A Dynamic and Component-Based Process Scheduler Framework for Heterogeneous Many-Core Systems

vorgelegt von
Dipl.-Ing.
Anselm Busse
geb. in Berlin

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr.-Ing. Stefan Tai
Gutachter:	Prof. Dr. Hans-Ulrich Heiß
Gutachter:	Prof. Dr.-Ing. Gero Mühl
Gutachter:	Prof. Dr. César De Rose

Tag der wissenschaftlichen Aussprache: 13. Dezember 2016

Berlin 2016

Abstract

Since the first decade of the 21st century, improvements in computer performance are no longer achieved through increasing clock rates but parallelization and specialization. As a result, heterogeneous many-core systems are becoming more and more common. Leaving their niche in highly specialized computing, they have to be supported by a general purpose operating system. A major part of the support that has to be delivered will be done by the process scheduler as it has to tailor the task order and placement to those systems. Current approaches to scheduler architecture are not capable of coping with the resulting challenges. In particular, the state of the art falls short to provide ways and means to react to new developments in hardware technology quickly and lacks the capability to enable the adaptation of the scheduler to the changed environment. Therefore, the original contribution to the knowledge of this dissertation is a new approach to the architecture of the operating system's process scheduler.

This dissertation introduces a component-based approach to the process scheduler architecture that allows the decomposition of the scheduling problem into distinct parts rather than the monolithic approach as it is state of the art. Components are connected through Pipes that are an advanced form of runqueues suitable for the component-based approach. Besides Pipes, information is distributed among the components through a publish–subscribe-based message system. Components can change the order of tasks and distribute or merge tasks from or to several pipes. This allows a flexible scheduler design both during development and runtime: The developer can reuse existing components and build new schedulers from existing implementations. Through the explicit layout of the components code-paths, possible bottlenecks become easier to identify compared to a monolithic approach. Through the separation, it is also feasible to change the entire scheduling policy during runtime. This enables an optimal shaping of the scheduler to the needs of the current working set and the properties of the underlying hardware architecture.

The components are embedded into a framework that allows a comprehensible development of schedulers that scale even in many-core systems. The framework approach permits the integration into several runtime systems without changing the actual scheduler implementation. This dissertation proves the feasibility through integrating the framework into major open source kernels: Linux and FreeBSD. Based on this exemplary implementation, the properties of the approach are evaluated. The studies show that the component-based scheduler framework is scalable for hundreds of cores, the overhead is quantified and the benefits of having well-defined interfaces are demonstrated. Finally, the advantages of a dynamic adaptation of scheduling strategies at runtime are shown.

Zusammenfassung

Leistungssteigerungen in Computern werden seit dem ersten Jahrzehnt des 21. Jahrhunderts nicht mehr durch fortlaufende Erhöhung der Taktfrequenzen erreicht, sondern durch zunehmende Parallelisierung und Spezialisierung. Auf Grund dessen gewinnen heterogene Many-Core-Systeme zunehmend an Bedeutung. Somit müssen auch zunehmend Betriebssysteme Unterstützung bieten. Eine wichtige Rolle hierbei wird die Prozessablaufplanung und -zuordnung spielen, die für diese Systeme angepasst werden müssen. Aktuelle Architekturen für die Implementierung der Prozessablaufplanung können mit den resultierenden Herausforderungen nicht Schritt halten. Insbesondere ist es beim aktuellen Stand der Technik nicht möglich, auf Innovationen im Hardwarebereich rasch zu reagieren und die Prozessablaufplanung an die neuen Gegebenheiten anzupassen. Aus diesem Grund besteht der Beitrag zur Wissenschaft dieser Dissertation in einer neuartigen Architektur für die Prozessablaufplanung.

Diese Arbeit präsentiert einen neuen komponenten-basierten Ansatz zur Architektur der Prozessablaufplanung, welcher die Aufteilung des Planungsalgortithmuses in Komponenten erlaubt. Komponenten werden durch Pipes verbunden, welche eine Weiterentwicklung der bisher verwendeten Runqueues darstellen. Ferner werden Informationen unter den Komponenten durch ein publish-subscribe-basiertes Nachrichtensystem verbreitet. Komponenten können die Reihenfolge und die Verteilung der eingehenden Prozesse auf die ausgehenden Pipes bestimmen. Der Entwickler kann bestehende Komponenten wiederverwenden und neue Prozessablaufplanungsimplementierungen aus bestehenden erzeugen. Die vorgestellte Architektur erlaubt das einfache Auffinden von Flaschenhälsen durch ein expliziteres Layout des Prozessplaners verglichen zum monolithischen Ansatz. Ferner erlaubt die Architektur durch ihre expliziten Schnittstellen die Änderung der Planungsimplementierung zur Laufzeit. Der Prozessablaufplaner wird hierdurch optimal auf die Gegebenheiten des Systems angepasst.

Die Komponenten sind in ein Framework eingebettet, welches die transparente Entwicklung von Prozessplanungsimplementierungen ermöglicht. Der framework-basierte Ansatz erlaubt die einfache Einbindung bestehender Implementierungen ohne größere Änderungen in verschiedenen Laufzeitumgebungen. Diese Dissertation demonstriert die Machbarkeit des Ansatzes durch die Integration in zwei der größten offenen Betriebssystemkerne: Linux und FreeBSD. Basierend auf dieser Beispielimplementierung werden im Verlauf dieser Arbeit die Eigenschaften des Ansatzes evaluiert und diskutiert. Die Arbeit zeigt, dass der Ansatz für hunderte von Rechenkernen skaliert, sie quantifiziert den Overhead des Ansatzes und stellt die Vorteile wohl definierter Schnittstellen in diesem Anwendungsbereich am Beispiel vor. Schlussendlich werden die Vorzüge des Wechselns der Planungsstrategie zur Laufzeit dargestellt.

Acknowledgment

The research included in this dissertation could not have been performed if not for the assistance, patience, and support of many individuals. I would like to extend my gratitude first and foremost to my thesis advisor Hans-Ulrich Heiß for the continuous support of my doctoral study and related research, for his patience, motivation, and immense knowledge. His guidance helped me during all the time of research and writing of this thesis.

Furthermore, I like to thank Gero Mühl for motivating me to join the research group and pick up my doctoral study and for the many inspiring and insightful discussions also beyond the scope of operating systems research. I would also like to thank César De Rose for his insightful comments and encouragement in order to finish my thesis and broaden my research from various perspectives.

My sincere thanks also go to Reinhardt Karnapke, Helge Parzyjegla, Matthias Diener, and Jan Richling for their support in both the research and especially the revision process that has lead to this document. Their knowledge and understanding allowed me to fully express the concepts behind this research.

I thank my fellow colleagues Daniel Graff, Jörg Schneider, Arnd Schröter, Jan Schönherr, Mohannad Nabelsee, and Stefan Sydow for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the recent years. I would also like to extend my appreciation to Gabriele Wenzel, Daniela kleine Burhoff, and Matthias Druve for their administrative support and helping me to keep my mind focused on my research.

I would additionally like to thank Anton Eisenbraun, Jens Krieg, René Sechting, and Daniel Röhrig who helped to implement the prototype and assisted me in bringing my ideas and concepts to reality.

Finally, I would like to extend my deepest gratitude to my mother Lilli Busse: Without her love, support, and understanding I could never have completed this doctoral degree.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AOP	Aspect-Oriented Programming
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
ASIC	Application-Specific Integrated Circuit
BFS	Brain Fuck Scheduler
CFS	Completely Fair Scheduler
CoBaS	Component Based Scheduling
COP	Component-Oriented Programming
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DSL	Domain Specific Language
DSP	Digital Signal Processor
ELF	Executable and Linking Format
FCFS	First-Come, First-Served
FLOPS	Floating-Point Operations per Second
FPGA	Field-Programmable Gate Array
GPGPU	General-Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HPC	High Performance Core
IC	Integrated Circuit
I/O	Input/Output
ISA	Instruction Set Architecture
KLT	Kernel-Level Thread
KVM	Kernel-Based Virtual Machine
LCFS	Last-Come, First-Served
LPC	Low Performance Core
LUT	Lookup Table
MIC	Many Integrated Core
MIMD	Multiple Instruction, Multiple Data
NAS	Numerical Aero Dynamic Simulation
NoC	Network on Chip
NUMA	Non-Uniform Memory Access
OOP	Object-Oriented Programming
OpenCL	Open Computing Language
PE	Processing Element
POSIX	Portable Operating System Interface

Pthread	POSIX Thread
QoS	Quality of Service
SIMD	Single Instruction, Multiple Data
SISD	Single Instruction, Single Data
SLA	Service-Level Agreement
SMT	Simultaneous Multithreading
SoC	System on Chip
SRT	Shortest Remaining Time
TCB	Thread Control Block
ULT	User-Level Thread

CHAPTER 1

Introduction

1.1 Motivation

Computer technology is one of the most influential inventions in human history and a central driving force behind innovation in the 20^{th} and early 21^{st} centuries. It was indispensable for several scientific and technological breakthroughs like space exploration, gene sequencing, and the creation of the Internet to name only a few. The most valuable property, which made the computer successful, is its programmability. This distinguishes the computer from all other tools humans invented. From the beginning, it did not serve a specific purpose but could be used ubiquitously. This implicates that a computer cannot fulfill a meaningful purpose on its own; it needs a programming to do so.

In the early days of computer history when computers were slow and had only basic input and output systems, the programming was entered by a human manually. The programs back then were not interactive and did not depend on each other. So the operator could decide in which order the programs were run. As computing time was extremely precious, this order was based on a given schedule that was created based on different factors, e.g., importance, job submission order, or costs. When computers became faster, the manual program input was replaced by punched cards. The job scheduling was now given by order of the programs in the punched cards stack but did not differ much from the manual input.

With the introduction of personal computers, a new challenge was introduced. Now, the work consisted not only of jobs that had to be executed without additional input from the beginning to the end, but the user interacted with it. This led to the introduction of multitasking, where two or more jobs run concurrently on one machine. In the beginning, it was limited to cooperative multitasking, where a job or application voluntarily cedes processing time to another. This made it necessary to introduce a new facility to the system: the *process scheduler*. It decides which program is served next as soon as another program releases the processor. With increasing computational power, the concept of cooperative multitasking evolved into preemptive multitasking, where a program no longer needs to release the processor voluntarily but can be forced to release it by the operating system. With this evolution, the process scheduler is not only responsible for

deciding which program to run next, when consulted, but also which share of processing resources gets allocated to every executed program.

Today, the scheduler is an indispensable component of nearly every operating system and fundamentally determines the general behavior of the operating system itself and the whole system overall, e.g., it decides whether the system is more suited for embedded, desktop, server, or mainframe requirements. With the omnipresent use of computer technology, the diversity and complexity of scheduling algorithms also increased over time. The diversity, too, grew by the fact that single computer systems are not sufficient to solve more complicated problems or sustain service for a large user base.

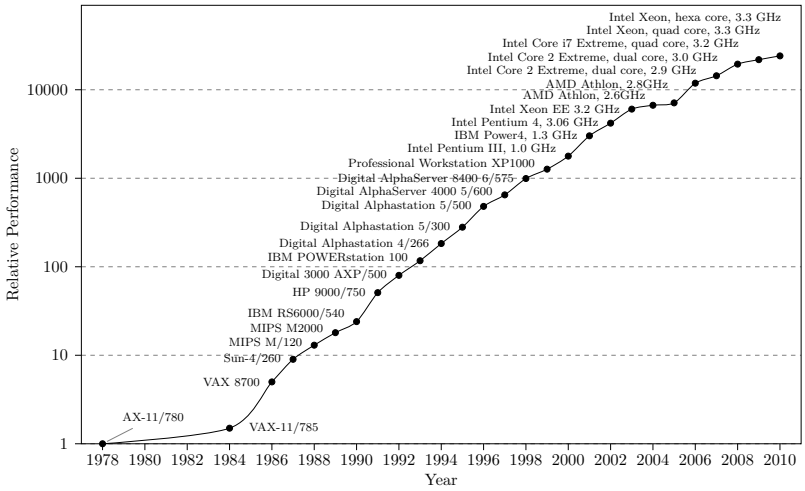
Without exaggeration, it can be stated that the scheduler is a fundamental component in system software design to unfold the computational power of future systems. However, the design and implementation of scheduling policies is a complicated process. Most operating systems have grown over many years and do not use modern software design methods. An accelerating pace of changes in the hardware architecture and rising degree of concurrency make the task of developing future scheduling strategies even harder.

1.2 Future Challenges

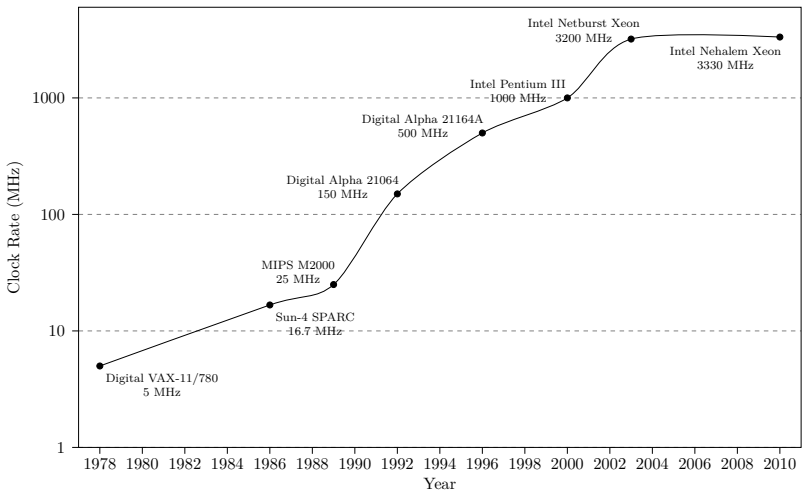
Operating system design has been facing more and more challenges in the recent years. They stem from the increasing pace and fundamental changes in innovation in information technology, especially computer architecture. This section gives an introduction to those challenges.

1.2.1 Hardware Innovation

In computer technology, innovation cycles are becoming shorter and shorter, which can be attributed to the exponential growth in informatics. This can be traced back to *Moore's* law [112] and is further elaborated by Kurzweil [93] in the *Law of Accelerating Returns*. This was no issue for system software design as long as performance growth was achieved by increasing clock rates. It was not necessary to adapt the system software widely to make the additional power accessible to the user. However, this paradigm radically changed around the year 2003 when processor designs hit the power wall [70]. As the increase of clock rates flatlined (cf. Fig. 1.1b), the performance improvements were still significant even though not as high as between the years 1986 and 2003 (cf. Fig. 1.1a). Since 2003, performance improvements through increased clock rates were only marginal. Now, the main performance growth was achieved through other means. The first of those was moving from single-core to multi-core CPUs. At first, this did not pose an issue for the operating systems, as the multi-core model was already known from multiprocessor systems in high-performance computing environments. Early multi-core CPUs even had a similar architecture as multiprocessor systems. However, multi-core systems evolved and became diverse, which made it more challenging for the operating system to manage them.



(a) Growth in processor performance since the late 1970s [70, p. 3].



(b) Growth in processor clock rates of since the late 1970s [70, p. 24].

Figure 1.1: Comparison between processor performance and clock rates since the late 1970s. The performance of the different systems is plotted in relation to the VAX-11/780 based on the SPEC benchmark [S40]. The machines were benchmarked with the current SPEC versions of their time. The performance of newer machines is estimated through a scaling factor between the different SPEC versions.

One example of that issue was the introduction of the Bulldozer microarchitecture by AMD in 2011. Bulldozer CPUs were unique compared to other CPUs back then, because, unlike in a real multi-core CPU, two Bulldozer cores shared one floating-point unit. Because the Bulldozer based CPUs were designated as desktop CPUs, one of the main operating systems running those CPUs was Microsoft Windows. However, because of the unique architecture, using Windows together with a Bulldozer based CPU resulted in performance issues with multi-threaded applications [7]. The issue could be traced back to the Windows process scheduler that treated every processing unit as an independent physical core, which they were not because of the shared floating-point unit. Even though the product was already publicly available in October 2011 and undoubtedly even earlier to Microsoft, they released the first scheduler fix [108] only in December 2011. Even worse, this patch did not fix the problem and had to be withdrawn. A functional patch was only released in January 2012 [109, 110]. This example is just one brief episode giving an idea of the challenges operating systems in general and the process scheduler, in particular, will face in the future due to changing architectures. However, more significant changes in hardware architecture can be expected in the future. Based on *Amdahl's* law [5] and its extension for multi-core systems [73], the performance improvement through additional cores cannot be maintained indefinitely. Therefore, new fundamentally different architectures have to be employed, if performance improvements are still to be achieved. The Intel Tera-Scale research program [79] for example gave first insights how such architectures could look like. Since then, many new system architectures emerged.

1.2.2 Heterogeneous Systems

Another challenge in scheduler design are heterogeneous systems. In such a system, the cores or Processing Elements (PEs) of the system have different properties. The degree of heterogeneity can range from very low like, e.g., minor differences in energy consumption to very high like, e.g., differences in the programming model or even special purpose PEs. Depending on the degree of heterogeneity, issues of different severities can arise.

The following example illustrates the struggle in scheduler design again, this time originating from the emergence of heterogeneous systems. The fast rise of mobile computing in the recent years lead to new requirements in CPU design [104], which is especially the need to conserve energy. First, it was countered with microarchitectures optimized for high efficiency, e.g., maximizing the number of Floating-Point Operations per Second (FLOPS) per Watt. The most prominent example of this effort is the ARM architecture developed by *ARM Holding*. However, with the rising computational demand of mobile applications, a specialized microarchitecture was not any longer sufficient to keep the balance between high efficiency and peak performance. For this reason, ARM introduced the big.LITTLE architecture, which was announced in October 2011 [8]. The big.LITTLE concept combines highly energy efficient low-performance cores with energy demanding high-performance cores in one multi-core CPU. It is an example

of a heterogeneous multi-core system even though the degree of heterogeneity is very low as the cores have the same instruction set and mainly differ in performance. The performance difference poses a challenge for the scheduler as a past common assumption in multi-core scheduler design was that cores are similar and therefore interchangeable. This disruption led to the fact that a scheduler for the Linux kernel – which is the main system software running this sort of CPUs – that can handle all cores at the same time was only available in July 2013 [67]. The changes necessary to the Linux scheduler were so severe that it remained until 2015 only available for the kernel that was stable in July 2013.

Again, this is only one brief episode of challenges in the scheduling of heterogeneous systems. For example, recent research has shown that even small differences in hardware architecture have to be considered by the scheduler to reach optimal performance [33]. However, other more disruptive challenges regarding heterogeneity are present today or will arise in the future. For example, certain computational problems can benefit from processing accelerators that yield a better performance both in absolute processing time and energy consumption per solved problem [23, 68, 75, 106, 167]. Graphics Processing Units (GPUs) can be considered as such a processing accelerator. With the availability of the Compute Unified Device Architecture (CUDA) [S9] in 2007 and Open Computing Language (OpenCL) [S29] in 2009, General-Purpose Computing on Graphics Processing Units (GPGPU) [168] that uses GPUs as processing accelerators became widely available. GPGPU has a completely different computing and programming model compared to general purpose PEs [151]. However, from the resource perspective of the operating system, those devices are not significantly different from general purpose CPUs: They consist of PEs that can be used by different tasks in a time division manner. Therefore, processing accelerators should be managed and assigned by a process scheduler. Currently, the management is realized through specialized libraries often in the user space and has only limited scheduling capabilities. However, the scheduling and sophisticated management of processing units for GPGPU are ongoing research, e.g., Bautin et al. [21] or Kato et al. [88, 89].

