We use Charmon for long-term monitoring, and continuously run it while sampling multiple hardware metrics through the PMU [83]. A PMU is an hardware implemented event counter, and it can autonomously count the occurrences of the specified event after it has been programmed. PMU events [104] that are common for many hardware architectures are for example cache misses, RAM accesses, branch misses and similar issues. There are also other types of events that are unique to each architecture, for instance, related to the execution pipeline, memory subsystems and similar.

5.2.5 Service Performance Monitoring

There are numerous tools [124] available for measuring the performance of a computer system. Some tools measure many aspects of a complete system such as Spec [127] or LM-Bench [204] while others target a subset such as the CPU [126], floating point unit (FPU) or the cache/memory [201]. One common thing among all these tools is that they describe the performance through a high-level performance metric such as operations/sec, messages processed/sec or similar. There are well-founded reasons [93] for using a high-level performance metric when describing the application performance. We could, for example, try to express the performance of an application by measuring the R usage, such as the number of Instructions Per Second (IPS). IPS counts the number of instructions executed by the CPU, which would be high if the software iterates in a spin-lock but the user-experienced performance is low [93]. Therefore, mea-

Event	Data Source	Description
CPU-load	OS	The OS tracks execution statistics for all $p \in P$ and $appl \in APPL$.
Message RTT	Application	
Number o fusers	Application	The number of concurrent users in the system.
Operations/second	Application	The number of system or application operations executed per second.

Table 5.2: Some examples of software performance metrics.

suring R may not be a good way to describe the user-experience performance. We propose to use a high-level metric to measure the experience performance.

Definition 11 The *performance* is denoted by $x \in X$ where X denotes the set of all performance metrics. The *performance* of process, p, is denoted x_p and x_{appl} for application $appl \in APPL$ and x_{sys} for system $sys \in SYS$.

Definition 12 The bounded series of *performance metric samples* of x for process p is denoted $m_{x,p}$.

Charmon measure x in a similar way as each $r \in R$ and continuously monitor process-specific and user defined x with fixed time period. Some examples of x is shown in Table 5.2. It is useful to have one tool to simultaneously capture both x and R since each measurement gets the same time-base and is therefore easily comparable.

5.3 Implementation

We have implemented Charmon with the explicit requirement to be continuously running within the platform. Charmon samples both hardware and software metrics. The hardware-usage is sampled by periodically setting and reading the PMU [102, 103]. Each individual counter inside the PMU can be configured to count one hardware-event. Charmon stores counted events in a database together with x measurements. Charmon has a very low probe effect [108] since the PMU is implemented in hardware located inside the CPU with negligible performance penalty. The data read from the PMU is infrequently stored in a memory-based database that is flushed to disk at user-demand.

Charmon was first implementation for Enea OSE running on a Freescale p4080 [102] with 13 concurrently running sets of performance events, see Listing 5.1 for examples of the PMU configuration. The implementation targets various parts of the functionality described in Table 5.1. We designed each event set to help providing an understanding of one particular hardware function. Each set uses between one and six PMU counters depending on the desired functionality and the number of counters supported by the hardware.

The second type of counter set utilized by Charmon are software based. A software counter can in practice be anything that is countable, but the two primary software metrics monitored in Charmon are CPU-load, supplied by the OS, and message round trip time. Our initial implementation of Charmon supported two software metrics related to the application performance. The supported metrics were CPU load and message round-trip time. CPU-load shows

Listing 5.1: ”Example PMU configuration for sampling hardware usage.”

```
PME_ITEM sampleConfig[] = {
  /* 0. IPC */
  {{{ E500_PME_INSTRUCTIONS_COMPLETED, E500_PME_PROCESSOR_CYCLES,
      E500_PME_NOTHING, E500_PME_NOTHING }}, &phInstr, &ptInstr },
  /* 1. DLS: Data Load/Store ratio
   * DLS below 1 indicates odd memory usage behavior and potentially poor cache usage as
   * data is more frequently written than read. DLS is typically above 3. */
  {{{ E500_PME_LOAD_MICRO_OPS_COMPLETED,
      E500_PME_STORE_MICRO_OPS_COMPLETED,
      E500_PME_NOTHING, E500_PME_NOTHING }}, &phDls, &ptDls },
  /* 2. BTB Hit/Miss: Branch Target Buffer Hit/Miss Ratio
   * Low BTB hit ratio indicates BTB is turned off and instruction execution is poor, generally
   * also giving poor IPC. BTB Hit should be above 0.7. Consider using likely()/unlikely() in
   * branch statesment to improve BTB Hit. */
  {{{ E500_PME_BTB_HITS_AND_PSEUDO_HITS, E500_PME_BRANCHES_FINISHED,
      E500_PME_NOTHING, E500_PME_NOTHING }}, &phBtb, &ptBtb },
  /* 3. L1 I$ Hit and Miss Ratio
   * Low I cache hit ratio indicates poor execution flow, likely due to jumpy code or poor BTB hit. */
  {{{ E500_PME_INSTRUCTION_L1_CACHE_RELOADS_FROM_FETCH,
      E500_PME_INSTRUCTIONS_COMPLETED, E500_PME_NOTHING,
      E500_PME_LOAD_MICRO_OPS_COMPLETED }}, &phL1IHitMiss, &ptL1IHitMiss },
  /* 4. L1 D$ Hit and Miss Ratio */
  {{{ E500_PME_DATA_L1_CACHE_RELOADS, E500_PME_LOAD_MICRO_OPS_COMPLETED,
      E500_PME_STORE_MICRO_OPS_COMPLETED,
      E500_PME_INSTRUCTION_L1_CACHE_RELOADS_FROM_FETCH }},
      &phL1DHitMiss, &ptL1DHitMiss },
  /* 5. L2 Hit and Miss Ratio */
  {{{ E500_PME_L2_CACHE_INSTRUCTION_HITS,
      E500_PME_L2_CACHE_INSTRUCTION_ACCESSES,
      E500_PME_L2_CACHE_DATA_HITS, E500_PME_L2_CACHE_DATA_ACCESSES }},
      &phL2HitMiss, &ptL2HitMiss },
  /* 6. Interrupts per Cycle
   * A high Int.pC indicates the core is not smootly executing but
   * constantly being interrupted, this will give poor performance. */
  {{{ E500_PME_PROCESSOR_CYCLES, E500_PME_INTERRUPTS_TAKEN,
      E500_PME_EXTERNAL_INPUT_INTERRUPTS_TAKEN,
      E500_PME_SYSTEM_CALL_AND_TRAP_INTERRUPTS }},
      &phIrqPerCycle, &ptIrqPerCycle },
  /* 7. ITLB hit/miss rate */
  {{{ E500_PME_INSTRUCTIONS_COMPLETED, E500_PME_INSTRUCTION_MMU_TLB4K_RELOADS,
      E500_PME_INSTRUCTION_MMU_VSP_RELOADS, E500_PME_NOTHING }},
      &phITLB, &ptITLB },
  /* 8. DTLB hit/miss rate */
  {{{ E500_PME_LOAD_MICRO_OPS_COMPLETED, E500_PME_DATA_MMU_TLB4K_RELOADS,
      E500_PME_DATA_MMU_VSP_RELOADS, E500_PME_STORE_MICRO_OPS_COMPLETED }},
      &phDTLB, &ptDTLB },
  /* 9. L2TLB hit/miss */
  {{{ E500_PME_PROCESSOR_CYCLES, E500_PME_L2MMU_MISSES,
      E500_PME_INSTRUCTIONS_COMPLETED, E500_PME_NOTHING }},
      &phL2TLB, &ptL2TLB },
};
```

the number of processes in the ready-queue [119]. The round-trip message time describes performance on a system level. We have introduced many other metrics in later version of Charmon, see Table 5.2 for some examples.

Sampling Frequency Selecting a sampling frequency was not a simple task. We had to deal with some critical demands during the implementation of the first version of Charmon. The most important ones were:

- We are using the performance-related data to create a semi-static model that does not require higher sampling frequency.
- A customer is very sensitive to any disturbance of their system. It is, therefore, challenging to get consent to running testing tools in production systems. We must be certain that Charmon does not affect the system performance or behavior in any way.

The general discussion revolved around not being able to run any monitoring tools, because of the scare of probe effects and using a very low sampling frequency. We opted for the latter solution and selected to use a 1Hz sampling frequency, which is undoubted to low for capturing a dynamic behavior but sufficient for detecting semi-static behavior or trends. The reason for using such a low frequency is not technical; it is a matter of ensuring that it will never impact the behavior of the system being observed.

5.3.1 Measuring Characteristics

Charmon iterates over a list of hardware event sets that is each programmed to the PMU for a period. As shown in Figure 5.2 Charmon is awoken ① by a fixed interval timer interrupt and sleeps in between. Charmon starts by reading ② the resulting values for the previous hardware counter set. Reading hardware counters is, for the legacy OS, low-intrusive by utilizing the `mfspr` [104] assembly instruction. The PowerPC® instruction set defines this particular instruction, but there are similar instructions for other architectures. On Linux, we use the Perf-API [62] for reading hardware metrics and our implementation for reading software metrics. The logical functionality, which is the major part of the Charmon application, is the same for both OSes. For both OSes, the measurements are stored ③ in a local database (DB). Next, the subsequent hardware performance counter set is read ④ from a table and programmed ⑤ into the PMU registers. The PMU programming is similar to reading and we use the `mtspr` for the legacy OS. The Linux Charmon implementation utilizes the Perf-API, which implements the PMU interaction.

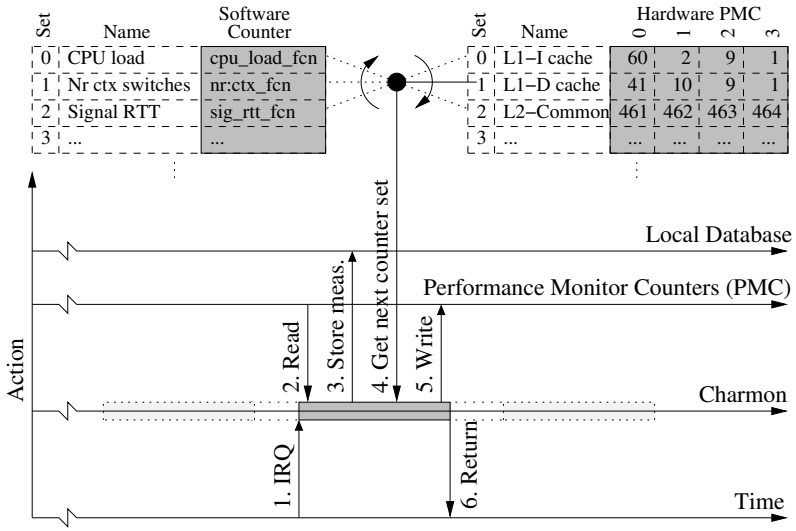


Figure 5.2: Hardware Characteristics measurements using Charmon.

It is also possible to add other software metrics, such as CPU-load, process context switches, signal turn around time. Our implementation uses CPU-load, which is supplied by the OS, and round-trip message time, which is supplied by the messaging application. Measurements for software metrics are stored in the DB to provide a contextualized and time-stamped log of both hardware and software utilization. Charmon provides the possibility to have a mix of both low-level and high-level metrics, which is useful when debugging/investigating performance related problems. After setting a new set ⑤ of hardware counters, Charmon sleeps for a predefined interval, then restarts at step ①. When using multi-core CPUs we follow a similar procedure where Charmon simultaneously programs all cores with the same counter set.

5.3.2 Counter Sets

Charmon implements two types of counter sets. The first and by far largest set uses hardware PMU counters. The second set uses software counters. We started by investigating the first set that contains hardware metrics describing the system performance, for example, instructions per second and cycles per second. By using these two metrics, it is possible to calculate Cycles Per Instruction

(CPI), which to some extent describes the efficiency of the system [94]. The next area of interest is to understand where the system loses performance. It is well-known from interviewing senior technicians within the organization we belong to that the target system we are investigating is memory-bound. Therefore, we implemented several counter sets to observe all cache usage regardless of the cache level. Using the CPI-metrics [94, 95] as a guideline we implemented many more metrics, such as counters for Translation Lookaside Buffers (TLB), branches, floating point units and other. We know that we must be careful when using CPI-stacks since they can be misleading [7], especially for multi-core CPUs. We also include counters for all pipeline stages since that is helpful to gain further knowledge of where stalls could occur.

Charmon has been designed and implemented to allow easy addition of more counter sets. Our aim has been to ease the extension of Charmon with additional counter sets whenever the need arise. In the future, we expect that memory subsystem metrics may be of specific interest because new hardware architectures introduce more multi-level cache hierarchies, non-uniform memory accesses, and other complex techniques.

5.3.3 Second Generation Implementation

The original version of Charmon has been used for an extensive period of time and in several ways. We received several new requirements during the initial Charmon usage, which related to both new functionality and what was conceived as functional limitations.

The second generation of Charmon was ported to the x86 architecture as a natural continuation of previous work. The PowerPC® target was not further developed in the context of our target system and the functionality of old Charmon suffice for the time being. The OSE-version of Charmon was also discontinued at the same time and we focused all our implementation efforts into further development of the Linux version [9].

The original Charmon supported a low, user-defined, sample frequency but in practice we never used a higher sample frequency than 10Hz because we were investigating semi-static systems and not their dynamic behavior. The performance debugging methodology implicitly stated that we should use Perf [62] when investigating performance-related problems related to dynamic behavior. We received several questions related to much higher sample frequencies causing us to revisit this issue in the second generation Charmon. Our changes included changes to the overall software structure and some code optimizations to reduce the sampling overhead.

A more generic request related to the generality of Charmon. Both ourselves and other users wanted to use Charmon on several different CPU architectures. The PMU hardware differs slightly between chip architectures, manufacturers and even between chip revisions. Running on two x86 CPUs does not necessarily mean that they have the same PMU counter hardware implementation. We addressed this issue by implementing a target specification JSON [232] file. The file specifies the available hardware counters making the source code generic. We also implemented a more generic interface to support sample-event configuration through configuration files.

5.4 Experiments Using the Performance Monitor

There are numerous ways to use a performance monitor such as Charmon. We have utilized Charmon in several ways. The first way was to find specific performance-related problems and understand why they occur. The second way is to create a Cycles Per Instruction (CPI) stack, which is a slightly more advanced way to debug performance issues. Finally, we have used Charmon by connecting it to other system functions providing a closed-loop.

5.4.1 Debugging Performance Related Problems

A typical use-case for Charmon is when a system designer encounters a performance related problem. We promote a methodology that starts with using Charmon with the default event-set, see Listing 5.1, on the system under investigation. The default event set gives an overview of the system resource usage, and an experienced engineer can quickly see if some events indicate a problem. When a problem-area has been found we advocate defining one or several new event-sets that encapsulate and investigate a particular problem in more detail. These new sets can explain why the problem happens. We have used this methodology on several occasions when performance debugging and many system engineers believe that Charmon is an adequate tool to improve the understanding of a systems' execution characteristics. We present three examples of successful performance investigations using Charmon in the following text.

1) Investigating a performance reduction between two system versions

The performance of our target system was not tested during the design phase at the time of this investigation. The performance was validated much later, during the system-testing phase, where system testers detected that the performance,

x, of the system had dropped between two subsequent versions A and B of a production-system with identical $appl$. The only difference between A and B was 10 bug fixes inside the OS, none of which addressed the functionality used by $appl$.

Setup The test setup consisted of a cabinet with 3 subracks each containing 20 PowerPC® 750GX [132] CPU-boards interconnected with a Gigabit backplane. Each board ran a full version of the telecommunication system including the production application. The application performs message processing (memory strides) on received messages before sending them to its communication partner. We suspected that the memory system was congested and therefore decided to monitor the following resources $R = \{\text{L}_1\text{D-cache}, \text{L}_2\text{D-cache}, \text{L}_1\text{I-cache}, \text{ITLB}, \text{VSP-ITLB}, \text{DTLB}, \text{VSP-DTLB}\}$. The performance of $appl$ is specified by $x = \{\text{CPU-load}\}$

Execution This is the seminal test of the Charmon monitoring tool. We designed and implemented Charmon and gave the system-testers instructions on how to run it. The system testers configured a test system similar to the production system and started Charmon measuring the system performance, x_A, before and after the OS upgrade x_B. The measurements resulted in extensive logs $m_{r,A}$, $m_{r,B}$, as well as $m_{x,A}$, $m_{x,B}$ for $r \in R$,

Evaluation We quickly deduced that x increased 10 percentage points (pp) between A and B. We investigated and compared the measurements $m_{r,A}$ and $m_{r,B}$ showing that the 4 KB instruction TLB usage had increased tenfold between A and B. Similarly the VSP-ITLB was close to zero for B.

Remarks The functionality added in B compared to A was not directly related to the system function that suffered the performance degradation. Further investigation showed that the size of the kernel monolith had grown with as little as a couple of KB so that the system kernel-trap interface entry point ended up outside the 4 MB VSP-ITLB mapping the complete kernel monolith.

The kernel part located outside of the 4 MB VSP-ITLB caused a radically increased load on the 4 KB TLB throughout the system. It was easy to fix the problem by realigning the monolith so that the VSP-ITLB covered the complete code section, and the performance returned to an acceptable level. We conclude that the Charmon tool quickly indicated the source of the performance loss. Contrary, it would have been very difficult to pinpoint the problem by measuring the execution environment only.

Feature	PowerPC® p4080 [264]	PowerPC® T4240 [105]
Core	8xe500mc (32bit, 1.5GHz)	12xe6500 hyperthreaded → 24 virtual cores (64bit, 1.6GHz)
L_1-cache	Two separate/core 8-way (I+D) 32 KB set-assoc. pseudo LRU	Two separate/core 8-way (I+D) 32 KB set-assoc. pseudo LRU
L_2-cache	One separate/core 8-way 128 KB set-assoc. unified I and D pseudo LRU	3 x 16-way set-associative 2048 KB unified, shared among four e6500
Platform Cache	2 MB platform cache/CPU	3 x 512 KB platform cache
MMU	6 TLB in total, 8-entry, fully-associative, I/D L_1TLB arrays for Variable Size Pages (VSP), 64-entry, 4-way set-associative I/D L_1TLB arrays for 4 KB pages, 64-entry, fully-associative unified L_2TLB array for VSP, 512-entry, 4-way set-associative unified L_2TLB array, for 4 KB pages.	6 TLB in total, 8-entry, fully-associative, I/D L_1TLB arrays for Variable Size Pages (VSP), 64-entry, 4-way set-associative I/D L_1TLB arrays for 4 KB pages, 64-entry, fully-associative unified L_2TLB array for VSP, 1024-entry, 8-way set-associative unified L_2TLB array for 4 KB pages, hardware page table-walk.

Table 5.3: Cache specifications for PowerPC® p4080 [264] and T4240 [105].

2) Comparing two operating systems

The legacy system in this experiment was running on Enea OSE real-time [39, p430] OS. There was a general opinion in the design department that OSE was more efficient than Linux. This opinion was further strengthened by earlier performance tests. We formulated an M.Sc. thesis [9] to investigate the accuracy of this belief in more detail. The M.Sc. thesis worker used Charmon to measure and compare the execution characteristics for OSE and Linux.

Setup The test setup contained two p4080 [102] PowerPC® CPU-boards, see Table 5.3, interconnected with a 100Mbit ethernet link. Each board ran a minimalistic version of our basic telecommunication system and a test application. The test application replicates the behavior of the production application and performs some basic message processing (memory strides) on received messages before sending them to its communication partner. We defined $x = \{\text{message Round Trip Time (RTT)}\}$.

Execution We ran Charmon with the configuration, R, listed in Table 5.1 when monitoring the execution characteristics and application performance of the system. We ran the two Test Cases (TC), the first on Enea OSE (TC_{OSE}) and the other on a Linux system with kernel 2.6 (TC_{Linux}).

Evaluation The first measurements of x showed that the message RTT was about 300% higher in TC_{Linux} vs. TC_{OSE}. We saw a similar pattern when we evaluated the R. The D-cache and I-cache hit ratio in TC_{OSE} is almost 100% while it was substantially lower in TC_{Linux}. Investigating the TLB measurements show that the number of 4 KB and VSP-DTLB evictions in TC_{OSE} is 0, but $800k$ 4 KB DTLB reloads/sec and $750k$ 4 KB ITLB reloads/sec. The VSP-TLB usage in TC_{Linux} us 0.

Remarks It is apparent from the evaluation results that Linux uses a much larger working set, both for data and instruction. This is in fact not surprising since Linux is a general-purpose OS while OSE is tailor made for the type of system we have evaluated. Other results came as a surprise. We also monitored the number of system traps and interrupts for each TC_{OSE} and TC_{Linux}. Our measurements showed that Linux generated between 100-250 times more interrupts than OSE for the same test run. A radical increase of interrupts is very expensive from an execution point-of-view. Our recommendations for the software department was to 1) start using VSP-TLB for the Linux system and 2) investigate why network traffic generates so many interrupts.

3) Comparing two hardware architecture

We have also compared the execution characteristics of a Linux system running on two different CPUs. We performed the tests within the context of the same M.Sc. thesis [9] as in Example 2. The goal was to evaluate and predict the system performance if the system was migrated from one type of CPU to another type.

Setup We used a p4080 [102] PowerPC® running Linux kernel 3.12.19 in the first test case, which is denoted by TC_{p4080}. The second test case, TC_{T4240}, utilized a T4240 [105] PowerPC® CPU running Linux kernel 3.813. See Table 5.3 for hardware information related to the two CPUs. The reason for choosing different kernel versions was that both our hardware was not supported in any of the two kernels. See Table 5.3 for hardware specifications. We configured the $R = \{\text{D-cache}, \text{I-cache}, \text{DTLB}, \text{ITLB}\}$. We had two concurrently measured x.

The first was the total runtime for all test applications and the second one was the Dhrystone [297] runtime. We ran the memtester [43] and IPerf [74] as test applications. Memtester load the memory subsystem and IPerf is an TCP, UDP and SCTP test application. Dhrystone performs integer arithmetics and cause heavy CPU load.

Execution We started Charmon and the test applications according to the following affinity settings: memtester ($A_{mem} = \{c_2\}$), Dhrystone ($A_{dhry} = \{c_4\}$) and IPerf with 1G link ($A_{IPerf} = \{c_7\}$).

Evaluation The total test runtime was 1m33s for TC_{p4080} and 1m04s for TC_{T4240}. Dhrystone executed 55% faster on TC_{T4240} compared to TC_{p4080}. The L_2D-cache hit ratio was 80% on TC_{p4080} and almost 100% on TC_{T4240}.

Remarks One apparent difference is that the core frequency is higher in TC_{T4240} compared to TC_{p4080}. Increasing the CPU frequency between 1.5GHz and 1.6GHz cannot by itself explain the massive performance improvement. We attribute the increased L_2D-cache hit rate in TC_{T4240} to the larger L2 cache. The performance is much higher in TC_{T4240} because the cache hit ratio is almost 100%. The TLB handling is another difference between TC_{p4080} and TC_{T4240}. The p4080 in TC_{p4080} uses software table walk when the requested memory location is not in the TLB. software table walk can be a costly operation since parts of the OS must operate in page-fault exception context when fetching the demanded $virtaddr \rightarrow physaddr$ memory mapping and reprogramming the TLB. The T4240 in TC_{T4240} uses hardware table walk meaning that the OS configure a memory region describing the current memory setup. The hardware automatically locate the memory mapping when a memory request cause a page fault without any page handling software being executed. Other researchers have earlier stated that software table walk has some other advantages, such as flexibility [143] as hardware table walk [14, Ch. 6.1]. In this case the hardware table walk is very efficient and improves the performance.

5.4.2 The Cycles Per Instruction (CPI) Stack

We have created a cycles per instruction (CPI)-stack to get an indication of the most relevant hardware metrics to monitor. Eyerman et al. [94] describes CPI as the total execution cost of a single instruction, including wasted capacity caused by processor stalls such as branch prediction misses, TLB misses, cache misses. Splitting the CPI into each contributing cost builds a CPI stack and the CPI

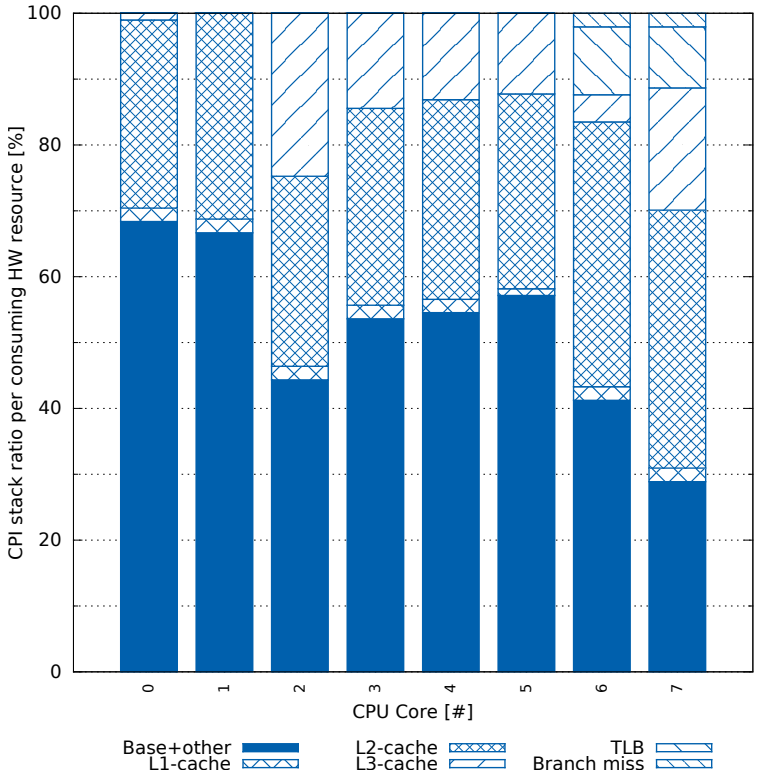


Figure 5.3: CPI stack for the production system.

stack can be used to illustrate how big part of the total execution time is spent doing real work and how much time is spent stalling for congested resources. A system engineer can evaluate how well a set of processes executes on a core by building a CPI stack per CPU core.

We are using the PowerPC® p4080 CPU in our target system, which has no specific hardware counters for measuring the number of lost cycles contributed by individual shared resources. We have therefore calculated each CPI-stack contributor by using the arithmetic mean [198] cost of each access and multiplied it by the number of accesses. The performance monitor counters can measure the number of accesses. The mean performance penalty was obtained

by interviewing hardware designers and through testing. Following our reasoning, we estimate the CPI-stack as:

$$CPI = CPI_{\text{L}_1\text{-cache}} + CPI_{\text{L}_2\text{-cache}} + CPI_{\text{L}_3\text{-cache}} + CPI_{TLB} + CPI_{branch} + CPI_{base+other}$$

We agree with other researchers [8, 94, 208] that our simplified CPI-stack is blunt, but it confirmed the opinions expressed by senior hardware and software designers working within our target organization. Figure 5.3 illustrates our measurements. In this particular test run, we focus on cores 6 and 7, which is where our target application runs. Cores 0-5 runs other applications not closely related to this investigation. They will not have much impact on the tested application apart from their sharing the same L_3-cache and other peripheral resources. Figure 5.3 shows that the majority of wasted execution time is spent waiting for L_1-cache, L_2-cache, and L_3-cache.

5.4.3 Closed Loop Interaction

We have implemented Charmon to support a closed-loop interface. We want to provide other applications the possibility to directly access performance-related measurements and also act upon the current system performance. The concept of acting on performance measurements is something that we will discuss further in Chapters 6, 7 and 8.

5.5 Related Work

We have found exciting relations to our work in the area of continuous system monitoring. For example Anderson et al. [10] implements a low intrusive (the execution impact is 1%-3%) sample based mechanism to gather system-wide information. Their implementation samples hardware performance counters when they generate overflow interrupts. Our implementation uses a timer to sample the hardware performance counters periodically.

One of the standard work when monitoring or measuring system performance is the LM-Bench suite by Mcvoy and Staelin [204]. It measures and calculates cache and memory timings to understand the system behavior. Unfortunately, our legacy platform does not support all API-calls and development tools required by LM-Bench. The lack of API support is one major factor for us implementing our hardware monitor.

Eranian [83] claims that performance monitor counters are an essential component in performance measurements and when evaluating system performance. They have investigated resource usage on the Intel® architecture. We started our resource and performance investigations on the PowerPC® architecture and then, some years later, continued on the Intel® architecture. The fundamental approach is similar to our monitor and theirs. They run a measurement application that gathers resource and performance information for later evaluation. In our case, we have extended this idea to let the samples provide input to a feedback control algorithm that can later mimic the monitored system.

Diniz et al. [69] have investigated how to use feedback control mechanisms to improve program compilation. They have modified a compiler to use performance feedback results from an initial application test-run. Their method allows the compiler to utilize complex optimization mechanisms that are not usually possible to utilize when using static methods at compile time. Lau et al. [180] extend the work by investigating how feedback control techniques can improve the performance of JAVA programs executing in a Virtual Machine (VM). They state that there are plenty of known optimization techniques available, but it is hard for a VM to know which one to use in particular cases. Sometimes a function optimization decreases the overall performance because the function may operate on different data during the next iteration.

Eyerman et al [94] describe an architecture for measuring the CPI-stack via tailored PowerPC® Power5 hardware performance counters. They also describe the difficulties and errors when using more straightforward methods, denoted "naive", like multiplying the number of misses with the average cost. The total number of misses may, for example, contain entries in mispredicted execution paths that should be omitted in the real CPI calculation. In our case, we could only use the naive method because our target hardware architecture did not support extended CPI counters. Additional research by Eyerman [93] suggest that CPI-stacks simplifies the execution environment and that the result may be misleading. Their suggestion is to use a system-level metric to describe the performance as described by Alameldeen and Wood [7] and Eyerman [93]. We have used CPI as a low-level metric to get an indication of which shared hardware resources to synthesize in our models. We use message round-trip time as the system level metric when measuring the system performance.

Sherwood et al. [268, 269] deduce that it is possible to model subsections of an application by dividing it into basic block vectors and providing a hardware-independent metric. In our case we have problems simulating the production application, because of its size and complexity, making it difficult to use this

approach. Our opinion is that their technique can act as an good complement to ours.

Demme [65] states that it is possible to accelerate system development by using and evaluating the result from PMU counters. They have devised an own tool, *LiMits*, and used it to perform case studies on Firefox and MySQL.

5.6 Summary

We have answered Q1 (Section 3.2.1) by the work presented in the text above and through our publication A, which is further expanded in the technical reports M and N. We have reused knowledge, techniques and tools in our later papers J and K.

We have shown that it is possible to monitor the production environment of a large-scale industrial system with our implemented Charmon tool. Charmon can measure both hardware resources and the system performance continuously and present the results in a user-friendly manner. Charmon also implements an API where other applications can extract resource and performance data for selected processes.

We use this feature in our research presented later in this thesis. We have used the hardware resource measurements and performance measurements to debug performance-related problems. The measurements aided us to find solutions and give advice to system engineers investigating performance-related problems. The Charmon tool is part of the industrial system.

There are numerous ways to improve our current work. We have made substantial changes to the Charmon tool over the time of our research, adding new features and porting it to new hardware architectures. There is a natural extension to this. We should widen the perspective and make the tool freely available. The next step is to generalize it further to make it even easier to add new architectures, PMU events, performance metrics. There are also several improvements to be made in the current sampler. We would like to optimize the PMU sampler to reduce the impact on the observed system further. The cause of action will follow the industrial demand, and we will address the issues with the highest industrial impact on the product portfolio.

Det här är inget man kan diskutera, jag har rätt och du har fel.[1]

My own translation:

This isn't something to discuss, I am right and you are wrong.

— *Hans Rosling* [248]

[1] Hans Rosling exclaims "This isn't something to discuss, I am right and you are wrong." during a danish DR2 TV-interview, when the program leader says that world is in chaos with regards to war and refugees.

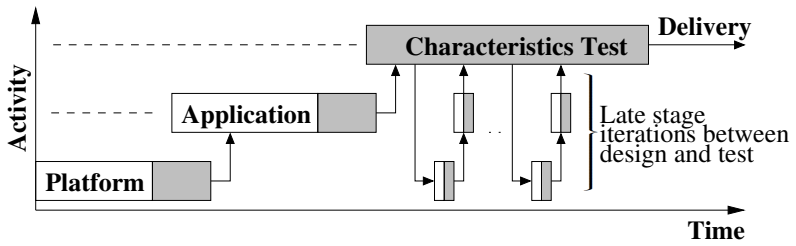
6
Load Replication

This section corresponds to research question Q2 (Section 3.2.2), which we have adressed in the technical report N based on papers A, B and M.

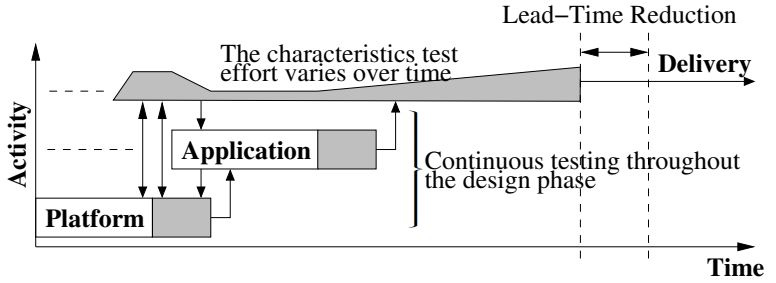
> *How to automate modeling and replication of hardware usage for a production system?*

PERFORMANCE is an important issue for most software development projects, and it is one of the major differentiating factors for most new software releases in a highly competitive market. Time-to-market is also a key factor [110,293] for large-scale industrial systems [122]. The ability to deliver performance improvements for the next generation of an existing product is an ever-more important issue in the software industry. This chapter describes a technique that firstly models the execution characteristics of a production system. Secondly, it replicates the execution characteristics on a test system. The ultimate goal of our method is to make it possible to test newly developed system functions with regards to performance issues much earlier than with traditional functional testing.

We begin this chapter in Section 6.1 by giving an introduction to load replication. We continue by describing our theoretical approach in Section 6.2, more specifically how we create the execution characteristics model. Section 6.3 describe our implementation specific details and lists two successful experiments replicating load from a production system in Section 6.4. We continue with related and future work in Section 6.5. We conclude the chapter by presenting our conclusions in Section 6.6.



a) Characteristics testing and corrections are iteratively performed at the end of the design process.



b) Performing characteristics testing throughout the development process shortens the total development time.

Test phase Development phase

Figure 6.1: Two performance verification approaches of an industrial system.

6.1 Introduction

Many companies use various degrees of traditional waterfall [252] development processes when implementing industrial systems. Agile development methods continuously make their way into design organizations although large organizations still have specific checkpoints to pass before passing on a product to later stages in the development process.

Figure 6.1a depicts the current, semi-sequential, development process with a slight overlap between some development activities. Platform implementation starts the chain of development activities. When the platform starts to reach a finished state, it is subjected to various function-tests during the same time the application development starts in parallel. A similar scenario applies to

the application development. A system test organization usually gets involved just before the system deployment. The system verification stage is where testers verify the functional properties, absence of bugs and overall system performance. There are also many iterations between the test-, the design- and application departments because various bugs needs to be fixed and retested. The latter is depicted to the right in Figure 6.1a.

The extended development time results in long lead-times between the start of platform development and system performance test. Long lead-times coincide with the fact that it is expensive to fix bugs late in the development process as stated by Boehm [28]:

> Finding and fixing software problems after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.

Some important reasons why it is important to get early-stage characteristics feedback are that:

- The lab costs for hardware and personnel when testing a complete production system are magnitudes larger compared to using small test nodes.
- It is expensive to fix bugs late in the development phase [282]. One reason for this is that the developers that made the initial code might not be available for troubleshooting and fixing the problem.
- It is vital to get an early performance evaluation when performing cost-reduction activities for an existing product. Several functions previously executing on different CPUs may be co-located to one CPU in an effort to reduce cost. System function consolidation may have undesirable effects on system performance.

Our idea is that it is possible to reduce the development time by moving all or part of the execution characteristics testing to earlier design phases in the development process, as depicted in Figure 6.1b. The main benefit is that performance related bugs are easier to detect, isolate and correct because engineers are in the middle of implementing and debugging the software. An additional bonus-effect is the feedback that the engineer receives when his/her code is tested for performance issues.

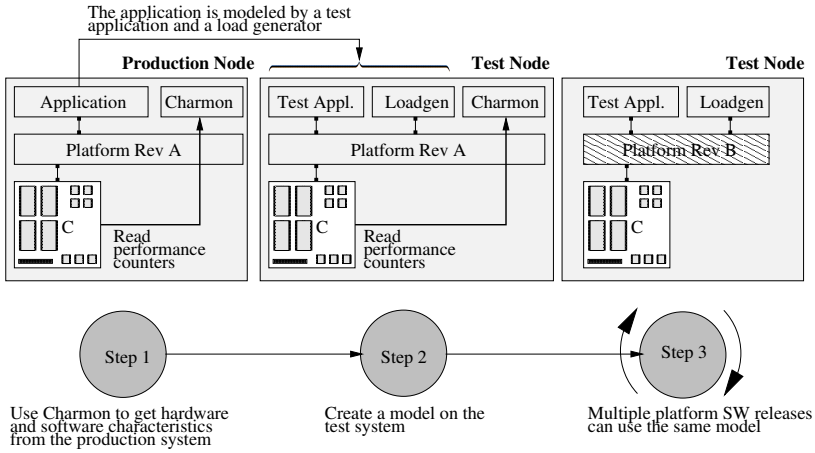
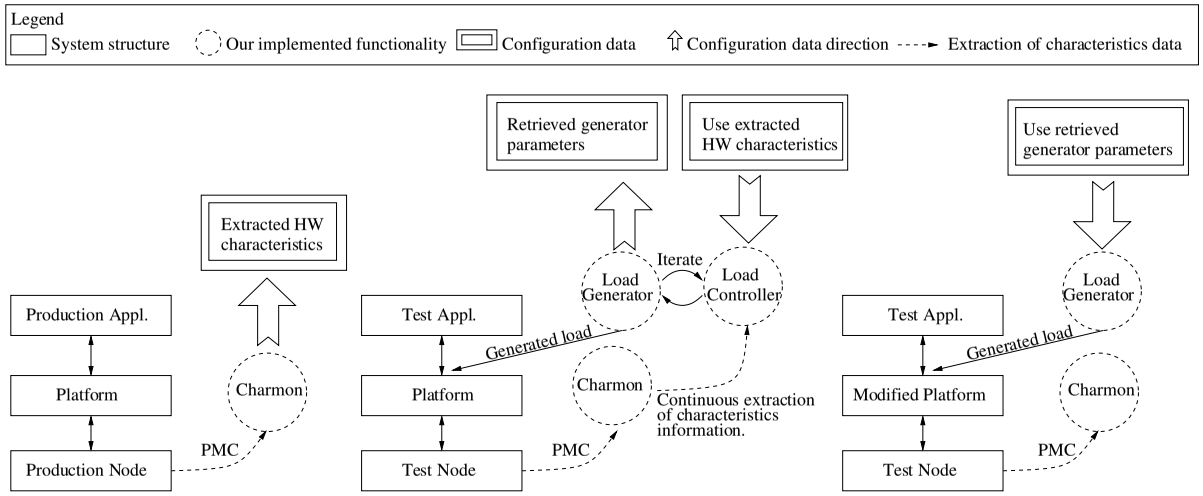


Figure 6.2: Three steps in the modeling process. (1) Gather execution characteristics from a production node; (2) Create a model of the production system using the original platform A; (3) Use the production node model for testing purposes when a new platform B is released.