Besides GPUs, several other kinds of accelerators exist with different programming models. On the one side of the spectrum, there is, for example, the Intel Many Integrated Core (MIC) architecture [42, 49], which is marketed under the name Intel Xeon Phi. Even though the first generation of the MIC architecture was released entirely and the second one partially as dedicated processing accelerator, it has a programming model that is very similar to the model of multi-core CPUs. On the other side of the spectrum, there are Application-Specific Integrated Circuits (ASICs) that have little to none programmability and are heavily tailored to special purposes like e.g. data compression [173], mining of crypto currencies[1] [26, 27], machine learning [86], or image processing [111].

[1] The money supply of a crypto currency is controlled by a cryptographic algorithm with a high computational complexity rather than a central bank [66].

1.2.3 Reconfigurable Computing

With the challenges pointed out in the previous subsections, the scheduling problem became more complex but remained mostly the same: several tasks with different requirements had to be assigned over time to a fixed number of PEs with specific capabilities. The hardware architecture is static and does not change during the runtime of the system. With Field-Programmable Gate Array (FPGA) technology,[2] this might change drastically in the near future. Today, FPGAs are commonly used as processing accelerators as mentioned in Section 1.2.2 or as co-processors to achieve hard real-time through jitter and latency reduction. In those scenarios, FPGAs are statically configured and do not change their functionality during runtime. Therefore, in those scenarios, they are not different from specialized PEs. However, this limitation is not necessary, because they can be reconfigured with almost no limitation during runtime. Considering this aspect, the scheduling problem becomes entirely different from many points of view.

A logic block or even a Lookup Table (LUT) might be considered as the smallest PE of a FPGA. A FPGA has hundreds of thousands of those, which is an order of 100 or 1000 more than PEs in a many-core chip. This turns the scheduling problem towards a continuous problem rather than a discrete one. First because of the sheer number, and second because single LUTs, contrary to other PEs, cannot fulfill computations completely on their own. Furthermore, many side conditions have to be considered. In practice, as a LUT virtually cannot process data on its own, several adjacent LUTs have to be combined. Moreover, certain timing conditions have to be met also influencing the number of LUTs needed and have to be taken into account during the scheduling decision. In addition to simple logic blocks, modern FPGAs incorporate specialized processing blocks like Digital Signal Processors (DSPs) or I/O blocks that have to be considered as well when assigning the resources of a FPGA.

Today, FPGAs can still be considered a niche product in servers and especially workstations and desktops. This might change soon as Intel, by far one of the biggest suppliers for data center processors, will integrate FPGA technology in its server processors [30]. Also, with the Zynq series [180], Xilinx combines established and widely used ARM cores tightly with their FPGA technology. This will make FPGAs more widely available and therefore another processing resource for the operating system to manage. The benefits of such systems are discussed, e.g., by Chung et al. [38].

The management of FPGAs in the operating system was already suggested by Brebner [29] in 1996. Since then, several projects emerged with the goal of making FPGAs readily available to the user. Such projects are BORTH [145] that supports hardware tasks abstracting the hardware configuration as Unix process or ReconOS [99] that enables a programming model for a FPGA similar to POSIX Threads (Pthreads).

[2] For details on FPGA technology itself refer to, e.g., Compton and Hauck [40].

1.2.4 Dark Silicon

The notion of *dark silicon* was introduced by Esmaeilzadeh et al. [53]. In their 2011 paper, they project that multicore microchips produced with a 8 nm process required more than 50 % of the chips' transistors to be powered off at all time. The powered off part of a chip is denoted as dark silicon. The reason for that restraint is the same as the introduction of multi-core and heterogeneous systems, the breakdown of *Dennard scaling* [43] beginning in the mid of the first decade of the 21st century. Dennard scaling states that the total chip power for a given die size stays the same in each process generation. However, this statement holds no longer as transistor energy efficiency only improves by a factor of 1.4 every two years while transistor density continues to improve, following *Moors* law, by a factor of two in the same time frame [158]. A consequence of this development is the possibility that in the future energy will be by far the biggest constraint regarding chip design in contrast to the die area and transistor count in the past.

The previous sections of this dissertation already discussed initial approaches to reduce the impact of the breakdown of Dennard scaling. However, those approaches will most likely not overcome the necessity that certain parts of a chip have to be dark and even further specialized in the future [cf. 159]. This development has a significant impact on the operating systems scheduler. Even though today certain power constraints have to be considered, with dark silicon the whole problem becomes much more complicated. Whereas today the power management can be seen as a micromanagement problem as only the overall system energy consumption is important, in the era of dark silicon the scheduler has to do micromanagement to ensure that the microchips of a system are not overheating and that the system can still unfold its full processing power potential.

1.2.5 Virtualization

Virtualization is another answer to the increasing computational power of modern hardware. As modern servers are often not fully utilized by a single service, for economic reasons it makes sense to run several services on one machine. However, contrary to multi-tasking, different services need different runtime environments. Furthermore, especially in the context of cloud computing, various services from different clients have to be fully isolated from each other. Another benefit of virtualization is that running services can be seamlessly migrated to a new host with minimal interruption.

Regarding the process scheduling, virtualization introduces another challenge, as the virtual machine brings its own scheduler that is completely isolated from the host scheduler,[3] interference between these two can be expected. Even when the virtual machines' scheduler tries to achieve the same optimization goal as the host, taking actions to achieve this might interfere with the steps taken by the host's scheduler. This is because, today, the host and guest scheduler are completely separated and do not share information.

[3] An exception is operating-system-level virtualization that uses the host scheduler.

With increasing degree of heterogeneity and new computational approaches as described in the previous sections, the question arises how the guest of a virtual host can benefit from those improvements. It is plausible to assume that in such a situation it becomes even more viable to provide additional information both to the guest and host scheduler to reach an optimal scheduling decision.

1.2.6 Complexity

The growing complexity of operating systems in general and schedulers in particular paired with low-level programming languages with small expressiveness resemble another challenge in scheduler development. The scheduler subsystem e.g. of the Linux Kernel v4.4 alone consists of approximately 16,500 lines of low-level C code and is tightly coupled to the rest of the kernel that has about 150,000 lines of code excluding device drivers. Besides the technological changes described in the previous subsections, the increasing complexity of scheduler designs can be found due to economic and convenience reasons.

In the past, operating systems were tailored to a specific application domain like embedded, desktop, or server systems. Today, one operating system is used in multiple domains. Take for example Windows. At the end of the 1990s, there were three operating system lines each with a distinct kernel: Windows 9x for desktop environments, Windows NT for workstations and servers, and Windows CE for embedded and later also mobile devices. With the introduction of Windows 10, Microsoft started using one operating system kernel for all of those domains. The same trend can be observed for the Linux kernel that is also used for many different domains ranging from high performance and supercomputing down to embedded devices. This trend makes it necessary to have a kernel that can fulfill all the specific needs of the different domains, therefore, increasing code complexity.

1.2.7 Monopolization

The last major challenge in scheduler development was indirectly already shown in the example in Section 1.2.2 with the big.LITTLE architecture. So far, there is only one notable effort to enable an adapted scheduling to this platform for the Linux kernel but no other operating system. The reason can be easily concluded when examining the number of active committers to several open source operating systems as depicted in Table 1.1. It can be observed that Linux has by far the most contributors and therefore is the most likely platform when adapting new technologies. It even wins more developers year by year than other platforms have developers at all. This would not be an issue if implementations from one operating system could easily be used in another operating system. However, most implementations are extremely specific to the operating system they are developed for. Therefore, porting a feature from one operating system to another mostly means a complete rewrite or at least a significant effort.

Table 1.1: Number of committers per year for selected open source operating systems between 2010 and 2015. The numbers were acquired from the respective source code repositories.

Operating System or Kernel	Year					
	2010	**2011**	**2012**	**2013**	**2014**	**2015**
Linux	2757	2852	2934	3237	3548	3656
FreeBSD	195	193	205	205	201	183
NetBSD	138	128	126	113	114	103
DragonFly BSD	4	22	28	31	30	25
Haiku	43	51	78	61	66	56
Barrelfish	–†	14	19	18	13	12
Minix	11	15	19	28	26	15
GNU Hurd	14	14	15	8	12	10
GNU Mach	4	10	11	9	7	9

† Source code history not available prior to 2011.

1.3 About this Dissertation

The thesis that this dissertation supports is:

> *To harvest the capabilities of future computer systems to their fullest extent, a novel dynamic approach to scheduler architecture is needed that is capable of being integrated into legacy operating systems.*

The previous sections have shown that operating system development in general and process scheduler development in particular, are facing many challenges while, at the same time, they have to be highly innovative to enable operating systems to fulfill their purpose as an intermediary between the hardware and the user applications. This dissertation means to overcome those obstacles for the process scheduler and puts the developer in a position where he or she can apply their knowledge to the fullest extent.

To achieve that goal, the development process has to become more transparent. Today's operating system schedulers of well established operating systems are scattered all over the operating system code with unmentioned interdependencies, written in low-level programming languages, and have poorly documented interfaces. All this has to change to enable a fast and prospering innovation process. Furthermore, the implementations have to be testable in a convenient way, allowing a fast prototyping and straightforward way to try new ideas. The latter one is inevitable for the future because the increasing pace of architecture changes will require new possibly bold approaches to process scheduling where a development and real life testing process lasting several months simply cannot be afforded.

Based on the increasing complexity of system architectures, the scheduler developer cannot be an expert on both architectural properties and specifics of several different operating systems. That means that the development process should be as independent as possible from the runtime system; yet, the developer should be able to test his or her approach in well established operating systems. To test the implementation easily not only in a simulated environment but also on the desired target systems will give an estimate whether it is worth optimizing the implementation for the target system.

Finally, to further speed up the development process, the reuse of existing and upcoming implementations has to be increased and promoted. With constantly changing architectures, a developer has to rely on previously found solutions to specific problems and should not be bothered with reimplementing them. Instead, he or she has to be enabled to reuse the existing solution in the new scheduling approach in an easy way.

With these properties fulfilled, it will be possible to design holistic schedulers that are entirely optimized for the specific system. It will no longer be necessary to conduct scheduling tasks in the user space, especially regarding processing accelerators. The operating system will, again, be in charge of the complete resource management of the machine.

This dissertation proposes a Component Based Scheduling (CoBaS) framework for fast prototyping of new scheduler algorithms. The framework is not limited to a specific architecture and does not use a greenfield approach. Instead, it fits into the existing runtime environments. It enables the scheduler developer to quickly try new ideas both in artificial as well as real life environments overcoming the limitations mentioned above.

1.4 Contributions

The contributions of this dissertation fall into four areas:

Heterogeneous Many-Core Support: The proposed scheduler framework provides an infrastructure to build schedulers for heterogeneous many-core architectures. By design, it can be scaled to a significant number of cores and can handle different architectural characteristics.

Adaptability: Changes to the system architecture during runtime might become common in the future. CoBaS can support this by changing the scheduler during runtime. Both the structure of the scheduler and the scheduling policies can be modified without rebooting the system. Distinct resource managers for specialized PEs are no longer necessary.

Composition: Scheduling policies can be composed of independent components. This dissertation discusses what components are composable based on their functionality and gives a classification of the components used for the CoBaS framework.

Runtime System Independence: Through a case study, this dissertation shows that the proposed framework is mostly independent of the runtime system. Existing components can be reused beyond the boundaries of a specific runtime system.

1.5 Outline of the Dissertation

The remainder of the dissertation is structured as follows: The next chapter delves further into the functional challenges that were initially laid out by this chapter. Chapter 3 continues to deduce requirements based on the discussion of the first two chapters of this dissertation that have to be addressed to tackle the outlined challenges. Following, Chapter 4 presents work related to this dissertation. It, on the one hand, presents previous research that tried to tackle some of the challenges discussed and, on the other hand, presents technology relevant to this dissertation. Chapter 5 introduces the CoBaS framework that tackles the challenges discussed in the previous chapters. This includes, in particular, those problems not addressed by previous research. The subsequent Chapter 6 gives an initial theoretical evaluation of the CoBaS framework before Chapter 7 presents a prototypical implementation of the CoBaS framework in detail. The following Chapter 8 presents an extensive practical evaluation of the CoBaS architecture based on the prototype. Chapter 9 concludes this dissertation. It revisits the contributions claimed in Section 1.4, discusses the strengths and weaknesses of this work, and gives directions for future research. In addition, Appendix A gives further technical details of the CoBaS prototype implementation and Appendix B records detailed data generated from the experiments in Chapter 8.

The bibliography consists of two parts. The first part contains literature that is referenced throughout this dissertation. The second part contains references to software or software projects. To distinguish between references to these two parts, references to software are prefixed with an *S*.

CHAPTER 2

Analysis of the Functional Challenges

The introduction gave an extensive overview of challenges that designers of system software in general and the developer of process schedulers, in particular, are facing. However, every area was only discussed briefly. This chapter discusses the functional challenges in more detail, especially with respect to the process scheduler. It covers the execution granularities of applications in the broadest sense in the first section. Section 2.2 gives more detailed insights of the execution environments and the challenges that arise from them. Section 2.3 discusses to what extent a scheduler has to be flexible and dynamic to changes in the system. Section 2.4 concludes this chapter with further considerations regarding process scheduling.

2.1 Execution Granularity

A system that is supposed to process several computational tasks is faced with the question in what granularity these should be managed and executed. This section shortly discusses the different approaches that are relevant today. However, it only gives a short introduction to the different concepts as a more detailed discussion would go beyond the scope of this dissertation. For more detailed explanations, please refer to the respective references mentioned throughout this sections.

2.1.1 Batch Jobs

Batch processing is the simplest and oldest form of multi-program execution and assumes that data is processed without any user interaction. In batch processing, jobs are assigned to a queue. The computer executes the jobs in the queue one by one. Each of these jobs is entirely independent of the others and runs from the beginning to the end without interruption or user interaction. Even though already introduced in the 1950's, batch processing is still used today, for example in mainframe systems [51, pp. 273–290]. However, the idea of batch processing is also still realized in other operating systems like, e.g., Linux (cf. Section 4.3.1).

2.1.2 Processes

As described above, batch processing is not intended for interactive program execution and assumes that tasks are executed from the beginning to the end. This made it complicated to implement multi-tasking as a task cannot be interrupted. That problem leads to the introduction of the concept of a process. Compared to a batch job, a process represents not only the program code and input data but also the current state of the job [147, pp. 130f.]. With further advancements in memory architecture, each process was assigned a dedicated address space, which is the only address region a process can access. Therefore, the kernel gets widely protected against direct manipulation by processes and processes are protected from each other as well. Saving the job state made it possible to interrupt a process at any point in execution and no longer rely on cooperative multitasking, which required the program to yield the CPU.

2.1.3 Threads

Further advancements in computer architecture and software engineering lead to a point where several parts of a program could be executed concurrently. Therefore, a program did no longer necessarily consist of only one set of instructions that had to be processed sequentially, but several sets of instructions that could be executed concurrently or even in parallel in the case of multi-core systems. However, as mentioned in Section 2.1.2, processes are only allowed to communicate and share data with each other through the help of the kernel as they are protected from each other. This separation is feasible for programs that do not share any or very little data but leads to a significant overhead if several processes belong to one program, thus possibly sharing a significant amount of data.

To solve this issue, the concept of threads or light-weighted processes was introduced. Threads, usually, reside in a process, so one process can have one or several threads. Still, every thread has its state, but the threads in one process share their address space. This sharing allows direct communication between the threads without the involvement of the kernel. Threads can be implemented in user space, making them opaque for the operating system, or can be supported by the kernel. In the latter case, the operating system can handle the scheduling of the threads, whereas in the former case the scheduling has to be performed entirely by the user space code, often trough a library like, e.g., Pthreads [161]. A more detailed discussion of both threads per se and user and kernel space implementation is done, e.g., by Stallings [147, pp. 177f.].

2.1.4 Virtual Machines

Even though virtualization was already introduced in the mainframe domain in 1967 with the IBM System/360 Model 67 [178, p. 57], it became mainstream in the 1990's with commodity hardware. Virtual machines can be managed either by a dedicated hypervisor (type-1 hypervisor) or by an operating system acting as a hypervisor (type-2

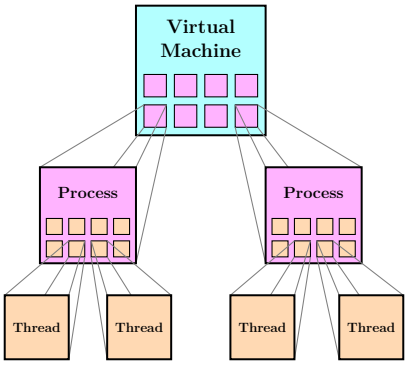


Figure 2.1: Simple relation between virtual machines, processes, and threads.

hypervisor) [127]. However, the impact on the scheduling problem of virtual machines is only minor regarding the differences between type-1 and type-2 hypervisors. Note that the overall scheduling problem might become more complex having a mixture of user processes and processes belonging to the hypervisor.

The scheduling of virtual machines on a single host can be compared to the scheduling of processes. In type-2 hypervisors, virtual machines are often treated as such (cf. Fig. 2.1). For example, the Kernel-Based Virtual Machine (KVM) [S23] assigns to every virtual core a dedicated thread. With virtual machines running an operating system that possibly runs several tasks itself, a parallel can be drawn to processes that run several threads. However, virtual machines have additional requirements regarding scheduling, especially in a multi-core context. As the guest does not know that it is virtualized, it assumes that all of its cores are available the whole time. If this assumption gets violated, the guest suffers from severe performance degradation. Therefore, hypervisors prefer to schedule all PEs of a virtual machine at the same time [140, 174]. Furthermore, virtual machines are completely opaque to the hypervisor, where threads are sometimes visible to the operating system. This opaqueness makes the scheduling more complicated as the host cannot leverage knowledge of task relations inside the guest. Current research shows that this knowledge could improve the overall system scheduling [34].

Looking at virtual machines from a broader perspective shows another scheduling problem. Contrary to processes in general,[4] virtual machines can be migrated from one computer to another while executing. Migration can increase both reliability and scalability of virtual machines. They can be, for example, evacuated from a host when it is failing or going towards an overload situation. Especially the latter scenario is a typical scheduling problem as a certain number of virtual machine hosts (PEs) have to be assigned to virtual machine guests (jobs) with different properties.

[4] Note that the live migration of processes between different hosts is ongoing research [S8] especially in the context of operating-system-level virtualization.

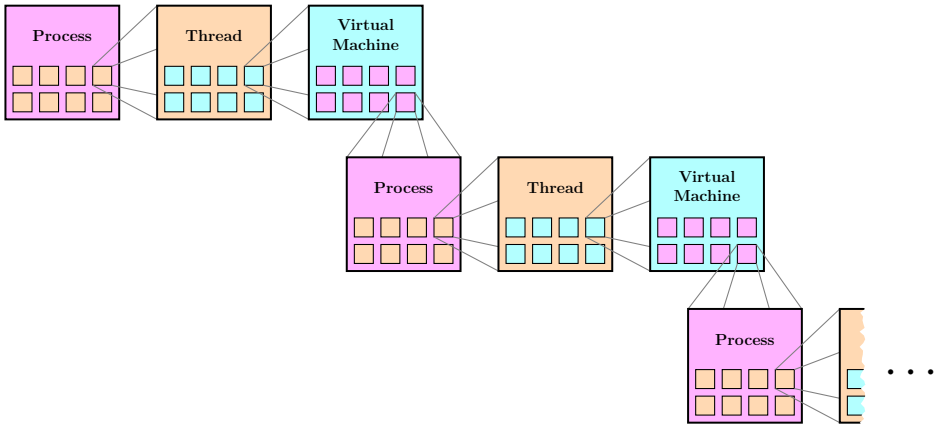


Figure 2.2: Nested virtual machines.

2.1.5 Granularity Relations

The discussion above already implied a relation between the various degrees of granularity. Figure 2.1 illustrates that threads are commonly encapsulated in processes and processes might be encapsulated in a virtual machine. However depending on the underlying hardware architecture or Instruction Set Architecture (ISA) respectively, this relation can be nested as depicted in Fig. 2.2. The virtual PEs are embedded in threads of the host operating system. Therefore, a thread can, as of today, opaquely hold other processes and threads. The simple configuration in Fig. 2.1 already is a challenge regarding the scheduling [34], where a situation as illustrated in Fig. 2.2 makes the situation even more complex from the perspective of process scheduling.

2.2 Execution Environment

The future challenges for scheduling arising from changes in hardware architecture were already introduced in Section 1.2. This chapter discusses the actual implications for the scheduler that results from such changes in detail. At this point, naturally, only certain developments can be considered. Figure 2.3 depicts a fictional architecture incorporating several components that poses challenges for the scheduler and scheduling policies in the future. This design can be considered as a kind of worst case scenario for current runtime systems. It combines several challenges in one system:

- Many-Cores
- Heterogeneity
- Specialized computing elements
- Reconfigurable computing

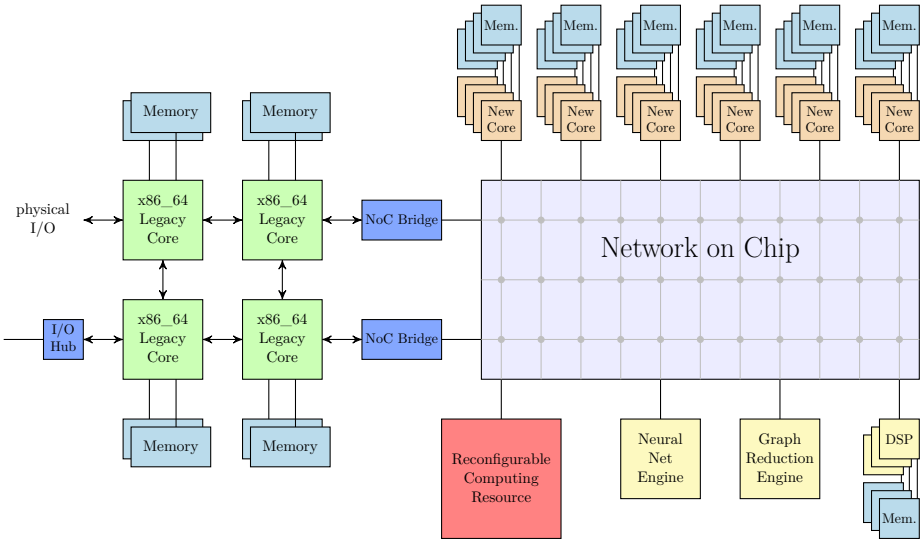


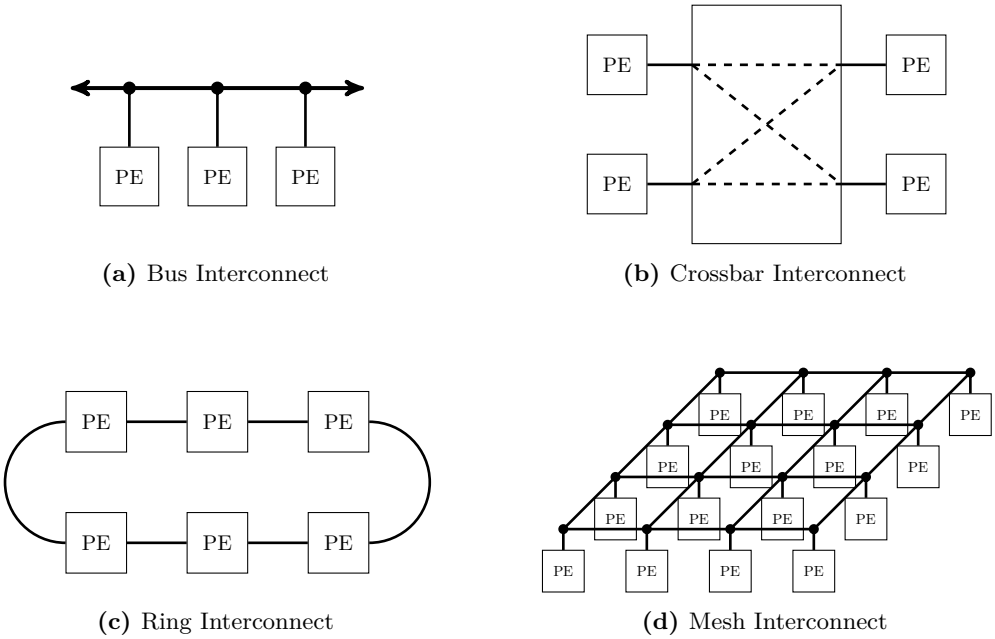
Figure 2.3: The fictional *Metropolis* architecture [144], depicting several architectural challenges that are or will become relevant for future scheduler and scheduling policies.

2.2.1 Many-Cores

In literature, there is no precise definition for many-core systems; however, it is widely accepted that systems with several tens or more cores can be considered as such a system (cf. Vajda [171, p. 3]). Increasing the number of cores on one chip above a certain level introduces new challenges for the hardware developer. On the one hand, it must be ensured that the cores have some way to interact with each other. On the other hand, every core has to have appropriate access to memory to work to its full potential. As the memory bandwidth cannot be scaled infinitely, caches become more important.

Interconnects

The interaction between different PEs on a chip can be achieved by various means. The four most fundamental ways are depicted in Fig. 2.4. The most basic one is a shared bus that connects every PE to the memory and to one another (cf. Fig. 2.4a). Such an architecture allows the communication between cores through the memory or directly. An example is the front side bus. The programming and resource allocation for those kind of multi-core systems is very simple. However, having a common bus for several cores communicating with each other directly or indirectly through memory introduces a critical bottleneck. Therefore, this interconnect architecture scales only for a very limited number of cores and will most likely not be found in future many-core systems.



(a) Bus Interconnect

(b) Crossbar Interconnect

(c) Ring Interconnect

(d) Mesh Interconnect

Figure 2.4: Multi-Core interconnect architectures

A more sophisticated approach is the crossbar interconnect (cf. Fig. 2.4b). This interconnect allows the direct communication between different cores or memory controllers depending on the particular architecture. An example for multi-core systems utilizing a crossbar is the SPARC architecture [166]. In this specific architecture, the crossbar only connects the cores to the memory and not the cores to one another. Compared to the shared bus, the crossbar approach does not show a severe performance degradation when the number of cores rises. Still, it remains a relatively simple programming model as every core has a similar connection to the memory or the other cores. However, the crossbar architecture will probably only have limited use for many-core systems as the hardware complexity of the crossbar itself rises significantly with the complexity of both wiring and switching being $\mathcal{O}(n^2)$, where n is the number of elements connected to the crossbar.

Another approach for on-chip interconnects is a ring (cf. Fig. 2.4c). The ring interconnect, for example, is used in Intel's MIC architecture [37]. With the ring-based interconnect, the communication of a PE is limited to its direct neighbors. As a consequence, to communicate with non-neighboring elements, several hops are necessary. This topology results in different communication times between cores and memory controllers depending on the distance. In contrast to the previously presented interconnects, the ring architecture is suitable for a many-core system, e.g., the MIC architecture having up to 61 cores [81].

Figure 2.4d depicts the on-chip interconnect approach of a mesh. The mesh is the most commonly used topology for a Network on Chip (NoC). Examples for such interconnects in the multi- and many-core area are the Adapteva Epiphany multi-core architecture [50] or the announced EZChip TILE-Mx processor [47]. The mesh approach is very scalable and most likely to be used for future many-core architectures. However, other topologies for a NoC are researched and discussed, e.g., by Sewell et al. [142]. Depending on the concrete topology of the NoC, the communication between cores becomes even more complex as there are now potentially several ways to reach another PE. The main challenge is to find a process placement, scheduling, and communication routing with as little congestion on the links as possible at the same time.

Another aspect that has to be taken into account regarding the different interconnects is the memory. The memory is either directly connected to the PEs or it is connected to controllers that are attached to the interconnects as separate elements. In both scenarios with a sophisticated interconnect, a Non-Uniform Memory Access (NUMA) behavior will be most likely. For the scheduler, NUMA introduces another degree of complexity as the scheduling does not only have to consider the optimal placement regarding communication between PEs but also regarding the access to the working set that is held in memory.

Caches

The memory wall [179] already poses a challenge for single-core systems as the computational power rises much faster than the memory bandwidth. With multi- and many-core systems, the situation becomes increasingly problematic as the memory bandwidth demand rises even faster. The common approach to solve the issue of insufficient memory bandwidth is the use of caches. The use of caches in multi-core systems, yet, introduces a new challenge to the hardware designer stemming from the fact that caches create a copy of data. If two copies exist and they are modified by different cores, data inconsistencies most certainly occur. To avoid inconsistencies, hardware designers today try to maintain cache coherency, meaning the modification of data at one point is reflected on other copies.

Even though many of today's many-core architectures employ a cache coherent, shared memory model like [16, 47, 80] and it is argued that cache coherency will not vanish [100], examples exist or are researched that do not employ cache coherence [165] or ensure cache coherency only amongst a particular group of PEs [61]. Having such islands of coherency makes the task scheduling even more challenging. The scheduler has to assess which task groups and possibly tasks of that task group profit most from placing them on such an island. It can be expected that the impact to performance is significant regarding this placement.

2.2.2 Heterogeneity

The aspect of heterogeneity in computer architecture covers a wide range. It can range from a difference in performance characteristics between cores over different instruction sets to entirely different programming models. This heterogeneity also leads to various challenges regarding the design of a scheduling policy. Those differences will be discussed subsequently based on three examples, representing a limited selection of what is possible.

Heterogeneous Performance

Heterogeneity regarding performance is mostly caused by the need to conserve energy while still maintaining performance capabilities when needed or during the absence of strict energy limitations. Such a system, in the simplest case, consists of one or more Low Performance Cores (LPCs) and one or more High Performance Cores (HPCs). The LPC shows a better energy efficiency than the HPC, which is achieved, for example, by using transistors with a higher energy efficiency [117] or by a more efficient microarchitecture [9].

Such a system can run with an unoptimized scheduler. However, it will result in a bad performance regarding both computation and energy. A low priority background task might get scheduled on a HPC, thus wasting energy, and highly computational intensive tasks are possibly scheduled on a LPC, resulting in a slow program progress. Therefore, an adapted scheduling policy is needed. The decision whether to run a task on a LPC or HPC is not trivial as previous research shows [115, 138]. It depends on many different factors, e.g., whether the system is running on battery, what the tasks characteristics are, or which tasks might be relevant to the user. The situation becomes increasingly complex as the number of efficiency domains rises.

Heterogeneous Instruction Set Architectures

Heterogeneity regarding the ISA means that the cores of a multi- or many-core system have different instruction sets. This heterogeneity can, again, be introduced to conserve energy, but the reasons can also be more diverse. For example, it is possible that a certain number of cores can only be reached with a simplified or reduced instruction set on the limited chip area or that the ISA is optimized for certain tasks. An actual example of such a system is the Parallela platform [119], which combines a dual-core ARM A9 processor with an Epiphany processor that consists of 16 or 64 cores that can only execute the entirely different Epiphany instruction set.

This kind of heterogeneity introduces the same challenges as systems with heterogeneous performance do; yet, it also introduces new challenges beyond the aforementioned. Today, it is common that the core that needs to have assigned a new task also executes the scheduler logic. With a different instruction set, this becomes difficult or even impossible.

Take for example a dual-core system with two distinct ISAs α and β. As it is a multi-core system, some load balancing or task distribution has to be conducted. So, either there is a load balancer code path that can be executed by both cores or the load balancing is only performed on one core or ISA respectively. The former approach is very challenging, as it requires two sets of code and, even more critical, the same representation of the processed data. This might be difficult or even impossible depending on the actual differences of the two ISAs. Potentially simpler is the second solution where the load balancing is only conducted on one core or ISA. Still, the challenge exists that information and, therefore, data, in possibly different representations have to be exchanged between the two architectures.

Heterogeneous Programming Models

The most extreme case for heterogeneous systems are systems that have very different programming models for their cores. This is a reality today, for example, in GPGPU programming. There, the host uses a Single Instruction, Single Data (SISD) or Multiple Instruction, Multiple Data (MIMD) programming approach whereas the GPU uses a Single Instruction, Multiple Data (SIMD) programming approach.[5] Different programming models make it particularly difficult to schedule tasks as some of the cores might not even be able to execute the scheduler code, which is, for example, the case with GPGPU programming. Furthermore, the PEs have different requirements regarding the computation granularity. For example, saving the register set of a GPU for a task switch became only available with the latest generation of GPUs and is not very sophisticated.

2.2.3 Specialized Computing Elements

Specialized computing elements are hardware that is designed to fulfill a very particular purpose like graph reduction, encryption, or hashing. It is disputable whether they can be considered as PEs or have to be treated as special I/O devices. They cannot execute a complete task and are used from a coding point of view in a very small part of a program. Still, the execution time of a task might have a significant percentage dedicated to the specialized computing element depending on the complexity of the function it realizes.

Today, it is not clear how specialized computing elements will evolve and how much autonomy they will have. Currently, they commonly need a conventional PE that feeds them data into and instructs them how to process it. So, it is possible that they will be handled like Direct Memory Access (DMA) controllers today, but also that they have to be considered by the scheduler like any other PE.

[5] Refer to *Flynns Taxonomy* [57] for explanation of the programming models.

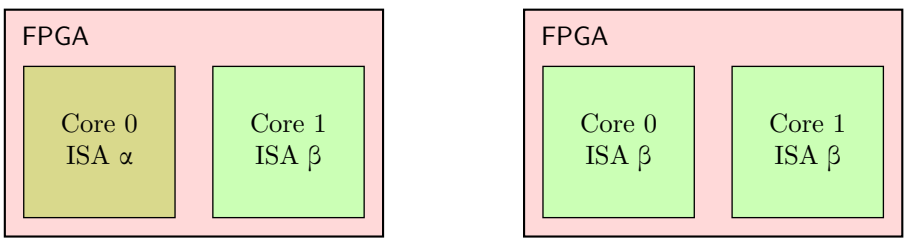
2.2.4 Reconfigurable Computing

Reconfigurable computing is mainly based on FPGA technology. Section 1.2.3 already discussed their properties. This section takes a closer look at the requirements arising from the design of scheduling policies. First, FPGAs can be used like specialized computing elements, which were discussed in the previous subsection. However, as the designation of reconfigurable computing implies, they can be reconfigured to fulfill different purposes at runtime. For example, a FPGA can be changed from being a graph reduction processor to an encryption processor.

However, the capabilities and possible applications of FPGA technology go far beyond that. It is also possible to create complete Central Processing Units (CPUs) in a FPGA. So, a possible scenario, for example, could be a heterogeneous system regarding the ISAs of its cores with an additional FPGA. The FPGA could be configured to act as a CPU of a certain ISA depending on the current task requirements. Take, for example, again the situation with two ISAs α and β. Now, if the scheduler detects a high load on the core that executes ISA β, it could reconfigure the FPGA from acting as a CPU running ISA α (Figure 2.5a) to act as a CPU that executes ISA β (Figure 2.5b). As even current FPGAs are so big that they can implement more than one CPU core, the situation can get increasingly complex. Take the same example, but the FPGA is able to act not only as a CPU executing ISA α or β, but, for instance, as two cores of ISA α and four cores of ISA β. This capability would result in many different configurations, making it necessary for the scheduler to be highly adaptive.

Another application for FPGA technology could be the augmentation of existing instruction sets with new instructions that are executed in hardware in the FPGA. With this approach, a new complex instruction is created that would originally need several instructions of the native ISA.

During the execution of the task with the new instruction, the process will run partially on the FPGA, if it is or will be configured to execute the new instruction for the task. If it is not available, the execution can fall back with, e.g., a trap to the original code with several instructions. Again, this technique introduces new challenges for the scheduler.



(a) Initial configuration with two cores and two ISAs.

(b) Reconfigured FPGA with two cores and one ISA.

Figure 2.5: System reconfiguration due to a changed load situation.

It has to decide when to execute such a task based on the current configuration of the FPGA, whether it is worth to reconfigure the FPGA for the current task set, and manage the utilization of the FPGA by several tasks. Again as in the CPU example above, several new instructions can be created in the FPGA concurrently. The augmentation of existing ISAs with specialized instructions was researched for example by Pittman et al. [126]. The resulting scheduling problem was discussed by Sheldon and Forin [143] on the example of the NetBSD [S28] operating system.

Besides the impact of the specialized hardware configuration for the task, the scheduler has to take into account the time to reconfigure the FPGA and possibly weigh up the reconfiguration against the processing of data on the FPGA. This consideration becomes even more complex as the reconfiguration time is not constant and depends on many aspects like the actual FPGA technology, vendor, or the size of the area that has to be reconfigured. The reconfiguration time is already identified as an issue having hardware supported tasks and ranges today in a magnitude of hundreds of microseconds (cf. Duhem et al. [48]).

2.3 Scheduler Flexibility

The scheduler flexibility considers how good the scheduler can react to changes in the requirements. Literature traditionally defines static and dynamic scheduling, where the former one is, today, almost exclusively relevant in real-time scheduling (cf. Ramamritham and Stankovic [129]). A static scheduling policy is the least flexible of all scheduling policies. It requires that all tasks and PEs are known before the runtime of the system. The schedule is created offline and then hard coded to the system. Such a policy cannot handle the introduction of new tasks or changes regarding the PEs, e.g., a core fails and can no longer execute tasks. Even though such a scheduling approach is very limited, it is common in embedded and real-time environments, because it is simple to implement and easy to analyze.

Dynamic scheduling is common to most modern operating systems. It determines the task order at runtime and allows the creation and the removal of processes at runtime as well. Furthermore, current dynamic scheduling policies can compensate the removal or addition of cores, which is, from an architectural point of view, not complicated. Failed cores, for example, can just be ignored in future scheduling decisions and new cores can easily be added to the list of available PEs. Note that the exact implementation of this behavior is, depending on the actual operating system architecture, not necessarily simple.

With new architectures, as described earlier, the flexibility of dynamic scheduling will not be enough. The limits are already assessable looking at the challenges introduced by heterogeneous many-core systems. Even though it is possible to create a standard dynamic scheduling policy for heterogeneous multi-core systems, this needs specific tailoring to the particular system. Even though it is often feasible to parametrize those scheduling policies, this might not be sufficient for future architectures. Parametrization

is only feasible to a certain extent as long as the systems do not differ severely. For example, for a heterogeneous system with differences in processing efficiency of the PEs, it is feasible to be able to parametrize the number of efficiency domains or the exact impact on energy consumption. However, it would be unrewarding to simply parametrize difference in programming models as this would result in the main differences in the handling of the task.

In case simple parametrization is not sufficient, an *adaptive* scheduling approach is necessary. An adaptive approach allows significant changes to the scheduler and scheduling policy during runtime. The adaption process can also include the injection of new code paths on demand. It enables the runtime system to change the scheduler whenever necessary and creates new possibilities regarding scheduling optimization. For example, it would be possible to ship a computation accelerator with its own specifically tailored scheduling policy and integrate that policy into the runtime systems' native scheduler.

2.4 Further Considerations

Besides the main challenges discussed in the previous sections, other problems exist that cannot be classified. Following, those challenges are discussed briefly.

2.4.1 Self and Foreign Hosted Scheduling

The task assignment to PEs can, in general, happen in two ways. Either the PE selects the process that it wants to execute next from the ready list by itself, or it is assigned a new task by another PE. In a single-core system, of course, only the former approach is feasible as no other PE can perform the selection. Because several current operating systems evolved from a single core operating system, the choice of the next task by the PE itself is a very common approach. The task assignment by another PE is mostly found when the PE itself is not capable of executing the scheduler routine, which is the case, for example, with GPGPU computing.