6.2 System Model and Definitions

We have depicted our three-step modeling process in Figure 6.2. The first step is to sample the execution characteristics of the target system (Platform revision A) when it is running in a production environment. The second step is to create a hardware characteristics model on a test system to emulate the hardware usage of the target system. In the third and final step, we use the model on a test system together with a function-test suite to detect if there are any performance deviations for new software releases (Platform revision B). We use the Charmon tool to sample the execution characteristics, see Chapter 5.

The characteristics model maps hardware characteristics from the production node to a smaller test node. The main goal is to provide a more realistic execution environment for the test node similar to the production node environment. It is well-known in the industry, that functional test suites are good at testing the required functions, but they do not stress the system in the same way as the real production system. Running tests while stressing the system increases the ability to provoke congestion scenarios that may lead to the detection of hidden bugs.



a) Extract hardware characteristics from a production node execution environment.

b) Use the extracted characteristics information to create an execution model on a test node.

c) Use the execution model to simulate the production node execution environment on a test node.

Figure 6.3: Schematic description of characteristics extraction from a production system, how to create an execution model and how to simulate the production system execution environment on a test system.

6.2.1 The Modeling Method

Our method to mimic the execution characteristics of a production system requires an understanding of the resource usage of a production system. We sample the hardware resource usage, R, and application performance, x, for the target system. An $r \in R$ can be any measurable hardware resources such as cache misses, branch prediction unit statistics, floating point counters and similar, as described in Section 5.2.4. It is necessary for a system designer to determine and specify what r that will accurately describe the execution characteristics of the target system. The performance, x, is any application-specific metric that accurately describe the application performance, see Section 5.2.5. We describe the process to obtain the execution characteristics from a production application and mimicking it in a test environment by three steps. We assume that the same type of hardware is used in the production system and in the test system.

1. The modeling procedure is started by sampling $r \in R$ for process p in the production system running in its target environment resulting in a bounded series of measurements, $m_{r,p}$. See Figure 6.3a for an illustration.

2. The second step, as depicted in Figure 6.3b, is to create a simulated environment on a test node. We achieve a simulated environment by substituting the production application with a function test application together with a load generator mimicking the characteristics obtained in Step 1. We use the average value for each modeled metric, $\overline{m_{r,p}}$.

   (a) Run the test application, t, on the same platform as in Step 1, i.e. exactly the same software release of the platform.

   (b) Continuously evaluate $\delta_r = \overline{m_{r,p}} - m_{r,t}$ for all $r \in R$. Let a proportional–integral–derivative (PID) control algorithm [22] converge until δ_r is sufficiently small, as decided by a system engineer.

   (c) Retrieve metrics from the control algorithm.

In the process above we have sampled the execution characteristics from a production system and then mimicked a similar execution environment for a test application that performs a function-test. We can generate the same rate of cache misses without using the control algorithm by storing the internal load-generation parameters retrieved in Step 2c. This allows us to change the platform and then apply the same rate of cache misses. Investigating the ratio

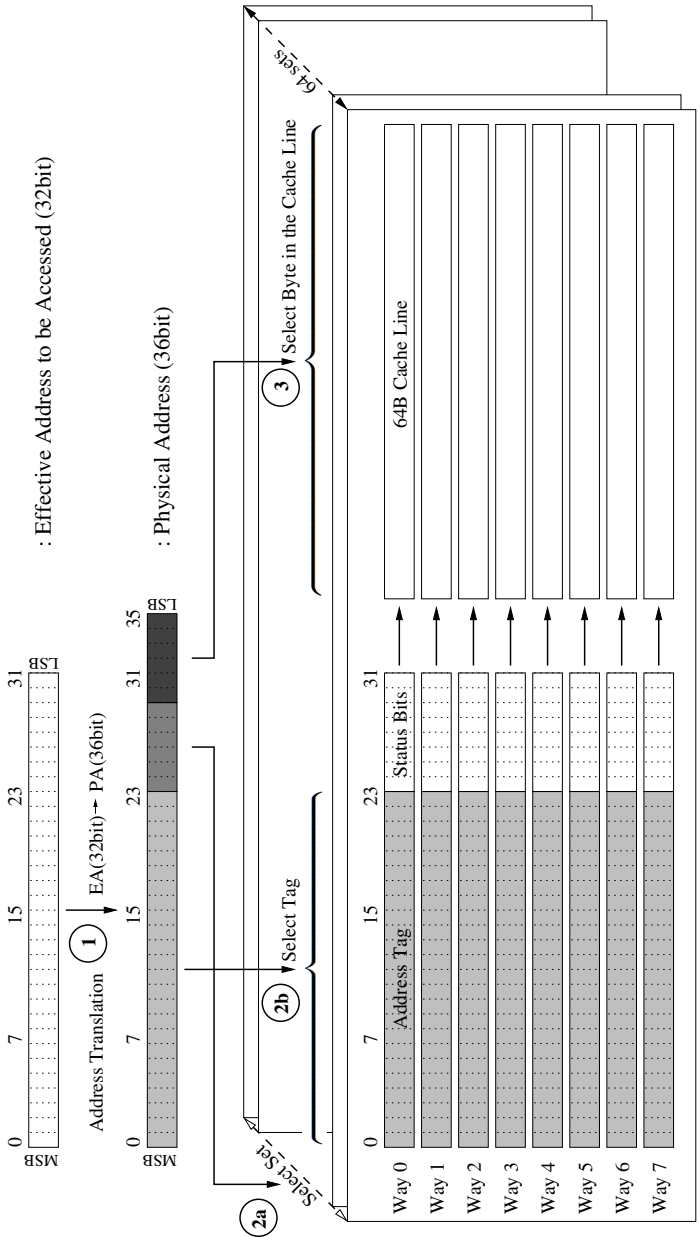


Figure 6.4: Address translation and physical address usage in the cache structure.

of misses allows us to detect changes in platform behavior. In the continued procedure below we can measure x for a different releases of the platform to get an indication of how it will perform when running the production system.

3. In this last step, see Figure 6.3c, we can detect changes to x for a modified platform without running the production application.
   (a) Start the modified platform together with the test application, t.
   (b) Generate the same resource usage R as obtained for t in step 2c.
   (c) Measure x for t and changes of x indicates a performance change.

The characteristics of the modified platform is different from the original one if x has changed in Step 3c. Low-level changes to the OS can influence the overall performance of the applications drastically if there are performance-related problems in cache handling or if the memory footprint has changed.

6.3 Implementation

The modeling process is generic and can use any hardware metric that is 1) possible to measure and 2) for which we can create an adaptable load generator mechanism. By load generator, we mean a specific part of the test application that can explicitly generate accesses to the resource we want to target. A system designer could investigate their target system with Charmon and decide which hardware metric has the highest effect on the performance. In the research leading up to this thesis, we have focused on modeling the cache usage. We selected to model the cache-usage mainly because the system we are investigating is memory-bound and depends heavily on cache and memory subsystem. The CPI-stack [94] in Section 5.4.2 further strengthen our opinion that our target application is memory-bound. Additionally, utilizing few metrics reduces the modeling complexity. The modeling application has been implemented using several PID-controllers. There is one PID-controller for each cache modeled property (L_1I-cache, L_1D-cache, and L_2D-cache). The modeling procedure is fully automatic and the model is typically created after 1–5 minutes.

6.3.1 Address Translation

The p4080 [102,103] Freescale processor has an 8-way set associative cache [274]. The address translation starts with a user-accessible 32-bit Effective Address (EA). When a user accesses an EA, the Memory Management Unit (MMU)

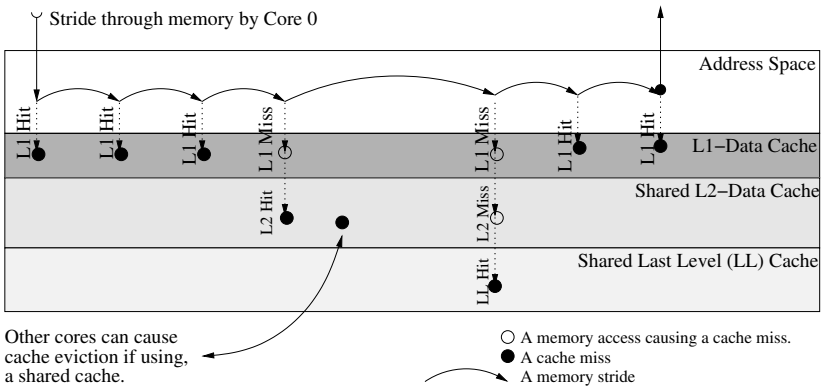


Figure 6.5: A memory stride through various data cache levels.

converts it to a 36-bit Physical Address (PA), step ① in Figure 6.4. There are eight address tags for each cache set, which each points to a corresponding 64B cache line. The selection of address tag ②a is done with bits PA[0:23] in parallel to the set selection ②b with bits PA[24:29]. A specific Byte within the cache line is selected using bits PA[30:35], see [102, 103]. We can use the same reasoning for Intel CPUs [140] although the section above describes a Freescale CPU. It is just to add or remove sets according to the CPU specifications.

6.3.2 The Load Controller

The load controller is designed to replicate the execution characteristics, i.e. hardware usage, of the modeled system. The model is synthesized by increasingly generating hardware-load to reach a user-supplied limit. When reaching the desired limit, a model is automatically created, which is possible to store for later use. The desired limits can be obtained through extracting production system characteristics, as in Figure 6.3a. Current measurements are obtained through the Charmon utility while automatically creating the model, see Figure 6.3b. When the model synthesis process has reached a stable state, it is possible to extract the controller parameters for offline storage. As shown in Figure 6.3c, the stored parameters can later be directly fed into the load controller. When the load controller receives the model parameters, it can start generating hardware load according to the model. The main benefit of using this procedure is that the model is synthesized once and then deployed widely without the need for remodeling.

Each hardware metric has its own PID-controller. It has proved difficult to avoid oscillations when implementing an autonomous feedback control loop that simultaneously considers multiple metrics. For each additional metric, the convergence is slower, and the oscillation tendencies increase. We are aware that there are more advanced techniques that may support multi-metric control, but due to lack of time, we have not investigated this issue further.

We have implemented a cache miss generator for L_1I-cache, L_1D-cache and L_2D-cache because they are substantial lost-cycle contributors, see Section 5.4.2. We have actively chosen a subset of the characteristics contributors because adding more metrics increase the complexity of the control algorithm. Increased complexity leads to oscillations and longer convergence time.

6.3.3 Generating Cache Misses

Cache memories has been around since 1968 with the introduction of IBM system 85 [190]. The main goal by using caches is to reduce the average access time when accessing memory. The cache usage is even more important with the systems of today since the CPU capacity increase quicker than the bandwidth towards the RAM. There are two type of cachees. The first cache-type aims to reduce the access time when fetching instructions. Instruction cache misses are typically an effect of code with frequent jumps or context switches. The other cache-type is data cache and contains a snapshot of the most used data that applications access and operate on. Code that access its data structures inefficiently generate extensive data cache congestion. All caches utilize a replacement strategy [274] that decides what to do when the cache is full and a new item needs to be placed in the cache. The PowerPC® p4080 and T4240 utilize a pseudo LRU replacement strategy.

We use the cache structure to generate a desired cache hit/miss ratio. It is easy to generate data cache misses by striding through memory as shown in Figure 6.5. In general, a short stride (jump) cause less cache misses while longer strides cause more cache congestion. It is possible to generate the desired cache hit/miss-ratio by varying the stride length and access pattern. A similar reasoning applies to the instruction cache.

Generating I-cache Misses

We use a large switch-case statement [275] to generate L_1I-cache misses, see Listing 6.1. The `bigswitch()` function is called with varying argument values for the switch-case index `n` depending on the desired outcome. We can

Listing 6.1: Generating L_1I-cache misses.

```
1  int bigswitch(int n) {
2    switch (n) {
3      case 1: n += 10;
4        break;
5      case 2: n += 11;
6        break;
7      case 3: n += 12;
8        break;
9      ...
10     case 99999: n += 50009;
11       break;
12     default: n+= 20;
13   }
14    return n;
15 }
```

foresee at least three possible outcomes depending on the hardware architecture used and the n function argument value. The first is when using a small n resulting in a short program jump. Jumping a short distance does not cause any instruction cache miss since the destination address is already in the cache. Jumping further by using a larger n results in a L_1I-cache miss because the cache needs to evict the current content and load the content for the destination address. The instruction pipeline also needs to be filled with new instructions from the destination address. Jumping even further, caused by a very large n, cause a L_2I-cache miss with similar results to a L_1I-cache.

The feedback controller varies the jump distance to produce the desired amount of cache misses. The main advantage of using this method is its simplicity while minimizing the data accesses, which would affect other feedback controllers working with data cache modeling.

Generating L_1D-cache and L_2D-cache Misses

We generate L_1D-cache and L_2D-cache misses by utilizing various memory strides [148]. Our implementation determines the memory address to access by different cache tags, sets, and offsets, see Figure 6.4, to reach the desired ratio of cache misses. The general approach to generating D-cache misses is straightforward. Striding within the number of sets/tags/offset supported by the L_1D-cache results in a cache hit. Striding outside the cache results in a L_1D-cache miss. Similar reasoning applies to generate L_2D-cache misses but

Listing 6.2: Function to generate D-cache hits and misses.

```
1  function gen_Dcache_misses(addr, accesses, ratL1, ratL2)
2  % PowerPC® p4080 hardware definitions.
3  define TAGS (8)
4  define SETS (64)
5  define CACHELINE_SIZE (64)
6  define L2_SIZE (128KB)
7  int s
8  int t
9  i = 0
10 int v
11 % Start with the last used address.
12 a = addr
13 forever
14    % Select a set.
15    s = 0
16    for s < SETS do
17       t = 0
18       for t < TAGS do
19          % Change offset to (maybe) generate L1miss.
20          if i % ratL1 then
21             % Move outside cache line.
22             offset += CACHELINE_SIZE
23          endif
24          % Change offset to (maybe) generate L2miss.
25          if i % ratL2 then
26             % Move outside the L2 cache.
27             offset += L2_SIZE
28          endif
29          % Access data.
30          v += a[offset+s+t];
31          % Quit if we have made enough accesses.
32          if i > accesses then
33             goto exit
34          endif
35          t++
36       endfor
37       s++
38    endfor
39 endfor
40 exit:
41 return a
```

it requires a larger working set. We have designed the cache miss generator function to generate cache misses with few side-effects, such as cache hits, instruction cache misses and so on.

We present the pseudo code for our data cache-miss generator algorithm in Listing 6.2. There are several input parameters to the `gen_Dcache_misses()` function. The first is `addr` which contains the base address of the allocated working set memory. The next is `accesses` which denotes the total number of accesses that should be made by the function before returning to the caller. The following arguments, `ratL1` and `ratL2`, denotes the ratio of memory accesses where we should try to generate a L_1D-cache orL_2D-cache miss. We have not specifically optimized the function but rather focused on making it generic to support various hardware architectures. Our function tries to excercise the entire cache, i.e., all sets, ways, and cache lines. Lines ③ – ⑥ define our target. The PowerPC® p4080 L_1D-cache architecture is implemented with 64 sets where each set has 8-ways. The L_1D-cache has a total size of 32 KB divided into 64 Byte cache lines. The algorithm begins with an eternal loop at line ⑫. It starts by selecting the set ⑭ and the tag ⑯. If we should generate a L_1D-cache miss, we make sure that we use an offset that is outside the cache line. We use a simpler approach for generating L_2D-cache misses and adds the size of the L_2D-cache to the offset. The memory access at line ㉗ triggers the actual cache hit or miss. We exit the function at ㉙ if we have reached the total number of accesses. We have also implemented various controls in the `gen_Dcache_misses()` function to ensure that we do not access memory outside the allocated working set memory. We have omitted this code to improve readability in Listing 6.2.

A side effect of striding through a multi-level cache is that misses at a certain cache-level will most likely also generate misses for all preceding cache-levels. Also, note that explicitly generating cache misses are difficult. It is not an exact science because all processes running on the system affects the system-wide cache and will affect the cache miss generator function. Also, generating a L_2D-cache miss will trigger a L_1D-cache miss. These are contributing reasons for connecting the PID controller to the cache miss generating function. We want to define the cache miss rate or ratio and then let the PID controller determine the actual function-argument values. We control the cache hit-miss ratio by giving the total number of accesses as well as the desired ratio for cache misses. The PID control algorithm, see Section 6.3.2, provides the parameters given to the function.

6.4 Experiments Using Execution Characteristics Modeling

We will describe the outcome from four experiments in the following sections. We begin our experiments with Section 6.4.1 where we verify that our execution characteristics method. In Section 6.4.2 we compare the hardware characteristics of a production node with characteristics from a test node, both with and without a characteristics model. In Section 6.4.3, we present an example of finding a performance related bug in the early phases of the development process. In our last experiment, see Section 6.4.4, we use our modeling mechanism to derive an estimate of the performance impact when switching from a legacy-OS to Linux.

6.4.1 Running a Test Application With The Load Generator

The goal of this experiment was to verify that it was possible to replicate a synthetic execution environment while running a standard packet processing test application. The test application sends messages between two applications running on the same system. Each application performs some basic packet processing before sending the packet back to the other communication part.

Setup We had already sampled the execution environment of a production system, see Section 5.2.4, so we knew the system cache usage. We decided to set the following desired values: L_1I-cache miss ratio 0.74%, L_1D-cache miss ratio 3.3% and various L_2D-cache miss ratio [Core 0=22%, Core 1=14%, 2=15%, ..., 7=20%].

Execution The test application and the load generator was started at the beginning of the test run, see Figure 6.6. The feedback controller was aborted after approx. 35 minutes, returning the load on the node to its normal state with a load provided only by the test application.

Evaluation We use one PID-controller per core and hardware metric. The test application has an initial characteristic (to the left) that differs from the final characteristics (to the right) that corresponds to the desired values as given above, see Figure 6.6 Each of the eight graphs in Figure 6.6 shows CPI and cache misses when running a signaling application sending signals between two processes located on the same core. The effect of an increased cache miss

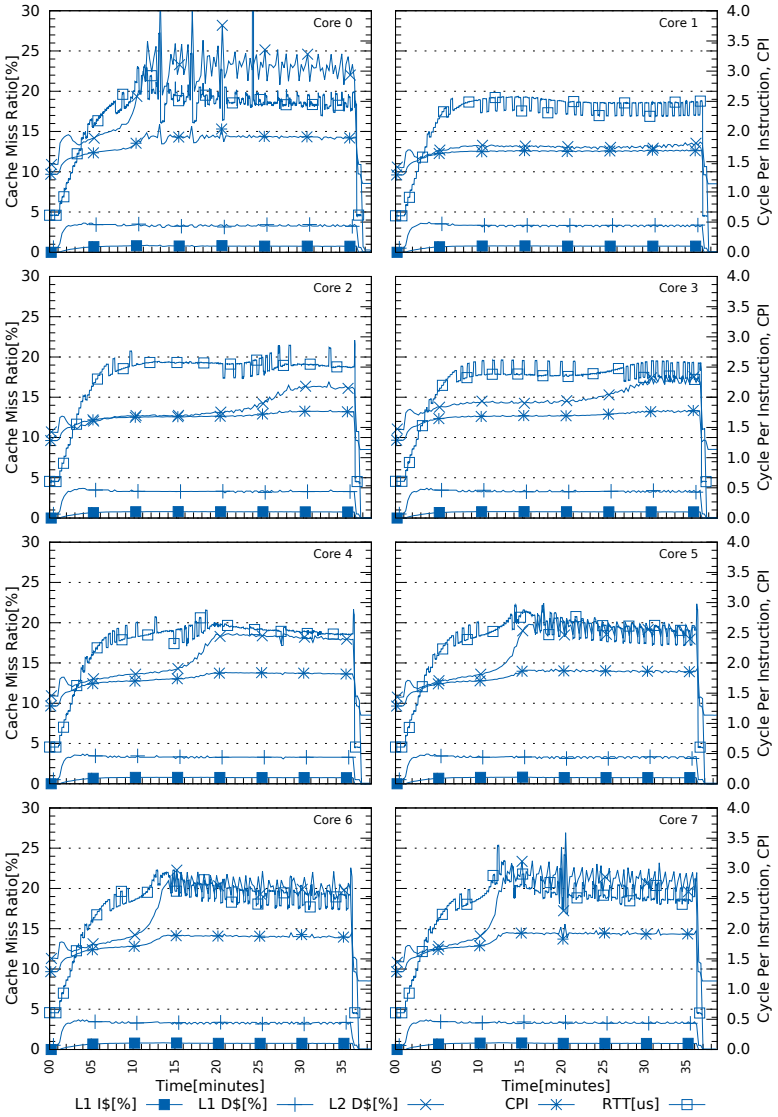


Figure 6.6: CPI and cache miss ratio when bouncing signals.

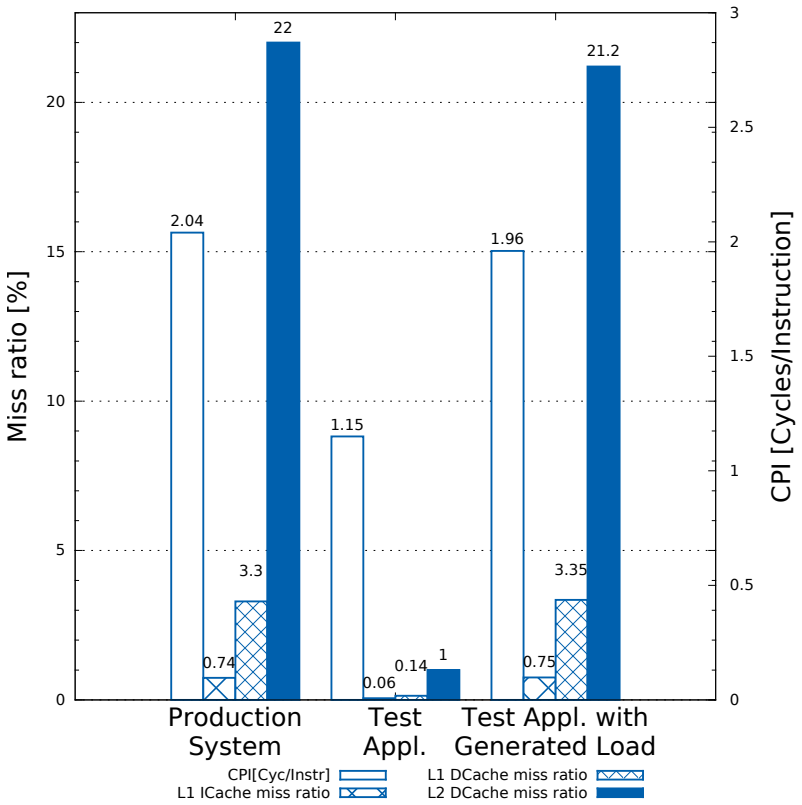
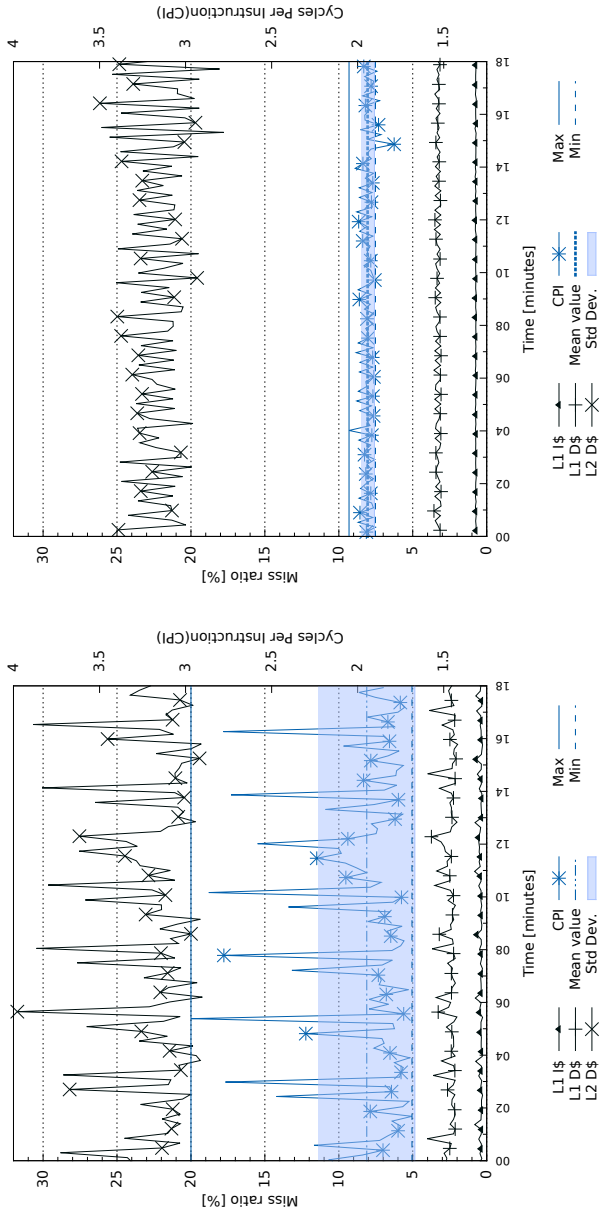


Figure 6.7: CPI and cache miss ratio for core 6.

ratio is an increasing CPI, which affects the performance of other processes executing on the same core. We conclude that it is possible to replicate the load of this production system on a test system. We need to perform additional tests to evaluate how generic the results are and if we can replicate the load of any system.

6.4.2 Production vs. Modeled Execution Characteristics

The goal of this experiment was to compare the original and replicated execution characteristics. We looked closer at core 6, which runs the main application



(a) Production system.

(b) Modeled test system.

Figure 6.8: A comparison of hardware characteristics for a production system and a modeled test system.

functionality on the production system. The other cores handle various support services related to other types of traffic processing, but we have omitted them due to space constraints.

Setup We use the same test setup as in Section 6.4.1 but configure core 6 with the following desired values: L_1I-cache miss ratio 0.74%, L_1D-cache miss ratio 3.3% and various L_2D-cache miss ratio [Core 0=22% and then Core 1=14%, 2=15%, ..., 6=22%, 7=20%].

Execution We use our modeling method to create a hardware characteristics model that converge the cache usage to the desired level, see Figure 6.6. All cores in the CPU are modeled during the experiment and shows similar characteristics as the production node. We have implemented three Test Cases (TC). TC1 shows the original samples from the production system. TC2 shows the execution characteristics when running only the function-test suite on the test system. TC3 shows the function-test suite together with the load generator, i.e. on a replicated execution environment.

Evaluation The execution characteristics, i.e. cache usage, for the production system (TC1) is shown in the leftmost stacks in Figure 6.7. It is easy to see that the application is a heavy user of cache-memory. This is in itself an important conclusion for the engineering department when they investigate the possibility to make performance improvements. The cache usage for TC2 is shown in the middle of the figure. Comparing the cache usage between TC1 and TC2 shows an important shortcoming with the function-tests. The memory footprint and memory usage are far too small when running TC2. Even though the function-test suite exercise the functionality of the system it does not generate remotely enough stress on the system. In TC3 we add load-replication to the function-test suite. The execution characteristics is almost identical to the production system.

A closer investigation of the measurements in Figure 6.7 reveals that the cache usage of the production system in TC1 jitters over time, Figure 6.8a. The jitter in TC1 is much higher than in TC3, see Figure 6.8b. Such jitter has a two-fold meaning. On the positive side, the model system is within the limits of the production system execution characteristics. The mean cache usage ratio is similar to the production system. There may also be downsides to the jitter difference. The explicit effects are difficult to predict, but although the mean cache-usage is similar, memory access bursts tend to congest memory subsystems. Memory congestions will have negative effects on the memory access time.

Core	Original Release [ms]	Modified Release [ms]	Comparison Increase [ms]	[%]
0		Omitted from the simulation.		
1	0.5934	0.5982	0.0048	0.81%
2	0.5935	0.5999	0.0064	1.08%
3	0.6017	0.6068	0.0051	0.85%
4	0.6022	0.6057	0.0035	0.59%
5	0.6022	0.6058	0.0036	0.60%
6	0.6025	0.6060	0.0036	0.59%
7	0.6015	0.6061	0.0046	0.76%
Average				**0.75%**

Table 6.1: Mean message RTT for a test application w/ and w/o a particular software change while running on a test system. The data was sampled on a second-level basis during several hours.

Core	Original Release [ms]	Modified Release [ms]	Comparison Increase [ms]	[%]
0		Omitted from the simulation.		
1	2.0238	2.2492	0.2254	11.14%
2	2.0937	2.3423	0.2487	11.88%
3	1.9284	2.1527	0.2243	11.63%
4	2.0195	2.2548	0.2353	11.65%
5	1.9945	2.1778	0.1832	9.19%
6	2.1637	2.4100	0.2463	11.38%
7	1.9952	2.1704	0.1752	8.78%
Average				**10.81%**

Table 6.2: Mean message RTT for a test application w/ and w/o a particular software change while simultaneously running a load generator on a test system. The data was sampled on a second-level basis during several hours.

Remarks It is also possible to detect periodically reoccurring events by looking for peaks in graphs. Our measurements imply that the production node executes a memory intensive task roughly every second minute because there are repeating CPI peaks occurring with this interval, see Figure 6.8a.

	Original Release					Modified Release					Comparison				
Core	**Instr.**	**Cycls.**	**IPC**	**CPI**	**CPU**	**Instr.**	**Cycls.**	**IPC**	**CPI**	**CPU**	**Instr.**	**Cycls.**	**IPC**	**CPI**	**CPU**
0	Omitted from the simulation.					Omitted from the simulation.					Omitted from the simulation.				
1	1111M	1498M	0,74	1,35	63,7%	1106M	1499M	0,74	1,36	69,3%	-0,49%	0,09%	-0,49%	0,49%	8,79%
2	1040M	1498M	0,70	1,44	63,8%	961M	1502M	0,64	1,56	69,2%	-7,54%	0,26%	-7,96%	8,65%	8,46%
3	1043M	1498M	0,70	1,44	62,8%	966M	1500M	0,64	1,55	68,7%	-7,38%	0,15%	-7,51%	8,12%	9,39%
4	1046M	1498M	0,70	1,44	63,3%	967M	1499M	0,65	1,55	68,6%	-7,50%	0,05%	-7,37%	7,95%	8,37%
5	1044M	1498M	0,70	1,44	64,0%	963M	1502M	0,64	1,56	69,1%	-7,74%	0,29%	-7,83%	8,49%	7,97%
6	1042M	1498M	0,70	1,44	63,3%	959M	1496M	0,64	1,56	67,9%	-8,04%	-0,16%	-7,89%	8,57%	7,27%
7	1044M	1501M	0,70	1,44	63,3%	966M	1500M	0,64	1,55	68,5%	-7,46%	-0,02%	-7,44%	8,04%	8,21%
$\overline{x}$	1053M	1498M	0,70	1,42	63,5%	984M	1500M	0,66	1,53	68,8%	-6,6%	0,1%	-6,6%	7,2%	8,4%

Table 6.3: Comparison of two releases of a telecommunication system running in a production environment. The modified system contains a hardware errata fix related to cache handling. The characteristics data was sampled each second during several hours at a customer site. The table shows the average value.

6.4.3 System Performance Measurement

The goal of this experiment was to verify if it is easier to detect performance-related problems (bugs) when running a function-test suite in an execution environment that is similar to the production environment.

Setup We use the same test setup as in Section 6.4.1.

Execution TC1 was running on the original production system with no bug-fix. In TC2 we run the function-test suite to verify that the bug-fix does not affect the function and performance of the system. In TC3 we run the function-test suite in a load-replicated execution environment. Both TC2 and TC3 contain the same bug fix related to a CPU errata. The errata was at the time of implementation very difficult to understand and no engineer could forsee the effects it could have on system performance. TC4 denotes the actual outcome of running the production system including the bug-fix.

Evaluation Executing TC2 showed a minor performance degradation where the message Round-Trip-Time (RTT) increased by 0.75%, as shown in Table 6.1. When running TC2 the message RTT increased to 10.8% with the bug-fix compared to without it, see Table 6.2. Such a considerable performance degradation clearly indicated that the bug-fix could cause a performance problem for the production system. To validate our method we delivered the bug-fix for formal performance verification. The average CPU load increased by 8.4%, see Table 6.3, when the bug-fix was tested together with the production system.

Remarks It is difficult to make a direct comparison between the message RTT on the test system, t_s, and the CPU load on the production system. However, we estimate that $t_s = t_p + t_t$ where t_p is the time it takes to process a message and t_t is the time in transit between nodes. In the system we are investigating $t_t \ll t_p$ since the available bandwidth is high and the communication path is relatively short. Thus, we assume that t_s is proportional to t_p, and that CPU load is a major contributor to the message RTT. A higher CPU-load results in an increased message RTT due to longer processing time.

6.4.4 Performance Prediction When Switching OS

The goal of this experiment [9] was to predict the application performance impact when switching the OS from Enea OSE [273] to Linux. Many industrial

systems still run on a legacy-OS that is expensive to maintain. The maintenance cost drives the demand to migrate such systems to Linux, which supports many more architectures and has a well-maintained execution environment and tools. It is useful to get a performance indication before deciding to port the complete system. In a practical sense, this experiment evaluates if a performance test is more accurate when using a synthesized hardware characteristics model together with a test suite compared to running the performance test with the test suite only.

Setup We use a similar test setup as in Section 6.4.1 and utilize two interconnected PowerPC® p4080 [102] CPU boards for Enea OSE and two additional boards running Linux. There were no significant background activity for either OS.

Execution In the first scenario, we run the function-test suite on two Enea OSE and two Linux system (Scenario S1 in Table 6.4). In our second test scenario), we run the function test-suite together with our load generator (Scenario S2 in Table 6.4).

Evaluation The function test run, S1, results in an 183% increase in message RTT when using Linux compared to Enea OSE. However, in the load-replication Linux runs we obtained a 14% message RTT degradation, S2. Later, when we had ported much of the system and its production applications to Linux, it was possible to verify that the average message RTT performance degradation was 15%, which is very close to our prediction.

Remarks We have revealed some clues to the radical drop in performance ($+183\%$ message RTT increase) when interpreting the results in S1. Starting from the top of Table 6.4 the first metric that stands out is a decrease of 1.2 percentage point (pp) for L_1I-cache hit ratio. It may, at first glance, look like a negligible change but we know from experience that even small decreases in the cache hit ratio affects the performance. The next metric is L_1D-cache hit ratio, which has decreased with 0.6pp. Such a hit rate reduction gives the first hint of a larger working set for Linux. The platform and test application remain the same for both OSes, which suggests that Linux by itself causes the increased working set size. Investigating the Linux source code shows that it is much more complicated than the legacy-OS. We attribute much of the complexity to the generic and modular design of Linux. We can observe a similar increase for the shared L_2-cache where the L_1D-cache and L_1I-cache spillover affects

	S1: Only Test Prog.			S2: Test Prog. w/ Loadgen			
Metric	**Legacy**	**Linux**	**Increase**	**Legacy**	**Linux**	**Increase**	**Comments**
Signal RTT	6us	17us	11us(+183%)	25us	29us	4us(+14%)	Big difference w/ and w/o Loadgen.
L_1I-cache hit ratio	100%	98.8%	-1.2pp	99.3%	98.8%	-0.5pp	Legacy fits in the cache.
L_1D-cache hit ratio	99.8%	99.2%	0.6pp	98.2%	98.5%	0.3pp	Similar cache usage for both OSes.
L_2-cache hit ratio	–	100%	–	85.0%	90.0%	5pp	L_2 is not used for original legacy but
L_2-cache accesses	0	16M	16M	6M	13.5M	6.5M(+108%)	modeled with Loadgen.
ITLB 4 KB reloc.	0	750k	750k	0	450k	450k	The Linux code base is much larger.
DTLB 4 KB reloc.	0	750k	750k	0	550k	550k	The Legacy system uses Variable Size Pages
VSP-DTLB reloc.	50k	0	-50k	20k	0	-20k	to reduce-TLB pressure.
L_2TLB reloc.	0	0	0	0	110k	110k	The Linux data set is larger for S2.
Branch hit ratio	100%	84%	-16pp	94%	85%	9pp	Smaller code base for the legacy system
Branch hit rate	200M	120M	-80M(-40%)	80M	90M	10M(12.5%)	results in good branch prediction.
Interrupts	0	230k	+230k	0	250k	250k	The network driver implementation differs.

Table 6.4: Scenario S1 shows a messaging test application. S2 show the same test application running with an additional hardware load generator. (pp=percentage point).

the number of accesses. The L_2-cache hit ratio is negligible for the legacy OS because there are too few accesses.

The TLB is closely related to the cache. The number of ITLB relocations have increased from $0 \rightarrow 750k$. We conclude that the total size of Linux is substantially larger than OSE. If we investigate the DTLB we can observe that Linux does not use Transparent Huge-Pages (THP) [179] because the number of DTLB relocations has increased from $0 \rightarrow 750k$. At the same time, the usage of VSP-DTLB is reduced from $50k \rightarrow 0$.

The execution flow is also affected by the increased code base and leads to a branch hit ratio drop ($100 \rightarrow 84$). The last counter in this experiment is the number of external interrupts. It seems likely that the Linux network driver implementation uses much more interrupts than the legacy OS. A polling driver would probably reduce the number of interrupts for a messaging application such as ours.

The message RTT has increased for both the legacy OS ($6us \rightarrow 24us$) and Linux ($17us \rightarrow 29us$) in scenario S2. The difference between the OSes is much smaller, only $+14\%$. We can still see that the instruction flow is slower for Linux, L_1I-cache ($99.3\% \rightarrow 98.8\%$) and ITLB relocations ($0 \rightarrow 450k$). For the data flow, we can observe better cache hit ratio for Linux ($98.2\% \rightarrow 98.5\%$) but there are still many more 4 KB DTLB relocations ($0 \rightarrow 550k$) due to missing VSP/THP support. The L_2-cache usage shows a greater number of accesses by Linux but also with higher hit ratio. Branch hit ratio still shows a performance impact for Linux, and there are still a great number of interrupts.