Decentralized scheduler implementations can be considered as a hybrid approach to self and foreign hosted scheduling. Within this method, every PE gets assigned a certain number of tasks by a load balancing mechanism that can be executed by the PE itself or by another PE. The PE then selects only tasks from the set of tasks it was previously assigned.

2.4.2 Multikernel Operating Systems

The multikernel approach was proposed by Baumann et al. [20] to cope with some of the upcoming challenges presented, among other things, in Section 1.2. Baumann et al. described a new architecture for operating systems that considers many-core systems

rather as a network than a system with shared resources. Even though a system running a multikernel operating system has shared resources and communication capabilities, e.g., through shared memory, the operating system will allow communication between cores only through explicit message passing. It is argued that such an architecture is more apt for many-core systems. This claim can be backed by the fact that some of the current many-core architectures use a NoC to communicate. This makes it natural to construct the operating system from the beginning with such a network in mind.

Baumann et al. argue that a multikernel operating system is independent of the underlying communication architecture [20, pp. 38f.(9f.)], which would, in theory, mean that such an operating system could use many different communication methods, e.g., shared memory or a NoC. The approach is very close to cluster operating systems that also manage a bigger number of PEs, often through message passing. However, nodes with more than one PE each are not considered as multiple systems, whereas in a multikernel operating system, every PE is managed by one kernel individually. Another aspect of multikernel operating systems are heterogeneous ISAs, which are not considered by cluster operating systems. Besides the Barrelfish operating system [S2], which is one of the first multikernel operating systems, Popcorn Linux was introduced lately [17]. It can also be seen as a multikernel operating system. However, it does not consider every core strictly as a single system but mostly as individual ISA domains.

The research regarding the scheduler architecture for such systems is still limited. Peter [125, Ch. 3] describes the scheduling in the Barrelfish operating system. With the current implementation, task placement and scheduling are divided. While the task placement is conducted in a centralized manner, the scheduling itself is carried out on every core individually.

2.4.3 Cross-Cutting Concerns

Concerns in software design are, in general, everything that has to be considered during the design including features, nonfunctional requirements, design idioms, and implementation mechanisms [135]. Software engineering tries to separate the concerns and assigns them to distinct design entities like, e.g., modules [121]. This approach can also be observed in operating systems, where, for example, the process scheduler is often a distinct subsystem. However, this *separation of concerns* [77] is not always possible. *Cross-cutting concerns* [90] exist that are a common challenge in software architecture in general and system software architecture in particular.

Take the example of hard realtime requirements. Figure 2.6 depicts a common system stack with the hardware, the operating system with its various subsystems, and the application. The concern *hard real-time* cannot be addressed by a single layer in this exemplary stack. All parts have to be aware of it. The hardware has to be deterministic, the operating system has to be real-time capable, and the application has to provide information like, e.g., its runtime and deadline to enforce a real-time scheduling. For the operating system to be real-time capable, most of its subsystems have to be real-time capable as well. For example, the hardware abstraction layer has to have deterministic

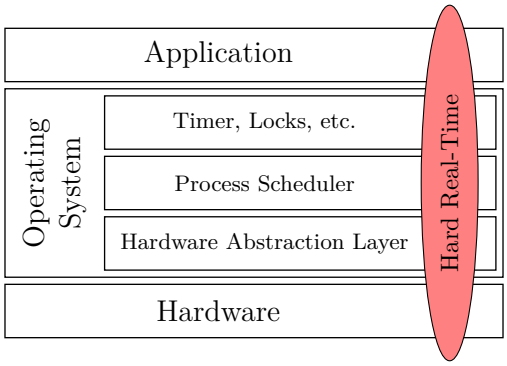


Figure 2.6: Hard real-time as an example for a cross-cutting concern.

code paths, the process scheduler has to employ a real-time scheduling algorithm, and the handler for, e.g., timers has to execute with an upper bound of time. Real-Time is one extreme example that touches all layers of the operating system. However, other concerns exist that touch only a subset. Take for instance the work by Richling et al. [133] or Schönherr et al. [141]. They have shown that the cross-cutting concern of energy consumption can be handled by a closer integration of the scheduler and the systems energy governor, while also improving system performance.

As cross-cutting concerns due to their nature have to be handled in different parts of a system, it will not be possible to have a solution regarding these issues that is restricted to the scheduler. In general, two options exist to handle cross-cutting concerns: integration and information distribution. Integration would be contradictory to the concept of separation of concerns and is, therefore, often not applicable. Hence, most of the time information distribution and communication between several software entities is the only solution.

CHAPTER 3

Requirements for a Future Scheduler Architecture

After discussing the problem domain covered by this dissertation in the previous chapters, this chapter substantiates the requirements that a novel scheduler architecture has to consider. Not every requirement can be tracked down to a single problem or issue, most of them will rather relate to several ones at the same time. The following sections are naming the requirements that have to be considered with a new approach to scheduler architecture.

3.1 Flexible Task Model

A future scheduler architecture has to be as general as possible regarding the task model. This requirement stems from two facts: First, the tendency that single operating systems are used across several application domains and, second, the differing execution models of PEs and processing accelerators. A flexible task model is a key to enabling the operating system to manage arbitrary PEs and accelerators.

Regarding the application domain and the utilized scheduling policies, different properties of a task are relevant. For example, a First-Come, First-Served (FCFS) algorithm would require the arrival time of a task, while a Shortest Remaining Time (SRT) based algorithm needs information about the remaining runtime. This means that a suitable scheduler architecture should not limit the developer regarding the management of such properties. Therefore, an approach is needed that allows a dynamic definition and introduction of relevant properties.

The execution model has to be general in a way that it does not matter how and where a task is running. The runtime system will decide how it uses the scheduling decision and puts it into action. The scheduler only has to decide which task will be most suited to the current situation when queried by the runtime system. That especially rules out one assumption made by operating system schedulers today, namely that the scheduling for each independent PE happens on that PE itself. Processing accelerators often cannot compute a schedule on their own and therefore a decision has to be made for them,

e.g., by a general purpose PE. Furthermore, a rising number of cores might enable the possibility of dedicated scheduling cores or even dedicated schedulers in hardware based on FPGA technology.

3.2 Scalability and Contention

Scalability is, in general, a desired property for most kinds of software; however, for the process scheduler, it is an indispensable property. The processing time spent in the scheduler is lost for the actual computational tasks assigned to the system. A scheduler that does not scale would increase the amount of processing time lost and might even slow down the whole system. Two aspects are relevant toward the scalability of the scheduler. The first aspect is determined by the complexity of the scheduler implementation and the second one by possible contentions in the implementation.

For the scope of this dissertation, the scalability of the architecture and the scalability of the underlying scheduling algorithm has to be separated. The complexity of a specific scheduling algorithm can de facto not be improved by the given architecture itself. However, on the other side, the architecture should also not worsen the complexity by enforcing its structure. Yet, the most determining factor besides the computational complexity, especially in a many-core system, regarding scalability is contention. The main cause for contentions in the process scheduler are shared data structures that are accessed and modified by several or all PEs of the system. This could be, for example, a shared runqueue. To be able to create a scalable scheduler for many-core systems, it is necessary for a future scheduler architecture to make it possible to implement scheduling policies that minimize the degree of contention.

3.3 Adaptability

Future performance improvements might only happen through specialization; therefore, it can be expected that the scheduler has to be specialized to certain workloads especially with a very heterogeneous system. Hence, a scheduler architecture should support changes to the scheduling policy during runtime. As both hardware configuration and workload scenarios become more unpredictable, it will not be sufficient to have a limited selection of scheduling algorithms that can be chosen from at runtime, but it must be possible to add new policies during runtime as well. Taking reconfigurable computing with FPGA technology into account, this property becomes indispensable. Hardware reconfiguration becomes only meaningful when the operating system can adapt to the new system as well.

A second aspect that makes adaptability indispensable for future systems is availability. An adaptable system can also increase the availability of a system. This feature is helpful in two scenarios. On the one hand when the need for a bug fix occurs and on the other hand in large scale computers with multiple tenants. With an adaptable system,

a faulty scheduler implementation could be replaced during runtime, thus removing the requirement to reboot the system and increasing its availability. While in the latter scenario as, e.g., in HPC, the scheduler could be adapted for every customer without a time consuming reboot and reconfiguration of the whole infrastructure.

3.4 Runtime System Independence

The requirement of runtime system independence is mainly based on the insights of Section 1.2.7. From the perspective of a single operating system, the independence of the scheduler implementation seems unnecessary. However, looking at the bigger picture, having a runtime system independent scheduler would make it much easier for specific system software projects with a smaller community and smaller workforce to profit from innovations of bigger projects with a bigger workforce. Also, the creation of new projects would be easier as an existing, well-tested scheduler implementation could serve as a solid foundation. This could increase the pace of the overall innovation process, which would be beneficial to all new, small, and big software projects. Moreover, it would be helpful to keep up with changes in hardware architecture as developers and researchers might concentrate on vital tasks instead of low-level problems.

Another argument for runtime system independence stems from the research point of view. With an independent scheduler architecture, it is easier to identify the properties of a scheduler policy and separate it from the effects of a specific runtime system. It would be possible to evaluate the same implementation in several runtime systems and, through statistics, obliterate the biases introduced by the specific runtime system.

3.5 Reusability

As future systems become more complex, future scheduling policies might become much more sophisticated as well. Particularly in heterogeneous systems, different PEs might require different scheduling strategies. Therefore, it will be infeasible to have a single homogeneous scheduling strategy for the whole system. To simplify the development of such scheduling policies, the reuse of existing scheduling implementations is desirable. This would dramatically increase the pace of innovation as existing implementations could be combined to form a holistic scheduler for the whole system. Also, reusability reduces development costs and testing efforts as existing, known to work implementations can be used.

In the ideal case, a new scheduler for a very heterogeneous system could be constructed with little implementation effort. It should be possible to take existing implementations for scheduling strategies peculiar to the used PEs and to combine them to an entirely new scheduler. That way, it is possible to create a scheduler that is suited and perfectly tailored to the specific system.

3.6 Information Distribution

As the heterogeneity in a system rises, so does the need for information to achieve the scheduling goals. The same applies to an increased degree of virtualization that potentially requires the exchange of information between the virtual machines' requirements and the host systems' scheduler. Therefore, a flexible infrastructure is needed in a future scheduler architecture that supports the propagation of information. This is valid for all three kinds of communication: inside the scheduler subsystem, from the runtime system to the scheduler, and from the scheduler to the runtime system.

An information distribution infrastructure might also tackle the issue of cross-cutting concerns. Cross-cutting concerns influence, by definition, several or all parts of a system. To avoid a tight integration and keep the independence of specific subsystems, a communication infrastructure is needed. Even though the overhead introduced by the communication and coordination might be significant, the separation of concerns might still be more desirable to secure maintainability and testability as will be shown in the next sections.

CHAPTER 4

Related Work

This chapter discusses related work and technology relevant to this dissertation. The chapter is divided into three sections. The first section gives an overview mainly on other scheduler frameworks that have goals similar to this work. Section 4.2 discusses research focusing on changing the scheduling policy or modifying an operating system kernel at runtime. The last section gives a brief insight on scheduler interfaces that have to be taken into consideration when designing a scheduler framework for modern operating systems.

4.1 Scheduler Frameworks

This section presents related work that describes a scheduler framework or similar approaches. Each sub-section is dedicated to a different research approach, with the exception of the last subsection that discusses the heterogeneity problem in distributed computing.

4.1.1 Bossa

The *Bossa* project started with the introduction of the *Bossa language* by Barreto and Muller [19], which was first intended as a Domain Specific Language (DSL) to develop real-time schedulers and later extended to a framework for general purpose process scheduling. The research is mainly driven by programming language aspects. Barreto and Muller based their DSL on the insights of their previous paper that pointed out the limited code reuse and the problem of limited extensibility in operating system development [113]. Barreto and Muller emphasize that experimentations with operating systems even in microkernel architectures are tough, because existing scheduler implementations are spread over the whole kernel and written in low-level languages that make the understanding of the code difficult and the implementation error prone. The proposed scheduler DSL enforces a fixed process state model, uses queues as the data structure to hold existing tasks and implicitly assumes a computer model that is a single core machine. *Bossa* uses events to propagate information from the kernel to

the *Bossa* runtime system and from the runtime system to the scheduler descriptions, which is discussed thoroughly in [95]. Lawall et al. [94] also examine the theoretic extension of the *Bossa language* to support some modularity that allows creating scheduler families that share common properties. The *Bossa* project in its entirety is presented and summarized in [96] and [114].

Bossa was implemented for the Linux kernel versions 2.4.20, 2.4.24, and 2.6.11 [S4]. It is claimed that the approach should be feasible for other operating systems, yet, the proof is still to be delivered. Furthermore, the Bossa framework does not take into account many-core systems, heterogeneity regarding the underlying hardware, or dynamic changes to the system architecture.

4.1.2 Hierarchical Loadable Schedulers

The Hierarchical Loadable Schedulers (HLS) framework strongly focuses on real-time scheduling. It was originally introduced by Regehr [131][132] for the Windows 2000 kernel and ported to the Linux kernel v2.4 by Abeni and Regehr [2][S21]. Its main goal is to extend the general-purpose operating system with general, heterogeneous hierarchical process scheduling and enable easy prototyping of real-time schedulers. The heterogeneous aspect of this approach does not regard heterogeneity of the underlying hardware but takes into account the scheduling strategies for different processes.

The general architecture of HLS is depicted in Fig. 4.1. The *hierarchical scheduler infrastructure* (HSI) acts as glue logic between the operating system and the HLS framework. Inside the framework, there are different loadable schedulers that are each connected to a top and a bottom scheduler. The upper termination of the hierarchy lies in the HSI. Every physical processor represents one of these connections, hence in an n processor system n such connections exist. The lower termination of the hierarchy is also located in the HSI. Here, every task is assigned one connection. When a task is selected for a processor, the hierarchy is traversed from the top to the bottom, ending at the task that is supposed to be dispatched next.

HLS does not completely remove the existing scheduler in a legacy operating system but relies on it. Furthermore, it was not designed for multi- or even many-core systems as it uses a single scheduler lock and the hierarchy of scheduling policies is iterated in a serialized manner. This means that only one processor can run the scheduler code at a time [131, pp. 36f.], which represents a significant bottleneck, especially in a many-core system.

Abeni and Regehr also emphasize the necessity to enable simple development of scheduling policy implementations. For that purpose, besides the Windows and Linux kernel implementation of HLS, they provide a simulator to test the scheduling algorithms in the user space. They acknowledge that their simulator is limited regarding multi-core operations [2, p. 4].

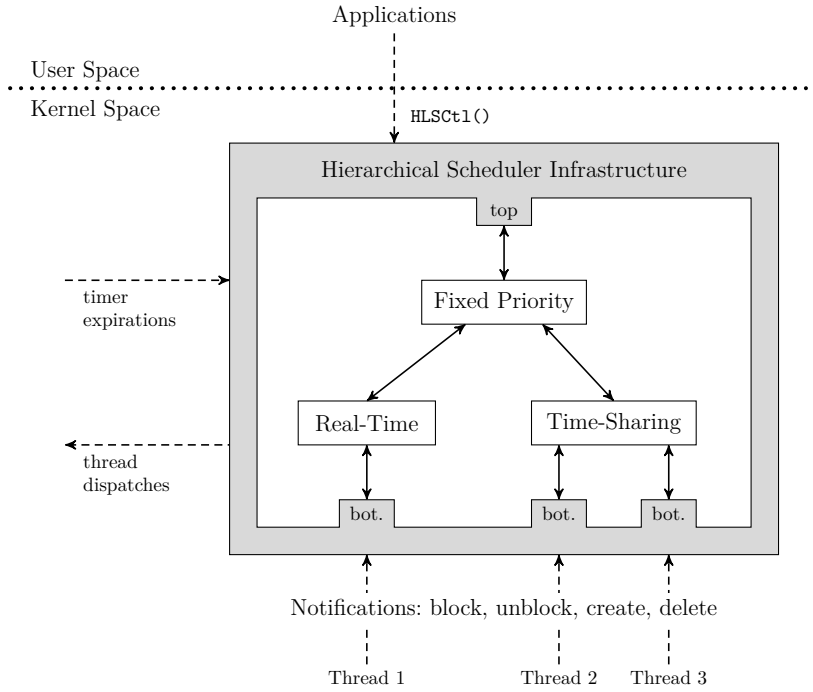


Figure 4.1: The general architecture of the HLS framework [131, p. 35]. Three schedulers – *Fixed Priority*, *Real-Time*, and *Time-Sharing* – are loaded into the hierarchical scheduler infrastructure (gray). Three tasks are assigned to the schedulers. The communication flow to and from the framework is indicated through dashed arrows.

4.1.3 SF3P

Gomez et al. [65] present the *SF3P* framework that also uses a similar hierarchical scheduling approach as the HLS framework presented in the previous sub-section. However, the main focus of Gomez et al. is the prototyping of real-time scheduling algorithms. In their work, they derive an interface for scheduler implementations tailored to the real-time domain. Based on this interface, they build a scheduling framework for the userland. In their evaluation of their approach, Gomez et al. implemented a prototype of their framework on top of a POSIX layer.

The SF3P approach shares the same drawbacks regarding the scope of this dissertation as the HLS framework. Moreover, they demonstrated the feasibility of their approach only in the userland on top of the POSIX layer. This design decision makes it easier for them to realize their goals; however, it strongly tailors their approach to the POSIX standard.

4.1.4 The Flux OS Toolkit

The *Flux OS Toolkit* was introduced in 1997 by Ford et al. [58]. The authors identified the main challenge in operating system design by the specifics, complexity, and missing documentation of existing kernel implementations that hinder the exploration of new ideas and the creation of simple research prototypes. To tackle this problem, the authors created a toolkit that consists of several components offering common operating system functionalities like, e.g., process management, memory management, or file system support. They further argue that it is not always necessary to implement certain functionalities, but that they can be reused from different existing operating systems. However, those implementations might not be well documented, and the implementations are hardly integrable in another code basis. Ford et al. identified as those functionalities the file system, the networking stack,user space and generic device drivers that are used from NetBSD, FreeBSD, and Linux respectively. To still make them usable in the *Flux OS Toolkit*, they embedded every component in glue code that is compatible with the rest of the toolkit.

The Flux OS Toolkit is a collection of libraries and wrapper functions and cannot offer the functionality of a framework. The main goal is to support the building of an operating system from scratch and not specifically the reuse of an implementation in multiple systems. The challenges of current and future system architectures are not considered either explicitly or implicitly.

4.1.5 The S.Ha.R.K. Soft and Hard Real-time Kernel

The S.Ha.R.K. Soft and Hard Real-Time Kernel was proposed by Gai et al. [62] and was actively developed until 2008 [S38]. S.Ha.R.K. provides a generic real-time kernel that handles the resource management through *Modules*. When a task is created, it can specify its requirements dependent on their task model through Quality of Services (QoSs). A *Generic Model Mapper* iterates over the Modules until it can find one that can satisfy the requested QoS. The task is then assigned to that Module, and the QoS parameters are converted by a *QoS Mapper* to a Module specific representation. The QoS Mapper is specific to and provided by every Module individually. The whole process is outlined in Fig. 4.2.

When the kernel needs to schedule a new task to a CPU, it iterates over all scheduling Modules in a fixed order. Each Module has its private ready-list, and if a task is ready to be scheduled, it will be reported to the kernel during the iteration. This means that the tasks from the second Module only have a chance to be scheduled when the first Module has no runnable tasks at that point. Modules in general and scheduling related Modules, in particular, can be added to S.Ha.R.K. during runtime.