Conclusions Based on the data collected, we believe that there are strong indications that we can attribute the most of the performance degradation to 1) a larger code and data working set; 2) the number of interrupts handled by the network driver. One explanation for the significant difference in message RTT between S1 and S2 is the greater size of the complete code and data set in Linux. Even a small size increase of the working set cause many more cache-misses.

To overcome the performance degradation we suggest the following actions: 1) Enable VSP/THP or similar functionality to utilize large memory pages, which will reduce the pressure on 4 KB TLB, 2) Use interrupt coalescing or polling to reduce the number of handled interrupts within the Linux network driver, and 3) Investigate why the total code and data set is bigger for Linux.

6.5 Related Work

Performance monitoring is vital in today's system development, and we believe that it will become even more desirable when developing future systems because of their increasing complexity. The amount of software implemented in modeling languages increase to aid the designer developing complex business logic. Compiling the models to low-level code introduce an additional complicating factor compared to the implementation in legacy systems, typically implemented in C. An ever-increasing complexity in CPUs and memory subsystems require system engineers to put much effort into investigating multiple performance bottlenecks. To do so, system developers need much information on hardware resource usage, and system monitoring is one important building stone.

Bell and John [20] describe a method to model an application by synthesizing vital metrics. The metrics are used to create a representative test application automatically with similar characteristics to the original one. Starting with the synthesizing procedure, we use a feedback control loop to model the system while Bell and John [20] use statistical simulation with instruction traces, described by Nussbaum and Smith [223]. Bell and John state that the synthesis procedure is semi-automatic, and an average of ten passes with some manual intervention is needed to tune the synthesis parameters. As a comparison, our feedback control algorithm allows the synthesis procedure to converge with no user interaction. Additionally, our model is described by the resulting configuration parameters, which are fed to the generic method. For Bell and John model derivation is done at compile time thus requiring a recompilation when altering the configuration. Another difference between our approaches is that we use a signaling application to detect any performance changes between releases while Bell and John use IPC.

Joshi et al. [164] have formulated a concept called performance cloning, i.e. load synthesis, that can be used to synthesize characteristics from a proprietary application and create a model that mimics a similar behavior. In effect Joshi et al. implements a similar methodology as Bell and John in [20] but have refined the memory and branching model to be hardware agnostic.

Doucette and Fedorova [71] implements a similar functionality to ours when generating cache misses to determine application sensitiveness for different architectures. If an application is sensitive for accesses to a particular hardware resource and another architecture has a different amount of that resource, the application performance is related to the hardware upon which it runs. By using their approach, it is possible to estimate the suitability of a new hardware

platform for a particular application. The load configuration is static compared to our automatic mechanism.

Similar to the research by Eklov et al. [76,77] and Tang et al. [281] our load generator "steals" hardware resources from other applications ny starving them. In contrast to their method, we use a cache miss generator to mimic a certain execution environment while Eklov et al. and Tang et al. determine the cache and memory bandwidth demand for the application. Our work related to load synthesis is concentrated on core-private caches instead of the shared caches targeted by Eklov et al. and Tang et al.

Alameldeen et al. [6] investigate server platforms and concludes that it is difficult to synthesize and model production systems. We certainly agree to that conclusion. They mimic the desired characteristics by using a manually tailored workload suite in their work. In contrast, we use an automatic feedback-based load generator to achieve an approximation of the production application.

There are other attempts to measure the system performance. Podzimek2013 et al. [233] has created a benchmarking suite where multiple applications simultaneously runs on a test system. The effect of consolidating multiple application is similar to ours, i.e. shared resource such as caches cause the applications to affect the performance of each other. The authors have expanded their work and created a tool called *Showstopper* [234].

6.6 Summary

We have answered Q2 (Section 3.2.2) with theories, experiments and through our work presented in this chapter. Our Paper B describe the load synthesis method, which is further expanded in the technical reports M and N.

We have build upon our earlier research and used the Charmon resource and performance monitoring tool. We connected a load synthesis application to the Charmon supplied API. A feedback control loop continuously obtained the current resource usage measured by Charmon through the PMU. The feedback loop can then control the hardware resource usage of the load-replication application and automatically reach the desired resource usage. The load-replication application automatically reaches the same execution characteristics as the model of the production large-scale industrial system.

We have used the execution characteristics model to mimic the hardware resource usage of the production system on smaller and cheaper test nodes. Having a realistic test environment during the design phase makes it possible to move performance testing from late in the development cycle to the much

earlier software design phase. Detecting performance-related problems early in the software design phase reduce the cost of debugging performance-related problems. We have deployed the proposed solution in a telecommunication development organization, and the company evaluates it for production usage.

Our work synthesizes a resource usage model and mimics static load of a system. Static load synthesis is suitable for our type of system where the load is stable over time, i.e., the application typically executes with a similar load for the majority of its execution time. However, we recognize the need for synthesizing dynamic behavior and therefore see this as an upcoming future work. We would also like to investigate further and improve the feedback controller so that it converges faster.

Don't cry because it's over, smile because it happened.[1]

— Theodor "Dr. Seuss" Geisel

[1] This quote summarizes my own experience when writing this thesis. It has been great fun and at the same time very challenging. Sometimes it was so rewarding that I couldn't stop writing. But as always, things have to come to an end.

7 Automatic Message Compression

This section corresponds to research question Q3 (Section 3.2.3), which we have addressed in papers C (H).

> *How can the communication performance of a telecommunication system be improved through message compression while retaining the system load within pre-defined limits?*

THE massive deployment of Information Communication Technology devices drives [70] the demand for a radically increased mobile communication bandwidth. End-user demands are passed on to the operators that define requirements for a radical capacity increase [21, 84]. The telecommunication industry tackles the requirements in various ways and one of them is to increase the communication capacity.

The chapter starts with Section 7.1 that gives an introduction to the communication problem. We theorize the problem in Section 7.2 where we also describe one way to improve the messaging performance. Our implementation is described in Section 7.3 and we show the results form experiments in Section 7.4. We relate our work to other researchers in Section 7.5 and estimate possible future work. We conclude the chapter with our comments on messaging performance and compression in Section 7.6.

7.1 Introduction

There is a great demand for high-performance communication in today's industry, both internally and between systems. We are in the midst of the evolution into multicore CPUs which causes the computational capacity [120, 129] to grow quicker than the available communication bandwidth [218]. Large-scale industrial systems [122] have additional problem areas, for example, the large and very expensive legacy of already installed systems. It is not always economically feasible to replace current systems with newer ones just because they can provide higher performance. Industrial requirements explicitly state that older systems must coexist with more modern ones, which pose difficulties when requiring substantial performance improvements.

7.1.1 Communication Performance Problem

The general opinion in our design organisation that the control plane communication performance was sub-optimal. We started our performance evaluation by using Charmon, as outlined in Chapter 5, on the telecommunication system we were investigating. Two things were clear when we evaluated the execution characteristics; 1) The network congestion was high; 2) The CPU load was varying between moderate ($\sim 25\%$) and high (100%) depending on the execution pattern of the services sharing the same hardware resources. Network communication is a well-known bottleneck [299] when computational capacity grows quicker than the bandwidth.

From these data, we assumed that it would be possible to increase the bandwidth under certain scenarios by compressing messages [121, 165, 217]. Message compression require CPU capacity. The node may have CPU capacity to spare depending on the current operational scenario. Although, this is not always the case so the compression method must consider the system CPU-load before starting to compress messages. Furthermore, the data transferred by the system changes over time and depends on the deployment situation, making it is difficult to decide manually what compression algorithm to use.

Manual system configuration is usually frowned upon for large-scale systems because it introduces additional costly manual labor. The same applies for manual configuration of the messaging subsystem with changing message content. Industrial systems implicitly require automatically configurable messaging systems, removing the need for off-line decisions. Communication mechanisms must be able to handle different scenarios and a dynamic environment.

Message compression implies that we are trading CPU resources for a reduction in message size, which in turn leads to quicker message transit time. A message processing node in our industrial environment runs several essential services that must not have their execution disrupted. Since unrestricted message compression may overload the CPU, there must be an overload protection for message compression.

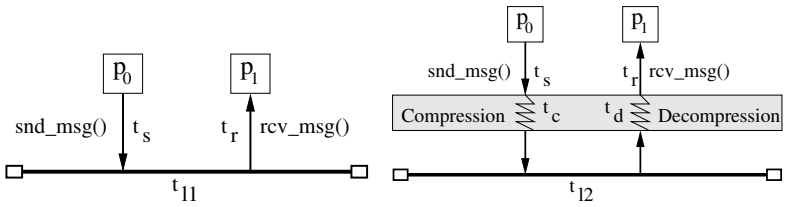
7.1.2 Improving the Communication Performance

We have designed and implemented an automatic compression method to improve the messaging performance and resolve the issues described in Section 7.1.1. The main features of our method are:

- A novel mechanism that automatically and transparently evaluate the messaging performance of different compression algorithms and select the most efficient one for the current communication stream.
- An automatic mechanism that detects both network congestion and message content changes and continuously selects the best compression algorithm.
- A selection mechanism that simultaneously can handle multiple communication streams. Each stream will have its own environment providing the possibility to have different compression algorithms for different streams.
- A Proportional Integrative Derivative (PID) feedback controller that evaluate the CPU usage and dynamically adapt the ratio of compressed and uncompressed messages, keeping the CPU usage to a specified limit.

We have implemented the method and tested it on a large telecommunication system. We have used a realistic test setup to show that our implementation improves the performance improvements to our system.

- We have used data extracted from a production system in our test environment.
- We have integrated 9 different compression algorithms with an additional 2 configuration variations in our test environment.



(a) Without message compression. (b) With message compression.

Figure 7.1: Schematic description of generic messaging system.

7.2 System Model and Definitions

In many cases it is reasonable to assume that the optimal compression algorithm for a specific data set is known. However, in some scenarios this is not the case. The message content may be unknown to the programmer, which makes it difficult to select an appropriate compression algorithm. To find the most suitable compression algorithm there are two approaches. The first is to manually, in an offline manner, select the compression algorithm that the operator think is the best one. The second method, the one we suggest in this paper, is to use a selection mechanism to automatically evaluate and select the compression algorithm that is most suitable for the current message stream. For systems with no compression this is a straight-forward procedure, which is depicted in Figure 7.1a.

> Process p_0 in Figure 7.1a communicates with p_1 running on a processor located on another node. Process p_0 and p_1 use legacy functionally without message compression. To increase messaging performance it is possible to use message compression, such as illustrated in Figure 7.1b.

With our novel solution, we can transparently improve messaging performance by adding message compression to the legacy application programming interface (API). The API already contains functions that handle message communication. We modify the existing API and add selective message compression functionality inside the `snd_msg()` function and corresponding decompression functionality in `rcv_msg()`. It is a substantial saving for any industrial software to improve performance without any API change. API changes have a drastic impact on context and dependencies and often requires substantial investments in modifying system components. System rebuild, regression testing

etc. will also cause an increased cost. Our solution is to update the existing API to monitor each communication instance. The communication API can transparently utilize the most suitable compression algorithm for different instances and types of message content.

> The message scenario is initiated by sending messages using all implemented compression methods in a round-robin fashion [15, Ch 7.7]. Compression and network statistics are stored for this communication instance. After an initial evaluation period one compression algorithm will be selected if it is predicted to give the lowest message round trip time.

The total time (t_{t1}) in Equation 7.1 is the time inside the send function (t_s), to travel on the link (t_{l1}) and inside the receive function (t_r). See Figure 7.1a.

$$t_{t1} = t_s + t_{l1} + t_r \tag{7.1}$$

As illustrated in Figure 7.1b our approach is to add compression (t_c) and decompression (t_d), see Equation 7.2.

$$t_{t2} = t_c + t_s + t_{l2} + t_r + t_d \tag{7.2}$$

Assuming that t_s and t_r are equal in both scenarios since the API send and receive call does not differ in functionality, we can conclude that message compression is beneficial if the time it takes to compress and decompress messages is lower than reduced link time, Equation 7.3.

$$t_c + t_{l2} + t_d < t_{l1} \tag{7.3}$$

The performance of each compression algorithm can be derived by gathering cumulative statistics. Such statistical data can then be used to predict future message compression, send, link, receive and decompression time for each compression algorithm. The prediction method selects the compression algorithm that gives the lowest message time (t_t). The selected algorithm is used for the majority of messages until a different algorithm outperforms the current one. To make sure that it is possible to detect a network- or message content change some messages are sent using other compression algorithms. The idea is to gather statistical data for all implemented algorithms, not only the one that is selected.

> Assume that one compression algorithm, F, is selected for a message stream. F will then be used to compress most of the transmitted messages. The rest of the algorithms will continuously be

evaluated by compressing a very small number of messages. If the content of the message stream changes, the algorithm evaluation may favour a different algorithm than F, which is subsequently chosen as the best compression algorithm.

If the CPU load increases beyond a limit a PID-feedback controller will reduce the message compression time-quota, causing messages to be sent uncompressed.

Process p_0 executes on the same CPU as p_1 sharing a common hardware. A process p_1 is ordered to perform CPU demanding calculations. The CPU usage throttling mechanism reduces the compression quota for p_0 resulting in fewer compressed messages. Reducing the resource usage allocated for message compression increase the available CPU resources for service S.

We think that it is difficult to manually, in an offline manner, consider all scenarios above selecting the most suitable compression algorithm. Our approach, using an automatic mechanism, greatly simplifies this task and provides the flexibility needed for a changing message stream while at the same time being able to provide CPU resources for other co-located services.

7.2.1 Definitions

A *host* in our communication system will typically communicate by first receiving a message and then spend some time processing it producing a result that is immediately sent onwards to another host. A host is usually, for cost-savings reasons, also configured to handle additional concurrent tasks, such as statistics measurements, user interaction, database management and/or other similar actions.

Definition 13 We define a *host* in our communication system to be a computer that performs a system vital task and that communicates with other hosts in the system.

We also need to define how we measure the communication performance of a host. The *messaging performance* varies depending on properties such as link speed, host distance in the network. In the case of message compression, additional properties can be added such as compression/decompression rate and compression ratio.

Definition 14 We define *messaging performance* as the time between the sender host calls the `snd_msg()` call together with the link time until the receiver obtains the message, i.e. $t_s + t_{l1} + t_r$ in Figure 7.1a.

We have also defined another way to measure the communication performance of our target system. The message processing performance includes the complete time until the sending node receives back a message from the host it is communicating with.

Definition 15 The concept *message processing performance* is a measurement of the ability to process messages per time interval. In our study, we measure this as the message round-trip time (RTT) between two interconnected nodes.

We need a way to differentiate the performance between various compression algorithms. We have opted to determine the compression algorithm suitability according to their ability to provide a minimum message RTT.

Definition 16 We define the *best compression algorithm* to be the one that gives the lowest message RTT.

To measure the performance of each compression algorithm we define three key properties: compression time, decompression time, and compression ratio. A compression ratio $H \leq 1$ indicates that no compression is achieved or even worse that the resulting message is larger than the uncompressed. Achieving a higher compression ratio $H > 1$ results in a smaller compressed message compared to the original one.

Definition 17 The *compression time*, $t_c = s/t_{cr}$ is the time to compress a particular message of size s. The compression rate, t_{cr}, is the number of bytes compressed per second, [B/s].

Definition 18 The *decompression time*, $t_d = s/t_{dr}$ is the time to decompress a particular message of size s. The decompression rate, t_{dr}, is the number of decompressed bytes per second, [B/s].

Definition 19 The *compression ratio* of a particular algorithm is $H = s_u/s_c$, where s_u is the size of the uncompressed message and s_c is the corresponding size when being compressed.

Definition 20 The *transmission time*, t_t, is the sum of message *compression time*, t_c, *send time*, t_s, one way of the *message round-trip time*, t_{rtt}, and *decompression time*, t_d, such that $t_t = t_c + t_s + \frac{t_{rtt}}{2} + t_d$.

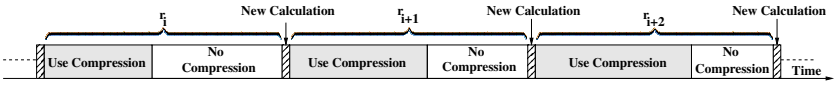


Figure 7.2: At the start of each round, r_i, messages are compressed. When the compression time budget is exhausted, messages are transmitted uncompressed to reduce CPU load. In this example the ratio compressed/uncompressed is increasing slowly.

The transmission time denotes the complete time it takes for a message from one node to another. It is difficult to measure the very short time a message spends on the link because the sender and receiver must have synchronized clocks. It is easier to measure the time it takes back and forth since we can measure the time it takes for a probe message. We, therefore, estimate the one-way time with half the round-trip time when calculating the transmission time.

7.2.2 Network Measurements

To measure the network capacity the sending rate, t_{sr}, and RTT rate, t_{rttr}, are continuously monitored. The data stored in the table is used for predicting future compression algorithm usages. Each measurement is a cumulative moving average to ensure that network changes, such as congestion, affect the algorithm selection procedure. In general, hard compression is favoured on slower congested networks with high transmission time.

7.2.3 Compression Measurements

A number of algorithm specific metrics are measured such as compression rate, t_{cr}, decompression rate, t_{dr}, compression ratio, H, see Table 7.1. The counters are updated each round and shows messaging properties for the system it is running on. The values will differ depending on the hardware it runs on and the message content. The values are calculated as cumulative moving average over a user-defined time to provide a good balance between quick response and stable behavior. If a particular system experiences problems with this approach it is simple to change this mechanism to a more appropriate one.

7.2.4 The Communication Procedure

We explain our method of automatic message compression by showing the *send* and *receive* procedure. The sending process owns all messages that are sent

Algorithm	**Processed Size** $[B]$	$\overline{t_{cr}}$ $[B/us]$	$\overline{t_{dr}}$ $[B/us]$	$\overline{H}$
Uncompressed	100k	0	12	1.00
LZFX	200k	0.00	10	0.90
LZO	2300k	0.05	50	0.5
LZO_SAFE	400k	0.003	32	0
LZMA	500k	0.002	43	0
LZW	5000k	0.001	32	0
BZ2	5000k	0.02	25	0
LZ4	5000k	1.0	100	0

Table 7.1: A number of metrics are monitored and stored in a database to be used in the decision process for future messaging. There is one table for each communication stream. Metric stored are the number of bytes passed through the communication subsystem, compression rate($\overline{t_{cr}}$), decompression rate($\overline{t_{dr}}$) and compression ratio($\overline{H}$). The values in the table are fabricated and will be different for each system.

and they belongs to a message stream, see Figure 7.2. The message stream is divided into rounds separated by an assessment period where the previous rounds are evaluated and the compression strategy is decided for the next round. It is desirable to keep as long round as possible while still letting adjustments take effect in a reasonable time. Additionally it must be long enough so that there is sufficient information to evaluate to make a good estimation of the compression algorithm performance. Both the round length and the number of rounds evaluated are configurable and system dependant.

Sending a message

We describe the communication procedure in the procedure below. The procedure shows the most significant steps together with links to later sections where we describe more details. There are no measured values when round 0 starts, so we use all compression algorithms equally much. For each subsequent round r do:

1. Evaluate previous statistics and assign r its parameters.
   (a) Calculate the algorithm distribution for r by setting a weighted probability, see Section 7.2.5. An algorithm that has a lower message RTT gets a higher probability for being selected than other algorithms.

   (b) Derive a compression time budget for r, see Section 7.2.7.

2. Send messages.
   (a) A random compression algorithm is selected from the list with its weighted probabilities. This means that for adjacent messages there might be different compression algorithms depending on message size, content etc. If the time budget is exhausted, don't compress the message.
   (b) Send the message.
   (c) Update the statistics for the compression algorithm used, also decrease the time budget with the time it took to compress the current message. Store the data at the sending node.
3. Until end of round, goto item 2
4. When the round ends, goto item 1.

The length of a round can be determined by time or number of messages depending on requirements and behavior for the system being implemented.

Receiving a message

The reception procedure is simpler than the sending procedure since no algorithm selection is needed. The following procedure outlines the necessary steps to be performed by the receiver.

1. The message is received.
2. The receiver reads the message header to determine which compression algorithm was used for this particular message.
3. The message is decompressed.
   (a) Destination memory is allocated for the uncompressed message.
   (b) The time it took to decompress the messages is stored in a database on the receiver node. The stored time is sent to the sender node with periodic probe messages, Section 7.3.3.
4. The application can resume operation with the newly received message.

7.2.5 Selecting the Best Compression Algorithm

It is possible to use many different approaches when selecting the most appropriate compression algorithm. For example, we could make a static selection based on the result of an initial evaluation run. This would initially provide

the best possible overall compression ratio but any content change in the communication stream would over time cause the selected algorithm to perform poorly. At the other end of the spectrum we could use all available algorithms evenly in a round robin [15, Ch 7.7] fashion. This provides a lower overall compression ratio but will be robust whenever there are content changes. We have defined three different algorithm distributions: Majority, One Algorithm or Round Robin. They are explained in more details below.

Compression Selection 1 - Majority Distribution

At the beginning of each round a compression algorithm distribution is calculated based upon measurements during previous rounds. The calculation is done according to Equation 7.6 for each algorithm providing an estimated cost for a typical message of size s. This produces a set of data. Depending on the communication stream users the message stream content may suddenly change. This may lead to vast differences in compression ratio and compression time for different algorithms. This is one reason for allowing all compression techniques to run in each round. Completely turning off a compression algorithm would make the strategy static and unable to cope with a changing situation. The internal distribution is decided by a simple scheme where the initial values have been decided by emprirical tests. The best algorithm is selected and all other algorithms gets 1% of the compression budget. The rest is allocated for the best algorithm. The effect of this is that the for the next round the majority of the messages will be sent using the best algorithm but it will still be possible to detect when another algorithm performs better and move it up the scale to be used more often. Higher performance can be obtained by more elaborate distribution schemes for compression algorithms.

Compression Selection 2 - One Algorithm

One algorithm is manually selected and gets the complete compression budget. No other compression algorithm will be used making this an offline method of determining which method is to be used. If it is possible to select the best algorithm it will provide the best compression ratio but as it is a static selection it provides poor performance for message streams where the content changes.

Compression Selection 3 - Round Robin

Round robin [15, Ch 7.7] apply an equal share of compression budget between the available compression algorithms. Each algorithm get the same amount of

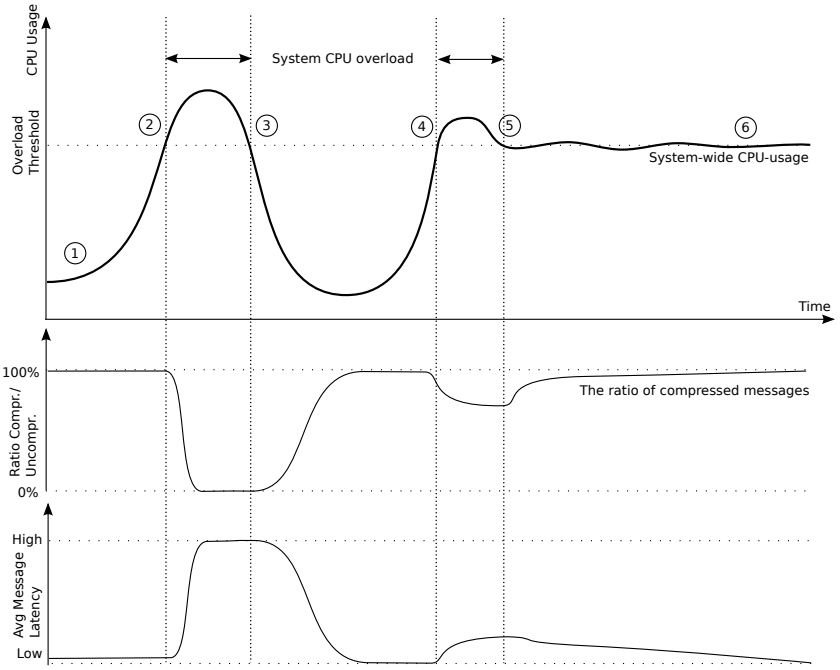


Figure 7.3: Adaptive online message compression overload protection.

slots with no regard to their performance. This results in a great flexibility but may lead to lower performance can be gained through compression since all algorithms are used.

7.2.6 Compression Overload Controller

Continuously compressing large quantities of data requires substantial computational capacity. We quickly realized that we needed to prevent overloading the CPU when enabling message compression.

The overload mechanism is illustrated in Figure 7.3. The CPU load is, in the left part of the figure, well below (1) the threshold. A temporary load-increase (2) surpasses the threshold, and our mechanism reduces the compression quota resulting in fewer compressed messages. Message compression is resumed when the system load is reduced (3). Our overload mechanism also handles scenarios with partially compressed message streams. If the total CPU load

caused by message compression and other services are above the threshold (4), our mechanism reduces the compression quota. A quota reduction may lead to a partially compressed message stream with compressed messages intermixed with uncompressed. Message compression is gradually resumed (5) as the total system load converges to the overload threshold (6).

7.2.7 Compression Throttling

We want to make sure that all processes gets a fair chance of running. In some cases high CPU-load and excessive message compression may starve other services running on the same CPU. We have implemented a control algorithm that throttles the amount of CPU cycles that can be used for message compression. The idea is to continuously monitor CPU-usage and current communication bandwidth. If the current CPU-load is low and the desired bandwidth is high it is possible to trade computational capacity for an increased compression level causing the perceived bandwidth to increase. This is achieved by assigning more CPU capacity to message compression.

Compression and Decompression Time

We measure the time spent on message compression and decompressing messages during a round. The total time is calculated by adding the compression time t_c and decompression time t_d time for each individual message n. Adding t_c and t_dc for all messages, N, during a round results in the total time, t_{tot}, spent performing compression activities, Equation 7.4. Aggregated decompression time measurements are piggybacked on probe messages sent from receiving nodes to the compressing node when measuring the message round trip time per compression algorithm.

$$t_{tot} = \sum_{n=0}^{n=N} (t_{c(n)} + t_{d(n)}) \tag{7.4}$$

Controlling the total compression time

We want to throttle the total time spent compressing messages, t_{tot}, per round to adjust the computational capacity spent for message compression if the CPU is overloaded. We use a feedback control algorithm which aims at maximizing the amount of time assigned for compression to match the desired CPU-load

target level. We want to find the optimal distribution of CPU-load between computational capacity and messaging compression to maximize the throughput without overloading the CPU.

The Control Algorithm

The (PID) control algorithm use CPU load as input and t_{tot} as output. The control algorithm will continuously adapt t_{tot} to converge to the target CPU load given at the time of system configuration. Increasing t_{tot} will cause more CPU time to be allocated for message compression.

Initial Values

We do not have any knowledge of the message stream content at the start of a new communication system or when creating new communication streams. In these cases we set the initial compression budget is 0 to avoid an instantaneous CPU overload. Initially, for the first round, there will be no compression and for the subsequent rounds the feedback control loop will increase t_{tot} to allow more compression. The desired value of CPU load is system dependent and therefore configurable. Most systems will have different services running at the same time as the messaging system so the CPU-load budget must be carefully determined.

7.3 Implementation

We have implemented the message compression functionality as part of a Linux-based telecommunication system. The application we are targeting is running inside an emulation layer that adds support for a legacy OS. We have only made changes in the communication API parts by adding the ability to support message compression before entering the TCP/IP layer inside the Linux kernel. No changes has been done to the Linux kernel.

We have modified an existing communication system by introducing a transparent message compression layer, as a response to the challenges above, see Figure 7.1. The legacy communication API containing `snd_msg()` and `rcv_msg()` is wrapped by `snd_msg()'` and `rcv_msg()'` to capture the message data. Our implementation of the API will transparently compress messages and then use the standard communication API supplied by the OS.

We have integrated eleven state-of-the-art compression algorithms in the telecommunication system we have investigated. Each algorithm has special

compression properties. One algorithm can, for example, provide high compression ratio but require much CPU time (LZMA [228]), which may be suitable for networks experiencing high congestion. Other algorithms may have special target areas, such as efficient text compression (SNAPPY [114]) or being fast but not so high compression ratio (QLZ [240]).

Our implementation requires the same communication API for both the sender and the receiver. The `snd_msg()'` function prepends a new header to each transmitted message. The header contains information on the compression algorithms used for compressing the particular message. When the receiving node calls the `rcv_msg()'` function, the API implements a transparent decompression of the message. The API sends decompression statistics to each sender making it possible to calculate the complete compression $\rightarrow$ transmission $\rightarrow$ decompression time.

7.3.1 Compression Algorithms

There are numerous compression algorithms, all designed for various uses and with different characteristics. Some [228] [266] of the algorithms focus on pure compression ratio with little regard to the compression- and decompression rate. Others [224] [48] provide a lower compression ratio but are faster. It is easy to understand that both approaches are useful in different situations. Additionally, it is possible to accelerate partial or complete compression algorithms by implementing them in hardware [139]. In this paper we focus on lossless [238] compression thus completely disregarding lossy [33] compression techniques. Lossless techniques could be applied to data that doesn't need to be transmitted in an unchanged manner. Our implementation of the automatic mechanism uses eleven compression algorithms/configurations; LZFX [49], LZO [224], LZO-SAFE which is a safe configuration of LZO, LZMA [228], LZW [298], BZ2 [266], LZ4 [48], FastLZ level 1 and 2 [128], Snappy [114], and QLZ [240]. The key properties are listed in Table 7.2. The list is extended from the results by Karlsson and Hansson [165].

It is simple to add new compression algorithms. The current implementation uses a list where the compression algorithms are transparently used. All algorithm measurements are generic and they do not depend on the specific implementation.

Table 7.2: Implemented compression algorithms and their characteristics.

Compr. Alg.	Key Characteristics
LZFX [49]	Fast compression, low H.
LZO [224]	Fast compression, low H.
LZO-SAFE [224]	A safe configuration of LZO, slightly slower than LZO.
LZMA [228]	Slow compression, high H.
LZW [298]	Medium compression, medium H.
BZ2 [266]	Medium compression, high H.
LZ4 [48]	Fast compression, low H.
FastLZ lv1 [128]	Fast compression, low H, suitable for small messages.
FastLZ lv2 [128]	Slightly slower than lv1, higher H than lv1.
Snappy [114]	Very fast compression, medium H, suitable for text messages.
QLZ [240]	Very fast compression, medium H, suitable for small messages.

7.3.2 Putting it all together

A message stream is divided into communication rounds, see Figure 7.2. Each round consists of two phases, see Definition 21. The first part is an evaluation of statistical data retrieved from previous messaging rounds. The evaluation procedure uses the historical data to predict if message compression should be used for subsequent messages, as defined by Equation 7.2. The most suitable compression algorithm is chosen depending on its ability to reduce the message transition time. Messages will be compressed using the algorithm distribution decided by the selection phase, during the second part, see Figure 7.2,

Definition 21 A communication *round* is started by an evaluation phase followed by a transmission phase, and it is delimited by a fixed number of messages.

Estimating the Compression Metrics

The time to transfer a message between two interconnected nodes is a central concept in this paper and is denoted transmission time (Definition 7.2). We have split this procedure in several parts that are individually measurable in a run-time environment.

The first part is the sending time, see Definition 22. It is the time it takes to prepare the message and present it to the driver that will perform the actual link

transmission. The time to send a message is defined as, $t_s = \frac{s}{t_{sr}}$, where s is the size of the message in Bytes and t_{sr} is the rate of transmission [Byte/s].

Definition 22 Sending time is the time spent inside the low-level send-message function call after all mandatory message processing has been performed.

The second part is the Round Trip Time (RTT), see Definition 23.

Definition 23 The round-trip-time (RTT), $t_{rtt} = \frac{s}{t_{rttr}}$, is defined as the time it takes for a message of size s to travel from node A to node B and back to A. The amount of data per time unit sent back and forth between two communication partners, RTT rate, is defined by t_{rttr} [B/s].

Estimating the Transmission Time

With Equations 7.1 – 7.3 we can describe the time it takes between the sending application calls the `send_msg()` function and the receiver obtains the decompressed message and can operate on it. For our system we assume that the link time (t_l) is roughly half the RTT, as defined by Equation 7.5. We assume that it takes equal time back and forth between two nodes, which gives an estimation for our proposed algorithm.

$$t_l = \frac{t_{rtt}}{2} \tag{7.5}$$

Combining Equation 7.2 with Equation 7.5 results in Equation 7.6.

$$t_t = t_c + t_s + \frac{t_{rtt}}{2} + t_d \tag{7.6}$$

The evaluation phase needs statistical data to be able to predict the need for message compression. Each algorithm will have its own set of measurements for compression time (t_c), send time (t_s) and decompression time (t_d). The round trip time (t_{rtt}) is a network dependent parameter and is stored on a per-network basis.

7.3.3 Real-World Compression Throttling

Compression throttling, see Section 7.2.7, is dependent on the ability of accurately measuring the current system CPU utilization. We assume that systems in which our method should be used have such metrics available. The CPU load is an input to the automatic throttling functionality when determining actions for current and future messaging strategies. In particular, we are interested in

the current CPU-load and communication properties since these are used to determine the compression technique. If the CPU is already saturated with other computational tasks it is possible avoid burdening it further with compression to allow the highest possible throughput while preserving the availability of other services.

Measuring the CPU-load

The CPU load is measured on a per-core basis, see Definition 24, and is used by the feedback control loop determining how much time should be spent on compression. Normally the target CPU load should be set to a value that allows other services to run in the desired way. Setting the target CPU load too high may cause the system to overload since all messages will be compressed and the worst case is that some process may starve out vital functionality on the system.

Definition 24 The *CPU-load*, L, is defined as the number of processes, ready to execute, in the run-queue of the operating system [119].

Linux implements the CPU load measurement through the `/proc` filesystem. An application can open a specific virtual file (`/proc/stat`) and read various CPU load figures. Linux supplies a 1 minute running average for the CPU-load to avoid jitter. We use the 1 minute CPU-load measurement because it greatly reduces oscillation problems in the feedback controller, where intermittent service usage may cause spikes in the CPU-load.

Measuring the RTT

We periodically issue probe messages to other nodes to measure the message RTT, indicating the network saturation. The periodicity is user-configurable.

7.4 Experiments Using Automatic Message Compression

We have devised four tests to verify that our automatic compression method improves the message performance, is fully automatic and that it fulfils industrial requirements. The first test presented in Section 7.4.1 verifies that it is possible to improve the messaging performance by compression. The second

test presented in Section 7.4.2 verifies that our method finds the most appropriate compression algorithm from the set of available algorithms. Our third test presented in Section 7.4.3 shows that the selection algorithm can handle a changing message stream. The fourth and final test presented in Section 7.4.4 makes sure that the number of compressed messages are reduced when the CPU is overloaded.

Data sets We use several different data sets, $\{\mathbb{D}_0, \mathbb{D}_1, \mathbb{D}_p\}$, in our experiments. The first data set contains only zeros, which we denote $\mathbb{D}_0$. The second set contains only ones, denoted $\mathbb{D}_1$. We also use message content gathered from the control traffic of a production environment in each test, denoted $\mathbb{D}_p$. The traffic was intercepted by Wireshark [50]. Our assumption is that production traffic data will provide a more realistic evaluation of our suggested techniques compared to pure synthetic test data. The data is control traffic to handle maintenance duties, internal communication etc.

7.4.1 Automatic Compression

The goal with this test is to verify that it is possible to increase the overall message processing performance by using message compression.

Setup The test setup is as follows: Node A is a Quad-Core AMD Opteron 2378@2.4GHz running Ubuntu Linux with kernel 3.2; Node B is a Intel® Core i7-4600U@2.10GHz running Ubuntu Linux 3.2; both nodes are interconnected by a 100Mbit Ethernet network that is shared with a large number of workstations and other office equipment.

Execution We have three test cases (TC1, TC2 and TC3). In TC1 we send uncompressed messages gathered from a production node between A and B. For TC2 we use data set $\mathbb{D}_0$ and our automatic compression mechanism selects how to send them most efficiently. Scenario TC3 is the same as TC1 but with data set $\mathbb{D}_p$.

Evaluation We display the measurements from TC1 in Figure 7.4a, which acts as a reference measurement representing the original system. In TC2, our compression selection algorithm automatically selects Snappy [114] as the best algorithm when using $\mathbb{D}_0$, see Figure 7.4b. Snappy is a fast and efficient compression algorithm for simple textual messages. External network disturbances triggers an additional compression algorithm selection at approximately 10k

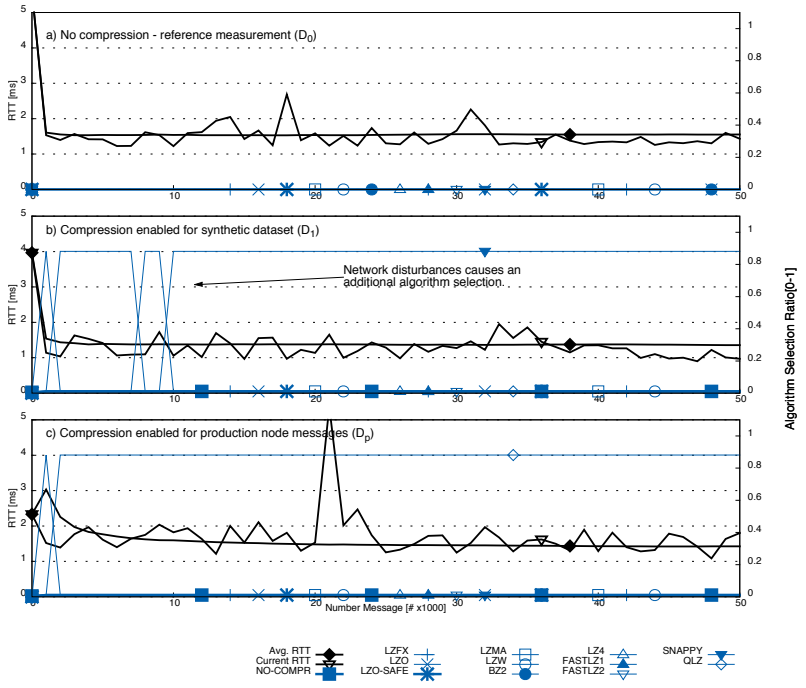


Figure 7.4: Three different test runs; a) Uncompressed messages; b) Compressed messages with zero-pattern payload; c) Compressed messages with production system pattern.

messages but the most optimal compression algorithm is still Snappy. For TC3, our compression selection mechanism selects QLZ [240] when we use a more complex, and realistic, data set $\mathbb{D}_p$, see Figure 7.4c. The transient at 20k messages may be caused by network congestion on the busy office network we are using. QLZ is very fast at compressing messages, similar to Snappy.

Remarks The reason for our selector to chose Snappy in TC1 and QLZ in TC2 is probably that the available communication bandwidth is fairly high. The benefit of spending time on compression can not be earned-back in the transmission phase. Our conclusion is that it is possible to increase the messaging performance by message compression.

Table 7.3: Relative improvements for different algorithm selection strategies.

Selection mechanism	**Test case**	$\overline{RTT}$ $[ms]$	**Relative time reduction**
Uncompressed	TC1	1.57	0.0 (Reference)
Round Robin	TC2	1.45	-8.8%
Automatic (QLZ)	TC3	1.35	-16.6%
Offline algorithm (QLZ)	TC4	1.30	-21.1%

7.4.2 Algorithm Selection Methods

Our goal with this experiment is to verify that the our automatic compression mechanism can provide the best message processing performance by selecting compression algorithm that gives the lowest RTT.

Setup We use the same test setup as in Section 7.4.1.

Execution We run four test cases in this experiment. All messages are replayed from the data set $\mathbb{D}_p$. TC1 is the reference test where we send all messages without compression. This scenario represents the unmodified communication system. In TC2, we use all compression algorithms equally much in a round-robin fashion [15, Ch 7.7]. We evaluate our method in TC3 by letting the system automatically select the compression algorithm providing the lowest message RTT. In the last test case, TC4, we turn off our automatic mechanism and manually evaluate the QLZ algorithm. We know, from the experiment in Section 7.4.1, that QLZ is the best algorithm for the production data set so it should be the ideal compression algorithm.

Evaluation We present the result from all tests in Table 7.3. We have normalized the RTT values according to the measurements for TC1. The measurements show that TC2 gives an 8.8% RTT reduction compared to TC1. It is interesting that even though TC2 uses all compression algorithms it has a lower average RTT compared to the reference value. Using our automated compression selection mechanism in TC3 gives a 16.6% RTT reduction. Manually assigning a compression algorithm reduce the RTT for TC4 by 21.1%. It is apparent that TC3 is not performing as well as TC4 when evaluating the results but it still provides a 16.6% reduction in RTT. The automatic selection mechanism is second best and provides the automatic selection functionality with the additional cost

of $21.1\% - 16.6\% = 4.5pp$ in RTT. The cost of automation can be attributed to intermittent use of non-optimal compression algorithms to be able to detect changes in the message stream. The cost can be reduced at the expense of not having the same ability to detect message stream changes.

Remarks Investigating TC3 in greater detail shows some interesting facts. The QLZ [240] compression algorithm is automatically selected in TC3, see Figure 7.5. T3 is inferior to T1 during the first part of the graph, up until about 6k messages. At that point the selected compression algorithm starts to contribute to an increased message processing performance and lower message RTT. Our conclusion is that our automatic selection mechanism finds the best compression algorithm, but the adaptability comes at a cost.

7.4.3 Automatic Compression Algorithm Selection for Changing Message Content

The goal of this experiment is to verify that our automatic selection mechanism will continuously ensure the best message processing performance even when the content changes in the message stream.

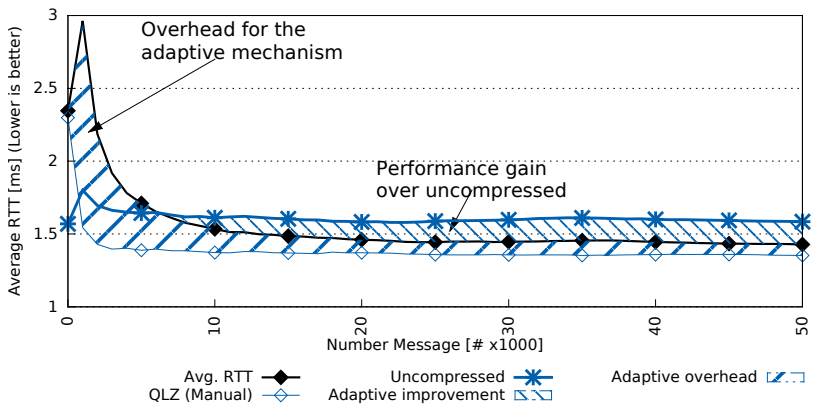


Figure 7.5: Cumulative average round trip time (RTT) for each of the selection methods.

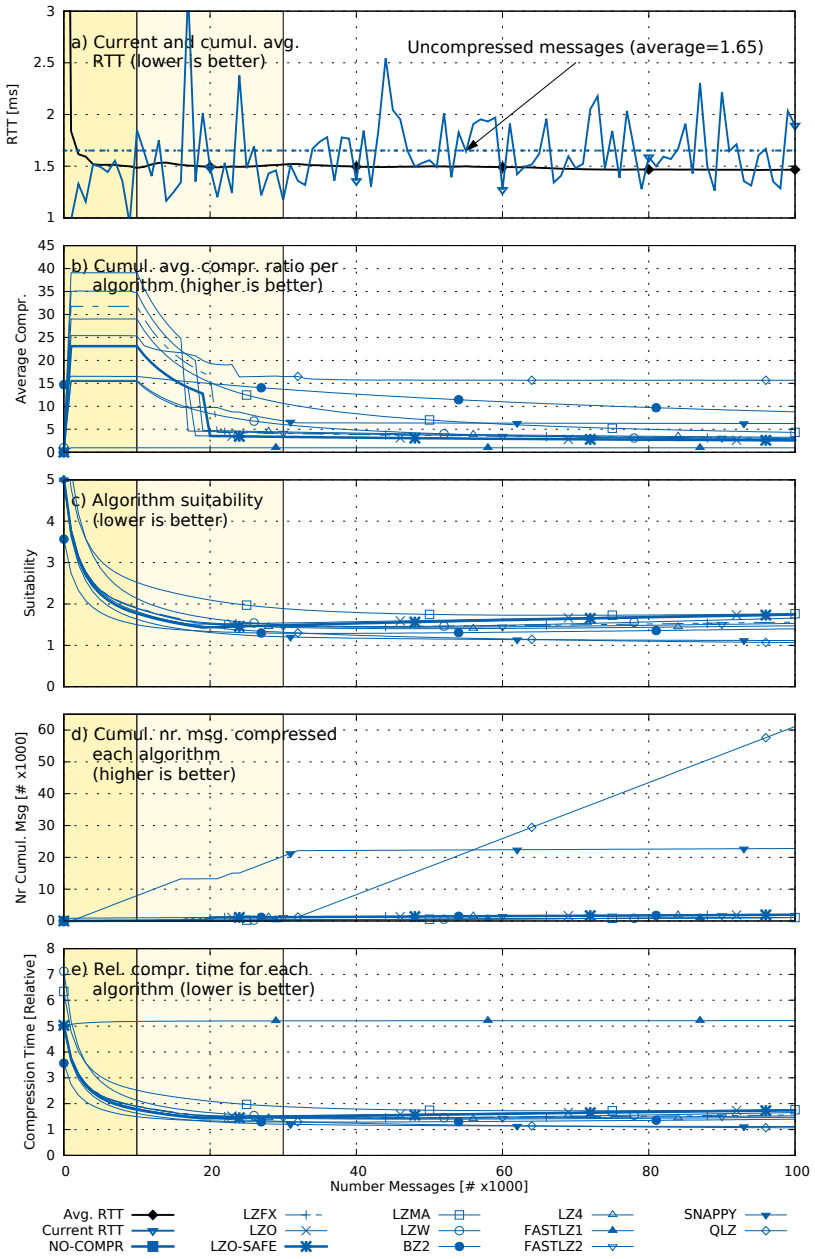


Figure 7.6: Message stream change at 10k messages triggers an re-selection (Snappy→QLZ).

Setup We use the same test setup as in Section 7.4.1. The system is configured to select the best algorithm from all implemented algorithms. We run 100k messages before ending the test.

Execution In this experiment we use two data sets. Our test run is depicted in Figure 7.6. The darker yellow field to the left, between 0 and 10k messages, shows the initial data set $\mathbb{D}_1$. At 10k messages we switch to the $\mathbb{D}_p$ data set. This is marked by a lighter yellow shade ending at 30k messages where our method has found the best compression algorithm and starts to use it exclusively.

Evaluation We show that the automatic mechanism selects Snappy [114] for $\mathbb{D}_1$ in the first part of the message stream. The selection mechanism detects that the message content changes and QLZ [240] is selected as a better match for $\mathbb{D}_p$.

The first and uppermost graph a) in Figure 7.6 shows the message RTT. The mean message RTT value is much higher in the initial part of the experiment but at 5k messages it is reduced to a value lower than uncompressed messaging.

The second graph b) shows the mean compression ratio for each algorithm being evaluated. Snappy is chosen as the most suitable compression algorithm for $\mathbb{D}_1$. Snappy results in a good overall RTT for the first data set. After changing the message content to $\mathbb{D}_p$, the QLZ compression algorithm is selected.

The third graph c) shows the suitability of each compression algorithm. This is a selection algorithm heuristics that we use to select the most appropriate compression algorithm.

The fourth graph d) shows the cumulative number of messages compressed by each algorithm.

The final graph e) illustrates the relative compression time for each algorithm. The FastLZ1 algorithm is by far the slowest of all compression algorithms.

Remarks We see that Snappy is the best compression algorithm when interpreting the graphs in Figure 7.6 within the interval $0 \rightarrow 10k$ messages. The $\mathbb{D}_0 \rightarrow \mathbb{D}_p$ data set change at $10k$ triggers a new selection procedure. Several compression algorithms compete in the interval $15 \rightarrow 30k$ messages and QLZ is finally selected at $30k$. The QLZ compression algorithm dominates in the remaining interval $30k \rightarrow 100k$.

We conclude that our section algorithm adapts to a changing content in a message stream by finding them most appropriate compression algorithm with respect to message RTT.

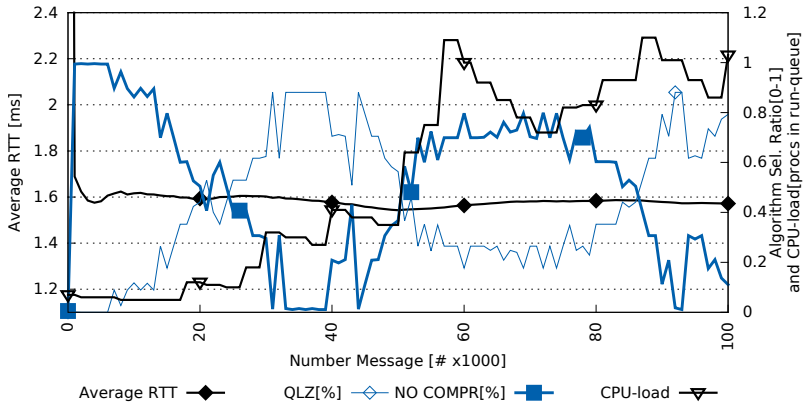


Figure 7.7: Overload handling by reducing compression time budget.

7.4.4 Overload Handling

The goal of this experiment is to verify that our message compression selection algorithm is able to limit the CPU resources available for message compression. As previously explained, this is to ensure that our algorithm does not cause any performance degradations for other services co-located on the same CPU.

Setup We use the same test setup as in Section 7.4.1.

Execution In this experiment we use messages from the $\mathbb{D}_p$ data set. An external application, cpuburn [209], runs in the interval $27k \rightarrow 50k$ messages with the sole purpose to overload the system, see Figure 7.7.

Evaluation We can see that the ratio of QLZ-compressed messages are increased in the interval $0 \rightarrow 30k$. Our overload controller mechanism detects the increased CPU-usage and reacts at $40k$ by reducing the amount of CPU-resources available for message compression. The resource shortage cause fewer messages to be compressed by QLZ in the interval $40k \rightarrow 60k$. The available compression resources are stable between $60k \rightarrow 80k$ resulting in a somewhat constant balance between uncompressed and QLZ-compressed messages.

Remarks We do not know why the CPU-load is increases in the interval $70k \rightarrow 100k$. One plausible reason is that the overload situation caused by cpuburn may have caused the queue in the send-buffer to increase. A large queue needs to be processed when the CPU is restored to a stable state.

After running this experiment we conclude that our overload mechanism can reduce the CPU-resources available for message compression in situations when the CPU is burdened with other tasks.

7.5 Related and Future Work

We start with general compression techniques then list relevant automatic compression techniques. At the end of the section general message compression techniques are described.

Comparison of Compression Algorithms

Ringwelski et al. [242] manually investigates a number of compression techniques with regard to compression ratio and computational resources. This is the starting point for our investigation, how can this task be fully automated? In the MSc thesis [165] by Karlsson and Hansson, performed as part of this investigation, a number of compression techniques are compared with regard to compression ratio and resource usage. Their work also investigates the suitability for each algorithm in the context of communication scenarios.

Automatic Compression

Jeannot [159] describes a way to automatically compress messages being sent through a POSIX-like API called adaptive online compression library for data transfer (AdOC). The library can be freely downloaded at the official web page [160]. The major differences compared to our work are:

- Jeannot uses POSIX standard calls while we have adapted an asynchronous messaging system to be legacy compliant.
- They use multiple threads to compress and communicate data. We have performed the compression in user mode and single thread mode.
- AdOC uses large 200 KB messages compared to ours that are usually a multiple of 100s of Bytes.

- Jeannot uses an ingenious technique looking at the current send queue length. If the length is increasing the network must be saturated and higher compression is requested.

In [171] by Knutsson and Björkman have changed the communication mechanism in Linux so that packets are transparently compressed/uncompressed as a function of available memory and communication capacity. In their paper they state that it would be desirable to only use idle CPU cycles. Knutsson describes [170] that it is possible to use an automatic scheme to use different compression depending on the availability of CPU, RAM and network resources. In [161] Jeannot, Knutsson and Björkman revisits earlier techniques being reimplemented outside kernel space to improve portability.

Sucu and Krintz [278] have created a communication compression environment called Adaptive Compression Environment (ACE). It aims to change the behavior of socket communication introducing compression for certain messages. Only messages larger than 32 KB are affected since smaller messages are sent uncompressed. Krintz and Sucu [177] revisit the earlier paper [278] and adds a number of compression algorithms such as Bzip2, zlib and LZO.

The trade-off between available bandwidth and the required computational capacity for compressing message is discussed in [236] by Pu and Lenin. They present a thorough investigation of simple schemes such as “compress-all messages” or “compress-none”. Furthermore, which is of special interest to this publication is the investigation of mixed messaging. In other words, when the messaging stream is intermixed (denoted Fine-Grained mixing) with both compressed and uncompressed messages. Gray et al. in [117] points out vital problems with automatic compression such as how to decide when to compress and not compress a message. The authors expands the work done by Pu and Singaravelu [236] taking mixed sets of compressed and uncompressed message further. In [36] by Brunet et al. describes a technique to auto-tune compression parameters depending on the actual hardware running the application.

Biederman [25] owns a patent that shows a general idea of receiving, compressing and sending messages. The method is similar to ours but not take into consideration the following issues. We adopt a feedback control loop to control the amount to CPU time spent compressing. This can control the maximum amount of CPU load in the system allowing coexistence of other services. Biederman uses different levels of compression. We suggest to use arbitrary compression algorithms used simultaneously with an additional machine learning to let the best algorithm dominate.

Message and Data Compression

Gutwin et. al [121] describes a transparent way of compressing Groupware messages in an efficient way. Both text and serialized objects are supported which makes it convenient for users to use the framework. Wiseman et al. [301] investigates loss-less compression of communication systems. They use off-line data to rate compression techniques and then use them in an automatic fashion. Nicolae [217] apply compression to cloud computing and its effect on cloud storage is investigated. Especially how to sacrifice a slight computational overhead. An interesting part of this paper is that it applies a practical implementation on the Grid5000 research network to obtain results. A significant reduction of network traffic has been detected using both LZO and BZIP2 compression algorithms. In recent CPUs there is a trend to include hardware support for compression, see [139]. The apparent benefit is that this function will offload the CPU with the heavy burden of compressing messages. In our investigation this means that such an algorithm will have special characteristics with relatively low compression ratio but very fast compression rate.

7.6 Summary

We have answered Q3 (Section 3.2.3) by our work presented in this chapter and in the publications referenced by this chapter and through our publication C, based on earlier publication H.

We utilized our Charmon resource and performance monitoring tool to detect and find a communication performance problem. By going through the performance and resource usage measurements, we noticed that the communication link was saturated and that the system did not fully utilize the CPU at all times. Based on the observations results we devised a transparent method that automatically evaluates the suitability of each one in a set of compression algorithms on the current communication stream. Our automatic compression method uses the compression algorithm that resulted in the shortest message RTT for the current stream. The messaging mechanism automatically considers the available CPU-resources to reduce its effect on other co-located processes, i.e., we reduce the CPU quota for compression if other processes want to use the CPU.

Adding more compression algorithms is the first and most apparent direction for future improvements. The current set of compression algorithms works well for the system we investigate, but it would be interesting to know if we can improve the performance further by implementing newer compression algorithms.

We would also like to implement and test hardware supported compression algorithms. Hardware supported compression have a slightly different approach than software based ones because of the significantly reduced compression time. A drastic compression time reduction changes the behavior of the complete system, but we believe that our mechanism would automatically support it without any changes.

– Jag vet ingenting om tur,
bara att ju mer jag tränar
desto mer tur har jag[1].

My own translation:

– I don't know anything about luck,
but the more I train,
the more lucky I get.

— Ingemar Stenmark

[1] Stenmark becomes somewhat irritated when a reporter implies that it was pure luck that he won the world alpine skiing tournament.

8

Resource Aware Process Allocation and Scheduling

This section corresponds to research question Q4 (Section 3.2.4), which we have addressed in Papers D [157] and E [147] based on and extending Patents O [155] and P [156].

> *How can an operating system process scheduler provide high performance and enforce shared resource quality of service by allocating and scheduling processes on a multicore CPU?*

THE telecommunication industry is currently consolidating [66] many services on multi-core CPUs in an effort to reduce the hardware cost and increase system capacity. Cost reductions [283] is a natural step in an increasingly competitive [166] telecommunication market. Consolidating multiple services with various QoS levels is difficult because all services affect the shared execution environment. Sharing resources between services makes it difficult to specify and enforce hardware resource access for individual services.

We begin this chapter with a short introduction to process allocation and scheduling, Section 8.1. We continue by describing the system model and give important definitions in Section 8.2. The implementation of our allocation and scheduling architecture is described in Section 8.3. We evaluate our research by testing our allocation and scheduling architecture in Section 8.4. We describe how our work contrasts to other researchers' work in Section 8.5 and conclude the chapter with our conclusions in Section 8.6.

8.1 Introduction

The demand for advanced telecommunication gadgets [146] and the desire to be continuously connected [90] to the Internet drives the telecommunication industry to continue developing systems with higher capacity [47, 89] for every release. The customer demand for increased network capacity coincides with fierce operator competition [166] resulting in a Capital Expenditure (CAPEX) [283] reduction, i.e., the amount of money a company invests in buying or improving their products portfolio. The CAPEX reduction causes operators to look for cost-efficient [92] ways to optimize the performance of legacy implementations while simultaneously implementing new 5G [2,11] functionality. Consolidating multiple system functions on a multi-core CPU [66] is one of the most common ways to improve the cost and efficiency of legacy functions. Consolidation means that software designed for running on tailored hardware should suddenly co-exist with other software functions on a common hardware with many shared resources such as for example caches and floating point unit.

8.1.1 Motivation for Resource Aware Scheduling

Migration of legacy systems to multi-core architectures poses new challenges when decade-old software should retain the agreed system level agreements (SLA) [305] of QoS while executing in the same environment as newly developed functionality. The performance of a throughput-sensitive application will suffer from the increased shared hardware resource congestion caused by other applications. Applications may also miss their deadlines if the system engineer does not change process priorities and/or deadlines to account for the uncertainty introduced by co-executing applications. Hardware overprovisioning is the standard industry practice to provide sufficient capacity in many currently existing system deployments [307]. Overprovisioning will empirically tell the system designer that the set of co-located applications fulfills each SLA. However, it is difficult to meet all SLA requirements without being pessimistic when calculating the amount of hardware to overprovision. Overprovisioning is very expensive, and it is, therefore, desirable to use mechanisms that can achieve a deterministic shared resource execution environment without adding extra hardware. Efficient resource handling is an important research question and there are many [311] techniques to optimize the application assignment. Optimized assignment of individual instances of virtual machine or containers with CPU as the lowest level of granularity may be too coarse for cost-efficient system deployment. It is hard to ensure that a process has sufficient shared

resources [5], even for processes running on single-core CPUs. Adding several concurrently executing processes makes it even harder to provide a deterministic execution environment. The systems become even more complicated with the introduction of multi-core CPUs and heterogeneous [178, 212] systems that add several levels of shared hardware resources with individual characteristics and behaviors.

8.1.2 Problem Description and Current Solutions

Operating system process schedulers follows similar trends as other software. System engineers tries to keep the source code as simple as possible with a minimal memory footprint and overhead [196]. Current process schedulers, like the completely fair scheduler (CFS) [172] in Linux, keep track of many scheduling parameters, such as each process' historical CPU-usage and its configurable process priority. The main aim of CFS is to provide a fair execution environment through process allocation and scheduling. The user should experience a high degree of interactivity with the OS while not wholly starve all other processes.

As far as we know, no process schedulers for currently existing OS'es evaluate shared hardware and software resource usage during process scheduling. A high priority process, p_h, may be starved by a low priority process, p_l because p_l uses substantial quantities of a shared resource that cause p_h to miss its deadline. An example of such a case is when the periodical process p_h finishes with its first execution instance and the scheduler swaps in p_l. Process p_l immediately strides through its extensive working set and evicts all entries in the cache shared by both processes. When the next instance of p_h starts to execute it is swapped in, but the cache is cold. The cache needs to be re-populated with data for p_h, causing long data access delays. The system designer must consider the effects of shared resource congestion when designing the system, especially if the system has hard real-time properties or requirements for high throughput.

The problem is twofold. The first problem is that shared resource congestion leads to performance degradation since processes may not be optimally *allocated* on the available hardware. The second problem is that two processes may affect the QoS of each other because they are *scheduled* to run on shared hardware. We will describe allocation and scheduling in detail in Section 8.1.3.

The individual per-process CPU-usage alone may not be enough to provide performance guarantees for multi-core systems because application processes running on one hardware core may, through accesses to shared resources, affect the performance and behavior of processes running on other cores. Shared resource congestion is particularly problematic for the type of memory-bound

applications commonly running in telecom systems and other large-scale industrial communication systems. There are several ways to address this kind of congestion problem. One commonly used method is to turn off adjacent cores that share a shared resource. Overprovisioning by using hardware with much higher capacity than needed is another way. Neither of the two above mentioned methods efficiently utilizes the available hardware resources and is therefore costly to implement.

There are some approaches to address the problem of shared resource congestion such as ARINC-653 [291]. ARINC-653 is an aviation standard to enforce the partitioning of shared resources such as CPU, memory and I/O to avoid resource congestion. Our target system is by no means as sensitive as an airplane, but the general idea of hardware partitioning is similar between the two environments.

8.1.3 What to do about it?

What is commonly thought of and often expressed as process scheduling really consists of two different parts [309]: 1) Process *allocation* and; 2) Process *scheduling* [61, p35:5]. Process allocation acts on the problem of *where* processes should run, i.e. on what CPU or core. Process scheduling decides *when* and *how* processes should run.

The correct result of a system function is vital for most systems. But in many cases the correct result by itself is not sufficient. Also, the timely arrival of the result is of vital importance [277]. Using a result that arrives too late may be inaccurate and even cause a system to malfunction, for example by using real-time scheduling algorithms.

Many OS process schedulers use CPU-load as the main input when making scheduling decisions. The main tasks for the OS process scheduler is to select which CPU cores to allocate processes on and the duration and the sequence of processes' execution schedule. The traditional scheduling approach works well for processes that are CPU-bound (turtles [304]), such as the case when the CPU computational capacity is the main limiting factor. The performance might drop significantly when scheduling memory-bound processes (rabbits [304]) on large CPU/core clusters with shared resources. Memory-bound processes make intense use of memory, causing strain on all levels of cache memory [5]. Concurrently running multiple memory-bound processes on the same hardware reduces the overall system performance [313] due to shared resource contention. Interference between processes makes it difficult to provide an efficient execution environment with a deterministic QoS. Our opinion

is that it is possible to address the overprovisioning issue by making the OS scheduler shared resources aware. We have addressed the resource congestion problem by devising a method that automatically allocate processes to achieve high performance while at the same time schedule processes to enforce QoS. Our method automatically performs the following actions:

1. Measure hardware resource usage and performance for each monitored process, see Section 8.2.4.
2. Correlate each measured hardware resource and the process performance, see Section 8.2.5.
3. Determine what hardware resources have greatest correlation to system performance by sorting the correlation list and selecting the top hardware resource, see Section 8.2.5.
4. Allocate processes efficiently so that they minimize shared resource congestion by using the resource-performance correlation and knowledge of the hardware architecture, see Section 8.2.6.
5. Use process scheduling constraints on monitored processes to avoid performance degradation caused by shared resource congestion, see Section 8.2.7.

The following sections describe our allocation and scheduling method in detail.

8.2 System Model and Definitions

We have mainly focused on the systems running in the telecommunication application domain [23, 144] but our opinion is that other industrial domains can easily adopt our proposed method. Our target system is thoroughly described in Section 2.5. The applications utilize a process model that allows individual processes to freely migrates over certain subsets of the cores in a node. A predefined QoS requirement is associated with each application. For example the number of served requests/second, or maximum number of deadlines missed/second. We assume open systems, which means that it is possible to add/remove/change applications during runtime. The application workload is dynamic and varies over the system runtime. However, in our experience, the frequency of significant configuration changes is low (at least several minutes time resolution) for our considered applications. The system should not be mistaken for an offline [100] method simply because it provides directives related to process allocation and scheduling directives at a low frequency.

8.2.1 Terminology

Computer systems that are designed to meet a well-defined deadline fall into one of the two following categories [250, p1]: hard or soft real-time systems. According to our definition, a hard real-time system fails catastrophically if it misses a single execution deadline. If a soft real-time system misses a deadline, it suffers a performance degradation or QoS reduction but not a complete system failure. A process with high priority is deemed to be more important than a process with lower priority or to ensure that a set of processes are schedulable. The scheduler makes sure that a high-priority process can execute while a low-priority process stalls.

A process scheduling algorithm is either preemptive or non-preemptive [39, p35]. A process can be interrupted at any time and being replaced by another process with higher priority when using a preemptive process scheduler. In a non-preemptive scheduling algorithm, a task that has started its execution will run until it has completed regardless of the state of other processes in the system. The benefit of preemption is that a system designer can prioritize processes in the system letting them execute in the same execution environment [271, p264]. The downside to preemption is that intermittent context switches are costly [186] because the process scheduler needs to push-pop registers and program the memory management unit with different context depending on what process should execute.

It is also common to classify scheduling algorithms whether they are static or dynamic [39, p35]. A static scheduling algorithm determines the scheduling parameters before starting the process. All subsequent scheduling uses the initial parameters. The static approach contrasts with the dynamic scheduling where scheduling parameters is selected at runtime. The offline and online concepts are related to the static-dynamic classification. An algorithm that is offline calculates all scheduling decisions at process start and stores them in a list covering all possible scheduling scenarios. An online algorithm acts on either the dynamic or static parameters during the process execution time.

Regardless of the system type, we also need to define some fundamental process scheduling concepts. We show some of the most significant execution time definitions for an aperiodic process in Figure 8.1 [39]. An arbitrary process in our target system is denoted p_i where i is an index number to use when denoting multiple simultaneously executing processes. In some cases, we will omit the index number for increased clarity. From an execution timing perspective, p_i can be periodic, τ_i, or aperiodic, J_i. A periodic process repeatedly executes with a fixed time-period T_i. An aperiodic process executes sporadically at dif-

ferent time intervals. The ready-queue contains processes that wants to execute.

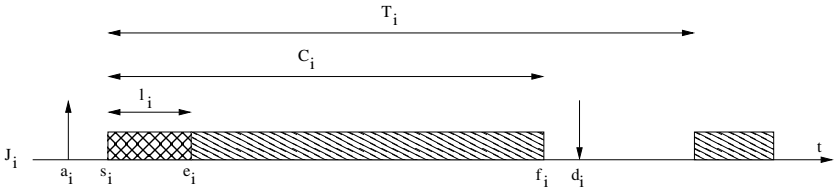


Figure 8.1: Timing definitions for an aperiodic process J_i.

As shown in Figure 8.1 an aperiodic process J_i is arrives in the ready-queue at a_i. The process starts its execution at s_i and continues its execution for a time C_i finishing at f_i before the stipulated deadline d_i. These variables describe the ideal execution of an aperiodic process without any explicit regards to shared resource congestions. A preemptive multi-process system implies that there are several processes, p_0 and p_1, sharing common resources such as CPU, cache and similar. When p_0 is swapped out it may have fully utilized the available shared resources causing performance disturbances for p_1 when it is swapped in. We have extended the original Figure 8.1 [39] with shared resource latency, l_i, i.e. the time it takes to swap in hardware resources after a context switch. Following this reasoning the execution start is delayed to e_i instead of the process swap-in at s_i.

8.2.2 Telecommunication System Requirements on Process Scheduling

Ericsson has a broad product portfolio, each with different requirements on the OS. The requirements vary from casual demands of necessary computational capacity to strict hard real-time execution requirements. The latter systems use either the general RT scheduling policy supplied by standard Linux or the more stringent RT-patched Linux kernel.

Other types of requirements arose with the introduction of multi-core CPUs and the drive to make systems more cost-effective by consolidating multiple functions on a shared CPU. The co-existence of multiple system functions on a shared hardware may cause shared resource congestion, which in turn cause a performance impact. A performance penalty is often acceptable, but in certain circumstances, the capacity and behavior of a system function must not be affected by other functions. This scenario introduced the requirement of a

shared resource aware process scheduler that can both limit the shared resource usage and also ensure a certain level of shared resource availability.

A typical use case for a shared resource aware process scheduling policy is when a customer would like to implement and deploy a tailored function inside the original system. Such deployment-specific system function must not affect the performance and behavior of the original system because that could affect the overall system functionality and QoS. A typical solution within the industrial community is to contain the new function in a virtual machine. The new function will still affect the performance of the other applications sharing the same hardware although the virtual machine encapsulates it. We need a method to not only encapsulate the execution environment, such as in a virtual machine but also to contain the hardware resource usage caused by the virtual machine.

8.2.3 Our Allocation and Scheduling Architecture

We have devised an architecture to achieve increased performance and deterministic QoS, see Figure 8.2 for an abstracted model. Our architecture follows the traditional closed-loop feedback principle consisting of sensing, controlling and actuation. In our architecture this becomes three tasks: The first task is to continuously monitor the hardware resource usage and the performance of selected processes through a performance monitor (PM). Monitoring corresponds to the sensing concept in closed-loop methodology. The output from the first step is a database of hardware usage and software performance. The second task is the controlling part. Here we utilize a decision engine (DE) to automatically analyze the data monitored by the PM and provide appropriate process allocation and scheduling directives. The output from the second step is process-allocation directives and scheduling parameters. The third and final task is the actuation, where we automatically act upon the allocation and scheduling directives. We allocate processes to the designated CPU/core and schedule processes so that they do not overspend their hardware resource quota and violates the QoS.

Industrial requirements reward flexible deployment, which caused us to aim for making each component in Figure 8.2 self-contained. Therefore, each element operates as a stand-alone component with a well-defined communication protocol. The dividing line between low-intrusive statistics collection (left) and continuous analysis (right) is typical for large-scale industrial systems where customers are sensitive to even very small application intrusions [241]. We also want to have an architectural border between collecting statistics and the analysis functionality. We want to ensure the possibility to perform the analysis-part

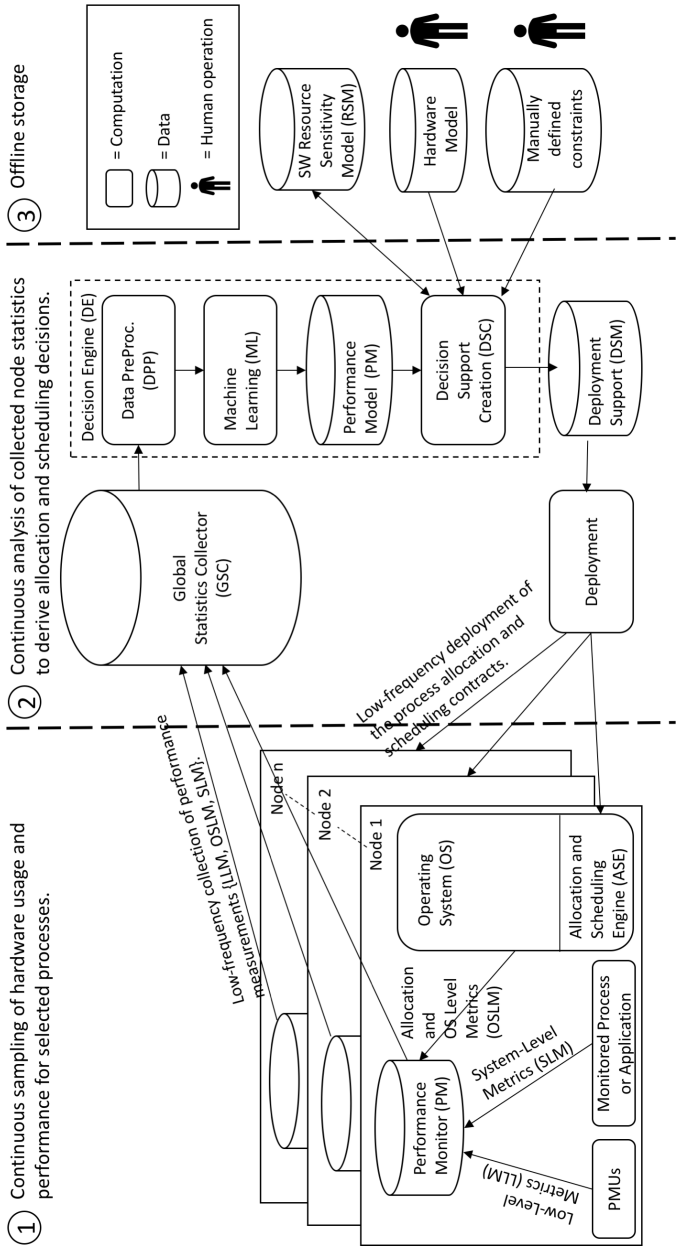


Figure 8.2: Our resource aware process allocation and scheduling architecture. Multiple nodes collect statistics ① that is transmitted to the decision engine ②. Offline data is stored in manually and automatically created files ③.

in a cloud environment because we estimate that the calculations requires lots of computational capacity when the size of the system increase.

The following sections will describe the most important components in the architecture.

Performance Monitor (PM) The PM continuously measures the hardware resource usage (R) and the system performance (x). We can configure the sampling frequency to provide a detailed execution profile while being low-intrusive. The PM is an extension of the Charmon tool described in Section 5.3. We have strived to make Charmon more generic and support various hardware and PMU event sets. The sampled data is combined and periodically transmitted to an analysis node. We store all needed parameters required by the PM in a configuration file that can be updated in runtime.

Global Statistics Collector (GSC) The architecture supports multiple nodes. Data gathered from one node can affect process allocation and scheduling for other nodes if they have a similar setup. Each node has an individual PM connected to the system-global GSC. The GSC collects data from all nodes and aggregate the information in a local database that is usable for other components in the scheduling framework.

Software Resource Sensitivity Model (RSM) The RSM describe how a process behaves and how sensitive it is to shared resource contention. The system continuously updates the RSM with new information deduced from R and x measurements. The RSM contains a snapshot of how processes behave in the system, and it is, therefore, possible to store the RSM for later usage. A stored snapshot is suitable for jump-starting systems before the initial deployed at a customer site. The RSM can, for example, consist of directives such as Process p_1 generates $1M$/sec memory accesses during normal operation, process p_1 and p_2 should not execute on the same core, or that the performance of p_1 correlates to a subset of R, such as cache-bound or CPU-bound [304].

Hardware Model The hardware model gives a low-level description of the hardware capabilities and capacity where the system executes. Information stored in the model can for example be cache architecture, size, and bandwidth; DRAM access time and bandwidth; or hardware floating point support and capacity. The hardware model is manually created offline for each supported hardware type. It is possible to create the hardware model using various types

of probing techniques that automatically detects the capacity and limitations to the hardware.

Manually Defined Constraints Experienced system engineers have vast domain knowledge of the system they implement. They may have well-founded opinions to consider when making process allocation and scheduling decisions. A manually defined constraint is for example: Do not run p_1 on the same core asp_2. A scheduling example is that p_1 should have a budget of $1M$ L_1D-cache accesses per second while p_2 does not have any limit.

Decision Engine (DE) and Data-Pre-Processor (DPP) The DE process performance samples from all nodes in the system. The primary task for the DE is to find correlations between hardware usage and process performance and to decide how to use this knowledge to make process allocation and scheduling decisions. The DE also tries to optimize what metrics to sample during the next evaluation period. The probe effect [108] and event multiplexing costs [67] are reduced by minimizing the number of concurrently monitored metrics (hardware resources).

The data sent from the nodes to the DE requires substantial data-processing to be suitable for automatic machine processing. It is also useful to get initial operator feedback on how the system behaves, like drawing hardware usage and performance graphs. An operator can quickly identify performance changes over time and determine if something is going wrong with the system. The Data Pre-Processor (DPP) aligns all performance samples so that it has the same time-base (start and stop time) and sampling frequency. Aligning data is vital for following data processing algorithms with strict requirements on the data format. The DPP also finds the correlation between R and x, as depicted in Figures 8.3a and 8.3b. There are numerous correlation metrics to use [187]. We selected the Pearson [125, 187] coefficient because it is simple to use and several scientific software libraries implement it.

The DE continuously update the RSM with information on how sensitive processes are for congestions. We have designed the architecture, so that already decided allocation and scheduling directives survive system crashes, upgrades, and other disturbances when clearing the internal memory. The DE contains several more components that we out of clarity describe in each of the following paragraphs.

Machine Learning (ML) The ML function process the well-formatted hardware usage and performance data together with the outcome from the analysis

step provided by the DPP. We can use the ML system to deduce bottlenecks, performance issues in the input data. First of all, we want to find performance bottlenecks. For example, if two processes that run on the same core reduce the overall performance because they utilize the same hardware resources. We also want to use the ML system to identify where to run processes and when to schedule them for minimal hardware usage and optimal performance. The performance model stores the result from the ML system.

Decision Support Creation (DSC) The DSC constructs and formats allocation and scheduling contracts by interpreting the performance model together with constraints and previously stored directives from the software resource sensitivity model, hardware model, and manually defined constraints. The output from the DSC is a set of contracts that describe process allocation and scheduling decisions. A process allocation example is how to distribute a set of processes over the available cores. Other types of scheduling directives can be that p_1 is not allowed to have more than $1M$ L_1-cache misses/sec. The Deployment Support Model (DSM) database stores the allocation and scheduling contracts. We exemplify an allocation and scheduling contract in Figure 8.9.