S.Ha.R.K. is tailored to the specific needs of real-time scheduling and does not consider many-core systems or heterogeneous architectures. Furthermore, the integration of S.Ha.R.K. in existing systems is questionable as it has a very specific task and scheduler model.

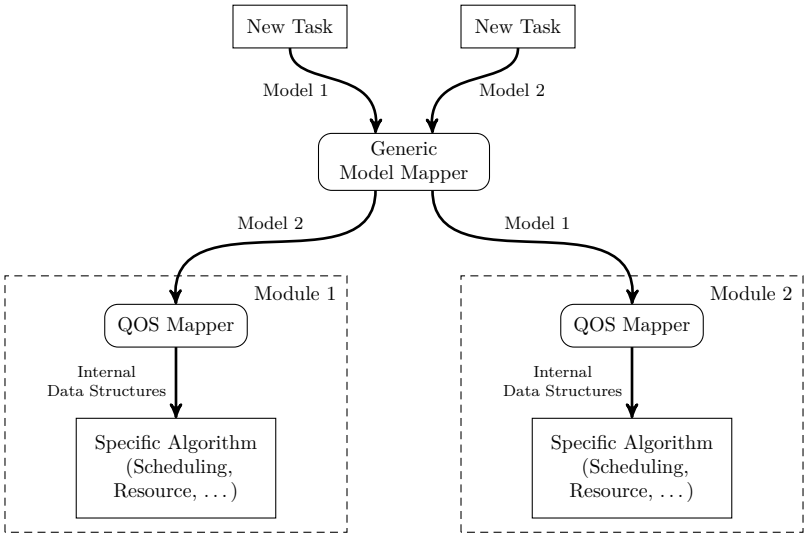


Figure 4.2: The interaction between the Model Mapper and the QoS Mapper in the S.Ha.R.K. soft and hard real-time kernel [62, p. 202(4)]. During task creation, a model is selected for each task based on their requested QoS. When a Module fulfills the requested QoS, the QoS mapper converts the requirements to the model specific data structure.

4.1.6 ExSched

ExSched was introduced by Åsberg et al. [11]. The goal of ExSched is to provide a framework to develop real-time schedulers while being minimally invasive to the existing system. It is available for VxWorks version 6.6 and the Linux kernel v2.6.36 [S13]. The architecture of the ExSched framework for the Linux kernel is illustrated in Fig. 4.3. In Linux, ExSched is loaded as a kernel module, while in VxWorks it is directly integrated into the kernel. ExSched utilizes the existing scheduler. In Linux, for example, it uses the `SCHED_FIFO` priority of the real-time scheduling class. A user space application can load a new scheduling policy implementation and control the behavior of ExSched through a given user space library.

Even though ExSched simplifies the implementation of new scheduling policies in the real-time domain, it is bound to the limitations and enforcements of the existing scheduler. For example, ExSched under Linux is bound to the constraints and properties of the decentralized scheduler approach used in the Linux kernel. Also, the approach is strongly tailored to the requirements of real-time scheduling and does not consider general purpose scheduling.

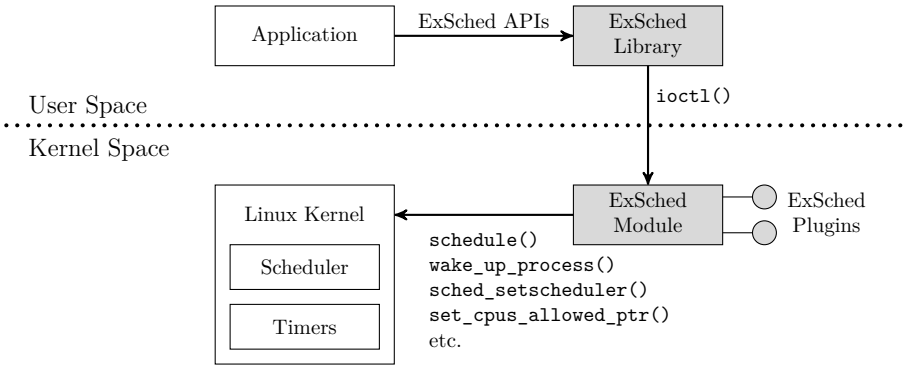


Figure 4.3: The ExSched Framework for Linux [11, p. 242(3)].

4.1.7 Group Scheduling

The concept of *group scheduling* was introduced by Aswathanarayana et al. [12] and is located in the real-time domain. The approach assumes that certain task groups have the need for common scheduling decisions. Therefore, in the model developed by Aswathanarayana et al., tasks are assigned to a certain group with a certain scheduling decision function. Furthermore, each scheduling group can be a member of another scheduling group. Hence, a hierarchical scheduling structure is possible. When a scheduling decision has to be made, for a particular group the assigned scheduling decision function is invoked. In case the function does not come to a result, the default operating system scheduler is used as a fallback. In their work, Aswathanarayana et al. briefly present two implementations of the group scheduling approach: one on the kernel level for the Linux kernel and one as middleware solution. An implementation for the Linux kernel of the group scheduling approach is pursued further by Watkins et al. [175]. They present an implementation of the group scheduling approach as a *flexible scheduling framework* for the real-time Linux patch [136, S34]. Watkins et al. extend the real-time scheduling strategy introduced by the real-time Linux patch through group scheduling capabilities.

The concept of group scheduling presents another approach to employ different scheduling strategies for various requirements. However, the approach is strongly tailored to the real-time domain and falls short to provide features for general purpose scheduling. Furthermore, the kernel-based approach to enable group scheduling is strongly related to the Linux scheduling architecture.

4.1.8 Scheduling in Distributed Computing

With an increasing number of cores and heterogeneity, the traits of former distributed systems can be found in nowadays individual machines. Therefore, it is worthwhile to look briefly into research in that area. Regarding the process scheduling, prior research

exists that looked into the challenge of heterogeneity in distributed computing. Two examples are the *SmartNet* scheduling framework introduced by Freund et al. [59] and the *unified resource scheduling framework* introduced by Alhusaini et al. [4]. The emphasis of both research projects lies on the heterogeneity of the resources of the computing hosts. Both Freund et al. and Alhusaini et al. show that the overall performance of the whole distributed system can vastly improve when optimizing the task assignments.

Drawing a parallel to the current development in computer architecture, it can be expected that the situation in a single computer might become the same as described by Freund et al. and Alhusaini et al. for distributed systems. Therefore, enabling a similar approach as they did with distributed systems in a single machine can be expected to yield similar results. However, as they only present a very high-level abstraction to scheduling, their approach cannot be applied entirely to the process scheduling of a single machine.

4.2 Scheduler Adaptability

This section discusses scientific research, existing operating systems, and technology regarding the adaptation of the process scheduler during runtime. Section 4.2.5 demonstrates also how important it is to avoid system rebooting. Therefore, the adaptation of the scheduler during runtime might not only be necessary due to dynamic changes of the underlying hardware, but also for other reasons.

4.2.1 User-Level Threads

User-Level Threads (ULTs) are, in contrast to Kernel-Level Threads (KLTs),[6] completely managed in user space. Examples for ULTs are GNU Portable Threads [S17], Green Threads [172, Ch. 6], or Fibers [56]. ULTs are encapsulated inside KLTs and can especially employ their own scheduling strategy. However, they suffer various drawbacks as the kernel has no knowledge of the existence of ULTs. The most noteworthy trait is the issue of I/O-operations when, e.g., the KLT gets blocked because of an operation of one of the ULTs. This means that also otherwise runnable ULTs inside the KLTs get blocked. Some of the issues raised by this kind of behavior are tackled by *Scheduler Activations* [6].

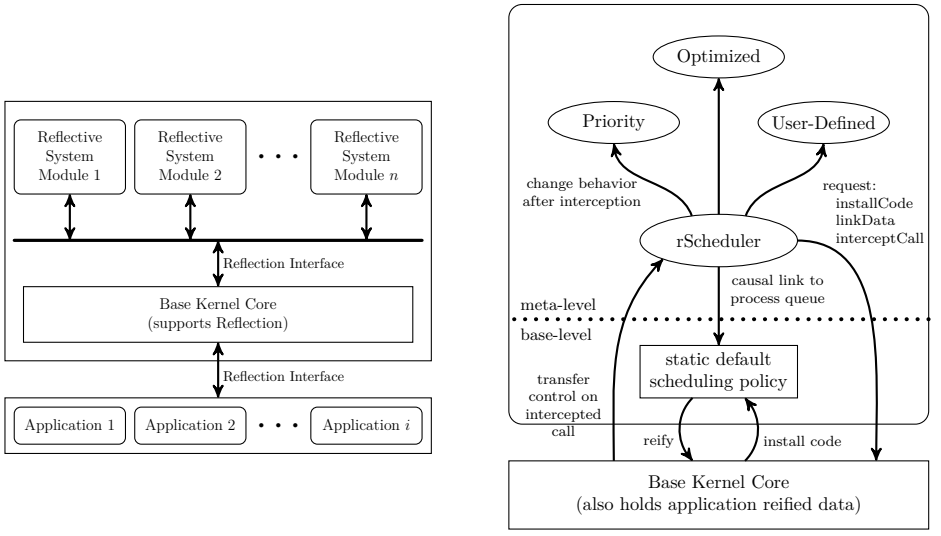
Even though it allows a custom scheduling of the threads inside KLTs, the approach of ULTs is not suited for the purpose of this dissertation. It is not possible to change the overall system behavior with ULTs. The behavior of the system is still determined by the scheduling of the KLTs, and the introduction of ULTs can be seen as hierarchical scheduling. It is neither possible in any way to handle changes in the hardware architecture with ULTs.

[6] In literature, KLTs are also referred to as *kernel-supported threads* or *lightweight processes* [147, p. 184].

4.2.2 DAMROS Reflective Scheduler

DAMROS is an acronym for Dynamically Adaptive Micro-Reflective Real-Time Operating System and was developed by Patil and Audsley [122]. DAMROS realizes the idea of a reflective operating system the general architecture of which is depicted in Fig. 4.4a. The system consists, apart from the core, of several system modules that support reflection. These modules are, e.g., the scheduler or memory management system. The behavior of those modules can be influenced by the applications. Each module has a *Base-Level Code* that represents the default behavior. This behavior can be changed through the reflection of the application via the reflection aware *Base Kernel Core*.

Based on this architecture, Patil and Audsley research the scheduler as a Reflective System Module. The resulting scheduler is depicted in Fig. 4.4b. The Base-Level Code, in this case, is a round-robin scheduling. As Meta-Level Code, it has a *rSchedule* called scheduling implementation that can change its behavior based on the reflection of the running applications. It can, for example, postpone energy demanding tasks when the system has little battery life left or help an application meeting its deadline by an adapted task order. This is realized by, e.g., user-defined scheduling policies. The changes to the scheduling are done mainly on a per application basis. That means an application can control how itself and its children are scheduled. However, it cannot influence how other applications are scheduled. Therefore, this approach has its main scope not on the whole system but rather on specific applications.



(a) General structure of the reflective operating system.

(b) Scheduler system module in the reflective operating system.

Figure 4.4: Architecture of a reflective operating system [122].

4.2.3 The SPIN Operating System

The SPIN operating system was introduced by Bershad et al. [22]. SPIN is designed as an extensible operating system. The extensibility also covers the thread management. SPIN does not provide a specific thread model but can by extension handle arbitrary models. “In SPIN, an application can provide its thread package and scheduler that is executed within the kernel.” [22, p. 274(8)] However, in SPIN there is also a global scheduler that runs beneath the application specific schedulers.

Even though SPIN is an interesting example for an extensible kernel, it cannot handle the issues covered by this dissertation. As with ULTs, the overall system behavior cannot be influenced by the task-specific schedulers.

4.2.4 Vassal

Vassal is an extension to the Windows NT 4.0 kernel that allows the dynamic loading and unloading of scheduling policies. It was introduced by Candea and Jones [35]. The architecture of Vassal and its integration into Windows NT 4.0 is depicted in Fig. 4.5. In Vassal, the implementations of the scheduling policies are treated like device drivers, and they are loaded and unloaded like drivers as well. The loaded schedulers register with the Vassal Dispatcher. Threads are assigned to one of the scheduling policies. Vassal is coexisting with the default Windows NT scheduler. The different loaded schedulers form a hierarchy inside which the default Windows NT scheduler is at the bottom. When a new task has to be dispatched, the loaded schedulers are queried by a Request Decision until a runnable task is found.

Candea and Jones acknowledge that their approach has some limits, especially employing different scheduling strategies in parallel. Their implementation, for example, is only able

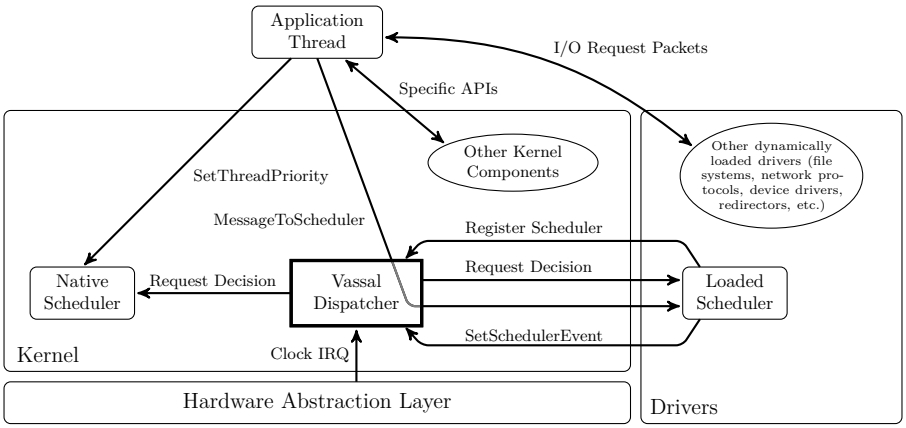


Figure 4.5: The Vassal architecture and its integration into Windows NT 4.0 [35, p. 6].

to have one scheduler loaded at a time. Furthermore, they argue that having multiple schedulers loaded at the same time might lead to problems when the schedulers have conflicting goals. The issue of having a multi-core system is not discussed by Candea and Jones.

Even though the Vassal extension is a good example for loadable schedulers, it is very limited regarding its flexibility as it still requires the native scheduler. Based on that design decision, the loaded schedulers are limited to a centralized scheduling approach. Furthermore, the approach cannot handle changes to the system architecture as the default scheduler is simply augmented with additional scheduling policies.

4.2.5 Kernel Live Patching

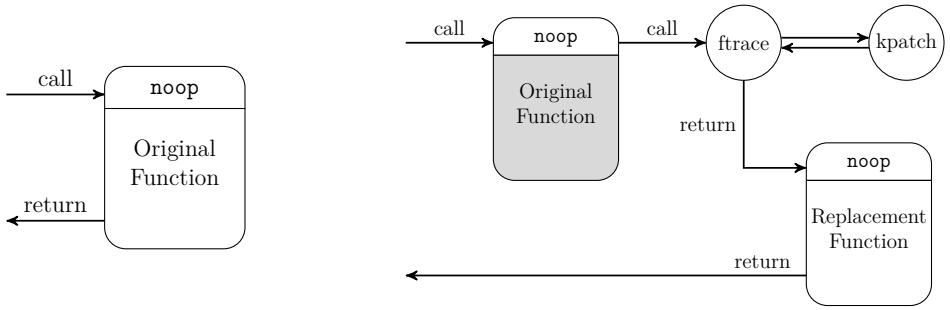
Certain application scenarios make it necessary to reduce the downtime of a system to a minimum. For example, a Service-Level Agreement (SLA) of 99.9999% results in a maximum downtime of the system of 31.5 seconds per year. In most cases, this is not even close to the reboot time of the system and bringing up all the services costs additional time. The importance of minimizing the unavailability of systems is backed by the fact that much literature exists that aims at system administrators to achieve that goal, e.g., Binnie [25]. For avoiding restarting a system due to the operating system kernel having to be patched, three approaches were developed to patch the Linux kernel during runtime.

Ksplice

Ksplice [10] allows the live patching of a Linux kernel by comparing the binary objects of the original and patched kernel on a by-function granularity. This will extract the differing functions and enable the creation of a patch for the running kernel. The patch is applied with the help of a kernel module by replacing the old function through a jump instruction to the new function. To avoid data corruption, the `stop_machine` function [S24, `kernel/stop_machine.c`] of the Linux kernel is used putting a core basically in a shutdown state no longer executing any tasks. Ksplice works fully automatically except for semantic changes to data structures. These require custom code to be handled. Ksplice was successfully used to apply several security patches to running Linux kernels. It also allows further patching of previously patched kernels; however, a rollback mechanism is not provided. Ksplice is limited to the x86 architecture.

kpatch

Another tool for live patching the Linux kernel is *kpatch* [S22]. Instead of comparing binaries, it can work on source diffs. It also uses the `stop_machine` function to ensure consistency. To apply the patch to the running Linux kernel, kpatch uses the *ftrace* facility [S15] of the Linux kernel. Ftrace is originally intended to profile the kernel.



(a) Before Patching. **(b)** After Patching.

Figure 4.6: The kpatch kernel live patching mechanism [82].

When activated, it is called at the beginning of every kernel function by replacing a previously inserted `noop` operation with a function call. This facility is reused by kpatch to introduce the replacement function (cf. Fig. 4.6). Kpatch allows the rollback of an applied patch by simply removing this code path again. However, with kpatch, it is not possible to apply patches that rely on changes to the kernel's internal data structures.

kGraft

The *kGraft* [152] live patching facility uses the same ftrace facility as kpatch to inject the patched code; however, contrary to Ksplice and kpatch, it does not require to stop the kernel for a short time through `stop_machine`. Instead, it uses two universes – the unpatched and the patched one – and migrates the existing tasks to the new reality when they leave the kernel space the next time (cf. Fig. 4.7). kGraft patches can be generated both from C source code and from object code.

4.3 Scheduler Interfaces

This section discusses details on the scheduler implementation of Linux and eCos. It gives a better understanding of how the state of the art schedulers are realized in different application domains.

4.3.1 Linux Scheduling Classes

The scheduler subsystem of the Linux kernel [S24] uses the concept of *Scheduling Classes*. A Scheduling Class can implement a certain scheduling algorithm in the boundaries of the Linux scheduler architecture. It has to comply to a certain programming interface.[7]

[7] For details on the interface refer to `kernel/sched/sched.h` Lines 1172–1231 in [S24].

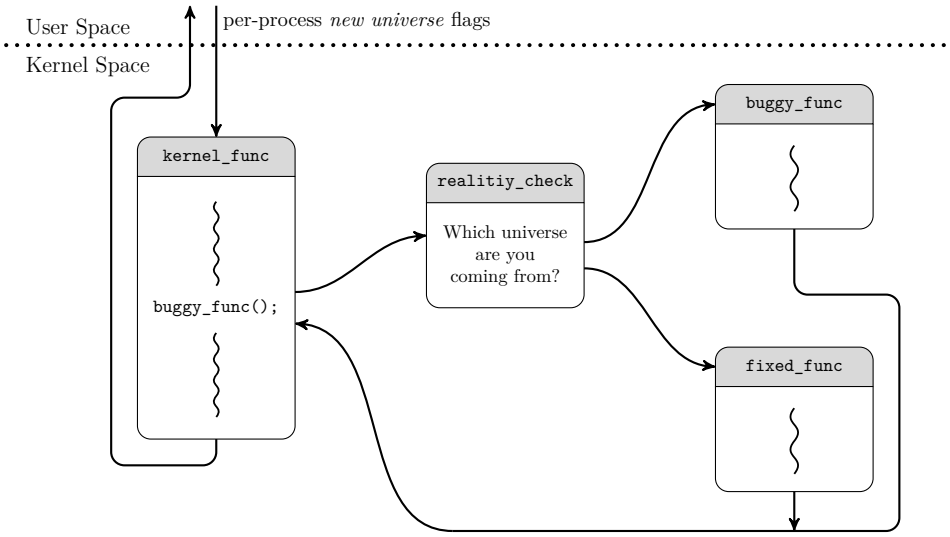


Figure 4.7: The kGraft kernel live patching mechanism [124]. Patched functions in the kernel are executed when the *new universe* flag of a user process is set, otherwise the old, buggy function is still used.