Deployment The DSM is parsed during the deployment phase and converted into a contract format that is distributable throughout the system. The contracts are designed to be easily transmitted to all nodes in large-scale systems. Message compression is typically not needed for standard systems but possible for large deployments. In our prototype, we use a textual format for the contracts because we want to simplify debugging and easy to visualize.

Allocation and Scheduling Engine (ASE) The ASE receives and parses the allocation and scheduling contract provided by the deployer. The ASE allocates processes to certain cores [155] by interpreting the allocation contract. The scheduling contract is forwarded to the OS process scheduler, which enforce the scheduling contract [156] when the demand for a shared resource is greater than the available capacity. The ASE forwards the sampling directive for hardware usage and performance to the PM. This mechanism ensures that the DE can vary the hardware resources to monitor without manual operations.

8.2.4 Resource and Performance Monitoring

We have utilized the performance monitor unit (PMU) [64, 83] via the Perf tool [188] to measure the hardware utilization of processes running in Linux.

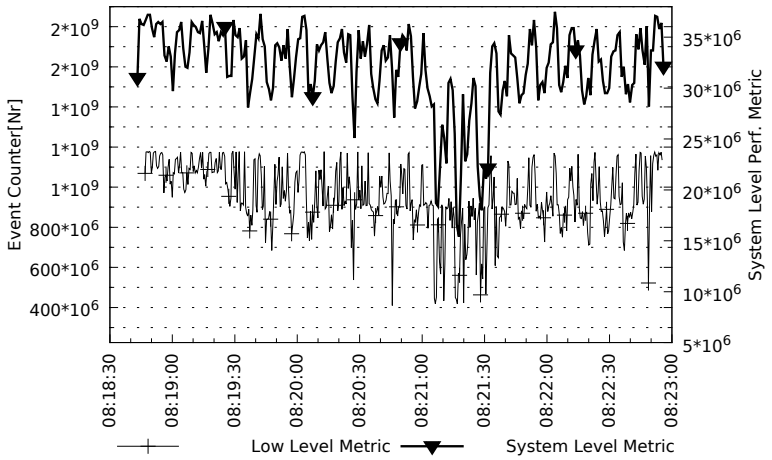
The user-space part of the Perf tool is well-tested and widely used in many projects [31]. There is also a kernel-space part of the Perf API used by various functionality in the Linux kernel. One example is the watchdog that uses the PMU to detect system lockups when the kernel or some driver runs in an eternal loop. The kernel part of Perf is less documented and not as widely used as the user-space interface. We have modified the process scheduler [258], which required us to use Perf in kernel space.

The scheduling performance monitor is an evolution of the monitor used by us in earlier work related to monitoring and modeling, see Chapters 5 and 6. We have developed the characteristics monitor further to support higher sampling frequencies and new hardware architectures. We have also added a convenient interface that is easily accessible via JSON [232] configuration files.

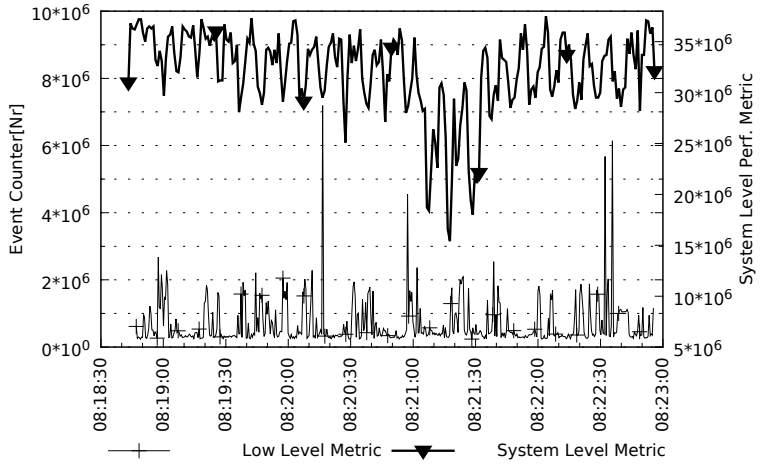
8.2.5 Resource and Performance Correlation

It is convenient to measure hardware resource usage by utilizing some of the available monitoring tools such as Charmon in Chapter 5, Perf [188], OProfile [184] or PAPI [109] just to mention a few. It is much more difficult to understand what PMU-events [226] to sample with the tools. It is convenient to describe the performance of a single-process-application with a set of low-level performance metrics R when the application is the sole user of a CPU. However, how can we be sure that the measured hardware metrics correctly reflects the perceived performance of the monitored application? Measuring the performance with R is more complicated when multiple processes share a common hardware resource, because each process may affect the performance of all other processes. A user-perceived application performance metric, x, is more suitable for describing the application performance [93] in such scenarios. The performance, x, is typically a humanly conceivable metric such as messages/sec, nr_operations/sec, user-requests served/sec or similar. It is usually difficult to compare x between two applications because the performance metrics may differ radically, for example x for p_0 is nr_operations/sec and x for p_1 is nr_simultaneous_users.

Our solution to the performance-comparison problem is to simultaneously measure R and x and then correlate the measurements for each $r \in R$ with x. We use R to compare the hardware resource usage of multiple processes and x to describe the performance of each process accurately. We quantify the correlation between measurements of R and x by using the Pearson [125, 187] coefficient. We denote the correlation between two sets a and b by $\rho(a, b)$. The value returned from the $\rho(a, b)$ spans between -1 and 1 where $\rho(a, b) = 1$ is



(a) There is *high* correlation between r_1 and x so that $Corr(r_1, x, p)$.



(b) There is *low* correlation between r_2 and x so that $\not Corr(r_2, x, p)$.

Figure 8.3: Correlation between two hardware resources, r_1 and r_2, and system performance, x, for a process p.

total correlation between the sets a and b, $\rho(a,b) = 0$ is no correlation, and $\rho(a,b) = -1$ is an inverted correlation.

Definition 25 Let $Corr(r,x,p)$ denote the *correlation* $\rho(m_{r,p}, m_{x,p})$ between the bounded series $m_{r,p}$ and $m_{x,p}$ for some $r \in R, p \in P, x \in X$, $|m_{r,p}| = |m_{x,p}|$ and where each element in $m_{r,p}$ is timely synchronized with the corresponding element in $m_{x,p}$.

It is often convenient to identify the maximum correlation value among all resources $r \in R$ and the performance x, which we denote $\widehat{Corr}(R,x,p)$.

Definition 26 Let $\widehat{Corr}(R,x,p) = max_{r \in R} Corr(r,x,p)$ denote the maximum correlation value for all resources $r \in R$ for a given process, $p \in P$, and performance metric $x \in X$

Similarly, it is convenient to identify the resource, $\hat{r}$, for which measured values have the highest correlation to the measured process performance.

Definition 27 Let $\hat{r} = \{r_i \mid \forall r_i \in R, p \in P, x \in X, Corr(r,x,p) > \theta\}$ denote the *set of resources* $r_i \in R$ with descendingly sorted correlation values larger than the threshold, θ.

We can quickly generate a list of sorted correlation values by calculating $Corr(r,x,p)$ for each $r \in R$. Identifying the resource $\hat{r}$ makes it possible to deduce which hardware resource has the highest impact on application performance. Figure 8.3a shows two graphs representing the hardware resource usage for two resources $R = \{r_1, r_2\}$ and the performance x for process p. Figure 8.3a shows a resource usage graph where the measurements for x is significantly correlated to the measurements for a a resource r_1. The other example, shown in Figure 8.3b, exemplifies a graph where r_2 and x has a low correlation. We therefore deduce that $\hat{r}$ is r_1 for $\widehat{Corr}(R,x,p)$.

For example, consider that a particular shared resource usage (such as L_1-cache) is correlated to the system performance for processes p_0 and p_1. A process scheduler can therefore deduce that p_0 should not be allocated in such a way that it shared a L_1-cache cluster with p_1, which typically happens in modern CPUs where multiple cores share common hardware resources.

Figure 8.3 acts as an example where we have configured the PMU to sample a set of hardware resources, R, simultaneously as the application performance, x. Our automatic correlation functionality process the sampled data, and we show the three hardware resources with the highest correlation to x in Figure 8.3.

This means in effect the resources with the highest correlation to system performance. The monitored application is an extensive user of L_1D-cache as well as L_1I-cache. Such information is useful for application engineers when they are improving the performance of individual applications. hardware congestion is also valuable for system engineers when they plan for future deployment schemes, i.e. where and how to run applications on shared hardware.
Understanding correlations between resources and performance is useful for automatic bottleneck localization [306] and has been used for a long time within industrial environments. The performance debugging possibilities is even further expanded when we add the possibility to automatically identify the resource having the highest impact on performance.

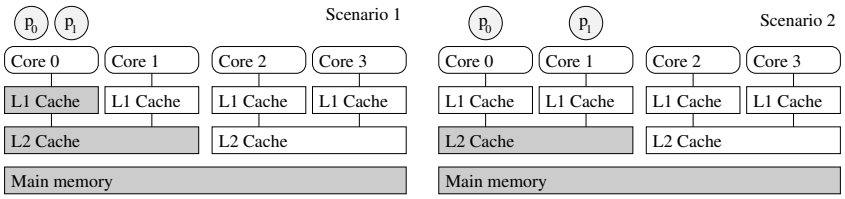
8.2.6 Resource Aware Process Allocation

The OS will in most cases automatically and dynamically allocate processes depending on the system load, process CPU usage, and power consumption settings. There are special cases when system designers need to statically bind, denoted *affine* in the Linux community, processes to a subset of the available set of CPU cores and most OS:es supply such functions through a privileged user accessible APIs.

Definition 28 The *affinity* $A_p \subseteq C$ for process $p \in P$ is the set of cores where p is bound/allowed to execute.

Process scheduling is a well-known problem. It has always been challenging to let processes efficiently co-exist on hardware with shared resources. There are many different implementations of process scheduling. Some schedulers targets a particular scenario, such as EDF [96], multi-resource servers [137], the lottery scheduler [205]. Yet others, like CFS [303], apply a more generic approach by being fair to all processes.

The standard CFS [163] scheduler in Linux is of the latter type, trying to be fair among all executing processes. In practice CFS balance between giving quick feedback for interactive processes while still allowing resource demanding background processes execute as much as possible. Some industrial systems have other constraints, such as real-time or high throughput requirements. We have identified resource constraints as another essential process scheduling requirement for our target system. In some process deployment scenarios, a process p_0 is not allowed to disturb the execution of another process p_1. A shared cache between two CPU cores is a typical use-case when one process can affect the performance of another one. In theory, our reasoning expands to



(a) L_1-cache congestion. (b) L_2-cache congestion.

Figure 8.4: Two processes p_0 and p_1 compete for the shared cache.

any shared resource such as cache, memory, FPU, branch prediction engines, or similar. We have opted to limit our reasoning to memory intensive application and their cache and memory usage in this thesis, although our reasoning and techniques are equally applicable to any measurable resource. The following scenarios exemplify the problem we address without allocation and scheduling framework, and the effect the problem has on our target system.

Scenario 1: Using the Linux CFS scheduler to allocate two processes p_0 and p_1 may end up in a non-optimal process allocation such as the one depicted in Figure 8.4a. In the scenario, p_0 and p_1 execute on the same core and share a common L_1-cache resulting in L_1-cache congestion. This type of scenario cause severe performance degradation if p_0 and p_1 are memory intensive processes.

Scenario 2: We can use a similar reasoning to the scenario shown in Figure 8.4b. The major difference is that p_0 and p_1 execute on adjacent cores that each has its L_1-cache but shares the L_2-cache. This scenario is an improvement from the first scenario because the number of L_1-cache misses will be significantly lower but p_0 and p_1 will still affect the performance of each other due to L_2-cache congestion.

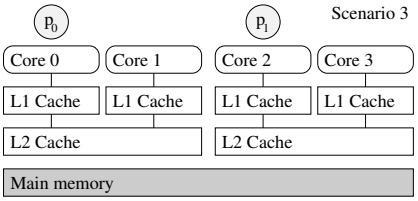
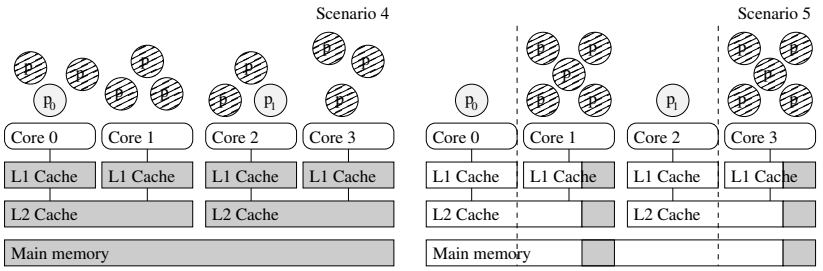


Figure 8.5: Memory congestion.

Scenario 3: The main problem with CFS is that it does not consider shared hardware resource usage when making its allocation and scheduling decisions.



(a) p_0 and p_1 coexist with many other processes.

(b) Efficient allocation of processes p_0 and p_1.

Figure 8.6: Deploying large number of processes on a target system.

Our tests [157] show that CFS can select any of Scenario 1, 2 or 3 depending on the other processes concurrently running on the system. The best solution would be to move p_0 and p_1 sufficiently apart so that they share as few resources as possible, like in Figure 8.5

Scenario 4: A more realistic real-world scenario is shown in Figure 8.6a where p_0 and p_1 co-exist with many other processes p. No process in the system is affined to a certain core and can float freely to any core depending on the available CPU capacity and the process execution pattern. The performance of p_0 and p_1 is difficult to evaluate because they will be severely affected by any other process p. The performance of p_0 and p_1 will be impeded by shared resource congestion even if we use an ordinary process priority scheme supported by most OS:es.

Scenario 5: Our target scenario is shown in Figure 8.6b where our Shared Resource Aware (SRA) process allocator efficiently allocates p_0 and p_1 on different cache clusters ($A_{p_0} = \{c_0\}$ and $A_{p_1} = \{c_2\}$), thus avoiding the L_1-cache and L_2-cache congestion described in scenarios 1 and 2. SRA also enforces the availability of shared resources by scheduling the processes so that any p cannot affect the performance of p_0 or p_1. SRA will in effect reserve a user-configurable number of cache accesses to the L_1-cache and L_2-cache and the main memory.

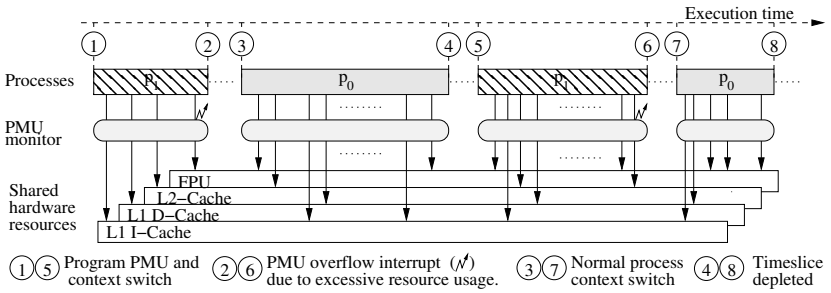


Figure 8.7: Enforcing hardware resource usage for process p_1.

8.2.7 Resource Aware Process Scheduling

Process scheduling decides how and when to run processes on a core. If many processes wants to utilize the CPU, some processes may need to share one core. The effect is, naturally, that the processes will affect the performance of each other.

We have devised a method [156] to schedule processes according to their resource usage rather than just considering their CPU quota. We limit the resources usage by processes through the use of quota for specific hardware resources. The scheduler suspends a process when it consumes its quota of any resource. The suspended process is resumed after a predefined time, i.e., get new quota for all resources. Our method is based on defining a resource overflow mechanism that generates an interrupt when reaching the configured resource quota. In that way we can enforce two scenarios: 1) set an overflow value for a particular process p so that it always get the desired amount of resources. 2) set an overflow value for all other processes sharing the resource, so that they cannot affect the performance of p.

Definition 29 The *access limit value*, $O_{r,p}$, is the number of accesses to resource $r \in R$ by a process $p \in P$ before the hardware should generate a resource usage overflow interrupt. We denote the set of all overflow values for p as $O_{R,p} = \{O_{r,p} : \forall r \in R\}$

Our method is illustrated by an example in Figure 8.7. The figure depicts a process p_0 implementing a system-critical task that must maintain a high processing throughput. Process p_1 co-executes on the same core sharing some resources with p_0 such as FPU, L_1I-cache, L_1D-cache, and L_2-cache. Process

p_1 is swapped in at time ① and the OS sets an O_{R,p_1} where $R = \{\text{L}_1\text{I-cache}, \text{L}_1\text{D-cache}, \text{L}_2\text{-cache}, FPU\}$ so that an interrupt should be generated when any O_{r,p_1} is reached for $r \in R$. The scheduler starts p_1 and no further software-based monitoring is needed until the overflow is triggered. At ②, a $r \in R$ reaches its access limit, and the PMU hardware generates an interrupt handled by the OS process scheduler. The scheduler swaps out p_1 because it has depleted its resource quota and p_0 is swapped in at ③. Process p_0 does not have any resource restrictions (for the sake of reasoning) and will, therefore, run until ④ when it has depleted its CPU quota, i.e., as a normal process. At ⑤ process p_1 is once again swapped in and the PMU is programmed with O_{R,p_1}. The scenario continues in a similar manner with resource exhaustion in ⑥ and CPU-quota based context switching [192] in ⑦ and ⑧. The example shows a typical scenario when a process with high throughout demand is protected from shared resource disturbances via a PMU counter overflow mechanism.

8.2.8 Integrating all Parts

A successful and industrially applicable implementation of our allocation and scheduling method requires several steps. We find the resources used by p which has significant influence on the process performance. As an example, it is of limited use to enforce the L_1D-cache usage if ITLB is the main limiting resource for the performance of p. Second, we need to quantify the access limit $O_{r,p}$ after finding r with $\widehat{Corr}(R, x, p)$. The resource limit describes the needed number of accesses to resources for maintaining the desired performance. Third, we need a way to monitor the resource usage continuously for processes assigned to our allocation and scheduling framework. Fourth and last, we deploy our scheduler with the access limit values, $O_{r,p}$, determined by previous steps so that it can enforce the performance of p.

Finding the Resource to Enforce

The first thing to do when deploying our method is to determine what hardware resources, $\hat{r} \in R$, for p, has the highest correlation to the performance, x. Many experienced system designers know what resource has the highest effects on the performance which makes it easy to find $\hat{r}$. The system engineer may be biased and it is not always correct to determine $\hat{r}$ from his or her knowledge. We therefore suggest a different approach where we have formalized an unbiased procedure to determine $\hat{r}$. The procedure described in Listing 8.1 has been

tested in our experiments [157]. The total measurement time, t, and the sampling period δ depends on the system being monitored.

Listing 8.1: Determine what hardware resource has the highest correlation to the performance.

```
1  start p
2  % Iterate for a predefined time.
3  for time t do
4     % Measure all hardware resources and the process performance.
5     measure R and x for p
6     delay δ
7  endfor
8  find Corr(R, x, p) and r̂
```

We describe the procedure in Listing 8.1 as follows. We start the process being investigated in step ①. It is important that p is deployed in a test environment that represents the execution environment of the real production system. We continue, in steps ② and ③, by monitoring p for time t. Wait for a pre-defined time before measuring again, step ④. Conclude the procedure, in step ⑤, by finding the resource that has the highest correlation to the performance.

The procedure above describes how to find the resources, $\hat{r}$, with the highest correlation to the performance. Our technique is generalizable so that we can extract an arbitrary number of resources with their corresponding performance correlation values.

Finding the Resource Usage Limit

We want to determine the actual resource usage for $\hat{r}$ on p. The limit for $\hat{r}$ is denoted by $O_{\hat{r},p}$. We determine the resource usage limit for p by co-executing a process, l, that increasingly consume the same resource as p. This is similar to previous work on cache boundness [77–79] but our method is generic and applies to any hardware resource as long as it is measurable. We define a performance degradation limit, ω, i.e., a relational value when the process performance is degraded by shared resource congestion. For example, we use $\omega = 0.9$ if the lowest acceptable performance is 90% of the maximum performance.

Listing 8.2 shows the procedure for finding a resource usage limit with acceptable performance degradation. We begin the procedure by starting process p, step ①, in an execution environment that correctly represents the production environment. In step ② we measure the maximum capacity for process p during a configurable and system specific time t. The time is chosen so that p is

allowed to reach a stable execution state. We continue in step ⑥ by calculating the performance limit, x_{limit}, for p and then start l, in step ⑦, that use the same hardware resource as p. In steps ⑩ – ⑯ p and l runs concurrently while l periodically increase its hardware resource usage, r. We stop iterating when the performance of p drops below x_{limit}. We measure the current resource usage, m_r, by p in step ⑲ for the last iteration when the performance was still acceptable. We then assign the resource usage as the overflow value in step ⑳, $O_{r,p}$, which defines the resource usage needed by p to maintain the desired performance and QoS.

Listing 8.2: Determine acceptable hardware resource usage limits.

```
1  start p
2  % Measure the maximum capacity for process p.
3  for time t do
4    measure x
5  endfor
6  % Determine the lower performance limit.
7  set x_limit = ω * x̄
8  start leech l
9  % Increase the leech resource usage until the performance is too low.
10 repeat until x of p < x_limit
11   % Store the last resource usage value that works.
12   store x of p in x_prev.
13   % Increase resource usage.
14   inc l usage of r
15   measure x of p.
16   delay δ
17 endrepeat
18 % Read the desired overflow value from the PMU.
19 measure r of p for x_prev store in m_r
20 set O_r,p = m_r.
```

Resource Monitoring

Our current monitoring algorithm implementation works in a periodic round robin fashion, iterating over all monitored processes and the set of events of interest for those processes. More formally; let P be the set of processes that we want to monitor. Let E be the set of PMU events available on the given hardware platform and let C be the capacity of the PMU. To each process $p_i \in P$ we assign a set of PMU event sets E_i. Each event set $E_{ij} \in E_i$ is a subset of E, i.e. $E_{ij} \subseteq E$, such that $|E_{ij}| \leq C$. The following pseudo-code illustrates how our current implementation work:

Listing 8.3: PMU sampling process.

```
1  % Iterate over each process.
2  foreach process p_i ∈ P do
3    % Iterate over each PMU event set belonging to p_i.
4    for each PMU event set E_ij in E_i do
5      program event set E_ij into PMU
6      reset timer T
7      wait for timer T to expire
8      read values of PMU counters into V
9      % If configured, read the performance metric for p_i .
10     if mon_perf(p_i) then
11       read perf_value(p_i) into x_pi
12     else
13       x_pi = 0
14     endif
15     store tuple {i, E_ij, V, x_pi} in database DB
16   endfor
17 endfor
```

Our resource monitor utilizes the PMU to sample pre-configured events, see Listing 8.3. We start the PMU sample procedure by defining a set of processes to monitor ① and iterating through each of them. We then find and iterate over each PMU event set belonging to the selected process ④. The event is programmed ⑤ into the PMU, and a timer is programmed with the sampling interval ⑥. The monitoring application yields and wait for the timer to expire ⑦ and read ⑧ the PMU values. Read the application performance for p_i if the system administrator has configured the monitor to do so ⑩. We have provided the ability to omit application performance for system evaluation reasons. Reading the application performance ⑪ requires an API call to the application so that it is possible to quantify the performance. It is possible to run the monitor without performance measurements if a user only wants to understand the hardware usage of a system. The monitoring procedure ends by storing the hardware measurements together with the performance measurement in the local database ⑭. The monitor repeats the process during the time span we want to monitor the system, and the database contains historical and current performance and hardware usage of each $p_i \in P$.

Enforcing the Resource Usage

We have, in the previous sections, determined the resources most correlated to the performance, $\hat{r}$ and its resource usage limit, $O_{r,p}$, for a process p so that it can maintain its performance x within a defined range, ω. There are multiple ways to schedule processes in an OS. A typical OS such as Linux has several

Listing 8.4: Process scheduling.

```
1  % Iterate over all sched. policies in priority order.
2  foreach po ∈ PO in prio_order do
3    % Get a process from the sched.policy waiting queue.
4    p = get_process(po)
5    % Schedule the process if any, otherwise skip to next sched.pollicy..
6    if p then
7      % Suspend the current process.
8      swap_out(p_curr)
9      % Start the new process.
10     swap_in(p)
11     % If the process to swap in is assigned to SRA.
12     if sched_class(p) = SRA then
13       % Program PMU events with the hardware resource usage overflow limit.
14       program O_{r,p} into PMU
15       % Wait until the process has depleted its resource quota.
16       reset timer T
17       wait for timer T to expire or PMU overflow interrupt
18     endif
19   endif
20 endfor
```

scheduling policies in an effort to meet each process' execution environment demand.

Definition 30 The Linux OS implements a set of scheduling policies, PO. Each scheduling policy $po \in PO$ implements a specific scheduling method. A process, p, belongs to one and only one scheduling policy such that $po_p \in PO$.

Each process in the system belongs to only one scheduling policy, po. We have implemented and added our Shared Resource Aware (SRA) scheduling policy to the Linux kernel so that it contains the following policies: $PO = \{RT, EDF, SRA, CFS\}$ in reduced priority order. As an explanation: RT=real-time, EDF=earliest deadline first, and CFS is the normal time sharing policy. The OS scheduler walks through PO asking each policy for a process to run when the current process is swapped out. This order defines the scheduling priority among the policies, i.e. RT is most important, then EDF etc. The procedure shown in Listing 8.4 gives an overview of how process scheduling is implemented in Linux and how SRA fits into the current scheduling framework. The OS calls the procedure in Listing 8.4 when it decides that the currently executing process should be context switched, called being swapped out. There are several reasons for the OS deciding on a context switch. One is that a tick

has arrived at the OS means that the process has executed too long and the OS needs to decide what process to run. Another reason is that a higher priority process has been created or wants to execute. The current process can also be stalled waiting for an hardware response.

We start the procedure, in step ①, by iterating over all scheduling policies $po \in PO$ in priority order with the most critical policy first. The scheduler asks the policy for a process that wants to run, in step ③. If the scheduling policy does not have a process that wants to run it moves on to the next policy, in step ⑥. The current process is swapped out, ⑧, and the new is swapped in, ⑨. This step was the last step in the generic scheduler, and we now enter the SRA scheduler. Start by programming the PMU, ⑬, to generate an overflow interrupt if p overuse its hardware resource quota. Continue by setting a timer so that p can only execute for a determinist time, ⑮. We then wait, in step ⑯, for the timer to expire or p to overuse its resources.

The resource usage enforcement procedure, Listing 8.4, is implemented inside the Linux kernel, which contrasts to the user space implementation of resource monitoring, Listing 8.3.

8.3 Implementation

We have implemented a simplified version of our allocation and scheduling architecture, see Figure 8.2, as a proof of concept. The following subsections describe the implementation in detail and our compliance and deviations from the theoretical design.

8.3.1 System Monitoring

The main requirement for our implemented PM is to sample different levels of process performance and store the result in a database with low impact on the investigated system [241]. The PM uses a configuration file that specifies sample frequency, the R to measure and on what CPU/core.

Performance Monitoring Our implementation of the performance monitor measure the performance and writes the result to a text file a specific format. Each line in the file contains date-time and the value of one measurement, m_x. Each entry looks as follows:

2017-05-09.19:20:06.991009 2415072

The PM creates multiple files if we decide to measure the performance of several processes at the same time.

Hardware Resource Monitoring We implement hardware resource monitoring by using the PMU [226]. Our implementation uses the Perf-API [112] provided by the Linux kernel [188]. Even though the PMU complexity varies between chip manufacturers, there are many PMU events available. The sheer magnitude of event availability poses a challenging problem because there are far too many events to be simultaneously monitored using the available hardware counters [67]. A typical chip implements several hundreds of PMU events but only 4-6 simultaneously usable counters [141]. It is hard to automatically deduce what events to program [310] and what sampling period to use [222] so we have made those parameters configurable. We minimize the number of simultaneously running events because event multiplexing causes performance penalties [220] and probe effects [108]. It is also possible to use the PMU from within a virtualized environment [265], which is typically useful when debugging a faulty kernel implementation. We have to state that not all events are implemented in a virtualized environment making it difficult to debug all aspects of hardware resource monitoring.

The PMU is also referred to as a hardware performance counter [141]. Such hardware performance counter is a special-purpose register built into modern microprocessors, which counts the number of hardware related activities related to the currently configured event. Although there are some efforts in simplifying PMU usage [183], it is difficult to interpret and understand the effects of different R measurements, which is one reason for our implementation of a graphical visualization tool. The PM is implemented in C and uses the Perf-API [188] to sample PMU events. We have chosen to use a 100Hz sampling frequency as a trade-off between fine-grained granularity and avoiding probe effect [108]. The pre-configured PMU event set in Perf is limited, and we are therefore using raw event sets. The PM keeps track of the events to program and utilize their event-id. In theory, our test hardware supports more than 300 PMU [65] events but in practice far fewer are usable for process allocation purposes.

Figure 8.8 shows the output from the Charmon tool. The preamble shows some basics such as the Charmon version and compilation date closely followed by sampling configurations such as a boolean showing if the sampler is running at the time of dumping the information. Charmon also shows the configured sampling frequency and the number of slots in the internal DB. We have made it possible to limit the number of entries in the DB to ensure that Charmon will have little impact on the memory footprint for the system it monitors.

```
PROGRAM          Revision: version 1.0.2       Compiled               : Apr  7 2017 08:42:09
STATISTICS       Sampling: on                  Sample frequency [Hz]  : 100                    Eviction limit [#]       : 30000
[SAMPLES]
2017-04-09.19:20:11.500562,0,00.500561,7283,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.500563,1,00.500563,7283,ICACHE.MISSES,1245,L1D.REPLACEMENT,635,ITLB_MISSES.MISS_CAUSES_A_WALK,44,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,42
2017-04-09.19:20:11.500563,2,00.500563,7283,ICACHE.MISSES,4729,L1D.REPLACEMENT,1997840,ITLB_MISSES.MISS_CAUSES_A_WALK,21,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,149
2017-04-09.19:20:11.500564,3,00.500563,7283,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.500564,4,00.500564,7283,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.500565,5,00.500564,7283,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.500565,6,00.500565,7283,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.500565,7,00.500565,7283,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.511288,0,00.511288,7286,ICACHE.MISSES,1245,L1D.REPLACEMENT,457,ITLB_MISSES.MISS_CAUSES_A_WALK,68,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,50
2017-04-09.19:20:11.511289,1,00.511289,7286,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.511289,2,00.511289,7286,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.511290,3,00.511290,7286,ICACHE.MISSES,3095,L1D.REPLACEMENT,1917245,ITLB_MISSES.MISS_CAUSES_A_WALK,2,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,78
2017-04-09.19:20:11.511290,4,00.511290,7286,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.511291,5,00.511290,7286,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.511291,6,00.511291,7286,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.511292,7,00.511291,7286,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.522004,0,00.522004,7289,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.522005,1,00.522005,7289,ICACHE.MISSES,1318,L1D.REPLACEMENT,1302,ITLB_MISSES.MISS_CAUSES_A_WALK,10,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,69
2017-04-09.19:20:11.522006,2,00.522006,7289,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.522006,3,00.522006,7289,ICACHE.MISSES,1140,L1D.REPLACEMENT,588,ITLB_MISSES.MISS_CAUSES_A_WALK,52,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,67
2017-04-09.19:20:11.522007,4,00.522007,7289,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.522007,5,00.522007,7289,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.522008,6,00.522007,7289,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.522008,7,00.522008,7289,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.532568,0,00.532568,7292,ICACHE.MISSES,1135,L1D.REPLACEMENT,730,ITLB_MISSES.MISS_CAUSES_A_WALK,32,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,54
2017-04-09.19:20:11.532569,1,00.532569,7292,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.532569,2,00.532569,7292,ICACHE.MISSES,1261,L1D.REPLACEMENT,498,ITLB_MISSES.MISS_CAUSES_A_WALK,51,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,50
2017-04-09.19:20:11.532570,3,00.532570,7292,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.532573,4,00.532572,7292,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.532573,5,00.532573,7292,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.532573,6,00.532573,7292,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.532574,7,00.532574,7292,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.543175,0,00.543175,7283,ICACHE.MISSES,0,L1D.REPLACEMENT,0,ITLB_MISSES.MISS_CAUSES_A_WALK,0,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,0
2017-04-09.19:20:11.543176,1,00.543176,7283,ICACHE.MISSES,1235,L1D.REPLACEMENT,714,ITLB_MISSES.MISS_CAUSES_A_WALK,52,DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK,64
```

Figure 8.8: A few hardware resource samples extracted from the Charmon monitoring tool. The sampling frequency is 100Hz and Charmon is configured to limit the number of samples in the database to 30000.

Performance Analysis and Decision Making It is easy to obtain vast quantities of data by monitoring the performance a system. One of the major challenges is to analyze the data and understand where it is possible to improve the process allocation and scheduling. We have implemented the DE in Python by using various scientific libraries including Numpy [290], Pandas [203], Scikit-learn [229] and Armadillo [260, 261]. The DE finds and displays the Pearson correlation $Corr(r, x, p)$ for process p. The automatic interpretation of the Pearson correlation has not yet been fully implemented and we currently perform it manually. We use a simple decision-making technique based on decision tree [237]. Our approach starts by finding the r with highest correlance, $\hat{r}$, to x. If several processes have the same $\hat{r}$ we allocate the processes so that they do no execute on cores that have that particular shared resource.

Allocation and Scheduling Output Contracts The output from the DE is an allocation and scheduling contract provided in textual JSON format [232] as exemplified in Figure 8.9. The contract begins with the process called *testrun_-core0*, denoted p_0 in the following text. The contract stipulates that p_0 has allocation constraints such that: Process p_0 is allowed to use any physical CPU (-1) in the system but has the affinity $A_{p_0} = \{c_0\}$. The contract also contains scheduling constraints such as $O_{\text{L}_1\text{D-cache},p} = 1000000$ and $O_{\text{L}_2\text{D-cache},p} = 100000$ over a period length of 10 ms. The contract gives other instructions to the process allocator and scheduler for process *testrun_core0*, denoted by p_1. The contract does not specify any CPU constraints but p_1 is affined $A_{p_1} = \{c_1\}$.

8.3.2 Allocation and Scheduling Engine (ASE)

The Linux process scheduler, Completely Fair Scheduler (CFS), keeps track of the historical execution time for each process. CFS aims to achieve high performance with low overhead while at the same time enforce priority-based scheduling and quick feedback for user-interactive processes. The ASE is implemented in C and interprets the allocation contract created by the DE. We have implemented the process allocation part of the ASE in user-space by using the Linux `sched_setaffinity()` kernel function to update the process core affinity bitmask, which results in the desired process allocation. This type of affinity implementation is commonly available and used in other projects [26].

```
[
  {
    "Name"  : "Sample allocation of proc #0",
    "Descr" : "This proc is a heavy cache user and want
               to execute on core 0.",
    "Pid"   : "testrun_core0",
    "Allocation" : [
      { "CPU"   : "-1" },
      { "Core"  : "0" }
    ],
    "Constraints'' : [
      { "Period"     : 10 }
      { "L1-DCache" : 1000000 },
      { "L2-DCache" : 100000 }
    ]
  },
  {
    "Name"  : "Sample allocatation of proc #1",
    "Descr" : "This proc is a heavy CPU user and want
               to execute on core 1.",
    "Pid"   : "testrun_core1",
    "CPU"   : "-1",
    "Core"  : "1",
  }
]
```

Figure 8.9: Process allocation and scheduling output contract in JSON format.

8.3.3 Implementing a Process Allocator

Our process allocator continuously monitors the hardware resource usage and application performance of the designated process. The process allocator uses the measurements to find a correlation between hardware usage and application performance. We have shown that it is possible to create a system that uses pre-programmed hardware architecture information, such as cache structure, together with the $Corr(r, x, p)$ correlation results to efficiently distribute processes over the CPU core cluster [157]. The Linux process scheduler is described in an article about the Lottery process scheduler [205].

We have used some basic Linux support for hardware resources governing such as:

- Support for hardware performance monitoring through the user- and kernel-space Perf-API.
- Core affinity support makes it possible to allocate individual processes to specific cores. The affinity functionality is reached through the system calls `sched_setaffinity()` and `sched_getaffinity()`.

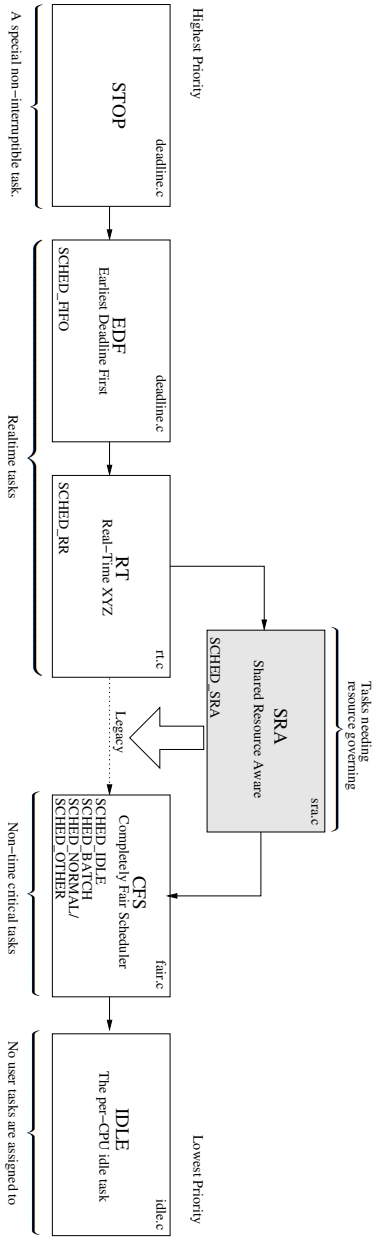


Figure 8.10: Linux scheduling policy priority.

8.3.4 Implementing a Process Scheduling Policy

We have added SRA to kernel version 4.6.2, it was the latest kernel when we started our kernel development. Changing the behavior of the Linux process scheduler is not an easy task [173, 174], from an implementation point-of-view. The code is complex and difficult to grasp. Austad describes the inner workings and structure of the Linux kernel, especially the scheduling framework [18]. The kernel structure is even further described in by Mauerer [200]. The Linux scheduler contains roughly 30000 SLOC [216] and the current, unoptimized, version of SRA adds 4300 SLOC to this. There has been many attempts to improve the scheduling performance, such as making the scheduler heterogeneous aware [38] and others have added new scheduling policies, such as the Lottery scheduler [205].

We have added hardware resource tracking support in kernel space by utilizing the PMU inside the process scheduler. Our new scheduling policy fits within the standard Linux scheduling framework [96]. The scheduling framework supports multiple concurrent scheduling policies, spanning from high-priority real-time behavior to low-priority batch processing. Each scheduling policy is implemented by a scheduling class, as shown in Figure 8.10. The most common scheduling policy in Linux is the CFS [172], which handles time-sharing processes. We have inserted a new SRA scheduling policy after RT to make sure that real-time tasks has higher priority. We added SRA before CFS so that SRA can throttle resource usage with higher priority than ordinary Linux processes. We have implemented the new scheduling policy as a separate scheduling class, named `sched_sra_class`. The framework explicitly defines necessary functionality for each class.

Processes can move freely between the different scheduling policies by using the `chrt` shell command. Such a command triggers the kernel to call the `switched_from()` function in the currently assigned scheduling class, asking it to remove the process from its internal storage. First, the scheduler removes the process from one class and then add it to the new scheduling class by calling the `switched_to()` function. There are many other functions implemented for each scheduling class but describing them is out of the scope of this text.

The process scheduler calls the generic `pick_next_task()` function whenever deciding that a new task should be swapped-in. The generic function iterates over a linked list containing all available scheduling classes. That iteration causes a priority between the scheduling classes, starting with EDF

and ending with the idle loop. As displayed in Figure 8.10 we have opted to insert our SRA scheduling class between RT ad CFS.

8.4 Experiments

We have designed two experiments to verify our allocation and scheduling framework. We verify several functional properties in each of the experiments. In the first, Section 8.4.1, we verify that the process allocator efficiently distributes processes over a CPU cluster. In the second experiment, Section 8.4.2, we instruct the process scheduler to preserve QoS for a process that runs in a complex shared resource environment. We have disabled CPU frequency scaling in all experiments to let the CPU run at the maximum frequency during our tests and Table 8.1 shows the hardware we have used. We ran each experiment multiple times with similar results. The results presented below are from individual test runs because they illustrate both resource and performance variations better than an average value.

8.4.1 Testing Automatic Process Allocation

The goal of this experiment is to ensure that we can reach a good process allocation automatically for multiple processes that utilize shared hardware resources. We use two types of processes in this experiments. The first process type, p_{mem}, is memory-bound which means that the performance of the process depends heavily on its ability to access memory. The p_{mem} process simulates a packet processing application by striding (reading and writing) through memory, see Section 6.3.3. The other process type, p_{cpu}, is CPU-bound and iterate in a short loop performing integer operations, resulting in a lower cache usage but heavy CPU usage. A p_{cpu} process simulates an application which performance depends on the availability of computational capacity in the CPU. Our target environment typically contains several message processing applications that co-exist with other processes consuming processing power for various reasons like calculating checksums, busy-waiting for external input and similar tasks.

Setup Our test system uses an Intel® Core™ i7-4600U CPU, see Table 8.1. We simulate the execution environment of a production system by simultaneously running four processes $P = \{p_1, p_2, p_3, p_4\}$. Processes $\{p_1, p_2\}$ are of type p_{mem} and $\{p_3, p_4\}$ are of type p_{cpu}. We have configured each p_{mem} to stride through a 128 KB working set, which is larger than the L_1D-cache

Feature	i7-4600U [52,53]	i7-4980HQ [54,55,158]
Core	2xIntel® Core™ i7-4600U CPU (Haswell/SharkBay) at 2.10GHz (4 virtual threads)	4xIntel® Core™ i7-4980HQ CPU (Haswell/Crystalwell) at 2.80GHz (A total of 8 threads)
L_1-cache	32 KB 8-way set assoc. instruction caches/core + 32 KB 8-way set assoc. data cache/core	32 KB 8-way set assoc. instruction cache/core + 32 KB 8-way set assoc. data cache/core
L_2-cache	256 KB 8-way set assoc. cache/-core	256 KB 8-way set assoc. cache/-core
L_3-cache	4 MB 16-way set assoc. shared platform cache	6 MB 12-way set assoc. shared platform cache
MMU	64 Byte line size, 64 Byte Prefetching, DTLB: 4 entries 1 GB 4-way set assoc., DTLB: 64 entries 4 KB 4-way set assoc., ITLB: 64 entries 4 KB 8-way set assoc., L_2Unified-TLB: 1 MB 4-way set assoc., L_2Unified-TLB: 1024 entries 4 KB/2 MB 8-way assoc.	DTLB: 64 entries 4 KB 4-way set assoc., 32 entries 2 MB/4 MB 4-way set assoc., 4 entries 1 GB 4-way set assoc., ITLB: 128 entries 4 KB 4-way se assoc., 8 entries per hardware thread 2 MB/4 MB L_2Unified-TLB: 1024 entries 8-way 4 KB/2 MB system wide

Table 8.1: hardware specifications for our test systems.

(32 KB) but sufficiently small to fit inside the L_2D-cache (256 KB), even when adding the additional data needed by the test application. We want to avoid undesirable spillover into the shared system-wide L_3-cache since that introduce a performance dependency towards an additional cache level. We have implemented a heuristic model in the DE, see Section 8.2, that allocate processes that depends on the same hardware resource as efficiently as possible. The target hardware has two virtual execution threads per physical CPU core. For convenience, we denote each virtual thread as a core in the same way as Linux perceives the hardware, i.e., such that the CPU contains $\{c_0, c_1, c_2, c_3\}$. The hardware model contains a subset of the information in Figure 8.1, such as the definition of CPU core clusters $\{c_0, c_1\}$ and $\{c_2, c_3\}$. Each CPU core cluster shares a L_1I-cache, L_1D-cache, L_2-cache. The model also contains information

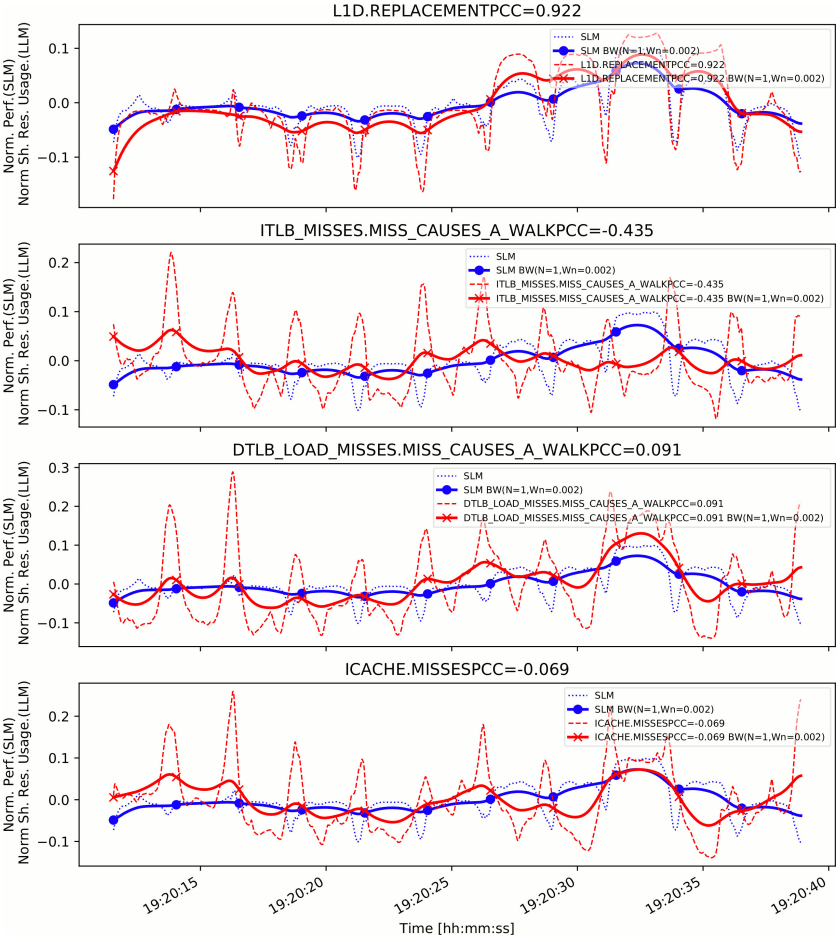


Figure 8.11: The graphs show the resource usage $R = \{\text{L}_1\text{D-cache}, \text{ITLB}, \text{DTLB}, \text{L}_1\text{I-cache}\}$ for process p_1 and the $Corr(r, x, p_1)$. The performance, x, is defined as the number of packets processed/sec. The dashed lines show individual measurements while the unbroken line shows measurements that has been passed through a Butterworth low-pass filter to remove oscillations.

Table 8.2: TC1: The sorted correlation between each $r \in R$ and x when running processes p_1 through p_4 using standard Linux CFS process scheduler.

Resource index	p_1	p_2	p_3	p_4	**Resource**(r)
0	0.92	0.91	0.41	0.41	L_1D-cache
1	0.43	1.19	0.25	0.07	ITLB
2	0.09	0.31	0.15	0.08	DTLB
3	0.07	0.29	0.28	0.45	L_1I-cache

of the unified and system-wide L_3-cache. We have performed our tests on an Ubuntu 14.04 x86_64 Linux system running a 3.13 kernel.

We have used the method described in Listing 8.1 to find the hardware resources most correlated to the performance. We have opted to monitor 4 resource events continuously per process to minimize the system performance impact. We define $R = \{L_1\text{D-cache}, \text{DTLB}, L_1\text{I-cache}, \text{ITLB}\}$ and measure R individually per processes and core. We normalize the sampled data and use a Butterworth [243] low-pass filter to remove high-frequency ripples before calculating the correlation between R and x.

Running TC1 We start the processes in the reference test, denoted TC1, without any core-affinity. The OS-scheduler is allowed to migrate processes between all cores in the system freely. We have assigned all p_{mem} and p_{cpu} processes to the SCHED_NORMAL class in the Linux process scheduler. The PM monitor continuously measures R and x for each process in P on all 4 cores.

Evaluating TC1 We have configured the allocation framework to evaluate the process allocation after running the system for 30 seconds. The PM measures R and x for all processes in P and forwards the result to the DE which correlates

Table 8.3: TC2: The sorted correlation between each $r \in R$ and x when running processes p_1 through p_4 using the SRA process allocator.

Resource index	p_1	p_2	p_3	p_4	**Resource**(r)
0	0.98	0.89	0.12	0.21	L_1D-cache
1	0.70	0.13	0.83	0.57	ITLB
2	0.81	0.24	0.58	0.64	DTLB
3	0.59	0.13	0.21	0.40	L_1I-cache

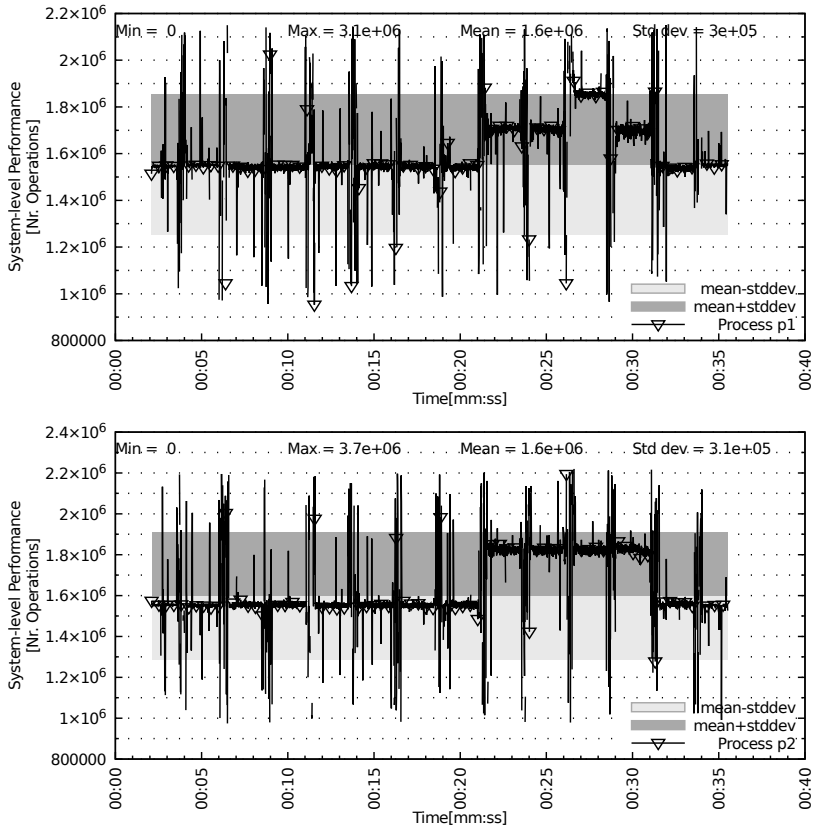


Figure 8.12: Running p_1 with the Linux CFS scheduler results in $x = 1.6M$(top), and for p_2 $x = 1.6M$(bottom).

the values. Figure 8.11 shows measurements of R and x for p_1. We have sorted the graphs according to their $Corr(r, x, p_1)$ with the highest value for the top graph. We deduce that the performance of p_1 has the highest correlation to L_1D-cache because of the high $Corr(L_1\text{D-cache}, x, p_1)$ correlation value. The complete list of R and $Corr(r, x, p_1)$ is displayed in Table 8.2. From the list, we can further deduce that p_1 and p_2 depends on the L_1D-cache because both have high correlation values. The DE uses our heuristic model, together with the

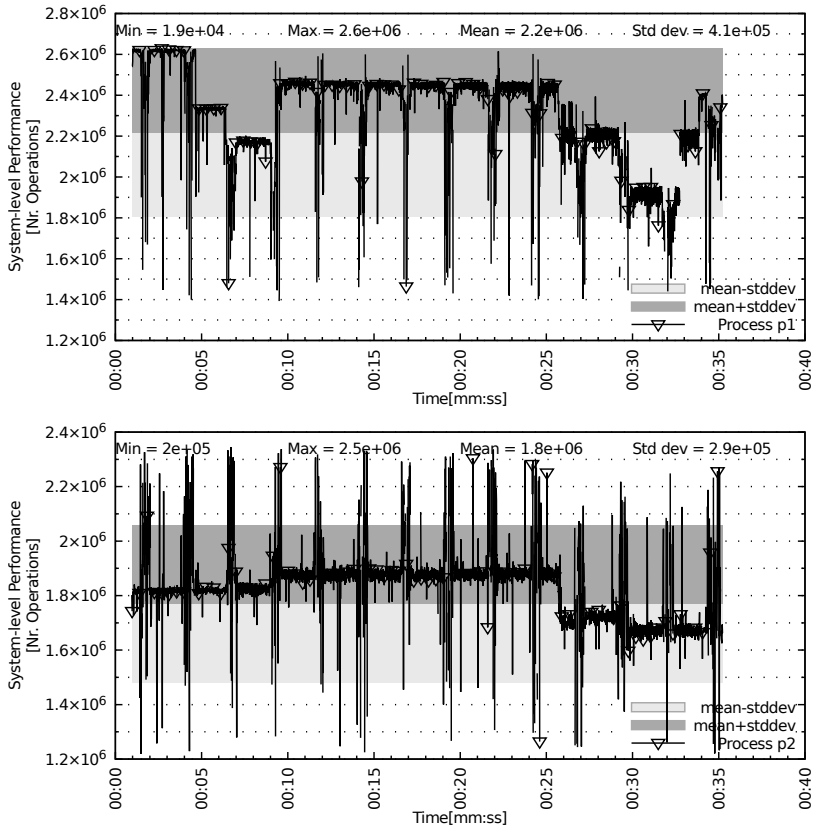


Figure 8.13: Using our shared resource process allocator results in $x_{p_1} = 2.2M$(top), $x_{p_2} = 1.8M$(bottom).

correlation information to decide that p_1 and p_2 should be affined to different L_1D-cache clusters.

Running TC2 In test, TC2, we want to reduce the L_1D-cache congestion by utilizing different cache clusters for p_1 and p_2. We start our scheduling framework to let the DE dynamically affine processes according to the heuristic and input from the PM. The effect is that DE affines $A_{p_1} = \{c_0\}$ and $A_{p_2} = \{c_2\}$. The DE allocate p_3 on $A_{p_3} = \{c_1\}$ because p_3 has low correlation to

L_1D-cache, it is of type p_{cpu}. The same rule applies to p_4 which is allocated accordingly $A_{p_4} = \{c_3\}$. The average performance for p_1 in TC2 is $x = 2.2$ and for p_2 the performance is $x = 1.8$.

Evaluating TC2 We start by evaluating R for p_1. We deduce that p_1 executed on core 2 by examining the measurements for R. The CFS process scheduler allocates, TC1, our test processes, p_1 and p_2, in such a way that the measured mean performance is $x_{p_1} = 1.6M$ and $x_{p_2} = 1.6M$ operations/sec. Details of the test application performance is shown in Figure 8.12. Allocating processes p_1 and p_2 using our proposed scheduling architecture, TC2, results in a drastic performance improvement where $x_{p_1} = 2.2M(+37.5\%)$ and $x_{p_2} = 1.8M(+12\%)$ operations/sec, see Figure 8.13. On average the performance increase for p_1 and p_2 is 20%. The main reason for the performance improvement is the reduced cache contention when moving p_1 and p_2 to cores that do not share L_1-cache. Our shared resource allocation architecture considers the adverse effects of shared resource contention and enforces the process reallocation.

Remarks Our tests show that going from scenario 4 (TC1) to 5 (TC2), illustrated in Figure 8.6, improves the performance of two memory-bound applications by 20% [157] (x_{p_1}: $1.6 \rightarrow 2.2$ and x_{p_2}: $1.6 \rightarrow 1.8$). Comparing the correlations for CFS (TC1), in Table 8.2, and SRA (TC2) in Table 8.3 shows that the increased performance in TC2 changes the balance of correlance. The increased performance utilization introduce ITLB, DTLB and L_1I-cache as the next bottlenecks for an event higher performance.

8.4.2 Testing the QoS Aware Process Scheduler

The goal of this test is to verify that the SRA process scheduler can ensure QoS by enforcing shared resource usage in a scenario where multiple processes compete for the same resource.

Test Setup

We have used the same hardware for all experiments in this test. Our test system is an Intel® Core™ i7-4600U, see Table 8.1. We have turned off CPU frequency scaling and reduced the performance impact of other processes by starting as few services as possible. The test system runs Ubuntu 14.04 x86_64 Linux system with a 4.6.2 kernel. We will further describe the exact test setup for each test case.

Leeches

We extensively use leeches in the following experiment. A leech is in the context of this thesis a process that heavily uses system resources such as caches, CPU or FPU. Benchmarking suites exploit this mechanism to generate hardware load on a system to determine its capacity. We have created three type of leech applications. The first leech generate substantial CPU load by iterating through a tight inner loop. The second leech generate heavy load on caches and the memory subsystem by iterating through several loops reading from and writing to memory. We can configure the stride to generate various access patterns as to saturate either the L_1D-cache, L_2D-cache or other levels. The third leech iterate through a tight loop executing FPU instructions. We exclusively use memory bound leeches in all the following tests.

TC1 : Enforcing QoS for a Media Player

It is common to use the MPlayer [292] multimedia player when verifying resource allocation scheduling algorithms [38, 308]. Other researchers [133] have concluded that MPlayer uses a significant amount of data when playing high-quality movies. MPlayer is a suitable test candidate for the SRA process scheduler because our target system also uses large volumes of data. MPlayer is also useful for many other memory-bound applications that require a stable execution environment and a minimum QoS level.

Setup We have configured MPlayer to show a high-resolution movie which causes heavy load on the memory subsystem due to the large volumes of data needed by the video/audio decoder. It is possible to measure the QoS for MPlayer in several ways. One way is to measure the used decoder memory bandwidth [267, p125]. Another way is to measure the perceived QoS by continuously measuring the number of lost Frames Per Second (FPS). We have chosen the latter performance metric because it efficiently describes the user-experienced application QoS. There are several ways to reserve resources, and the most common is to allocate a certain amount of computational power in the CPU for a specific media player [206, 207] while running some other application that tries to "steal" as much of the available resources as possible. A successful scheduling algorithm has a high QoS level [57] by showing a low number of lost FPS compared to less reliable process schedulers with a high number of lost FPS.

Execution We have followed the process described below:

1. Run MPlayer in a continuous loop showing the movie while storing the number of dropped FPS in a file. We use a 24 FPS MPEG4 test movie with a filesize of 81 MB, resolution 1920x1080, a bitrate of 10.4 Mbps, and a total playing time of 1 min 2 sec.
2. Start phase 1 (CFS with only MPlayer)
   (a) Continuously measure the number of dropped FPS.
3. Begin phase 2 (CFS with memory congestion)
   (a) Starting four leeches with 10 MB data working set each.
   (b) Continuously measure the number of dropped FPS caused by the leech memory congestion.
4. Start phase 3 (SRA with memory congestion)
   (a) Move MPlayer from CFS to SRA scheduling class.
   (b) Configure SRA to enforce L_1D-cache capacity for MPlayer such that $O_{\text{L}_1\text{D-cache},MPlayer} = 10^6$, meaning that we configure the PMU to generate an overflow interrupt, causing a process context switch, at 10^6 L_1D-cache replacements. We configure SRA to renew the quota every 10ms.
   (c) Continuously measure the number of dropped FPS caused by the leech memory congestion.

We empirically deduced the settings for the SRA scheduler. Using a too large access limit value does not yield any higher performance for MPlayer but limits other processes from executing. With similar reasoning, using a too small access limit value reduces the MPlayer performance.

Evaluation The reference measurement in phase 1 shows high QoS and MPlayer runs without any lost FPS. During phase 2, MPlayer shows clear signs of lower QoS by losing 10 FPS when it runs under CFS together with the leeches. The user can clearly see that the movie is choppy and shows clear signs of an overloaded system. Phase 3 restores the high QoS by switching MPlayer to SRA and reduce the lost FPS to 0. The movie plays smoothly, in the same way as phase 1 when running without leeches.

Remarks It is apparent that MPlayer does not perform well when coexisting with other application (leeches) that are also memory bound. The CFS scheduler does not provide enough shared resource isolation to retain the high QoS when multiple processes compete for the same shared hardware resource. Scheduling

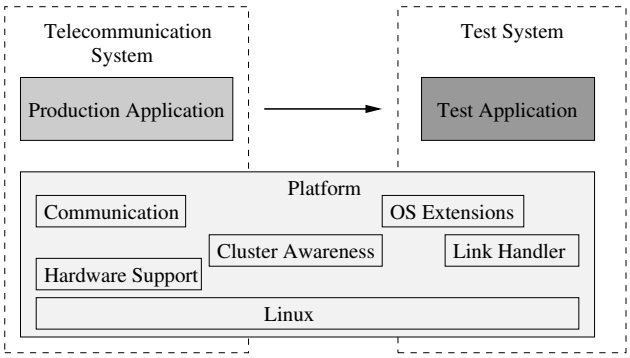


Figure 8.14: The production application is replaced by a test application while preserving the same platform implementation.

the MPlayer with SRA ensure the availability to the configured quota of the demanded resources. In this experiment we have ensured the availability to L_1D-cache.

TC2 : Enforcing QoS for a Telecommunication Application

The goal of this test case is to verify that it is possible to enforce resource usage resulting in a higher QoS level than CFS can provide.

Setup In this experiment we simulate a real-world execution scenario by emulating a real system, as depicted in Figure 8.14. The production application is huge, see Section 2.5, and is very complex to evaluate. We have therefore used the test application setup described in Chapters 5 and 6. The main functionality of the production application is to receive messages, process and then forward (send) them to another computer in the network. Processing messages cause heavy cache usage and memory bus congestion, as described in Chapter 6. We have emulated this behavior by using a system test application replicating the behavior of the production system. Apart from the application we use the same general system setup as the production application, see Figure 8.14. The system runs on Linux, and several extensions provide cluster awareness, communication capabilities, and other OS services to run efficiently.

Execution The test execution is described in the following procedure:

1. Start OS-extensions such as the process handling, IPC, communication and link handler daemons.
2. Start the test application, $appl$, and initiate message communication and processing.
3. Continuously measure the system level performance, x for $appl$, defined as the message round-trip time.
4. Start phase 1 that measure the performance of $appl$ when it runs by itself using CFS.
5. Reset message counter, $cnt = 0$.
6. When $cnt = 10^6$, start two leeches, each with 10 MB data workset.
7. Start phase 2 that measure the shared resource effects of $appl$ co-existing with leeches on the CFS scheduler.
8. When $cnt = 2 * 10^6$, move $appl$ from CFS to SRA.
9. Configure SRA to enforce resource usage. Start by ensuring that there are enough data cache capacity available for $appl$ such that $O_{\mathrm{L_1D\text{-}cache},appl} = 10^6$. Renew each quota every 10ms.
10. Start phase 3 that measure the shared resource effects of $appl$ co-existing with leeches on the SRA scheduler.
11. When $cnt > 3 * 10^6$, conclude the test.

Evaluation phase 1 : Running $appl$ without additional load We can see the reference performance to the left ($cnt = [0, 10^6]$) in Figure 8.15 where $appl$ runs on its own using the CFS scheduler. The average message RTT is approx. $x_{phase1} = 42.9\mu s$ for $appl$ and acts as a reference value of the highest achievable performance.

Evaluation phase 2 : Introducing additional load Introducing additional cache usage, through leeches, stresses the system and reduce the performance of $appl$. It is obvious that CFS cannot guarantee the QoS that $appl$ desire, as shown ($cnt = [10^6, 2 * 10^6]$) in Figure 8.15. The performance for $appl$ has dropped drastically to $x_{phase2} = 137.6\mu s$, which is an message RTT increase of $\frac{x_{phase2}}{x_{phase1}} = \frac{137.6}{42.9} = 321\%$.

Evaluation phase 3 : Enforcing QoS by running $appl$ in SRA We switch scheduling policy for $appl$ from CFS to SRA at $cnt = 2 * 10^6$ causing SRA to enforce the L_1D-cache. The resulting performance is $x_{phase3} = 81.7\mu s$, which

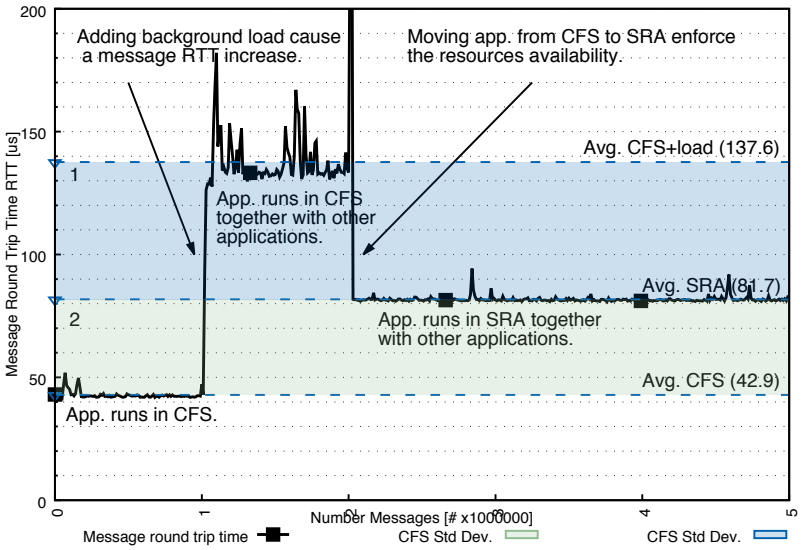


Figure 8.15: Enforcing hardware resources.

is substantially better than phase 2 and the message RTT is, compared to phase 2, reduced by $\frac{x_{phase2}-x_{phase3}}{x_{phase2}} = \frac{137.6-81.7}{137.6} = 40.6\%$.

Remarks We can draw some conclusions from this experiment. The first conclusion is that SRA can enforce QoS because x_{phase3} is significantly smaller than x_{phase2}, denoted by the colored field ① in Figure 8.15. The second conclusion is that x_{phase1} is significantly better than x_{phase3}, denoted by the colored field ② in Figure 8.15. There are several reasons for *phase 1* being more efficient than *phase 3*. We will further investigate the scheduling performance of CFS vs. SRA in TC3 and the performance variations over time in TC4.

TC3 : The Cost of Enforcing QoS

The goal of this test case is to mesure the scheduling cost of using SRA compared to CFS.

Setup We use the same test setup as in TC2.

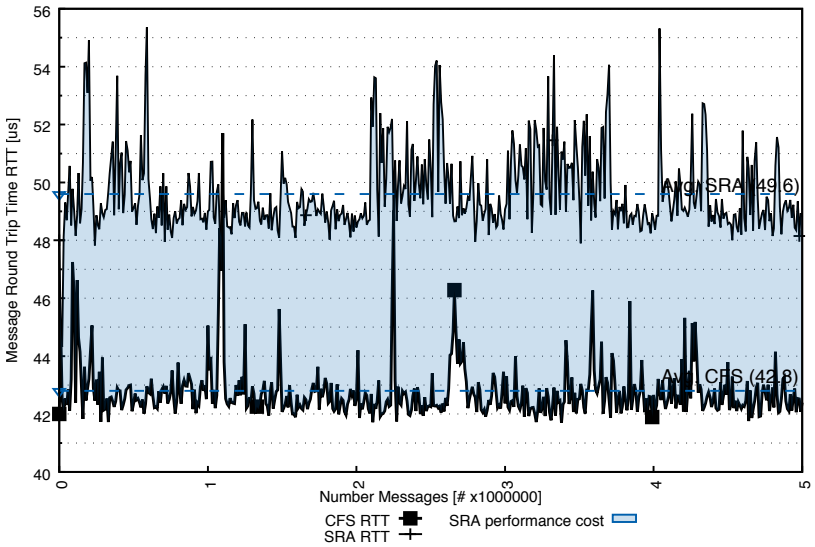


Figure 8.16: SRA vs. CFS scheduling performance.

Execution The setup procedure is as follows:

1. Start OS-extension, communication and link handler daemons.
2. Start the test application, $appl$, and initiate message communication and processing.
3. Continuously measure the system level performance, x, for $appl$ defined as the message round-trip time.
4. Reset message counter, $cnt = 0$.

In phase 1 we run the setup procedure and then the following:

1. Start phase 1 that measure the performance of $appl$ when it runs by itself on the CFS scheduler.
2. Stop the test when $cnt = 5 * 10^6$.

In phase 2 we run the setup procedure and then the following:

1. Start phase 2 that measure the shared resource effects of $appl$ when it runs by itself on the SRA scheduler.

2. Configure SRA to enforce resource usage. Start by ensuring that there are enough data cache capacity available for $appl$ such that $O_{\text{L}_1\text{D-cache},appl} = 10^6$. Renew each quota every 10ms.
3. Stop the test when $cnt = 5 * 10^6$.

Evaluation Scheduling the test application with SRA results in a message RTT $x_{SRA} = 49.6\mu s$ while CFS has $x_{CFS} = 42.8\mu s$, see Figure 8.16, meaning that scheduling in CFS is $\frac{x_{SRA}-x_{CFS}}{x_{CFS}} = \frac{49.6-42.8}{42.8} = 15.9\%$ more efficient than SRA.

Remarks There are several possible reasons for the performance degradation in SRA compared to CFS. For example:

1. We have only restricted the resource usage for L_1D-cache, thus other resources will still cause congestion-effects and ultimately the performance of $appl$.
2. Another issue that affects the SRA scheduling performance is that the SRA implementation is far from optimized. We removed several debug statements before performing these test cases, which reduced the SRA scheduling cost by approx. 30%. We estimate that it is possible to drastically improve the SRA performance by going through a normal productification procedure. We typically profile the code to find optimization opportunities. In general, we cannot see any reason why the runtime performance of SRA should be much different than CFS.

TC4 : Performance Variation in SRA is Smaller Than in CFS

The goal of this test case is to investigate how the application performance varies over time for CFS compared to SRA. A high performance variance can lead to coarse grained response times. It is therefore desirable to have a stable execution time.

Setup We use the same test setup as in TC2.

Execution We have divided our tests into two phases with a common setup procedure:

1. Start OS-extension, communication and link handler daemons.
2. Start two leeches, each with 10 MB data workset.

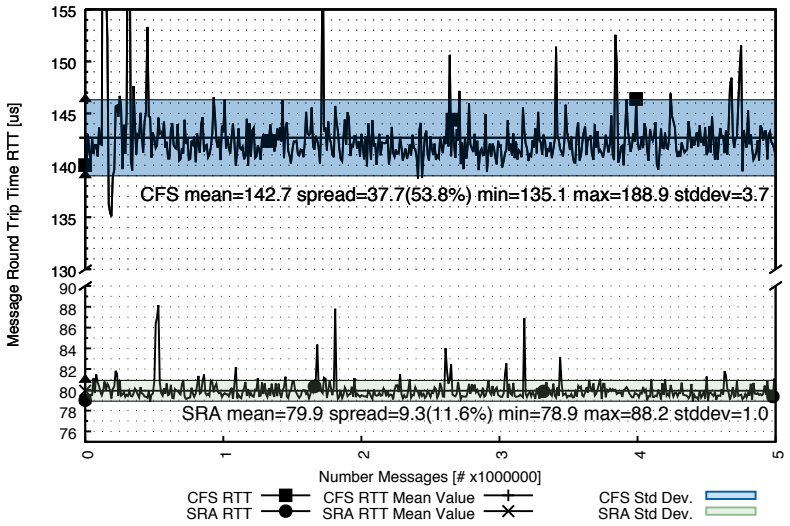


Figure 8.17: The application performance varies substantially running CFS compared to SRA.

3. Start the test application, $appl$, and initiate message communication and processing.
4. Continuously measure the system level performance, x for $appl$, defined as the message round-trip time.
5. Reset message counter, $cnt = 0$.

Phase 1: Run the setup procedure and then continue with the following steps:

1. Continuously measure the performance x_{CFS} when $appl$ runs with load using the CFS scheduler.
2. When $cnt = 5 * 10^6$, stop the test.

Phase 2: Run the setup procedure and then continue with the following steps:

1. Configure SRA to enforce resource usage. Start by ensuring that there are enough data cache capacity available for $appl$ such that $O_{\mathrm{L_1D\text{-}cache},appl} = 10^6$. Renew each quota every 10ms.
2. Continuously measure the performance x_{SRA} when $appl$ runs with load using the SRA scheduler.

3. When $cnt = 5 * 10^6$, stop the test.

Evaluation From the measurements in phases 1 and 2 it is apparent that the variations in SRA is substantially smaller than when using CFS, see Figure 8.17. The performance of $appl$ when using SRA is $78.9 \leq x_{SRA} \leq 88.2\mu s$, which has a span of $max_{SRA} - min_{SRA} = 9.3\mu s$. CFS varies much more, $135.1 \leq x_{CFS} \leq 188.9\mu s$, and has a span of $max_{CFS} - min_{CFS} = 37.7\mu s$.

Remarks We conclude that SRA manages to restrict the resource congestion caused by other processes. The reduction of congestion leads to a more stable execution environment and less performance fluctuation. Such considerable performance variation will have an impact on the application performance granularity, even when achieving the same mean performance. A significant performance variation may lead to missed deadlines if the response time cannot be met due to resource congestions. In this sense, SRA is a much better choice than CFS.

8.5 Related Work

Running a process, p, on a single-core system is an ideal situation from a performance evaluation perspective and is thoroughly evaluated for decades [191].

There are numerous different scheduling algorihtms [267] targetting various scheduling domains and system requirements [250]. Most process schedulers only act on execution time and do not consider the shared hardware resource usage. This type of scheduler assumes that execution time, t_{exec}, correlates to application performance, such that $Corr(t_{exec}, x, p)$. One benefit of this assumption is that it simplifies process scheduling. The execution time is easily comparable between all processes in a system. The weakness of the model is that t_{exec} does not always model the application performance. We show examples of several widely used scheduling algorithm in the following sections.

Deadline Scheduler There are various types of deadline schedulers. The Earliest Deadline First scheduler dynamically tracks processes so that the one with the shortest time to its deadline gets the highest priority. There are various types of real-time schedulers that tries to solve the problem of shared resource congestion (but not explicitly), such as EDF [57]. Another related area is how to control QoS. Many attempts have been made to control QoS, for example, QoS control within an OS [239] or QoS control within a middleware [246]. There is an EDF

implementation [97, 181] in Linux since kernel revision 3.14. RT-Muse [196] is one of the tools available for measuring the real-time characteristics of a system scheduler.

Fixed Priority Scheduler Many researchers attribute fixed-priority scheduling to the seminal paper published 1973 by Liu and Layland [191] although some researchers [17] claims that there are several earlier attempts to implement scheduling policys [51]. A preemptive fixed-priority process scheduling [191] ensures that at any given time out of all runnable processes in the system, the process with the highest priority executes.

Shared Resource Allocation and Scheduling The general idea behind resource aware process scheduling is to use any measurable metric when making scheduling decisions. CPU-time is traditionally used by process schedulers, but our opinion is that we should expand the reasoning to include any hardware metric.

A central issue for resource-aware schedulers is to efficiently measure the resource usage for each monitored process [257]. The scheduler can then use the resource usage when making allocation or scheduling decisions. The demand for heterogeneous CPUs has increased due to requirements of diversified application loads [210]. This demand causes, according to Bower et al. that schedulers needs to be aware of dynamically changing heterogenicity [32]. One process scheduler instance may span a big CPU-core cluster where the core capacity varies over time due to the utilization situation. A process scheduler can decide if a process should execute on a high-capacity or low-capacity core of a heterogeneous CPU by monitoring the number of execution stalls generated inside or outside the CPU [175]. It is possible to reach a similar conclusion by measuring the level of cache misses [38] and use the outcome as input to the scheduler. Some researchers measure the number of processor cycles used by a process [206, 207] and use it to schedule processes efficiently. We have not addressed heterogeneous hardware explicitly, but our opinion is that we can handle cores with varying capacity with our allocation and scheduling framework.

Other researchers have investigated how memory contention affects the performance of applications sharing an execution environment. One promising technique is to use the PMU to measure the memory bandwidth utilized by a process [136]. Using the memory bandwidth simultaneously with other metrics is possible by using a multi-resource server [137], which is demonstrated [135] by using the ExSched [16] framework in Linux. The concept of multi-resource servers is also usable when consolidating several software components on com-

mon hardware [19]. The main difference with these implementations is that they are implemented as user-space implementations and suffer from performance problems. Our scheduler executes inside the Linux kernel as has, therefore, much higher performance.

An efficient allocation of processes reduces the intercore [116] congestion while process scheduling reduce the intracore and interprocess congestion levels. Apart from process scheduling there are other techniques to reduce the effects of shared cache congestion such as page coloring [123] and cache partitioning [211]. We have not investigated these techniques because our target hardware do not always support for them.

Blagodurov et al. [26] states that cache and memory congestion is the main performance bottleneck for many system. They have implemented a scheduling algorithm for Linux that evenly distribute memory accesses over the available cache. Their distributed intensity online algorithm manage the scheduling of each process by assigning a value that quantifies the process' memory miss rate. There are other techniques to provide cache aware process scheduling. Knauerhase et al. [169] have devised a method to observe the cache usage of processes and schedule cache-intensive processes on cores with separate cache clusers. Our approach differs from Knauerhause because it automatically detects which of the shared hardware resources that have the greatest effects on the system performance. Our mechanism uses this correlation to optimize process allocation and scheduling parameters.