Currently, this means a decentralized scheduler with individual runqueues for every core and a centralized load balancing for the whole system. Every Scheduling Class has its runqueue. The Scheduling Classes are linked in the manner of an unidirectional list. Every time the scheduler subsystem has to determine the next task, the Scheduling Class at the head of the list is invoked and the task selected by the algorithm of that Scheduling Class is dispatched. If the runqueue of the first Scheduling Class is empty, the subsequent Scheduling Class in the list is invoked and so on.

The current version of the Linux kernel implements five different scheduling classes, which are linked as depicted in Fig. 4.8. The Scheduling Classes are compiled into the Linux kernel and cannot be changed during runtime. Each task can be assigned to a Scheduling Class. However, the functionality of the scheduling classes is limited to the scheduling model of the Linux kernel, hence resulting in a decentralized scheduling approach. It is, for instance, not possible to realize a centralized scheduler through the means of a Scheduling Class. Evidence for that is the centralized *BFS* by Kolivas [91] that implements an alternative scheduling algorithm for the Linux kernel. It is based on the *Earliest Eligible Virtual Deadline First* algorithm [150] and Staircase Deadline scheduler [92]. Kolivas refrains from using the Scheduling Class Application Programming Interface (API) as it is not possible to implement the algorithm with the restriction of that API.

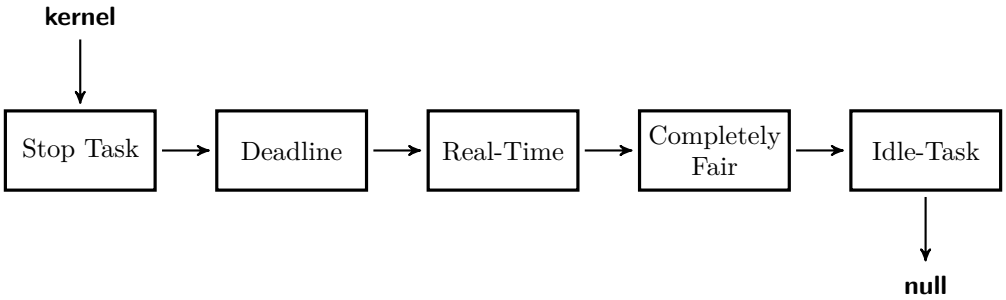


Figure 4.8: Sequence of Scheduling Classes in the Linux kernel v4.4. The *Stop Class* is used to overwrite any scheduling to shut down the machine or hot-plug a CPU. The *Deadline* class implements the Earliest Deadline First scheduling policy [97] with a Constant Bandwidth Server [1]. The *Real-Time* scheduling class follows the UNIX specification for real-time process scheduling [161]. The *Completely Fair* scheduling class implements the Completely Fair Scheduling policy [85]. The *Idle-Task* class is solely responsible for the dispatching of the idle task.

4.3.2 The eCos Scheduler

The eCos operating system [S12], like most other real-time operating systems, employs a priority-based scheduler by default. However, it is designed in a way that the scheduler implementation and therefore the scheduler algorithm can easily be replaced. For this purpose, eCos uses the inheritance feature of the C++ programming language. A new scheduler implementation is simply a child class of the main Scheduler class.[8] As in Linux, the scheduler cannot be exchanged during runtime. Even if that feature existed, it would be of limited use as eCos is not designed as a very dynamic system and has very limited user space capabilities. As a real-time operating system, it is focused on a fixed functionality.

Further limitations arise from the scheduler interface. Even though it allows a free implementation, the developer is very limited as in Linux he or she is bound to the scheduler architecture enforced by eCos. The interface does not allow an easy extension for acquiring additional information that might be useful to the scheduler or information that should be given from the scheduler to the operating system.

[8] For details on the class interface refer to `packages/kernel/v3_0/include/sched.hxx` in the source package in [S12].

4.4 Summary

The assessment of the related work has shown that the challenges in process scheduler architecture have been identified as a research area before. One of the main goals of the research efforts was the simplification of the scheduler design process. The prior work reached that goal to a certain degree, but always in a limited scope. The most noteworthy missing feature in such architectures is the support for many-core systems and the possibility to build scalable schedulers for such systems. Moreover, the previous work covers heterogeneous system only to a very limited extent.

The necessity to change fundamental parts of the operating system was also the object of prior research. As with the general scheduler architecture, the related work falls short to tackle many-core systems. The presented work regarding kernel live patching identified the problem of changing data structures that are actively used by the system. This is a special issue for the scheduler and not tackled by prior work regarding dynamic scheduling architectures.

CHAPTER 5

A Component-Based Scheduler Framework

Chapters 1 and 2 have outlined the challenges process scheduler design is facing and from that, Chapter 3 deduced requirements necessary to be addressed by a future scheduling architecture. The previous chapter has shown that the current state of the art falls short of addressing all the issues at the same time. For this reason, this dissertation introduces a novel approach to the architecture of process schedulers – the Component-Based Scheduler (CoBaS) framework. Figure 5.1 illustrates the overall architecture of the framework. It is designed as main scheduling facility for a runtime system like, in most cases, an operating system. However, the framework can schedule arbitrary task sets that fall into the task model outlined later in this chapter and is, therefore, not necessarily limited to operating systems. It can be used in every system that needs task management. The CoBaS framework consists of five major elements:

- **Components** that are enforcing the scheduling policies and enable to control the scheduling (cf. Section 5.2).
- **Pipes** that are an extension to the concept of runqueues and transport tasks from one Component to another (cf. Section 5.3).
- **A Notification System** that distributes events inside the framework on publish–subscribe based messaging pattern (cf. Section 5.4).
- **A Runtime System Adapter** that connects the runtime system independent parts of the CoBaS framework to the surrounding runtime system (cf. Section 5.5).
- **Topologies** that determine the layout and connection of Components to Pipes (cf. Section 5.6).

The framework does not have a general restriction how the tasks have to look like. They can be jobs, processes, threads, or whole virtual machines as described in Section 2.1. Through the standardized interfaces inside the framework, the Components are highly reusable and independent of the surrounding runtime system. Hence, the implementation of a scheduling policy within the CoBaS framework can be used in every system that integrates the framework. Furthermore, the encapsulation of the scheduling policy enforcement in Components allows the dynamic creation, deletion, and exchange of Component instances even during runtime. This approach allows the scheduler to be adapted to changes in the scheduling goals or even the underlying system architecture.

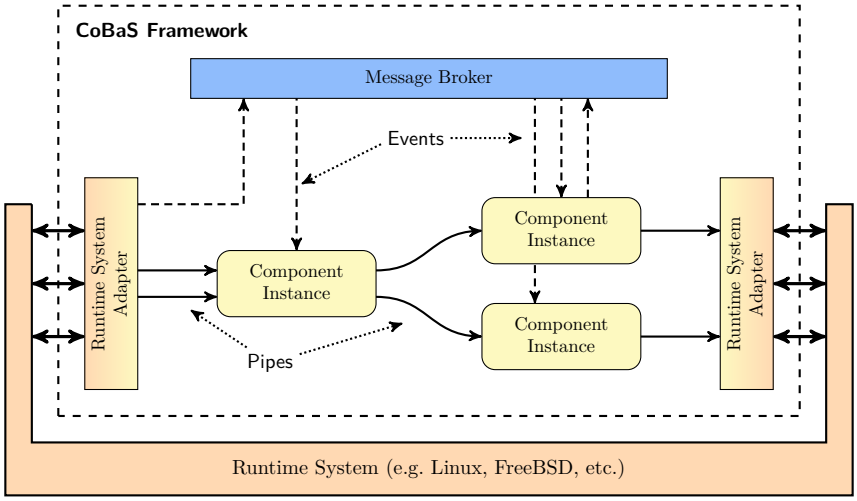


Figure 5.1: Overview of the CoBaS architecture.

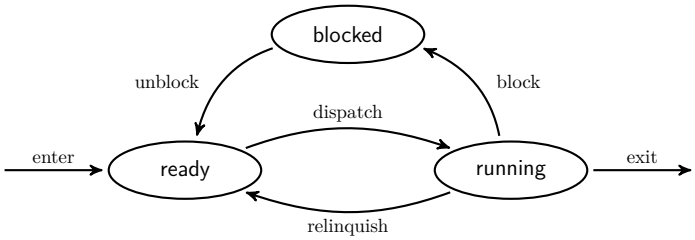
This chapter describes the CoBaS architecture in detail. It starts with the presentation and discussion of the system model that was used to create the framework. The Sections from 5.2 up to 5.6 are discussing the different elements of CoBaS in detail. Section 5.7 explains the interaction between the elements of the CoBaS framework with the help of examples to give a better understanding of the architecture and the interaction of the different parts of the framework among each other. Section 5.8 continues with a discussion on composability in the context of the CoBaS framework. Section 5.9 concludes this chapter with a discussion on the design rationales of the CoBaS framework, which focuses on the decision to use the component and the framework approach. Note that the concepts presented in Sections 5.1.2, 5.2, 5.3 and 5.5 of this Chapter were previously published in parts in Busse et al. [32].

5.1 System Model

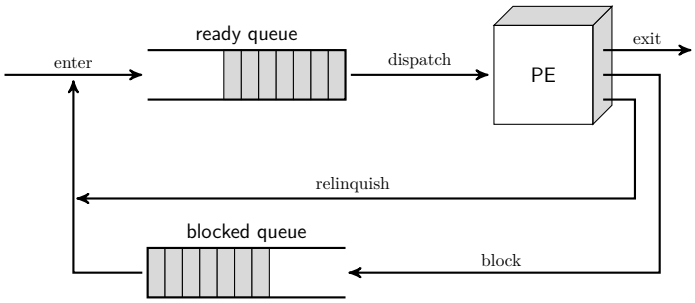
The model that is the foundation for the CoBaS framework can be subdivided into two parts: a task model and a scheduler model. The task model describes the properties of tasks the CoBaS framework can handle, while the scheduler model describes the scheduling process.

5.1.1 Task Model

Starting points for the task model are the traditional two- and three-state task models as described in literature (cf. Stallings [147, Ch. 3] and Tanenbaum [157, Ch. 2]). The two-state model is very limited and does not allow preemption. Therefore, it is only suitable



(a) State transition diagram [157, p. 88].



(b) Queueing diagram [147, p. 140].

Figure 5.2: The classical three-state task model. Tasks can be blocked because of the interaction with other tasks.

for batch processing granularity as described in Section 2.1.1 and not sufficient for the purpose of this dissertation. The three-state model (Fig. 5.2) is a further generalization of the two-state model that is not only suited for the batch processing granularity, but also for the other granularities as discussed in Sections 2.1.2 to 2.1.4.

However, it is questionable whether this rather simple academic model is also suitable for complex real-world systems. Take for example the task state machine of the µITRON system (Fig. 5.3). Although it has a `READY` state that is equivalent to the *ready* state of the three-state model and a `RUNNING` state that is equal to the *running* state of the three-state model as well, no single state can be identified as *blocked* state and a task can exit from a waiting state in contrast to the classical three-state model. Similar differences can be found looking at other legacy systems' task states, e.g., in UNIX systems (cf. Bach [13, p. 148]) or Windows (cf. Stallings [147, p. 200]).

To obtain a holistic task model for CoBaS, it is necessary to be independent of the actual state machine of the tasks. Looking closer at both the simplified three-state task model and the real-world examples reveals that the scheduler does not necessarily need all information in that task model. The only challenging part in task scheduling is the question which task to transit from the *ready* to the *running* state and vice versa. In the simple three-state model, the transitions from the *ready* to the *blocked* state and

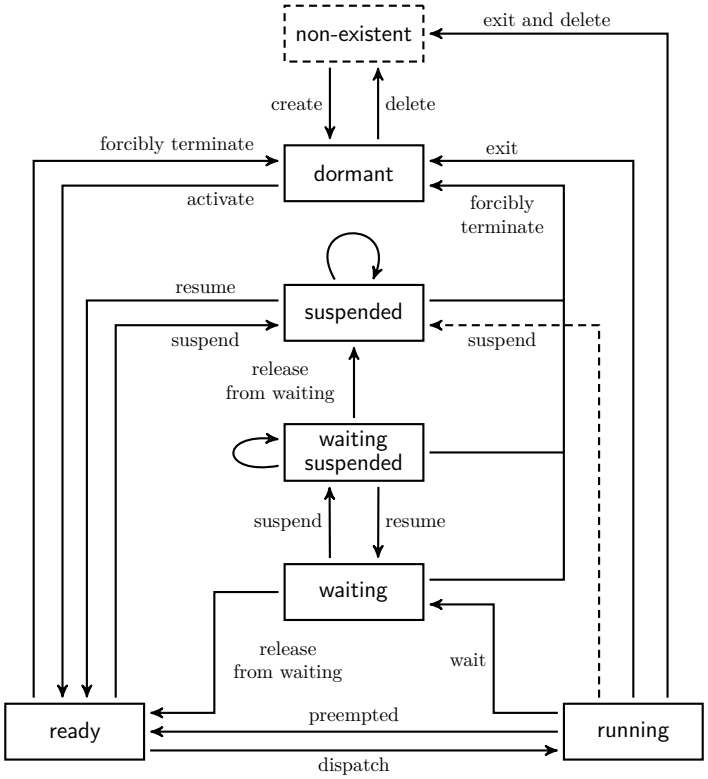


Figure 5.3: μITRON task states and transitions [139, p. 54].

from the *blocked* to the *ready* state are a given fact and not based on a decision. When a resource needed by the running task is not available, it is not questionable whether it gets blocked or not because there is no alternative. The same applies to the unblocking. Unlike the dispatching, the tasks do not get unblocked based on policy but on the fact that resources become available.

For more complex task state models, this reasoning might not be so obvious as there is a category of transitions that are not necessarily intrinsic but driven by a policy. Take for example the *swap out* of tasks, where the task is moved from main to secondary memory. This decision is based on policy. However, the memory management subsystem and not the scheduler subsystem is responsible for this decision. From the scheduler's point of view, the task is missing a resource, i.e. memory, to progress. The same line of argument can be applied to other properties. Take for example the `SUSPENDED` state of the μITRON state model. There, the task is forcefully denied the CPU, which can be modeled as a logical resource: *Permission to be dispatched*. This approach allows the same approach for tasks that are supposed to be put in the `SUSPENDED` state as it does for *swap out* tasks.

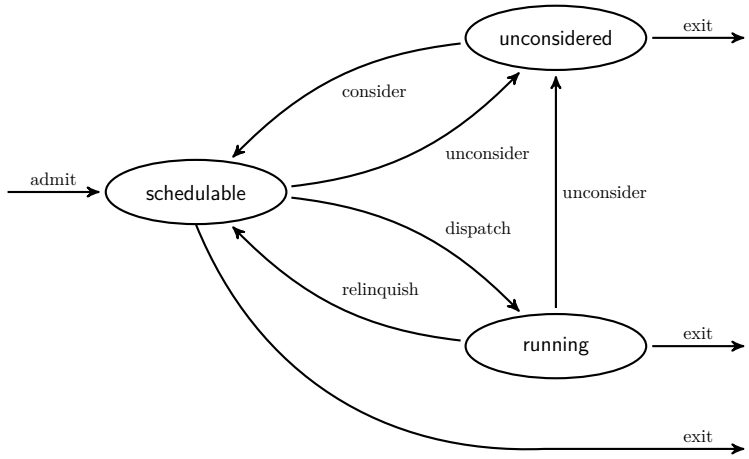


Figure 5.4: The CoBaS task state model.

Based on this insight, only the two transitions between the *ready* and the *running* state remain to be considered for the scheduler. The CoBaS framework uses an enhanced three-state task model as illustrated in Fig. 5.4. The framework only considers tasks that are assigned to it by the runtime system. Those tasks are either *running*, meaning they are currently processed by a PE similar to the classical three-state model or they are *schedulable*, meaning they can be dispatched to a PE. The latter state differs from the classical *ready* state as all tasks that are not blocked or currently executed are *ready*. With CoBaS, the runtime system can decide whether it wants a task to be considered for scheduling no matter if it is dispatchable or not. Tasks transit from the *ready* to the *running* state through dispatching and from the running to the ready state through relinquishment or preemption of the PE. The third state in the CoBaS model is the *unconsidered* state that, from the perspective of CoBaS, collects all the tasks that are not to be considered for the scheduling. The runtime system can put every task in that state if it comes to the conclusion that the task should not be considered by the scheduling algorithm. It has to be emphasized that the CoBaS framework only considers tasks in the *schedulable* and *running* state. Every task in the *unconsidered* state is not known to the framework and therefore not considered for scheduling.

Besides the changes to the meaning of the states, the CoBaS model introduces additional state transition to be as versatile as possible. With the CoBaS model, a task can be destroyed from every state, eliminating the need to dispatch a task before destroying it. Furthermore, the runtime system can remove a task from the *schedulable* state at any time, for any reason it deems necessary.

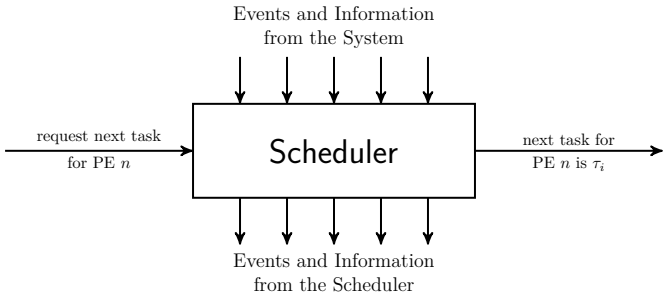


Figure 5.5: The CoBaS scheduler model.

5.1.2 Scheduler Model

After defining a task model, a scheduler model can be built that can accept such tasks and is suited for this dissertation. The model is depicted in Fig. 5.5. The scheduler can be queried for a task for a specific PE and will output a task for that PE. Furthermore, the scheduler can receive information and events from the system that may influence its scheduling decision. Those events could be for example a frequency change of a PE, the change from permanent to battery power, or reaching a critical temperature in the system. Also, the scheduler is informed about task state transitions over that interface and the system informs the scheduler that it has dispatched a task or that a task should no longer be considered for scheduling. On the other hand, the scheduler can notify the system about certain events like, e.g., an overload situation on a specific PE.

Even though the model is very simple, it is very flexible as it can be used to process every kind of information relevant to the scheduler or the system, while still fulfilling the main task of a scheduler: selecting tasks for specific PEs.

5.2 Components

Components are the core of the CoBaS framework and the means used to enforce scheduling policies. Components can order and filter tasks. An initial simple example is given in Fig. 5.6. In the example, Component A reorders the incoming task set, while Component B refines the ordered task-set by filtering.

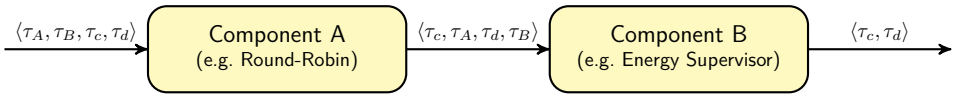


Figure 5.6: Example of task ordering and filtering by CoBaS Components. A task set is ordered by Component A. The ordered task set is further refined by Component B through filtering, in this case by filtering out the tasks identified by an uppercase letter index.

CoBaS differentiates between *Components* and *Component instances*. While the Component is the implementation, several instances of one Component can exist during runtime. For the remainder of this dissertation, the notion Component is used for both Components and Component instances. If a differentiation between those two is necessary and not obvious, it is pointed out.

Besides their direct functionality, Components serve three main purposes: Separation of concerns, workload distribution, and reusability. A Component is a stateless entity that holds instructions to perform a certain functionality like, e.g., task ordering or filtering. When using a Component, a Component instance has to be created that is stateful and can be executed by the runtime system. How Components serve their purpose is discussed subsequently.

5.2.1 Separation of Concerns

Why the component approach allows a separation of concerns shall be explained on the example of a decentralized multi-core scheduling with support for affinities to PEs and a round-robin policy for each PE. With a traditional scheduler architecture, the scheduler logic would be implemented as one entity. For example, the FreeBSD operating system uses a scheduling approach very similar to that, except that it also supports priorities. In the given example, three concerns can be identified:

- Load-Balancing for all PEs.
- The Assignment of an affinity to every task.
- A round-robin policy on each PE.

Each of these concerns can be handled by an individual Component or, to be more exact, by instances of Components. The resulting scheduler is depicted in Fig. 5.7 on the example of a quad-core system. It consists of a Component instance that can assign an affinity to every task, an instance that enforces the load-balancing between PEs with the support of the Affinity Component instance, and finally n instances of a Component that applies a round-robin policy, in the quad-core example n being four. The interaction between the Components is explained in detail later on in Section 5.7.

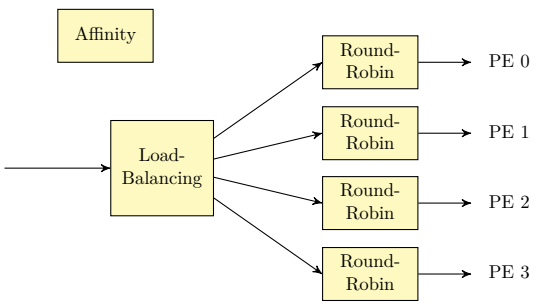


Figure 5.7: Separation of concerns for a multi-core scheduler.

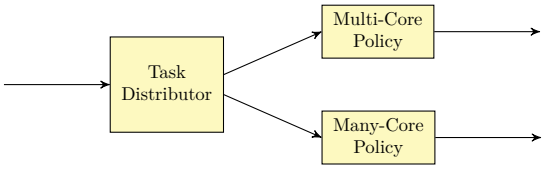


Figure 5.8: Code path separation for a heterogeneous system with dissimilar scheduling requirements.

5.2.2 Code Path Separation and Workload Distribution

Having a heterogeneous many-core system that, for example, consists of two parts – one part a classical flat multi-core topology and one a mesh-based many-core topology (cf. Fig. 2.3 on page 17) requires, possibly fundamentally, different scheduling strategies for the different parts. The state of the art approach to scheduler design would result in one scheduler implementation that needs to handle both scheduling approaches. Even though a differentiation in the code paths for the different architectures might exist, it would be hard to distinguish.

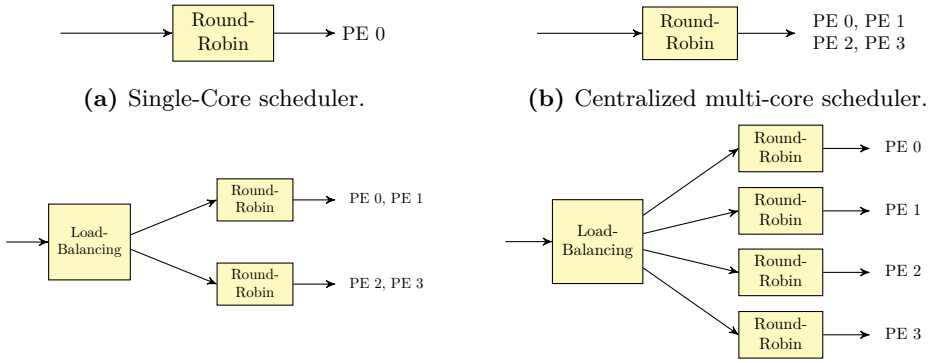
The component-based approach allows a much easier differentiation and division between the different scheduling strategies and therefore code paths. Figure 5.8 illustrates this separation on the example mentioned above: A shared load-balancing Component would assign the tasks either to the flat multi-core part of the heterogeneous processor or the mesh-based many-core part. The assigned tasks would then be scheduled based on the most efficient policy for the respective topologies.

The decomposition of the scheduler implementation in distinct Components also allows an easier to comprehend workload distribution. Assuming that the scheduler logic is executed by several PEs in parallel, bottlenecks are likely to occur and become more likely the more PEs try to run the scheduler at the same time. For example in the scheduler layout in Fig. 5.8, it can be reasoned that the Task Distributor might become a bottleneck because both parts of the heterogeneous system have to access it, while the dedicated implementations for the distinct scheduling policies can be accessed in parallel. How such contention is handled in CoBaS is discussed thoroughly in Section 6.1.

5.2.3 Reusability

Reusability consists of two aspects: The reuse of one scheduler implementation in different runtime systems and the reuse of existing parts of a scheduler implementation in the implementation of a new policy. The CoBaS framework allows both. The reuse between different runtime systems is given by the framework approach. A runtime system has to be adapted only once to the framework and can then use every scheduler implementation that was programmed against the framework's API.

The component approach also allows a broader reuse of existing parts of the scheduler. The round-robin Component from the example mentioned above can, for instance, be



(a) Single-Core scheduler.

(b) Centralized multi-core scheduler.

(c) Hierarchical multi-core scheduler.

(d) Decentralized multi-core scheduler.

Figure 5.9: Reuse of a single Component implementation in different scheduler and scheduling approaches.

used in different scenarios like a simple single-core scheduling (Figure 5.9a), a centralized multi-core scheduler (Figure 5.9b), a hierarchic scheduler (Figure 5.9c), or a decentralized scheduler (Figure 5.9d). Compared to the state of the art approach to scheduler implementation, the reuse is promoted by the clear functional distinction between individual Components.

5.3 Pipes

Pipes are one of two interfaces that are connecting Components to other Components or the framework. Components, as described above, are manipulating task sets by ordering or filtering of tasks. Those ordered and filtered task sets have to be handed over from one Component to another to compose several Components to a complete scheduler implementation. In current systems, tasks are managed in runqueues [147, Ch. 9][157, Ch. 2][S14, `sys/sys/runq.h`][S24, `kernel/sched/sched.h`]. However, for the CoBaS system, this data structure is both ineffective and not sufficient.

For the CoBaS framework, the concept of runqueues is extended to *Pipes*. A Pipe contains, besides a list of tasks assigned to the Pipe, a list of added and removed tasks, a list of moved tasks, a hook for the connected Component, and a lock (Fig. 5.10). When a task is added to or removed from the Pipe, the change will not only be applied to the task list but also noted in a dedicated list. This will take the burden from the receiving Component to determine the difference of the previous configuration of the task list to the current, which has, for example for the *Hunt–McIlroy* algorithm, a complexity of $\mathcal{O}(m \cdot n \cdot \log n)$ where m and n is the length of the input lists [76]. The same goes for changes to the task list that only change the relative order but do not add or remove a task. The reason to differentiate between these two kinds of changes is that it has no significant overhead to distinguish between in the Pipe but might introduce a significant

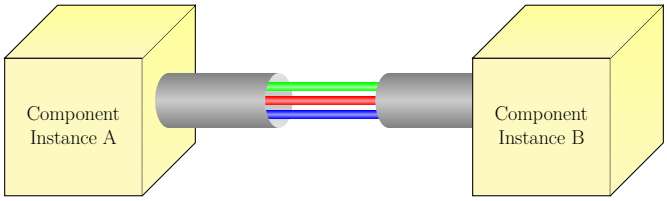


Figure 5.10: The structure of a CoBaS Pipe. The Pipe manages tasks in three distinct lists: A list for all tasks in the Pipe (green), a list with the tasks added and removed since the last access of the receiving Component (red), and a list with tasks rearranged since the last access of the receiving Component (blue).

computational overhead when handled in the Component as the two types of changes would have to be separated again if only one kind is needed. Furthermore, this allows an optimization of the Pipe structure. When, for example, a task is first added to the Pipe and later removed before the receiving Component instance reacts on these change, the two operations can cancel each other out. Besides the task management, the Pipe holds a hook for the connected Component. With this hook, one Component instance can hand over the flow of the scheduler execution to the next Component instance.

5.4 Notification System

To decide on the task order, a scheduling algorithm needs additional information about the system state as discussed in Section 5.1. The CoBaS approach assumes that changes to the scheduling decision are mainly event based, for example, the system changes from main power to battery operation, a timer runs out, the priority of a task changes, or a new PE arrives. Furthermore, the system is supposed to be dynamic and different Components should work together without necessarily knowing each other. To meet these requirements, CoBaS uses a topic-based publish-subscribe communication.[9]

Components and other elements of the CoBaS framework can both subscribe to topics and publish notification for topics. The CoBaS framework incorporates a *Broker* that manages all subscriptions and distributes events to the subscribed Components. The indirect communication pattern of publish-subscribe is not always sufficient for the communication of the framework. Certain aspects and information are at a certain location and might need to be queried by a Component or the framework itself. Therefore, the concept of a *Topic Responder* is introduced. The framework allows exactly one Component to register as a responder for a certain topic. This Component will answer to every query for the registered topic. The framework is limited to one responder to keep the Broker as simple as possible. More than one responder for one topic would raise the question on folding or reduction of the results collected from the responders.

[9] For more details on the publish-subscribe communication pattern refer to, e.g., Eugster et al. [54].

5.5 Runtime System Adapter

The *Runtime System Adapter* is the link between the actual runtime system and the CoBaS framework. It is unique to every runtime system and allows the runtime system to use the framework as well as allowing the framework to access facilities of the runtime system like e.g., memory management or logging facilities.

To work properly, an actual implementation of the CoBaS framework will need a certain set of functionalities. For instance, as it is designed in a dynamic way, it needs to offer some memory allocation functionality. The *Runtime System Adapter* has to realize this feature by either implementing a memory management algorithm itself or by forwarding memory allocation and deallocation requests to the surrounding runtime system.

The other way around, the *Runtime System Adapter* has to transform particular runtime system calls to, e.g., CoBaS notifications. Take for example the priority of a task in a priority-based scheduling policy. The priority can be set by a userland application via a system call. The system call would then be redirected to the *Runtime System Adapter*, which creates a notification that is submitted to the CoBaS framework. This approach is most feasible for information that is not part of the legacy system. Another method can be used for information or functionality that exists in the legacy system but is now handled by the CoBaS framework like, e.g., the creation of the new task. As this function might not only be used in system calls but also by other legacy code, the best approach is to provide the function in the *Runtime System Adapter* and link the existing code against the function in the Adapter. Finally, because of the flexible structure of CoBaS, it is also possible to directly interact with the framework bypassing the *Runtime System Adapter*. A notification in CoBaS can be created in the userland and, through a system call, directly inserted into the framework. All three interactions are illustrated in Fig. 5.11 on the next page on the example of an operating system with strict division between user and kernel space.

5.6 Topologies

To obtain a functional scheduler, CoBaS Components have to be instantiated and, where necessary, connected to Pipes. This goal is achieved with CoBaS *Topologies*. In general, Topologies can be considered as a set of rules how Components are instantiated and connected to each other. Furthermore, Topologies are responsible for assigning the entry and exit points of tasks into and out of the framework. The Topology decides to which Pipe a submitted task is added or from which Pipe a requested task for a PE is taken. Depending on how flexible they are regarding changes, Topologies can be divided into three groups: *Static Topologies*, *Dynamic Topologies*, and *Adaptive Topologies*. Note that the flexibility only reflects how well a scheduler can react to changes in the system, but not how well a system can handle heterogeneity.

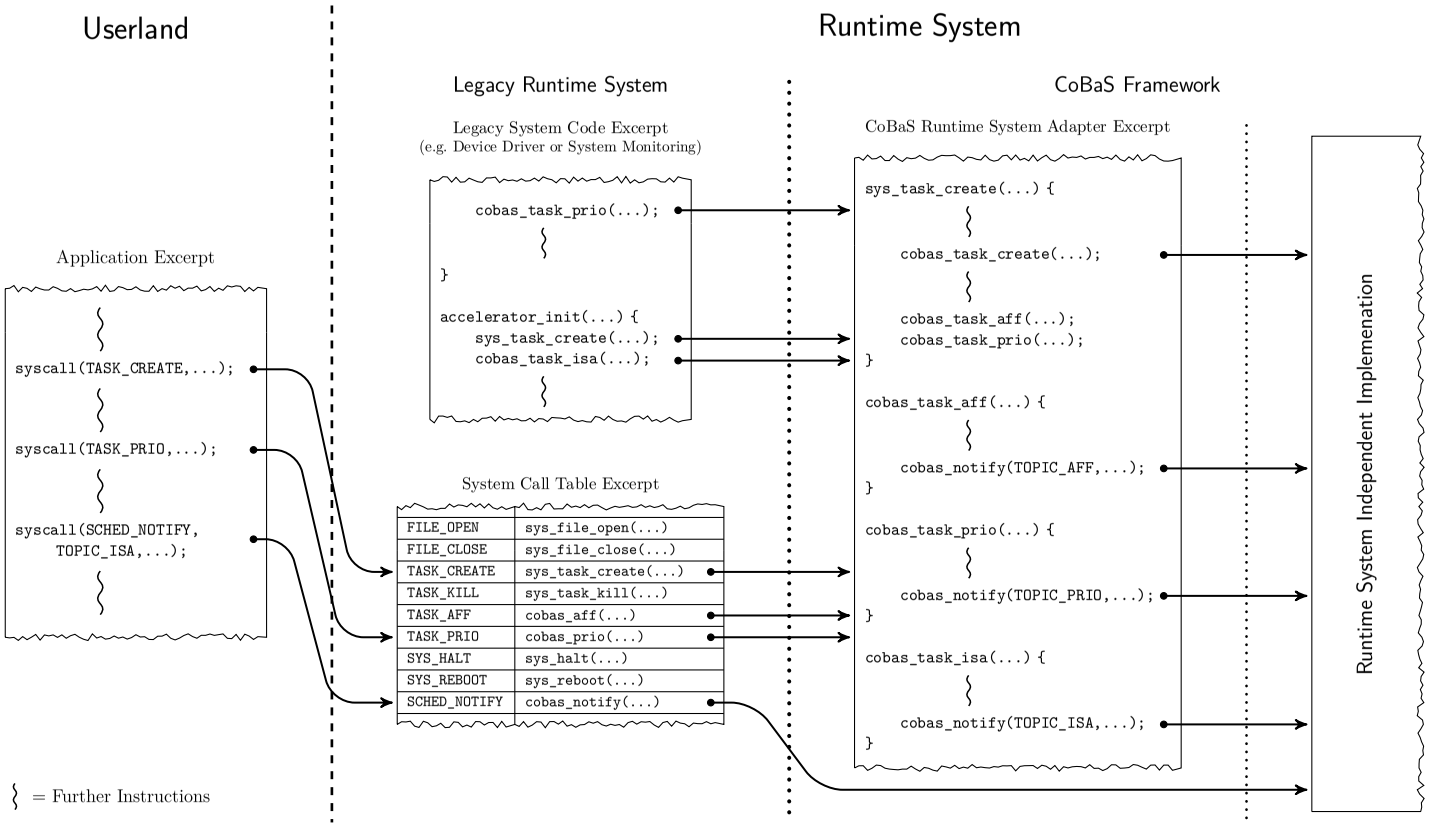


Figure 5.11: Example of a CoBaS Runtime System Adapter in an operating system with user and kernel space separation. Three different interactions are depicted: The interaction of legacy kernel code with the framework, the indirect interaction of the user space with the framework through modified system calls, and the direct interaction of the user space with the framework through a notification system call.

Listing 5.1: Example algorithm of a static Topology.

Precondition: $|PE|$ is the number of PEs in the system.

```
1: function BuildTopology
2:     for i ← 1 to (2 · |PE| + 1) do
3:         p_i ← new PIPE
4:     end for
5:     c_1 ← LoadBalancing(p_1 · p_(2·|PE|+1))
6:     for i ← 2 to (|PE| + 1) do
7:         c_i ← RoundRobin(p_i, p_(i+|PE|))
8:     end for
9: end function
```

5.6.1 Static Topologies

The simplest Topologies in CoBaS are *Static Topologies*. Static Topologies are part of the final system and configure the Component layout once at boot-time and will leave it untouched. Take the example of a decentralized multi-core scheduler for n PEs with load-balancing and a round-robin policy for the tasks assigned to each PE. Figure 5.12 depicts this scheduler Topology and Listing 5.1 shows the algorithm used to create this Topology. When the CoBaS framework is initialized, the code is executed. Based on the number of advised PEs, which is either given for the Topology or obtained from the runtime system via the Runtime System Adapter, the Topology computes the number of required Pipes, allocates, and initializes them (Lines 2 to 4). If, for example, the runtime adapter would report 4 PEs, the Topology would instantiate $p = 2 \cdot |PE| + 1 = 9$ Pipes. Next, the Topology creates one Load-Balancing instance (Line 5) and as many instances of the Round-Robin Component (Lines 6 to 8) as the number of PEs. During creation, it assigns the Pipes to the proper Components. Note that in Listing 5.1 it is assumed that the order of arguments of the Component instantiation implies which Pipe has which purpose.

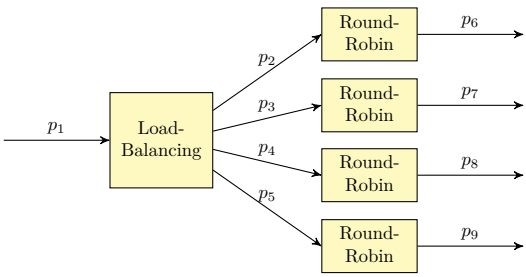


Figure 5.12: Example for a static Topology generated by Listing 5.1 with four PE.

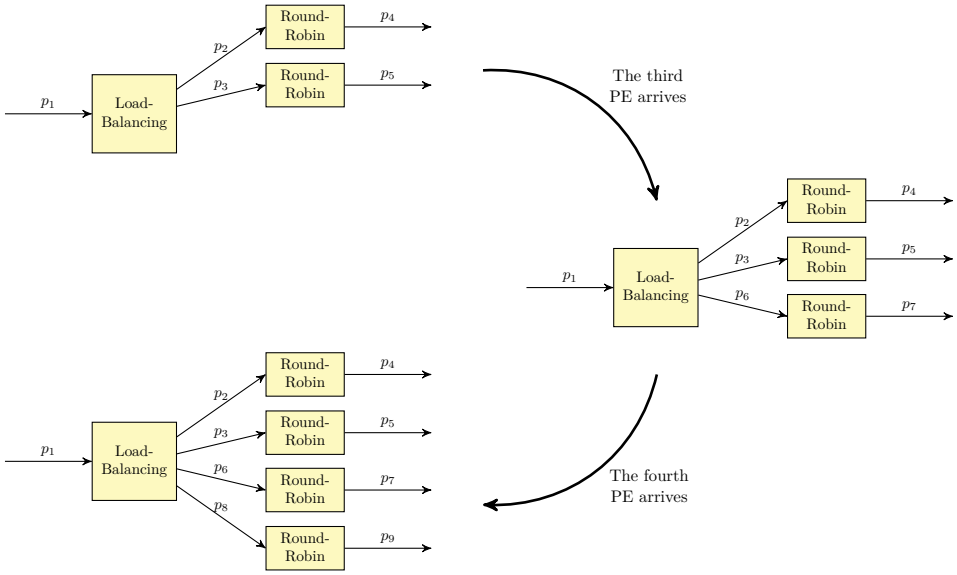


Figure 5.13: Changes in a dynamic Topology.