High Performance Computing (HPC) Feitelson [98] stated already 1995 that *"The main issue is how to share the resources of the parallel machine among a number of competing jobs, giving each the required level of service"*. This sentence condense the main idea for high performance computing (HPC) systems. The drastic development of HPC-tools and techniques is somewhat related to our research. One such example is the automatic memory bandwidth monitor [35]. Apart from monitoring the memory bandwidth, it can also allocates applications over a set of CPU cores to maximize the performance. The same research team shown that application co-location can drastically reduce power consumption while retaining performance [34]. Our techniques are more generic and can monitor any hardware resource and is, therefore, not limited to only the memory bandwidth.

Xiong defines and implements a method to schedule processes in a multi processor system according to their SLA [305] by aiming to minimize the cost of computation. Job scheduling is a large research topic and it incorporates three main areas [305]. The first area is how jobs are assigned to processors.

The second area is in what order a set of jobs should execute. The third area is how to assign enough resources so that the jobs can run efficiently and ensure the QoS requirements. The first area defines the problem we have addressed by our process allocator method. The second and third areas are addressed by our resource aware process scheduler.

Mars has identified the problem of *"Cross-core application interference due to contention for shared on-chip and off-chip resources pose a significant challenge to providing application level quality of service (QoS) guarantees on commodity multicore micro-architectures."* [197]. They use a simplified heuristics to determine if two applications are sensitive to the last-level cache contrasting to our approach of supporting any measurable hardware metric.

Process Classification Schemes There are many ways to classify processes as a function of their resource usage. We have opted to describe the hardware resource usage by values. We have, for example, described the cache usage by the number of accesses/sec or the floating point (FP) usage by the number of FP-instructions executed/sec. There are other process descriptions and classification systems that defines a certain behaviour. Xie and Loh [304] have introduced a classification scheme that defines four classes depending on the process' execution pattern and L_2-cache usage: 1) Turtles - rarely uses the L_2-cache; 2) Sheep - low L_2-cache usage; 3) Rabbits - high L_2-cache usage and sensitive to memory contention; 4) devil - high L_2-cache usage and still causes many L_2-cache-misses by itself. In a simular way, Zhuravlev et al. [313] has defined the pain classification scheme, which determines to what degree a process is cache-sensitive and how it affects the cache usage of other processes. Ren [241] describes the importance of system-wide performance sampling and understanding of a system. They also describe the importance of monitoring live systems running at customer sites, such that obtaining real-world measurements that may be difficult to obtain in a lab environment.

System- and Hardware Monitoring Software engineers have used PMU counters since they were first included in CPUs. One of the first implementations was the Intel Pentium CPU [199].

Many attempts have been made to use PMUs to understand performance bottlenecks, for example, hierarchical cycle accounting [221], source-code loop-level methodology [66]. Also, PMUs were used in [133, 308] to bound the interference between memory intensive processes that execute on different cores by assigning a budget on the number of memory requests for each core [308] or application [133] every predefined period. When a core/application consumes

its budget, monitored by a PMU, the scheduler suspends it until the next period replenishes its budget. The calculation of the budget for cores/application is based on the worst-case resource use, which might not be easy to obtain and often cause overprovision. Also, depending on the source of resources contention, suspending processes might not be the most efficient solution, and it might be more efficient to move them to other cores to decrease the interference, which our solution can explore better.

Many researcher have proposed the usage of feedback control approaches to control the scheduling of dynamic workload applications [3, 56, 58, 167, 195, 296]. In the literatures, different types of controllers, sensing and actuating parameters were proposed. Regarding controllers, PI/PID controllers were proposed in [3, 167, 195], Linear quadratic regulator LQR [167, 295], Fuzzy controller [167], stochastic controller [56], cascade controller [195], model predictive controllers [296]. Regarding sensing parameters, most of the mentioned works use either deadline miss ratio, utilization, overrun or a combination. While for the actuation (i.e., controlling the scheduling) CPU budget, process period, allocation of processes or combinations are used. Our solution is different from the aforementioned works in two aspects; first, we use both R and x to find as an input to the control loop to find the source of the problem and provide a proper action, while all mentioned works do not consider R. In addition they assume that all process have the same QoS/x requirement (i.e., each process should have a deadline miss ratio less than certain value) while in our solution different QoS/x can be used for different process which is a more practical assumption specially for the application domain that we are targeting. The second aspect is that we do not use traditional feedback control approaches to design the controller, which require control models that might be very complex to obtain/identify since such systems are non-linear and time varying systems. Instead, we use machine learning based controller to optimize the usage of resources and gurantee the required QoS.

Emulators There are several emulators where researchers can evaluate their work. One of the most well-known is Linsched [40] where Linux scheduling experts can implement and test their new schedulers. Another emulator is ExSched [16], which acts as a user-mode framework where researchers can implement and test their schedulers. These emulators are not suitable to use in a production environment because they run in user-space, which results in slow performance. On the other hand, one significant benefit is that user-mode allows easy debugging by utilizing common debugging tools, such as GDB. We have implemented part of our process scheduler inside a virtual machine to

ease debugging and, therefore, reduce the development time. We could deploy the functionality on a native Linux environment after verifying its functionality inside the virtual machine.

8.6 Summary

We have answered Q4 (Section 3.2.4) by the work presented in this chapter and through our Papers A and B that extends the Patents P and O.

Our target system requires high throughput at the same time as maintaining a certain level of QoS. The currently existing Linux process schedulers support various real-time schedulers that do not meet the requirements because they do not account for shared hardware resource congestions. The first part of our scheduling framework automatically correlates resource usage with performance, which indicates what hardware resource has the highest impact on the performance. We use the correlation information to allocate processes efficiently over a CPU core cluster so that processes do not affect the performance of each other through involuntary shared resource congestion. The second part of our scheduling framework uses PMU events to restrict the hardware resource usage so that a process does not affect the performance of other processes executing on the same core. We have verified our ideas on a test system replicating the environment of a telecommunication system [154]. Our shared resource aware allocation method shows an average 20% (37.5% for one process and 12% for another process) performance increase SRA vs. Linux CFS. We have also verified our QoS aware process scheduler by running several test cases. We reduce the effect of shared resource memory congestion by triggering context switched at PMU counter overflow.

The current implementation works but still requires some improvements to be production grade. We would like to develop it further to include it in the product formally. We would also like to automate the scheduling framework further to decrease the time for re-allocation and re-scheduling decisions. We have limited our experiments by implementation time and system access restrictions. We would like to experiment further by comparing our SRA scheduler with other process scheduling algorithms implemented in the Linux kernel, for example, EDF, RT and similar.

Adding machine learning is a natural extension to the current heuristic and rule-based system in the decision engine. Our current system depends on human input when setting the rules but our working assumption has been that a machine learning system would be able to find many more correlations and

problematic scenarios that we cannot find manually. We knew this at the beginning of our project phase but due to time constraints we decided to prepare but not implement advanced machine learning functionality.

It is untrue that happiness means a trouble free life. (A) happy life means overcoming struggles, fighting with struggles, resolving difficulties. The challange is that you just confront your challanges, you try your best, strain yourself and then you get the moment of happiness when you see that you have have controlled the challenges or the fate. Now, that is exactly this joy of overcoming difficulties of fighting with struggles, facing them point-blank and overcoming.

— Zygmunt Bauman in the film
"The Swedish Theory of Love" [1:06:10], 2016[1]

[1] This quote summarize all struggles in life, including my own experiences writing a thesis.

9 Conclusion and Future Work

ONE of the essential things in a thesis is the conclusions. What have we learned from all the research that is the foundation for the thesis and what are the ultimate conclusions? Each chapter has its own summary section that describes our conclusions. In this chapter, we lift our gaze and give a broader and more general conclusion to our complete work. We briefly answer the research questions listed in Section 3.2. We give our answers in the frame of the telecommunication system we have defined in Section 2.5, and delimited in Section 3.3.

We have divided this chapter into two parts. The first part lists our conclusions, Section 9.1, and shortly comment on each research question and our achievements. The second part lists some possible future work, Section 9.2. We describe, what we think is, the most interesting future research areas related to our research areas.

9.1 Conclusion

We have formulated four research questions. We have implemented a performance and hardware usage monitoring application that can observe large industrial systems in a production environment. Our implementation answers the first research question (Q1), Section 3.2.1. The monitoring application periodically samples hardware characteristics information with low impact on the system behavior. We describe our monitoring conclusions in more detail in Section 5.6.

As a response to the second research question (Q2), Section 3.2.2, we have devised a method to automate the synthesis process when modeling the hardware usage of a production system. We have tested our method by using hardware characteristics information sampled by our monitoring application to create an execution model on a much smaller and cheaper test system. The characteristics model makes it possible to run performance tests 1) without using the business logic of the production system and 2) much earlier in the development process. Both approaches aim to reduce the overall development time and cost. We present our load synthesis conclusions in Section 6.6.

To answer the third research question (Q3), Section 3.2.3, we needed to understand how the performance of our target communication system could be improved. As a first step, we implemented a message compression mechanism that automatically selects the most appropriate compression algorithm depending on the network congestion level, message content, and CPU load. Our mechanism uses the compression algorithm that provides the shortest round-trip message time for bulk message transmission. Our mechanism continuously assess the performance of all supported compression algorithms and adapt to a changing environment or message stream content. We plan to continue using the monitor-model-improve methodology to find additional performance improvements. There is a more detailed description of our message compression conclusions in Section 7.6.

We answer the fourth research question (Q4), Section 3.2.4, by designing a framework that efficiently allocates processes in a CPU-core cluster to improve performance while simultaneously scheduling processes to retain QoS. We have implemented our framework and tested it on a test system that emulates a real telecommunication system. We describe our conclusions in more detail in Section 8.6.

We formulated all research questions from requirements observed when being a part of the software design organization of Ericsson's telecommunication system. We investigated and designed our solutions to the research questions with the mindset and environment provided by the industrial settings. We there-

fore focused more on devising and implementing solutions that work in an industrial system rather than more generalized solutions with severe practical limitations in an industrial environment. We implemented and tested all of our research results within the environment of Ericsson's telecommunication system. Our opinion is that our generic methods should be usable for many other systems although we have mainly tested them on one particular system. The reason for this limitation comes naturally since we cannot get access to other commercial telecommunication system.

The corporate test department currently uses the monitoring and modeling tool for early-stage performance testing. We have patented the allocation [155] and scheduling mechanisms [156] for possible inclusion in future products.

Finally, It has been great fun and rewarding to do all this work. It has been a personality-changing event for me, and I wish that many more people would get the opportunity to pursue their wishes and dreams, whatever they are.

9.2 Future Work

Every researcher knows that it is difficult to limit the scope of one's work when conducting research. Plunging deeper into a problem and investigating it more thoroughly is always rewarding and gratifying but there must always be an end to the study. In this section, we list some areas where we would like to investigate further, given the time and resources. We divide this section into four parts, each describing one research area: monitoring, modeling, message compression and process allocation/scheduling.

Monitoring

Our opinion is that the techniques for hardware monitoring have matured significantly during the last decade. There are, however, still some distance to go. There are many academic techniques, but we think that they still need to be simplified and made much more available for the industrial community. Many companies do not have engineers explicitly dedicated to performance evaluation, so the state-of-the-art monitoring techniques should be implemented in user-friendly tools. Commercialization of performance tools is and will probably always be a market opportunity. We would like to investigate more detailed and efficient methods that can be used to monitor large-scale systems. In particular, how to efficiently and accurately monitor customer-deployed systems. It is probably not as easy as increasing the sampling frequency, as that will affect the performance of the system. Much remains to be done in the monitoring arena.

Load replication

We think that adding load replication support for dynamic behavior would make the model more accurate. We use the mean value of a metric when creating the hardware usage model in our current implementation. The mean-value-approach is sufficient for our current purposes but modeling the dynamic resource usage should make it possible to investigate additional performance-related areas. For example the undesirable memory bus side-effects caused by data bursts. It would also be useful to add additional hardware metrics to the model, such as branch misses, last level caches, and TLB misses just to name a few. Our opinion is that using modeling and load replication is beneficial for most design organizations. In an ideal case, we may be able to move parts of the performance testing to the early development phase. Imagine having a performance gauge in the source-code editor, telling how well the code utilizes the hardware. That could be an interesting research project as well as a commercial opportunity.

We have assumed, according to existing research, that finding performance related bugs in the initial phases of the development process will reduce the total development time. We would like to formally validate our assumptions by performing a study of an industrial system. We would also like to implement and test our methods on a broader range of systems to verify that they support varying types of systems.

Automatic message compression

It is easy to improve our automatic message compression method by adding additional compression algorithms. It would be particularly interesting to evaluate the hardware supported compression algorithms included in recent CPUs. It would also be rewarding to use machine learning techniques to predict recurring changes to the message stream and predict the most appropriate compression algorithm to use.

During our writing this thesis, we have deduced that there is an infinite demand for performance investigations and capacity improvements within the industry. We think that the demand for more advanced monitoring techniques will continue to be a vital issue in a competitive market environment. The continuous need for increased communication bandwidth is promising for the development of more advanced and efficient adaptive message compression techniques. We estimate that modern CPUs will increasingly support hardware acceleration for compression algorithms.

Process allocation and scheduling

The last research topic presented in this thesis is resource-aware process allocation and scheduling. We still have many issues to investigate in this area. This is particularly interesting as the hardware becomes more advanced and complicated with every new architecture and CPU. Our opinion is that there is a need to evaluate and improve scheduling algorithms continually. OS:es will always need the standard algorithms, but there is a place for tailored scheduling methods that targets a particular use-case. It would be interesting to investigate how to efficiently allocate processes on a set of heterogeneous CPU cores. We would also like to investigate how to guarantee QoS on heterogeneous hardware. How can we handle the increased CPU complexity and increasing number of cores, shared resources and bottlenecks? More complex cache with many levels, clusters, and various sharing between cores further complicate scheduling and is something that we need to handle in future resource aware process schedulers.

We would also like to formalize our reasoning related to process allocation and scheduling. It would be useful to find a theoretical connection between resource usage and performance for a given hardware architecture. Predicting the performance and QoS before deploying the system would be a great tool for system designers.

Jag är inte en petimeter, jag är en besserwisser![1]

My own translation:

I am no fusspot, I am a besserwisser!

— Amelie Jägemar, 2018

[1] Our daughter, nine years old, exclaimed that she wasn't a fusspot at all when she was told not to always correct parents and her younger sisters on non-essential matters.

10 Definitions

We use the definitions listed in Table 10.1 throughout the thesis. We have grouped the definitions in functionality order. The section reference describes where each defintion is defined in the thesis.

Definition	Page	Description
CPU and cores (C)	67	The CPU has a set of *cores* denoted C.
System (sys)	69	Let sys denote the system under investigation.
Application ($appl$)	70	We denote the application under investigation as $appl \in APPL$ where $APPL$ is the set of applications in sys.
Processes (P)	70	Let $p \in P$ be one *process* of the complete set of processes P executing on system sys. We use a subscript, $p_i \in P$, if we need to differentiate between multiple processes.
Core affinity (A_p)	168	The *affinity* $A_p \subseteq C$ for process $p \in P$ is the set of cores where p is bound/allowed to execute.
CPU-load (L)	138	The CPU-load, L, is defined as the number of processes, ready to execute, in the run-queue of the operating system.

Continued on next page →

← Continued from previous page

Definition	Page	Description
Hardware resource (r, r_i, R)	71	The *hardware resource*, r, is one of the total hardware resources, R, such that $r \in R$. We use a subscript i, such that $r_i \in R$, if we need to differentiate between multiple resources.
Hardware resource measurement ($m_{r,p}$)	71	The bounded series of *resource usage samples* of hardware resource r for process p is denoted by $m_{r,p}$.
Performance (x, X)	73	The *performance* is denoted by $x \in X$ where X denotes the set of all performance metrics.
Performance measurement ($m_{x,p}$)	73	The bounded series of *performance metric samples* of x for process p is denoted $m_{x,p}$.
Access limit value ($O_{r,p}$)	171	The *access limit value*, $O_{r,p}$, is the number of accesses to resource $r \in R$ by a process $p \in P$ before the hardware should generate a resource usage overflow interrupt. We denote the set of all overflow values for p as $O_{R,p} = \{O_{r,p} : \forall r \in R, p \in P\}$
Pearson correlation ($\rho(a, b)$)	165	The Pearson correlation coefficient, $\rho(a, b)$ quantifies the similarity between the two data sets a and b.
Resource and performance correlation ($Corr(r, x, p)$)	167	Let $Corr(r, x, p)$ denote the *correlation* $\rho(m_{r,p}, m_{x,p})$ between the bounded series $m_{r,p}$ and $m_{x,p}$ for some $r \in R, p \in P, x \in X$ and where $\lvert m_{r,p}\rvert = \lvert m_{x,p}\rvert$.
Maximum resource and performance correlation ($\widehat{Corr}(R, x, p)$)	167	Let $\widehat{Corr}(R, x, p) = max_{r \in R} Corr(r, x, p)$ denote the maximum correlation value for all resources $r \in R$ for some given process, $p \in P$, and performance metric $x \in X$.

Continued on next page →

← Continued from previous page

Definition	Page	Description
The resource with max correlation value ($\hat{r}$)	167	Let $\hat{r} = \{r_i \mid \forall r_i \in R, p \in P, x \in X, Corr(r, x, p) > \theta\}$ denote the *set of resources* $r_i \in R$ with descendingly sorted correlation values larger than the threshold, θ.
Transmission time (t_t)	127	The *transmission time*, t_t, is defined as the sum of message *compression time*, t_c, *send time*, t_s, one way of the *message round-trip time*, t_{rtt}, and *decompression time*, t_d, such that $t_t = t_c + t_s + \frac{t_{rtt}}{2} + t_d$.
Compression time (t_c) and rate (t_{cr})	127	Let compression time, $t_c = \frac{s}{t_{cr}}$ be the time to compress a particular message of size s. The compression rate, t_{cr}, is the number of bytes compressed per second.
Decompression time (t_d) and rate (t_{dr})	127	Let decompression time, $t_d = \frac{s}{t_{dr}}$ be the time to decompress a particular message of size s. The decompression rate, t_{dr}, is the number of decompressed bytes per second, [B/sec].
Compression ratio (H)	127	We define *compression ratio* as $r_c = \frac{s_u}{s_c}$, where s_u is the size of the uncompressed message and s_c is the compressed message size.
Data sets ($\mathbb{D}, \mathbb{D}_0, \mathbb{D}_1, \mathbb{D}_p$)	139	A data set, $\mathbb{D}$, is a bounded series of values. A data set containing only zeros is denoted $\mathbb{D}_0$. A data set with only ones is denoted $\mathbb{D}_1$. A data set that contains sampled production system message data is denoted $\mathbb{D}_p$.

Table 10.1: Definitions.

People who are really serious about software should make their own hardware.

— Alan Kay[1]

[1] talk at Creative Think seminar, 20 July 1982 `http://folklore.org`

11 Key Concepts

Table 11.1 lists the most common abbreviations used throughout the following text.

Key Concept	Description
2G (GSM)	The second generation telecom network, 1991, introduced digital communication.
3G	The third telecom network generation, 1998, enabled large scale digital communication with increased bandwidth and service availability.
3GPP	The 3GPP is a standardization organization created by the telecommunication industry. 3GPP aims to create a global standard that is used for development and maintenance of telecommunication systems.
4G (LTE)	Long term evolution is the fourth generation telecommunication network, 2008, with increased capacity.
5G	High bandwidth to mobile users with explicit focus on low response times.
Action Research (AR)	A research method where the researcher is an active part of an incremental procedure (plan, act/observe and reflect), which is repeatedly used to improve the object being investigated. AR was first expressed in 1946 by Lewin [185]

Continued on next page →

← Continued from previous page

Key Concept	Description
(Process) allocation	Process allocation [309] acts on the problem of *where* processes should run, i.e. on what CPU or core.
(Process) scheduling	Process scheduling decides *when* and *how* processes should run.
ASIC	Application specific integrated circuits are circuits that can be pre-programmed with specific functionality
Capacity	The Oxford english dictionary states that capacity means: *"Ability to receive or contain; holding power"*. We use capacity as a description of the maximal capability of a resource.
COTS	Common off the shelf are devices that does not need to be tailored for a specific need, they can be bought from other device manufacturer that produce common hardware for many purposes.
CPI	Cycles per instruction is a metric to determine the performance of a computer system. A CPI stack estimate how much of the total execution time is attributed to various HW resources such as cache misses, branch misses, TLB misses etc [94].
Five Nines	99.999% uptime, which results maximum of approx. 5 min downtime per year.
FPGA	Field programmable arrays are generic circuits that can be programmed in runtime with new functionality.
FPU	The floating point unit typically architecturally located inside modern CPUs.
HW	HW is an abbreviation for hardware, which means all physical parts in the network, including computers, cables, circuit-boards etc.
ICT	Information communication technology that makes it possible for people to communicate and easily access information.

Continued on next page →

← Continued from previous page

Key Concept	Description
L_1-cache, L_2I-cache, L_2D-cache	The cache acts as a small intermediate memory that is substantially faster than the RAM. The subscript index determine the cache level, starting with 1 for the first cache-level. Using the capital letter "I" indicate the instruction cache and "D" means the data cache.
L_1TLB, L_1ITLB, L_1DTLB	The translation lookaside buffers temporarily store memory mappings between the virtual and the physical address space. The index,"I" and "D" specifiers acts the same was as for caches.
Low-intrusive Monitoring	The monitoring mechanism does not affect the behavior or performance of the monitored system. There is no noticeable effect on the system.
Node	A computer designed for message processing, which is part of a telecommunication system.
OS	Operating system.
Performance	As specified by the Oxford English dictionary; *"The quality of execution of such an action, operation, or process; the competence or effectiveness of a person or thing in performing an action; spec. the capabilities, productivity, or success of a machine, product, or person when measured against a standard."* [225]. More specifically; a quantifiable metric on how good a particular action is performed. We denote system performance with x.
PID Controller	Proportional integrative controller [22].
PMU	The performance monitor unit [310] implements funtionality to measure many metrics (events) that describe the currently executing application. For example: cache usage, floating point usage, execution pipeline statistics. The PMU is completely implemented in HW making it an efficient tool for execution supervision.

Continued on next page →

← Continued from previous page

Key Concept	Description
PMU Overflow	It is possible to configure the PMU to generate an interrupt on an event counter overflow. We use this mechanism to enforce shared resource quota through process context switch on overflow.
Production Node	One node that is running at a customer site handling real end-user traffic.
(Process) Scheduling	Process scheduling [61, p35:5] decides *when* and *how* processes should run.
Superscalar Processors	Low-level instructions can be executed in parallel to achieve higher performance, typically more than one instruction per clock cycle. The first commercial appearance was in 1988 with Intel® i960CA [202].
SW	As specified by the Oxford english dictionary; *"The programs and procedures required to enable a computer to perform a specific task, as opposed to the physical components of the system"* [225]
Service Level Agreement (SLA)	The service level agreement stipulates guidelines and requirements for a particular application or process [305].
Test Node	Test nodes are typically smaller than production nodes and usually only accessible by corporate personnel. Economic reasons and keeping debugging simple drive the demand to keep test nodes being small.

Table 11.1: Key concepts.

Bibliography

[1] Josh Aas. *Understanding the Linux 2.6.8.1 CPU Scheduler*. Silicon Graphics International SGI, 22(February):1–26, 2005.

[2] Ericsson Ab. *5G systems*. Technical Report January, Ericsson, 2017.

[3] Luca Abeni and Giorgio Buttazzo. *Adaptive bandwidth reservation for multimedia computing*. In *IEEE real time computing systems and applications*, December, pages 1–8. 1999.

[4] Göran Ahlforn and Erik Örnulf. *Ericsson's family of carrier-class technologies*. Technical Report 4, Ericsson, 2001.

[5] Anastassia Ailamaki, David J Dewitt, Mark D Hill, and David a Wood. *DBMSs On A Modern Processor : Where Does Time Go ?* Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99), 1394:266–277, 1999.

[6] Alaa R. Alameldeen, Milo Martin, Carl J. Mauer, Kevin E. Moore, and Min Xu. *Simulating a $2M Commercial Server on a $2K PC*. IEEE Computer, 36(2):50–57, 2003.

[7] Alaa R. Alameldeen and David A. Wood. *IPC considered harmful for multiprocessor workloads*. IEEE Micro, pages 8–17, 2006.

[8] Osman Allam, Stijn Eyerman, and Lieven Eeckhout. *An efficient CPI stack counter architecture for superscalar processors*. Proceedings of the Great Lakes Symposium on VLSI, pages 55–58, 2012.

[9] Gabor Andai. *Performance monitoring on high-end general processing boards*. Master thesis, KTH Royal Institute of Technology, 2014.

[10] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. *Continuous profiling: where have all the cycles gone?* ACM SIGOPS, 15(4):357–390, 1997.

[11] J G Andrews, S Buzzi, W Choi, S V Hanly, A Lozano, a C K Soong, and J C Zhang. *What Will 5G Be?* Selected Areas in Communications, IEEE Journal on, 32(6):1065–1082, 2014.

[12] Apple. *Apples Revolutionary App Store Downloads Top One Billion in Just Nine Months*. `www.apple.com`, 2009. [Accessed 2015-03-04].

[13] Apple. *App Store Tops 40 Billion Downloads with Almost Half in 2012*. `www.apple.com`, 2013. [Accessed 2015-03-04].

[14] ARM. *Cortex -A8 Revision : r3p1*. ARM, edition i edition, 2009.

[15] Arpaci-Dusseau Andrea Arpaci-Dusseau Remzi. *Operating Systems: Three Easy Pieces*, volume Electronic. Arpaci-Dusseau Books, Wisconsin, 2015.

[16] Mikael Asberg, Thomas Nolte, Shinpei Kato, and Ragunathan Rajkumar. *ExSched: An External CPU Scheduler Framework for Real-Time Systems*. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 240–249. 2012.

[17] Neil C. Audsley, Alan Burns, Robert I. Davis, Ken W. Tindell, and Andy J. Wellings. *Fixed priority pre-emptive scheduling: An historical perspective*. Real-Time Systems, 8(2-3):173–198, 1995.

[18] Henrik Austad. *A Multicore-aware Deadline-driven Real-Time Scheduler for the Linux Kernel*. M.sc., Norwegian University of Science and Technology, 2009.

[19] Moris Behnam, Rafia Inam, Thomas Nolte, and Mikael Sjödin. *Multi-core composability in the face of memory-bus contention*. SIGBED Review, 10(October):35–42, 2013.

[20] Robert H. Bell and Lizy K. John. *Improved automatic testcase synthesis for performance model validation*. In *Proceedings of International Conference on Supercomputing*, pages 111–120. 2005.

[21] Bell-Labs. *Video Shakes - A Bell Labs Study on Rising Video Demand and its Impact on Broadband IP Networks.* Technical report, Bell Labs, 2012.

[22] S. Bennett. *Nicolas Minorsky and the Automatic Steering of Ships*. IEEE Control Systems Magazine, 4(4):10–15, 1984.

[23] Mikael Bergqvist, Jakob Engblom, Mikael Patel, and Lars Lundegard. *Some experience from the development of a simulator for a telecom cluster (CPPemu)*. In *Proceedings of the International Association of Science and Technology for Development*, pages 13–21. 2006.

[24] JO Best. *The race to 5G Inside the fight for the future of mobile as we know it - Feature - TechRepublic*. Techrepublic, 2015.

[25] Daniel Biederman. *Communication system with content-based data compression*. US Patent 7069342, 2001.

[26] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. *Contention-Aware Scheduling on Multicore Systems*. ACM Transactions on Computer Systems, 28(4):1–45, 2010.

[27] Critical Blue. *Process Scheduling in Linux*. Technical report, University of Edinburgh, 2013.

[28] Barry Boehm. *The Incremental Comitment Spiral Model. Principles and Practices for Successful Systems and Software*. Addison-Wesley Professional, 2013.

[29] Barry Boehm and Victor R. Basil. *Software Defect Reduction Top 10 List*. Computer Journal, 34(1):135–137, 2001.

[30] Barry Boehm and Philip N. Papaccio. *Understanding and controlling software costs*. IEEE Transactions on Software Engineering, 14(10):1462–1477, 1988.

[31] P. Bose and T.M. Conte. *Performance analysis and its impact on design*. Computer, 31(5):41–49, may 1998.

[32] Fred A. Bower, Daniel J. Sorin, and Landon P. Cox. *The impact of dynamically heterogeneous multicore processors on thread scheduling*. IEEE Micro, 28(3):17–25, 2008.

[33] Rupinder Singh Brar. *A Survey on Different Compression Techniques and Bit Reduction Algorithm for Compression of Text / Lossless Data*. 3(3):579–582, 2013.

[34] Jens Breitbart, Josef Weidendorfer, and Carsten Trinitis. *Case Study on Co-scheduling for HPC Applications*. Proceedings of the International Conference on Parallel Processing Workshops, 2015-Janua:277–285, 2015.

[35] Jens Breitbart, Josef Weidendorfer, and Carsten Trinitis. *Automatic co-scheduling based on main memory bandwidth usage*. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*. 2016.

[36] Elisabeth Brunet, Francois Trahay, Alexandre Denis, and Raymond Namyst. *A sampling-based approach for communication libraries auto-tuning*. In *Proceedings of International Conference on Cluster Computing*, pages 299 – 307. 2011.

[37] Mary Brydon-Miller, Davydd Greenwood, and Patricia Maguire. *Why Action Research?* Action Research, 1(1):9–28, jul 2003.

[38] Anselm Busse. *Load-aware Scheduling for Heterogeneous Multi-core Systems*. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1844–1851. 2016.

[39] Giorgio C Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 3rd Edition (Real-Time Systems Series)*, volume 24 TS - C. Springer, 2011.

[40] J. Calandrino, D. Baumberger, Tong Li, J. Young, and Scott Hahn. *LinSched : The Linux Scheduler Simulator*. Proceedings of the ISCA 21st International Conference on Parallel and Distributed Computing and Communications Systems, pages 171–176, 2008.

[41] Martin Carlsson and Jakob Engblom. *Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system*. In *Proceedings of the 2nd International Workshop on Real-Time Tools (RT-TOOLS'2002)*. 2002.

[42] R. L. G. Cavalcante, S. Stańczak, M. Schubert, a. Eisenblätter, and U. Türke. *Toward Energy-Efficient 5G Wireless Communications Technologies*. IEEE Signal Processing Magazine, accepted f(October):24–34, 2014.

[43] Charles Cazabon. *Memtester*. `http://pyropus.ca/software/memtester/`, 2018. [Accessed 2018-08-27].

[44] Marco Cesati. *Overview of the Linux Scheduler Framework*, 2014.

[45] Craig Chapple. *Count of Active Applications in the App Store*. `http://148apps.biz/app-store-metrics/?mpage=appcount`, 2014. [Accessed 2015-03-04].

[46] Winston Churchill. *Memorandum by the Prime Minister : Brevity.* Technical report, 1940.

[47] Cisco. *Cisco Visual Networking Index: Forecast and Methodology Cisco Visual Networking Index: Cisco Visual Networking Index: Forecast and Methodology*. Technical report, Cisco, 2015.

[48] Yann Collet. *lz4 Data Compression Library*. `http://fastcompression.blogspot.se/p/lz4.html`, 2013. [Accessed 2015-03-04].

[49] Andrew Collette. *LZFX Data Compression Library*. `http://code.google.com/p/lzfx/`, 2013. [Accessed 2015-03-28].

[50] Gerald Combs. *Wireshark*. `http://www.wireshark.org/`, 2014. [Accessed 2018-02-07].

[51] Richard Walter Conway, William L. Maxwell, and L. W. (Louis W.) Miller. *Theory of scheduling*. Dover, 1967.

[52] Intel Corporation. *Mobile 4th Generation Intel ® Core ™ Processor Family , Mobile Intel ® Pentium ® Processor Family , and Mobile Intel ® Celeron ® Processor Family - Datasheet Volume 1*. Technical Report September, 2013.

[53] Intel Corporation. *Mobile 4th Generation Intel ® Core ™ Processor Family , Mobile Intel ® Pentium ® Processor Family , and Mobile Intel ® Celeron ® Processor Family - Datasheet Volume 2*. Technical Report September, 2013.

[54] Intel Corporation. *Mobile 4th Generation Intel ® Core ™ Processor Family , Mobile Intel ® Pentium ® Processor Family , and Mobile Intel ® Celeron ® Processor Family - Volume 1*, volume 2. 2013.

[55] Intel Corporation. *Mobile 4th Generation Intel ® Core ™ Processor Family , Mobile Intel ® Pentium ® Processor Family , and Mobile Intel ® Celeron ® Processor Family - Volume 2*, volume 2. 2013.

[56] T Cucinotta, L Palopoli, L Marzario, G Lipari, and L Abeni. *Adaptive reservations in a Linux environment*. In *Proceedings - IEEE Real-Time and Embedded Technology and Applications Symposium*, volume 10, pages 238–245. 2004.

[57] Tommaso Cucinotta. *Access control for adaptive reservations on multi-user systems*. Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, pages 387–396, 2008.

[58] Tommaso Cucinotta, Fabio Checconi, Luca Abeni, and Luigi Palopoli. *Adaptive real-time scheduling for legacy multimedia applications*. ACM Transactions on Embedded Computing Systems, 11(4):1–23, 2012.

[59] Jakob Danielsson, Marcus Jägemar, Moris Behnam, and Mikael Sjödin. *Investigating Execution-Characteristics of Feature-Detection Algorithms*. In *Proceedings of Emerging Technologies and Factory Automation (ETFA)*, page 4. IEEE, Limassol, 2017.

[60] Jakob Danielsson, Jägemar Marcus, Moris Behnam, Mikael Sjödin, and Tiberiu Seceleanu. *Measurement-based evaluation of data-parallelism for OpenCV feature-detection algorithms*. In *Proceedings of Computers, Software and Applications Conference (COMPSAC)*, page 10. 2018.

[61] Robert I. Davis and Alan Burns. *A survey of hard real-time scheduling for multiprocessor systems*. ACM Computing Surveys, 43(4):1–44, 2011.

[62] Arnaldo Carvalho de Melo. *The New Linux 'perf' Tools*. Technical report, Redhat, 2010.

[63] Johan De Vriendt, Philippe Lainé, Christophe Lerouge, and Xiaofeng Xu. *Mobile network evolution: A revolution on the move*. IEEE Communications Magazine, 40(4):104–111, 2002.

[64] Roman Dementiev. *Processor Performance Counter Monitoring*. Technical Report July, Intel, 2010.

[65] John Demme and Simha Sethumadhavan. *Rapid identification of architectural bottlenecks via precise event counting*. In *Proceeding International Symposium on Computer Architecture (ISCA)*, page 353. 2011.

[66] Jeff Diamond, Martin Burtscher, John D. J.D. McCalpin, Byoung-Do Do Kim, S.W. Stephen W. Keckler, and J.C. James C. Browne. *Evaluation and optimization of multicore performance bottlenecks in supercomputing applications*. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 32–43. 2011.

[67] Maria Dimakopoulou, Stéphane Eranian, Nectarios Koziris, and Nicholas Bambos. *Reliable and efficient performance monitoring in linux*. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, page 34, 2016.

[68] Torgeir Dingsøyr, Sridhar Nerur, Venugopal Balijepally, and Nils Brede Moe. *A decade of agile methodologies: Towards explaining agile software development*. Journal of Systems and Software, 85(6):1213–1221, 2012.

[69] Pedro C. Diniz and Martin C. Rinard. *Dynamic feedback: An Effective Technique for Adaptive Computing*. ACM SIGPLAN Notices, 32(5):71–84, may 1997.

[70] Gordana Dodig-Crnkovic. *Cognitive revolution, virtuality and good life.* AI and Society, 28:319–327, 2013.

[71] Daniel Doucette and Alexandra Fedorova. *Base vectors: A potential technique for microarchitectural classification of applications*. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*. 2007.

[72] Angela Duckworth. *Grit: The power of passion and perseverance*. `https://www.ted.com/talks/angela{_}lee{_}duckworth{_}the{_}key{_}to{_}success{_}grit`, 2013. [Accessed 2017-09-17].

[73] Angela Duckworth. *Grit : the power of passion and perseverance*. Scribner, 2016.

[74] Jon Dugan, Seth Elliott, Bruce Mah, Jeff Poskanzer, and Prabhu Kaustubh. *IPerf*. `https://iperf.fr`, 2018. [Accessed 2018-08-27].

[75] Denis Duka. *Connectivity packet platform in the GSM/WCDMA network*. Proceedings Elmar - International Symposium Electronics in Marine, pages 163–166, 2006.

[76] David Eklöv, David Black-Schaffer, and Erik Hagersten. *StatCC: a statistical cache contention model*. In *Proceedings of the International conference on Parallel architectures and compilation techniques*, pages 551–552. 2010.

[77] David Eklöv, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. *Cache Pirating: Measuring the Curse of the Shared Cache*. In *Proceedings of International Conference on Parallel Processing*, pages 165–175. sep 2011.

[78] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. *Bandwidth bandit: Understanding memory contention*. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 116–117. 2012.

[79] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. *Bandwidth Bandit: Quantitative characterization of memory contention*. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013*. 2013.

[80] Enea. *OSE Kernel Reference Manual*. Enea OSE Systems AB, Stockholm, r1.0.4 edition, 1998.

[81] Enea. *OSE Architecture User's Guide*. Enea OSE Systems AB, Stockholm, bl410003 edition, 2010.

[82] Enea. *OSE Core User's Guide*. Stockholm, bl410003 edition, 2010.

[83] Stéphane Eranian. *What can performance counters do for memory subsystem analysis?* In *Proceedings of the ACM SIGPLAN workshop on Memory Systems Performance and Correctness*, pages 26–30. 2008.

[84] Ericsson. *Market Outlook.* Technical report, Ericsson, 2013.

[85] Ericsson. *Ericsson Consumer Lab: 10 Hot Consumer Trends 2014*. Technical report, Ericsson Consumer Lab, 2014.

[86] Ericsson. *5G Energy Performance - Key Technologies and Design Principles*. Technical Report April, Ericsson White Paper, 2015.

[87] Ericsson. *5G Radio Access - Technology and Capabilities*. Technical Report February, Ericsson White Paper, 2015.

[88] Ericsson. *Ericsson Mobility Report November 2015*. Technical Report November, Ericsson Consumer Lab, 2015.

[89] Ericsson. *Ericsson Mobility Report November 2016*. Technical Report November, Ericsson, Stockholm, 2016.

[90] Ericsson. *Hot Consumer Trends 2016*. Technical Report December 2015, Ericsson Consumer Lab, 2016.

[91] Fredrik Eriksson. *Porting OSE Systems to Linux*. Master thesis, Mälardalen University, 2010.

[92] Ernst & Young Global. *2015 Global telecommunications study: Navigating the road to 2020*. Ernst & Young Global Limited, pages 1–39, 2015.

[93] Stijn Eyerman and Lieven Eeckhout. *System-level performance metrics for multiprogram workloads*. IEEE Micro, 28(3):42–53, 2008.

[94] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. *A Top-Down Approach to Architecting CPI Component Performance Counters.* IEEE Micro, 27(1):84–93, 2007.

[95] Stijn Eyerman, K. Hoste, and Lieven Eeckhout. *Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware*. In *International Symposium on Performance Analysis of Systems and Software*, pages 216–226. 2011.

[96] Dario Faggioli, Fabio Checconi, Michael Trimarchi, and Claudio Scordino. *An EDF scheduling class for the Linux kernel*. Proceedings of the 11th Real Time Linux Workshop (RTLW), page 8 pp., 2009.

[97] Dario Faggioli, Fabio Checconi, Michael Trimarchi, and Claudio Scordino. *An EDF scheduling class for the Linux kernel*. Proceedings of the 11th Real Time Linux Workshop (RTLW), page 8 pp., 2009.

[98] Dror G. Feitelson, Larry Rudolph, Larry Rudolph Dror G. Feitelson, Dror G. Feitelson, and Larry Rudolph. *Parallel Job Scheduling: Issues and Approaches*. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 949, pages 1–18. 1995.

[99] Colin Fidge. *Fundamentals of distributed system observation*. IEEE Software, 13(6), 1996.

[100] Gerhard Fohler. *How Different are Offline and Online Scheduling?* In *2nd International Real-Time Scheduling Open Problems Seminar*, pages 2–3. 2011.

[101] Eelke Folmer and Jan Bosch. *Architecting for usability: A survey*. Journal of Systems and Software, 70(1-2):61–78, 2004.

[102] Freescale. *P4080 Reference Manual Rev F*. 2009.

[103] Freescale. *e500mc Core Reference Manual Rev F*. Freescale, 2010.

[104] Freescale. *Advanced QorIQ Debug and Performance Monitoring*. Freescale, rev. d edition, 2011.

[105] Freescale. *e6500 Core Reference Manual*. Freescale, revision f edition, 2012.

[106] Anders Furuskär, Jonas Näslund, and Håkan Olofsson. *Edge - enhanced data rates for GSM and TDMA/136 evolution*. Ericsson Review (English Edition), 76(1):28–37, 1999.

[107] Gert Fylking. *Gert Fylking bakom litteraturpris*. `https://www.svd.se/gert-fylking-bakom-litteraturpris`, 2000. [Accessed 2018-02-07].

[108] J Gait. *A Probe Effect in Concurrent Programs*. Software - Practice and Experience, 16(3), 1986.

[109] N Garner, G Ho, and P Mucci. *A Portable Programming Interface for Performance Evaluation on Modern Processors*. The International Journal of High Performance Computing Applications, 14(3):189–204, 2000.

[110] Gartner. *High Tech and Telecom Providers*. `http://www.gartner.com/technology/consulting/high-tech-telecom-providers.jsp`, 2012. [Accessed 2015-03-04].

[111] Brian Gildon. *ADVANTAGES OF ENEA OSE ® : The Architectural Advantages of Enea OSE in Telecom Applications (Product Marketing White Paper)*. Technical report, Enea OSE, Stockholm, 2017.

[112] Thomas Gleixner. *Linux Performance Counter announcement*. `http://lkml.org/lkml/2008/12/4/401`, 2008. [Accessed 2018-02-07].

[113] Adithya Gollapudi and Arvind Ojha. *Comparing Applicability of Test Design Techniques for Telecom systems*. Ph.D. thesis, Mälardalen University, 2009.

[114] Google. *Snappy Compression Library*. `https://code.google.com/p/snappy`, 2013. [Accessed 2015-03-28].

[115] Rasmuss Graaf. *Choosing Your Runtime*. Technical report, Enea, Stockholm, 2010.

[116] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. *A Survey on Cache Management Mechanisms for Real-Time Embedded Systems*. ACM Computing Surveys, 48(2):1–36, 2015.

[117] Michael Gray, Peter Peterson, and Peter Reiher. *Scaling Down Off-The-Shelf Data Compression : Backwards-Compatible Fine-Grain Mixing*. In *Proceedings of Distributed Computing Systems*, pages 112 – 121. 2012.

[118] GSM_World. *GSM Market Data Report*. Technical report, 2009.

[119] Neil Gunther. *UNIX Load Average*. Technical report, TeamQuest, 2009.

[120] JL Gustafson. *Reevaluating Amdahl's law.* Communications of the ACM, 31(5):532–533, may 1988.

[121] Carl Gutwin, Christopher Fedak, Mark Watson, Jeff Dyck, and Tim Bell. *Improving network efficiency in real-time groupware with general message compression.* In *Proceedings of Conference on Computer Supported Cooperative Work*, pages 119–128. ACM Press, New York, USA, 2006.

[122] Daniel Hallmans, Marcus Jägemar, Stig Larsson, and Thomas Nolte. *Identifying evolution problems for large long term industrial evolution systems*. In *Proceedings - IEEE 38th Annual International Computers, Software and Applications Conference Workshops, COMPSACW 2014*, pages 384–389. Västerås, 2014.

[123] AH Hashemi, DR Kaeli, and B Calder. *Efficient procedure mapping using cache line coloring*. ACM SIGPLAN Notices, pages 171–182, 1997.

[124] Nicholas Hatt, Axis Sivitz, and Benjamin a Kuperman. *Benchmarking Operating Systems*. In *Conference for Undergraduate Research in Computer Science and Mathematics.*, pages 63–68. 2007.

[125] Anthony Hayter. *Probability and statistics for engineers and scientists*. Brooks/-Cole Cengage Learning, Boston, 4th edition, 2016.

[126] J.L. Henning. *SPEC CPU2000: measuring CPU performance in the New Millennium*. Computer, 33(7):28–35, 2000.

[127] John L. Henning. *SPEC CPU2006 benchmark descriptions*. ACM SIGARCH Computer Architecture News, 34(4):1–17, 2006.

[128] Ariya Hidayat. *FastLZ*. `http://fastlz.org/`, 2014. [Accessed 2015-03-28].

[129] M.D. Hill and M.R. Marty. *Amdahl's law in the multicore era.* Computer, 41(7):33–38, jul 2008.

[130] Harri Holma and Antti Toskala. *WCDMA for UMTS, 3rd edition*. John Wiley & Sons Ltd., 2004.

[131] Jian Huang, Moinuddin K Qureshi, and Karsten Schwan. *An Evolutionary Study of Linux Memory Management for Fun and Profit Methodology*. 2016 USENIX Annual Technical Conference (USENIX ATC 16), 2016.

[132] IBM. *IBM PowerPC 750GX and 750GL RISC Micro- processor User ' s Manual*. Technical report, 2006.

[133] Rafia Inam. *Hierarchical scheduling for predictable execution of real-time software components and legacy systems*. Ph.D. thesis, Mälardalen University, dec 2014.

[134] Rafia Inam, Nesredin Mahmud, Moris Behnam, Thomas Nolte, and Mikael Sjödin. *The Multi-Resource Server for predictable execution on multi-core platforms*. 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pages 1–12, 2014.

[135] Rafia Inam and Mikael Sjödin. *Combating Unpredictability in Multicores through the Multi-Resource Server*. In *Workshop on Virtualization for Real-Time Embedded Systems*. IEEE, 2014.

[136] Rafia Inam, Mikael Sjödin, Marcus Jägemar, Mikael Sjodin, and Marcus Jagemar. *Bandwidth measurement using performance counters for predictable multicore software*. In *Proceedings of the International Conference on Emerging Technologies and Factory Automation (ETFA12)*. 2012.

[137] Rafia Inam, Joris Slatman, Moris Behnam, Mikael Sjödin, and Thomas Nolte. *Towards implementing multi-resource server on multi-core Linux platform*. In *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*, volume 1, pages 10–13. 2013.

[138] Nathan Ingraham. *Apple by the numbers: 30 billion app downloads, 650,000 apps available in the App Store*. `http://www.theverge.com/2012/6/11/3077792/apple-wwdc-2012-stats-ios-mac-growth`, 2012. [Accessed 2015-03-04].

[139] Intel. *LZO hardware compression*. `http://software.intel.com/en-us/articles/lzo-data-compression-support-in-intel-ipp`, 2013. [Accessed 2015-03-04].

[140] Intel. *Intel 64 and IA-32 architectures optimization reference manual*. Technical Report June, 2016.

[141] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, volume 3. Intel, 2016.

[142] Anand Padmanabha Iyer, Li Erran Li, and Ion Stoica. *CellIQ : Real-Time Cellular Network Analytics at Scale*. In *Nsdi*, pages 218–234. 2015.

[143] Bruce L. Jacob and Trevor N. Mudge. *A look at several memory management units, TLB-refill mechanisms, and page table organizations*. ACM SIGOPS Operating Systems Review, 32(5):295–306, 1998.

[144] Marcus Jägemar. *Utilizing Hardware Monitoring to Improve the Performance of Industrial Systems*. Licentiate thesis, Mälardalen University, 2016.

[145] Marcus Jägemar. *Mallocpool : Improving Memory Performance Through Contiguously TLB Mapped Memory*. In *Proceedings of IEEE Emerging Technologies and Factory Automation (ETFA)*. IEEE, Torino, Italy, 2018.

[146] Marcus Jägemar and Gordana Dodig-Crnkovic. *Cognitively Sustainable ICT with Ubiquitous Mobile Services - Challenges and Opportunities*. In *Proceedings - International Conference on Software Engineering*, volume 2, pages 531–540. 2015.

[147] Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl, Moris Behnam, and Björn Lisper. *Enforcing Quality of Service Through Hardware Resource Aware Process Scheduling*. In *Proceedings of IEEE Emerging Technologies and Factory Automation (ETFA)*. IEEE, Torino, Italy, 2018.

[148] Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl, and Björn Lisper. *Feedback-Based Generation of Hardware Characteristics*. Technical report, Mälardalen University, 2012.

[149] Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl, and Bjorn Lisper. *Technical Report : Feedback-Based Generation of Hardware Characteristics*. Technical report, Mälardalen University, 2012.

[150] Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl, and Björn Lisper. *Towards Feedback-Based Generation of Hardware Characteristics*. In *Proceedings of the 7th International Workshop on Feedback Computing*. 2012.

[151] Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl, and Björn Lisper. *Automatic Multi-Core Cache Characteristics Modelling*. In *Proceedings of the Swedish Workshop on Multicore Computing (MCC13)*, page 4. Halmstad, 2013.

[152] Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl, and Björn Lisper. *Adaptive Online Feedback Controlled Message Compression.* In *Proceedings of Computers, Software and Applications Conference (COMPSAC14)*. Västerås, 2014.

[153] Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl, and Björn Lisper. *Automatic message compression with overload protection*. The Journal of Systems and Software, 2016.

[154] Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl, Björn Lisper, and Gabor Andai. *Automatic Load Synthesis for Performance Verification in Early Design Phases.* Technical report, Mälardalen University, 2016.

[155] Marcus Jägemar, Andreas Ermedahl, and Sigrid Eldh. *Decision support for OS process scheduling based on HW-, OS- and system-level performance counters*, U.S. Patent 62/400353. 2016.

[156] Marcus Jägemar, Andreas Ermedahl, and Sigrid Eldh. *Process scheduling in a processing system having at least one processor and shared hardware resources*. U.S. Pat.Pending, PCT/SE2016/050317. 2016.

[157] Marcus Jägemar, Andreas Ermedahl, Sigrid Eldh, and Moris Behnam. *A Scheduling Architecture for Enforcing Quality of Service in Multi-Process Systems*. In *Proceedings of Emerging Technologies and Factory Automation (ETFA)*. 2017.

[158] Tarush Jain and Tanmay Agrawal. *The Haswell Microarchitecture - 4th Generation Processor*. International Journal of Computer Science and Information Technologies, 4(3):477–480, 2013.

[159] Emmanuel Jeannot. *Improving Middleware Performance with AdOC: an Adaptive Online Compression Library for Data Transfer*. In *Proceedings of International Parallel and Distributed Processing Symposium*, page 70. 2005.

[160] Emmanuel Jeannot. *ADOC homepage*. `http://www.labri.fr/perso/ejeannot/adoc/adoc.html`, 2012. [Accessed 2015-03-04].

[161] Emmanuel Jeannot, Bjorn Björn Knutsson, Mats Björkman, and Mats Bjorkman. *Adaptive online data compression.* In *IEEE High Performance Distributed Computing*. 2002.

[162] Oliver P. John and Veronica Benet-Martinez. *Handbook of research methods in social and personality psychology*. Wiley Online Library, 2000.

[163] M Tim Jones. *Inside the Linux 2.6 Completely Fair Scheduler*. IBM developerWorks, (December):1–9, 2009.

[164] Ajay Joshi, Lieven Eeckhout, Robert H. Bell, and Lizy K. John. *Distilling the essence of proprietary workloads into miniature benchmarks*. ACM Transactions on Architecture and Code Optimization, 5(2):1–33, aug 2008.

[165] Stefan Karlsson and Erik Hansson. *Lossless Message Compression.* Bachelor thesis, Mälardalen University, 2013.

[166] Rich Karpinski. *2016 Trends in Mobile Telecom.* Technical report, 451 Research, 2016.

[167] Nima Khalilzad. *Adaptive and Flexible Scheduling Frameworks for Component-Based Real-Time Systems*. Ph.D. thesis, Mälardalen University, nov 2015.

[168] Lars-örjan Kling, Åke Lindholm, Lars Marklund, and Gunnar B Nilsson. *CPP — Cello packet platform*. Technical Report 2, Ericsson Review, 2002.

[169] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. *Using OS observations to improve performance in multicore systems*. IEEE Micro, 28(3):54–66, 2008.

[170] Björn Knutsson. *Increasing Communication Performance via Adaptive Compression*. In *Proceedings of the Seventh Swedish Workshop on Computer Systems Architecture*. Gothenburg, Sweden, 1998.

[171] Björn Knutsson and Mats Björkman. *Adaptive end-to-end compression for variable-bandwidth communication.* Computer Networks, 31(7):767–779, apr 1999.

[172] Jacek Kobus and Rafał Szklarski. *Completely Fair Scheduler and its tuning*. Technical report, 2009.

[173] Con Kolivas. *Staircase Process Scheduling Algorithm*. `https://lwn.net/Articles/87729/`, 2004. [Accessed 2017-08-28].

[174] Con Kolivas. *Rotating Staircase DeadLine(RSDL)*. `https://lwn.net/Articles/224865/`, 2005. [Accessed 2017-08-28].

[175] David Koufaty, Dheeraj Reddy, and Scott Hahn. *Bias Scheduling in Heterogeneous Multi-core Architectures*. EuroSys 2010, pages 125–138, 2010.

[176] N Krajnovic. *The design of a highly available enterprise ip telephony network for the power utility of Serbia company*. Communications Magazine, IEEE, 47(4):118–122, apr 2009.

[177] Chandra Krintz and Sezgin Sucu. *Adaptive on-the-fly compression.* IEEE Transactions on Parallel and Distributed Systems, 17(1):15 – 24, jan 2006.

[178] George Kyriazis. *Heterogeneous System Architecture : A Technical Review*. Technical report, AMD, 2012.

[179] Christoph Lameter. *Bazillions of Pages - The Future of Memory Management under Linux*. In *Proceedings of the Linux Symposium*, volume 1, pages 275–284. Ottawa, Ontario, Canada, 2008.

[180] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. *Online performance auditing*. In *Proceedings of ACM SIGPLAN Conference on Programming language design and implementation*, pages 239–251. jun 2006.

[181] Juri (Scuola Superiore Sant'Anna) Lelli, Giuseppe (Scuola Superiore Sant'Anna) Lipari, Dario (Scuola Superiore Sant'Anna) Faggioli, and Tommaso (Scuola Superiore Sant'Anna) Cucinotta. *An efficient and scalable implementation of global EDF in Linux*. Proceedings of the OSPERT 2011, page 10, 2011.

[182] Robert Lenz. *Media reviews*, volume 4. Sage, 3rd edition, 1981.

[183] David Levinthal. *Performance Analysis Guide for Intel ® Core ™ i7 Processor and Intel ® Xeon ™ 5500 processors*. Technical report, Intel, 2009.

[184] John Levon. *Oprofile*. `http://oprofile.sourceforge.net`, 2002. [Accessed 2018-02-07].

[185] Kurt Lewin. *Action research and minority problems*. Journal of Social Issues, 2(4):34–46, 1946.

[186] Chuanpeng Li, Chen Ding, and Kai Shen. *Quantifying the cost of context switch*. Proceedings of the 2007 workshop on Experimental computer science - ExpCS '07, (June):2–es, 2007.

[187] M.H.T. Ling. *COPADS, I: Distance Coefficients between Two Lists or Sets*. The Python Papers Source Codes, 2:2, 2010.

[188] Linux. *Perf*. `https://perf.wiki.kernel.org`, 2009. [Accessed 2017-02-28].

[189] Linuxcounter. *Lines of code of the Linux Kernel Versions*. `https://www.linuxcounter.net/statistics/kernel`. [Accessed 2018-02-07].

[190] J. S. Liptay. *Structural aspects of the System/360 Model 85, II: The cache*. IBM Systems Journal, 7(1):15–21, 1968.

[191] C. L. Liu and James W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. Journal of the ACM, 20(1):46–61, 1973.

[192] Fang Liu and Yan Solihin. *Understanding the behavior and implications of context switch misses*. ACM Transactions on Architecture and Code Optimization, 7(4):1–28, dec 2010.

[193] S Loosemore, Rm Stallman, and R McGrath. *The GNU C library reference manual*. Technical report, 2016.

[194] Robert Love. *Linux Kernel Development*. Pearson Education, 3rd edition, apr 2010.

[195] Chenyang Lu, J.a. Stankovic, G. Tao, and S.H. Son. *Design and evaluation of a feedback control EDF scheduling algorithm*. In *Proceedings 20th IEEE Real-Time Systems Symposium*. 1999.

[196] Martina Maggio, Juri Lelli, and Enrico Bini. *Rt-Muse: Measuring Real-Time Characteristics of Execution Platforms*. Real-Time Systems, 53(6):857–885, 2017.

[197] Jason Mars, Neil Vachharajani, Robert Hundt, Mary Lou Soffa, and Others. *Contention aware execution*. Proceedings of the 8th annual IEEE/ ACM international symposium on Code generation and optimization - CGO '10, page 257, 2010.

[198] JR Mashey. *War of the benchmark means: time for a truce*. ACM SIGARCH Computer Architecture News, 32(4):1–14, 2004.

[199] Terje Mathisen. *Pentium Secrets: Undocumented features of the Intel Pentium can give you all the information you need to optimize Pentium code*. Byte Magazine, 19(7):191—-192, jul 1994.

[200] Wolfgang Mauerer. *Linux ® Kernel Architecture*. 2008.

[201] John D McCalpin. *Memory Bandwidth and Machine Balance in Current High Performance Computers*. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, (May):19–25, 1995.

[202] Steven McGeady, Randy Steck, Glenn Hinton, and Atiq Bajwa. *Performance enhancements in the superscalar i960MM embedded microprocessor.* COMPCON Spring '91 Digest of Papers, 1991.

[203] Wes McKinney. *Data Structures for Statistical Computing in Python*. Proceedings of the 9th Python in Science Conference, 1697900(Scipy):51–56, 2010.

[204] Larry Mcvoy and Carl Staelin. *lmbench : Portable Tools for Performance Analysis*. In *Proceedings of the USENIX Annual Technical Conference*, pages 279–294. 1996.

[205] María Mejía, Adriana Morales-Betancourt, and Tapasya Patki. *Lottery scheduler for the Linux kernel*. DYNA, 82(189):216–225, 2015.

[206] C Mercer, R Rajkumar, and J Zelenka. *Temporal protection in real-time operating systems*. In *IEEE Workshop on Real-Time Operating Systems and Software (RTOSS)*, pages 79–83. 1994.

[207] Clifford W Mercer, Stefan Savage, and Hideyuki Tokuda. *Processor capacity reserves: operating system support for multimedia applications*. Proceedings of IEEE International Conference on Multimedia Computing and Systems MMCS-94, pages 90–99, 1994.

[208] Pierre Michaud. *Demystifying multicore throughput metrics*. IEEE Computer Architecture Letters, pages 10–13, 2012.

[209] Michal Mienik. *CPU burnin*. URL `http://cpuburnin.com`.

[210] Sparsh Mittal. *A Survey Of Techniques for Architecting and Managing Asymetric Multicore Processors*. Computing Surveys, 9219(c):1–13, 2015.

[211] Sparsh Mittal. *A Survey of Techniques for Cache Partitioning in Multicore Processors*. ACM Computing Surveys, 50(2):1–39, 2017.

[212] Sparsh Mittal and Jeffrey S. Vetter. *A Survey of CPU-GPU Heterogeneous Computing Techniques*. ACM Computing Surveys, 47(4):1–35, 2015.

[213] Vilhelm Moberg. *Din stund på jorden*. 1963.

[214] Naveed Ul Mustafa. *Enriching Enea OSE for Better Predictability Support Master of Science Thesis*. Master thesis, KTH, 2011.

[215] Nicholas Nethercote and Julian Seward. *Valgrind: A program supervision framework*. In *Electronic Notes in Theoretical Computer Science*, volume 89, pages 47–69. 2003.

[216] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. *A SLOC Counting Standard*. COCOMO II Forum., 2007:1–15, 2007.

[217] Bogdan Nicolae. *On the benefits of transparent compression for cost-effective cloud data storage*. In *Proceedings of Transactions on Large Scale Data and Knowledge Centered Systems*, volume 3, pages 167–184. 2011.

[218] Jakob Nielsen. *Nielsen's law of internet bandwidth. (online)*. `http://www.nngroup.com/articles/law-of-bandwidth/`, 1998. [Accessed 2018-01-01].

[219] Nokia Siemens Networks. *Long Term HSPA Evolution: Mobile Broadband Evolution beyond 3GPP Release 10 HSPA has Transformed Mobile Networks*. Technical report, Nokia Siemens Networks, 2010.

[220] Andrzej Nowak and Georgios Bitzes. *The overhead of profiling using PMU hardware counters*. Technical Report CERN Openlab Report, CERN, 2014.

[221] Andrzej Nowak, David Levinthal, and Willy Zwaenepoel. *Hierarchical cycle accounting: a new method for application performance tuning*. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 112–123. 2015.

[222] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. *Establishing a base of trust with performance counters for enterprise workloads*. In *USENIX Annual Technical Conference*, pages 541–548. 2015.

[223] S. Nussbaum and J.E. Smith. *Modeling superscalar processors via statistical simulation*. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24. 2001.

[224] Markus Oberhumer. *LZO (Lempel-Ziv-Oberhumer) Data Compression Library*. `http://www.oberhumer.com/opensource/lzo/`, 2013. [Accessed 2015-03-04].

[225] Oxford. *English Dictionary (online)*, 2014.

[226] Dipak Patil, Prashant Kharat, and Anil Kumar Gupta. *Study of Performance Counters and Profiling Tools*. In *Proceedings of 21st IRF International Conference.*, March, pages 45–49. Pune, India, 2015.

[227] Ulf Paulsson. *Some short advice to a PhD student.* Technical report, School of Economics and Management, Lund University, Lund, 2018.

[228] Igor Pavlov. *LZMA Software Development Kit*. `http://www.7-zip.org/sdk.html`, 2013. [Accessed 2015-03-27].

[229] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. *Scikit-learn: Machine Learning in Python*. Journal of Machine Learning Research, 12:2825–2830, 2012.

[230] Kai Petersen, C Gencel, and N Asghari. *Action research as a model for industry-academia collaboration in the software engineering context*. In *Proceedings of the 2014 international workshop on Long-term industrial collaboration on software engineering*, pages 55–62. 2014.

[231] Kai Petersen and Claes Wohlin. *Context in industrial software engineering research*. In *International Symposium on Empirical Software Engineering and Measurement*, pages 401–404. Orlando, Florida, USA, 2009.

[232] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Mart\'\in Ugarte, and Domagoj Vrgoč. *Foundations of JSON Schema*. Proceedings of the 25th International Conference on World Wide Web, pages 263–273, 2016.

[233] Andrej Podzimek and Lydia Y. Chen. *Transforming system load to throughput for consolidated applications*. Proceedings - IEEE Computer Society's Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS, pages 288–292, 2013.

[234] Andrej Podzimek, Lydia Y. Chen, Lubomír Bulej, Walter Binder, and Petr Tüma. *Showstopper: The partial CPU load tool*. Proceedings - IEEE Computer Society's Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS, 2015-Febru(February):510–513, 2015.

[235] Ian Poole. *Cellular Communications Explained : From Basics to 3G*. Elsevier, 1st edition, 2006.

[236] Calton Pu and Lenin Singaravelu. *Fine-Grain Adaptive Compression in Dynamically Variable Networks.* In *Proceedings of the International Conference on Distributed Computing Systems*, pages 685–694. 2005.

[237] J R Quinlan. *Induction of Decision Trees*. Mach. Learn., 1(1):81–106, mar 1986.

[238] K.A Ramya and M Pushpa. *A Survey on Lossless and Lossy Data Compression Methods*. International Journal of Computer Science and Engineering Communications, 4(1):1277–1280, 2016.

[239] John Regehr and John A. Stankovic. *Augmented CPU reservations: Towards predictable execution on general-purpose operating systems*. Real-Time Technology and Applications - Proceedings, pages 141–148, 2001.

[240] Lasse Mikkel Reinhold. *QuickLZ - Fast compression library for C, C# and Java*. `http://www.quicklz.com/`, 2011. [Accessed 2013-05-31].

[241] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. *Google-Wide Profiling: A continuous profiling infrastructure for data centers*. IEEE Micro, 30(4):65–78, 2010.

[242] Martin Ringwelski, Christian Renner, Andreas Reinhardt, Andreas Weigel, and Volker Turau. *The hitchhiker's guide to choosing the compression algorithm for your smart meter data*. In *2nd IEEE ENERGYCON Conference & Exhibition*, pages 935–940. 2012.

[243] D. Gordon E Robertson and James J. Dowling. *Design and responses of Butterworth and critically damped digital filters*. Journal of Electromyography and Kinesiology, 13(6):569–573, 2003.

[244] Colin Robson. *Real world research*. Blackwell, Oxford, 2nd edition, 2002.

[245] James Roche. *Adopting DevOps practices in quality assurance*. Communications of the ACM, 56(11):38–43, 2013.

[246] Ronghua Zhang, Chenyang Lu, T.F. Abdelzaher, and J.a. Stankovic. *ControlWare: a middleware architecture for feedback control of software performance*. Proceedings 22nd International Conference on Distributed Computing Systems, pages 301–310, 2002.

[247] Jussi Rosendahl and Leila Abboud. *Nokia buys Alcatel to take on Ericsson in telecom equipment*. `http://www.reuters.com/article/2015/04/15/nokia-alcatel-lucent-ma-idUSL5N0XC0X220150415`, 2015. [Accessed 2018-02-07].

[248] Hans Rosling. *Hans Rosling is lecturing the Danish Radio Channel 2 program (deadline) host Adam Holm*, sep 2015.

[249] Steven Rostedt and Darren V. Hart. *Internals of the RT Patch*. In *Proceedings of the Linux Symposium*, page 318. 2007.

[250] Mehrin Rouhifar. *A Survey on Scheduling Approaches for Hard Real-Time Systems*. International Journal of Computer Applications, 131(17):41–48, 2015.

[251] Kim Rowe. *Time to market is a critical consideration*. `http://www.embedded.com/electronics-blogs/industry-comment/4027610/Time-to-market-is-a-critical-consideration`, 2010. [Accessed 2015-03-04].

[252] Dr. Winston W. Royce. *Managing the Development of large Software Systems*. Ieee Wescon, (August):1–9, 1970.

[253] Per Runeson. *Case Study Research or Anecdotal Evicende?* Technical report, Lunc University, 2010.

[254] Per Runeson and Martin Höst. *Guidelines for conducting and reporting case study research in software engineering.* Empirical Software Engineering, 14(2):131–164, dec 2008.

[255] Olof Rutgersson and Simon Boman. *Replacing OSE with Real Time capable Linux*. Master thesis, Department of Computer and Information Science, 2009.

[256] Rafael H. Saavedra and Alan J. Smith. *Measuring cache and TLB performance and their effect on benchmark runtimes*. IEEE Transactions on Computers, 44(10):1223–1235, 1995.

[257] J. C. Saez, A. Pousa, R. Rodriíguez-Rodriíguez, F. Castro, and M. Prieto-Matias. *PMCTrack: Delivering Performance Monitoring Counter Support to the OS Scheduler*. The Computer Journal, 60(1):60–85, jan 2017.

[258] J.C. Saez, A. Pousa, R. Rodrigues-Rodrigues, F. Castro, and M Prieto-Matias. *PMCTrack: Delivering Performance Monitoring Counter Support to the OS Scheduler*. The Computer Journal, 82(September):29–35, 2016.

[259] Aksel Sandemose. *En flykting korsar sitt spår*. Themis Förlag, 1933.

[260] Conrad Sanderson. *Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments*. Technical Report, NICTA:1–16, 2010.

[261] Conrad Sanderson and Ryan Curtin. *Armadillo: a template-based C++ library for linear algebra*. The Journal of Open Source Software, 1:26, 2016.

[262] Max Schuchard, Eugene Y. Vasserman, Abedelaziz Mohaisen, Denis Foo Kune, Nicholas Hopper, and Yongdae Kim. *Losing Control of the Internet: Using the Data Plane to Attack the Control Plane*. In *Computer and Communications Security*, pages 726–728. 2010.

[263] Carolyn B. Seaman. *Qualitative methods in empirical studies of software engineering*. IEEE Transactions on Software Engineering, 25(4):557–572, 1999.

[264] Freescale Semiconductor. *QorIQ Communications Processor Product Brief*. Freescale, 2008.

[265] Benjamin Serebrin and Daniel Hecht. *Virtualizing performance counters*. In *European Conference on Parallel Processing*. 2011.

[266] Julian Seward. *BZIP2, a program and library for data compression compression.* `http://www.bzip.org`, 2013. [Accessed 2015-03-04].

[267] L Sha, T Abdelzaher, K E Arzen, A Cervin, T Baker, A Burns, G Buttazzo, M Caccamo, J Lehoczky, and A K Mok. *Real time scheduling theory: A historical perspective*. Real-Time Systems, 28(2-3):101–155, 2004.

[268] Timothy Sherwood and Brad Calder. *Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications.* In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September. 2001.

[269] Timothy Sherwood and Greg Hamerly. *Automatically Characterizing Large Scale Program Behavior.* In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems*. 2002.

[270] Silberschatz, Galvin, and Gagne. *Module 20: The Linux System*. Technical report, 2002.

[271] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating systems concepts*. 9. Wiley, 2013.

[272] Dag Sjøberg, Tore Dybå, and Magne Jørgensen. *The Future of Empirical Methods in Software Engineering Research*. Future of Software Engineering, SE-13(1325):358–378, 2007.

[273] Viveka Sjöblom. *OSE och Linux - En studie om prestanda*. Master thesis, Uppsla University, 2008.

[274] Alan Jay Smith. *Cache Memories*. ACM Computing Surveys, 14(3):473–530, 1982.

[275] Stackoverflow. *Generate Instruction Cache misses*. `http://stackoverflow.com/questions/9793660/what-are-the-causes-for-instruction-cache-miss`. [Accessed 2018-02-07].

[276] Niklas Ståhle. *Implementing Transaction Tracing in Real-Time Systems Master of Science Thesis Implementing Transaction Tracing in Real-Time Systems*. Ph.D. thesis, Royal Institute of Technology, 2009.

[277] John A. Stankovic. *A Serious Problem for Next-Generation Systems*. Computer, 21(10):10–19, 1988.

[278] Sezgin Sucu and Chandra Krintz. *Ace: A resource-aware adaptive compression environment.* In *Proceedings of International Conference of Information Technology: Coding and Computing*, pages 183 – 188. 2003.

[279] Angelina Sundström, Gunnar Widforss, Malin Rosqvist, and Anette Hallin. *Industrial PhD Students and their Projects*. Procedia Computer Science, 100:739–746, 2016.

[280] Andrew S Tanenbaum, Jorrit N Herder, and Herbert Bos. *Can We Make Operating Systems Reliable and Secure?* IEEE, (May), 2006.

[281] Lingjia Tang, Jason Mars, Neil Vachharajani, and Mary Lou Soffa. *The Impact of Memory Subsystem Resource Sharing on Datacenter Applications Categories and Subject Descriptors*. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 283–294. 2011.

[282] Gregory Tassey. *The economic impacts of inadequate infrastructure for software testing*. Technical Report 7007, National Institute of Standards and Technology, 2002.

[283] Gary Taylor and David Russell. *PwC Capex is king - a new playbook for telecoms execs*. Technical Report December, PwC, 2014.

[284] Paul Taylor. *Battle lines are drawn for the future of 4G*. `http://www.ft.com/intl/cms/s/0/399b1508-d9d8-11dc-bd4d-0000779fd2ac.html{#}axzz1va5rEtRx`, 2008. [Accessed 2015-03-04].

[285] Techcrunch. *Apples App Store Hits 50 Billion Downloads, 900K Apps*. `http://techcrunch.com/2013/06/10/apples-app-store-hits-50-billion-downloads-paid-out-10-billion-to-developers/`, 2013. [Accessed 2015-03-04].

[286] Telecomasia. *Faster time to market with next-gen OSS*. `http://www.telecomasia.net/content/faster-time-market-next-gen-oss`, 2012. [Accessed 2015-03-04].

[287] Ericsson Nikola Tesla, Denis Duka, and Keywords Cpp. *Connectivity Packet Platform in the GSMIWCDMA Network*. In *Access*, June, pages 7–9. 2006.

[288] Linus Torvalds. *Talk is cheap. Show me the code.* `https://lkml.org/lkml/2000/8/25/132`, 2000. [Accessed 2018-02-23].

[289] Dennis Upper. *The unsuccessful self-treatment of a case of "writer's block"*. Journal of Applied Behavior Analysis, 7(3):1311997, 1974.

[290] Stéfan Van Der Walt, S. Chris Colbert, and Gaël Varoquaux. *The NumPy array: A structure for efficient numerical computation*. Computing in Science and Engineering, 13(2):22–30, 2011.

[291] Steven H. VanderLeest. *ARINC 653 hypervisor*. AIAA/IEEE Digital Avionics Systems Conference - Proceedings, pages 1–20, 2010.

[292] Girish Venkatachalam. *Multimedia Dynamite*. Linux Journal, oct 2007.

[293] Hans Vestberg. *Ericsson unveils new products, partnerships and increased market share.* In *Proceedings of at Mobile World Conference*. 2012.

[294] Wan Vinny. *CPP in LTE Overview*. Technical report, Ericsson, 2014.

[295] Xiaorui Wang, Xing Fu, Xue Liu, and Zonghua Gu. *PAUC : Power-Aware Utilization Control in Distributed Real-Time Systems*. EEE Transactions on Industrial Informatics, 6(3):302–315, 2010.

[296] Xiaorui Wang, Dong Jia, Chenyang Lu, and Xenofon Koutsoukos. *DEUCON: Decentralized end-to-end utilization control for distributed real-time systems*. IEEE Transactions on Parallel and Distributed Systems, 18(7):996–1009, 2007.

[297] Reinhold P. Weicker. *Dhrystone: a synthetic systems programming benchmark*. Communications of the ACM, 27(10):1013–1030, 1984.

[298] Terry A Welch. *A Technique for High-Performance Data Compression*. Computer, 17(6):8–19, 1984.

[299] Benjamin Welton, Dries Kimpe, Jason Cope, Christina M. Patrick, Kamil Iskra, and Robert Ross. *Improving I/O Forwarding Throughput with Data Compression*. 2011 IEEE International Conference on Cluster Computing, pages 438–445, sep 2011.

[300] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. *Dynamic storage allocation: A survey and critical review*. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 986, pages 1–116. 1995.

[301] Y. Wiseman, K. Schwan, and P. Widener. *Efficient end to end data exchange using configurable compression*. ACM SIGOPS Operating Systems Review, pages 4–23, 2005.

[302] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software - An Introduction*. Springer Science+Business Media LLC, Lund, 2000.

[303] C. S. Wong, I. K T Tan, R. D. Kumari, J. W. Lam, and W. Fun. *Fairness and interactive performance of O(1) and CFS Linux kernel schedulers*. Proceedings - International Symposium on Information Technology 2008, ITSim, 3(1), 2008.

[304] Yuejian Xie and Gabriel H. Loh. *Dynamic Classification of Program Memory Behaviors in CMPs*. Cmp-Msi'08, (June):1–9, 2008.

[305] Kaiqi Xiong and Sang Suh. *Resource provisioning in SLA-based cluster computing*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6253 LNCS:1–15, 2010.

[306] Ahmad Yasin. *A Top-Down method for performance analysis and counters architecture*. IEEE International Symposium on Performance Analysis of Systems and Software, pages 35–44, 2014.

[307] Murat Yuksel, K. K. Ramakrishnan, Shivkumar Kalyanaraman, Joseph D. Houle, and Rita Sadhvani. *Quantifying overprovisioning vs. Class-of-Service: Informing the net neutrality debate*. Proceedings - International Conference on Computer Communications and Networks, ICCCN, 2010.

[308] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. *Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality*. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 299–308. 2012.

[309] Omar U Pereira Zapata and Pedro Mej. *EDF and RM multiprocessor scheduling algorithms: Survey and performance evaluation.* In *Seccion de Computacion*, pages 1–24. 2005.

[310] Gerd Zellweger, Denny Lin, and Timothy Roscoe. *So many performance events , so little time*. APSys '16, 2016.

[311] Jiangtao Zhang, Hejiao Huang, and Xuan Wang. *Resource provision algorithms in cloud computing: A survey*. Journal of Network and Computer Applications, 64:23–42, 2016.

[312] Li Zhang, Dhruv Gupta, and Prasant Mohapatra. *How expensive are free smartphone apps?* ACM SIGMOBILE Mobile Computing and Communications Review, 16(3):21–32, dec 2012.

[313] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. *Addressing shared resource contention in multicore processors via scheduling*. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 129. 2010.

Nu är det slut. [*]

— Fem myror är fler än fyra elefanter

[*]This quote is taken from the concluding scene of a TV-show, famous to all Swedish children born during the 70's. A pink elephant exclaims "This is the end" and then skedaddles away while trumpeting with its trunk.