5.6.2 Dynamic Topologies

In contrast to *static Topologies*, *dynamic Topologies* can react to changes of the running system. Take the example above, but now the system starts with only two PEs and two more PEs will come up later through, e.g., CPU hotplugging (Figure 5.13). The initial scheduler is created as described above. However, the dynamic Topology is capable of reacting to the newly arriving PE. The runtime system adapter, for example, can create a notification once a new PE arrives. The Topology will change the scheduler layout accordingly, for instance by creating new Pipes and round-robin Components for the arrived PE and by replacing the existing two-output load-balancing instance by a four-output load balancing instance. Note that in CoBaS several possibilities exist to cope with this problem that is not only limited to the Topology. For example, also a *static Topology* could have been used and the arriving of new cores could be handled by the load-balancing Component.

5.6.3 Adaptive Topologies

For future systems, even *Dynamic Topologies* might not be sufficient. Take for example a system to which a processing accelerator like, e.g., a neural net engine is added. This would make it necessary to transform the CoBaS layout as depicted in Fig. 5.14. The initial Topology with the gray background had to be extended to the whole Topology that can distinguish different ISAs and is able to assign the task to the right Sub-Topology. Even though it is possible to achieve this with a *dynamic Topology* in general,

it would be necessary to predict all possible system changes and program the *dynamic Topology* accordingly. As this is infeasible, CoBaS offers *Adaptive Topologies*, which has two different forms: *Runtime System Controlled* and *Runtime System Assisted*.

Runtime System Controlled

The active CoBaS Topology can be controlled from user space to make changes to the scheduling policy. Take the example above: A user space daemon could monitor the system for changes relevant to the scheduling. Once it detects a change, it can proactively reconfigure the current Topology. In the example, it would create an instance of the *ISA-Splitter* Component, the round-robin Component, and connect the Pipes accordingly. The request goes through the Runtime System Adapter, as the call to CoBaS might be implemented as, e.g., a system call. The actual implementation depends strongly on the runtime system.

Static Topologies can always be converted to user space assisted *dynamic Topologies* as they only set up the CoBaS layout once. The situation is more complicated for *dynamic Topologies* as the changes triggered from user space might interfere with the Topology logic and invalidate assumptions it made. However, in certain situations, it is also possible. Take, again, the example: The general processing cores can be managed by a *dynamic Topology*, which is not necessarily influenced by changes induced from user space, hence making the old Topology a subset of the new one (cf. Fig. 5.14).

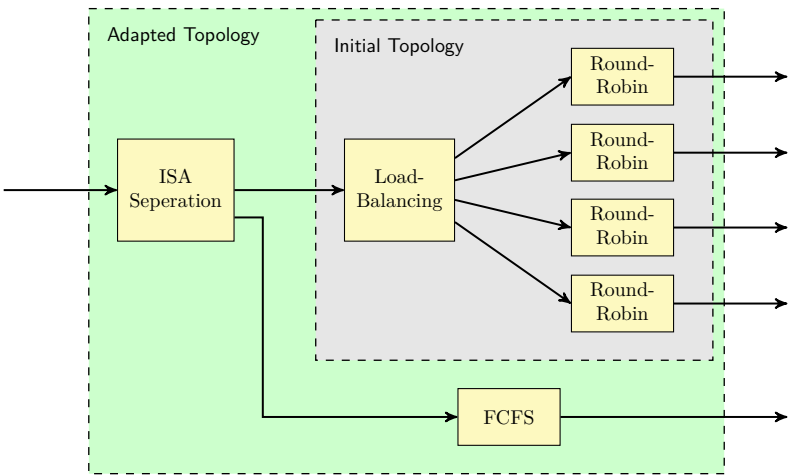


Figure 5.14: Example for an Adaptive Topology. Due to the introduction of a computation accelerator, the initial Topology in the gray box is no longer sufficient. It is extended towards the adapted Topology in the green box that includes the initial Topology.

Runtime System Assisted

The CoBaS architecture allows the Topology logic to be changed during runtime. As Topologies, as well as the Components, are only loosely coupled to the system, they can be easily swapped. This process needs assistance from the runtime system to induce the new Topology. In the example above, the vendor of the processing accelerator could ship a scheduler layout that has the necessary changes. During the initialization of the processing accelerator, the new Topology could be loaded, while replacing the old one.

Another approach than shipping a separate scheduler Topology could be the extraction and patching of the existing Topology. Compared to the runtime system controlled approach, this method has the benefit that, e.g., a user space tool could try and evaluate several new scheduler Topologies and finally submit the best one to the framework.

5.7 Scheduling Example

To achieve a better understanding of the interaction of the different elements of the CoBaS framework and how it forms a complete scheduler, this section shows how a simple CoBaS scheduler implementation behaves regarding different scheduler tasks. The example is based on a dual-core system that is supposed to have a priority based scheduling on the first core for compute intensive tasks and a round-robin scheduling on the second core for interactive tasks. The CoBaS Topology that can enforce such behavior could consist of the following Component instances. Note that this Topology is only one way to implement a scheduler in CoBaS with the behavior as described:

- A load-balancing instance that distributes the work among the two cores.
- A priority instance that applies a priority based policy on the first core.
- A round-robin instance that enforces a round-robin policy on the second core.
- A Component that assigns a PE affinity to every task.
- A Component that assigns a priority to every task.

Figure 5.15 shows the layout of such a scheduler. For a better understanding, the tasks are referred to in the figures as well as in the subsequent text through pictograms of colored balls, e.g.,  . Component instances are depicted as yellow boxes, whereas the Broker of the CoBaS framework is illustrated as blue box. The Component instances of the load-balancer, round-robin policy enforcement, and priority policy enforcement are connected by Pipes that are holding references to the submitted tasks. A table in every figure summarizes the properties of each task before the changes depicted in that figure. On the first PE, the task  is running, while the second PE is currently idle. The Runtime System Adapter is left out from the figures for an improved comprehensibility. Every event that does not originate from a Component is assumed to originate from or have as a target the Runtime System Adapter or the framework itself. The figures in this section show the state of the system at a certain point in time. That means in particular that most tasks are referenced by multiple Pipes at once. For example in Fig. 5.15, the task  is referenced by three Pipes: the initial Pipe and the two Pipes

connected to the round-robin policy Component. Even though from a conceptional point of view the tasks are handed over from one Pipe to another, to be able to avoid unnecessary re-computations of the whole task sets on every change, every Pipe can hold its state and therefore reference to a task. This does not limit the framework as it is still possible to do the computation at every change if that is desired. However, it is also possible to only consider changes. The examples given in the subsequent sub-sections are based on the idea to only consider changes and therefore, only changes are applied to the Pipes through the steps explained in the examples.

Sections 5.7.1 to 5.7.4 explain the simulation of the following list of events that happen successively after the initial situation depicted in Fig. 5.15:

- The next task for the second PE is requested from the CoBaS framework and subsequently dispatched by the runtime system.
- A new task is submitted to the scheduler.
- The priority of one of the tasks is changed.
- The affinity to a certain PE of one task is changed.

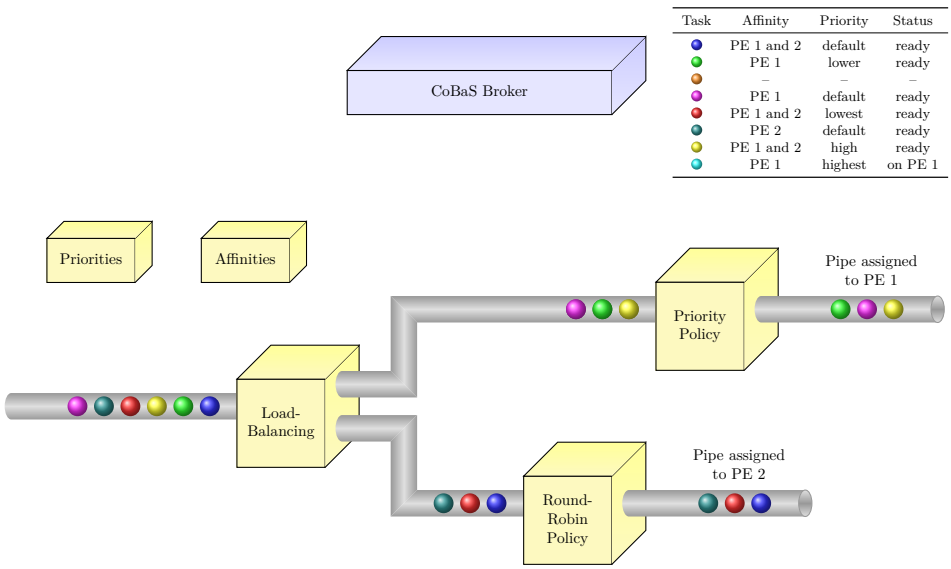


Figure 5.15: The initial state of the CoBaS scheduling example. Tasks are depicted as colored balls. Different balls with the same color indicate references to the same task. Yellow boxes are depicting *Components* instances. The blue box represents the *Broker* that is an integral part of the framework. The *Runtime System Adapter* is not illustrated for a better comprehensibility. Note that the figure depicts the state of the whole system at a certain point in time. That means in particular that most tasks are referenced by multiple Pipes at once.

5.7.1 Example for a Task Selection

In the first operation of the example, the runtime system queries the CoBaS framework for the next task that is supposed to run on the second PE and dispatches it to that PE. For that purpose, the runtime system invokes a function in the *Runtime System Adapter* that selects the task. In the example, the *Runtime System Adapter* takes the first task from the Pipe that is designated to the PE in question. For the example, this is the far right lower Pipe. Therefore, the *Runtime System Adapter* will report the ● task to the runtime system (① in Fig. 5.16). The ● task is then dispatched on the second PE, which results in a notification on the `DISPATCH` topic generated by the *Runtime System Adapter* (② in Fig. 5.16). The framework is by default subscribed to the `DISPATCH` topic as it always triggers a change in an initial Pipe (③ in Fig. 5.16). However, also other Components could be interested in dispatch events; therefore, the event is processed through the CoBaS notification system. The Runtime System Adapter removes the task from the initial Pipe and triggers a Pipe update (④ in Fig. 5.16). This Pipe update and, therefore, the task removal is propagated through the pipeline: The Load-Balancing Component removes the tasks from its outgoing Pipes and triggers a Pipe update on its own (⑤ in Fig. 5.16). The same happens in the Round-Robin Policy Component (⑥ in Fig. 5.16), which finalizes the process of task selection and dispatching.

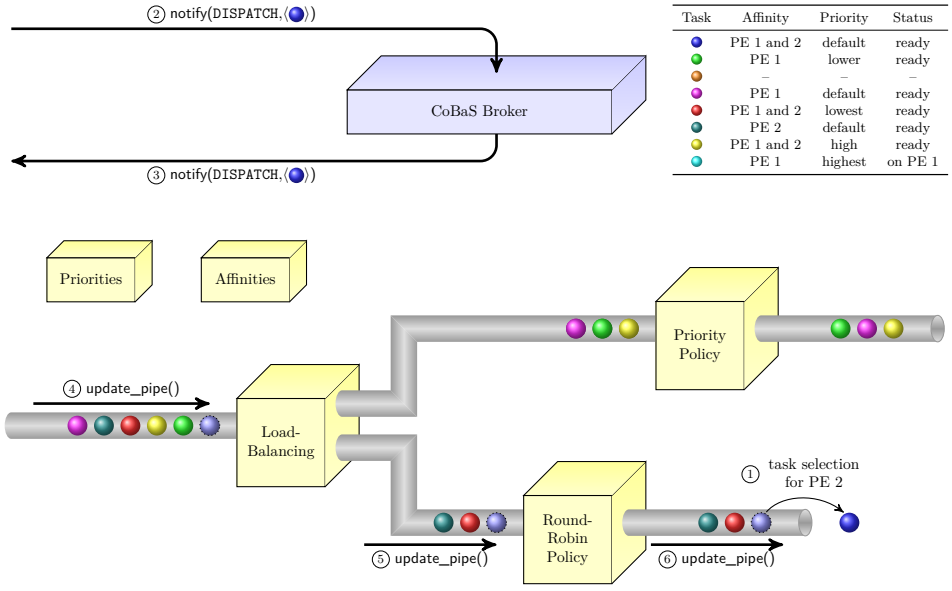


Figure 5.16: Example for the task selection by the CoBaS framework. The next task for the second PE is requested by the runtime system and subsequently dispatched.

5.7.2 Example for a Task Submission

In this example, a new task is submitted to the scheduler. The framework is notified about the task transmission via the *Runtime System Adapter* (①) in Fig. 5.17). The notification is forwarded to the framework (②) in Fig. 5.17) that puts the newly created task in the first Pipe of the Topology and causes a Pipe update that triggers the load-balancing Component instance (③) in Fig. 5.17). Again like in the previous example regarding the task removal, in this example, the task addition is propagated through the pipeline. However, additional steps are necessary to do so. To assign the newly added task to one of its outgoing Pipes, the load-balancing Component has to acquire the affinities of the tasks as it might be supposed to run on a certain PE. In order to do so, it sends a request for the `AFFINITY` topic to the *Broker* (④) in Fig. 5.17). During the instantiation of the Component instance, the affinities instance has registered for that topic as the responder, therefore the *Broker* will forward this request to it (⑤) in Fig. 5.17). The affinity Component will then reply with the default affinity, which allows the task to run on every PE, as no specific affinity was submitted for the task (⑥) in Fig. 5.17). The reply is then forwarded further by the *Broker* to the originating Component instance (⑦) in Fig. 5.17). The load-balancing instance can then assign the new task to one of its outgoing Pipes. It selects the lower Pipe that is connected to the round-robin Component, which causes a Pipe update for the round-robin Component (⑧) in Fig. 5.17). The change in its incoming Pipe makes the round-robin Component change its outgoing Pipe by appending the new tasks to the

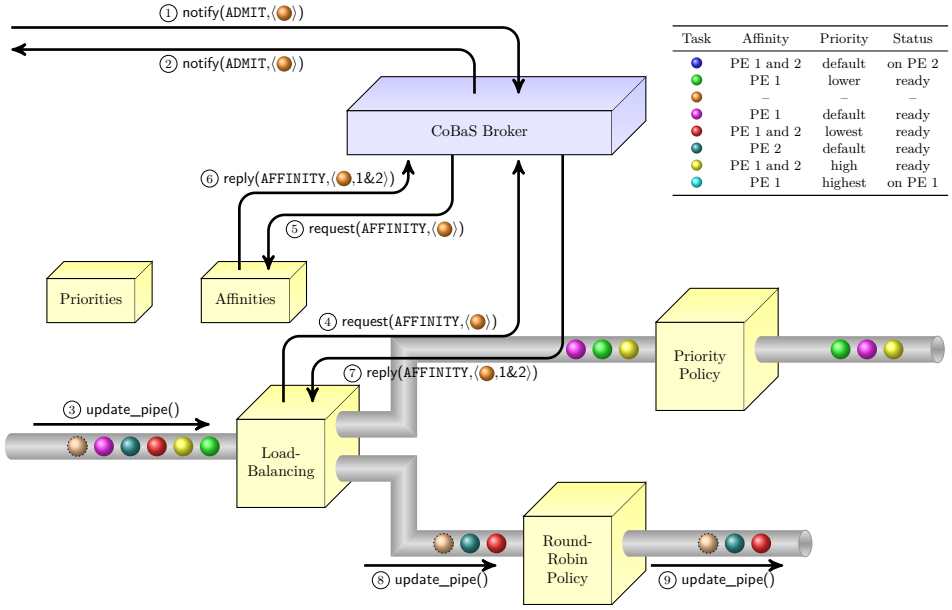


Figure 5.17: Submission of a new task to the CoBaS framework.

existing ones. This operation, again, results in a Pipe update. As in the first step of the operation of the example, the update has no effect as the outgoing Pipe of the round-robin Component is the last one of the Pipeline (⑨ in Fig. 5.17).

The example would work the same way for a task that is not newly created but woken up after it was, e.g., blocked. The only difference is that the affinities Component might have already stored information about the task. As a result, it has to be submitted to the other outgoing Pipe of the load-balancing Component.

5.7.3 Example for a Change of a Task Priority

The third operation in the example changes the property of one of the tasks. The priority of the ● task is changed from the default priority to the high priority level. The initial notification that triggers the change of priority comes from the *Runtime System Adapter* (① in Fig. 5.18). The change could, for example, be initiated from user space either through a dedicated system call that is translated by the Runtime System Adapter to a notification or through a user space tool that can send notifications through a generic system call to the scheduler. In the example, both the Priorities Component and the Priority Policy Component are subscribed to the `PRIORITY` topic. The Priorities Component keeps track of all priorities of all processes, so it is natural that it is subscribed to that topic. The Priority Policy Component is subscribed to the topic, as a priority change might require an immediate change to the outgoing

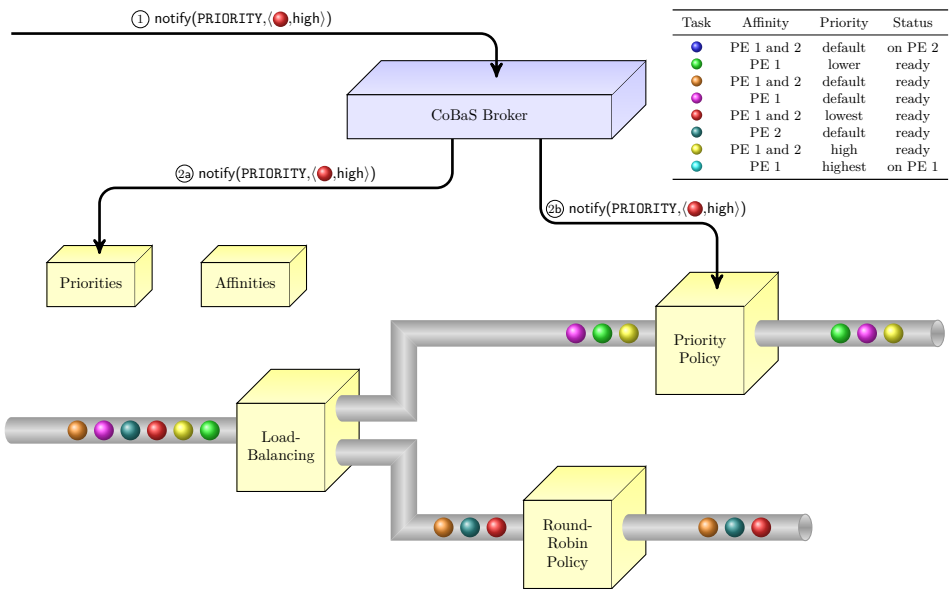


Figure 5.18: Change of the priority property of a task.

Pipe. Because of the subscriptions, the *Broker* distributes the notifications to those two Components (2a and 2b in Fig. 5.18). The order of delivering the notifications is not deterministic. As the task is currently not in the Pipes of the Priority Policy Component, the notification will simply be dropped by that Component. The Priorities Component will save the change for the specified task. As both the Load-Balancing and the Round-Robin Policy Component are not influenced by the priority, the change has no impact of the current task order in the system.

5.7.4 Example for a Change of a Task Affinity

The last operation of the example changes the affinity of the task. The initial step is the same as in the example above: A notification with the modification of the affinity from both PEs to an affinity towards only the first core is sent to the *CoBaS Broker* (1 in Fig. 5.19). Both the Affinities Component as well as the Load-Balancing Component are subscribed to the `AFFINITY` topic. Again, the distribution of the notification by the *Broker* to the subscribed Components is not deterministic (2a and 2b in Fig. 5.19). The Affinities Component will save the new affinity for the task the same way as the Priorities Component saved the task's priority in the operation above. However, contrary to the example above, the Load-Balancing Component does not simply drop the notification as the task is in one of its Pipes. It conducts a migration of the task

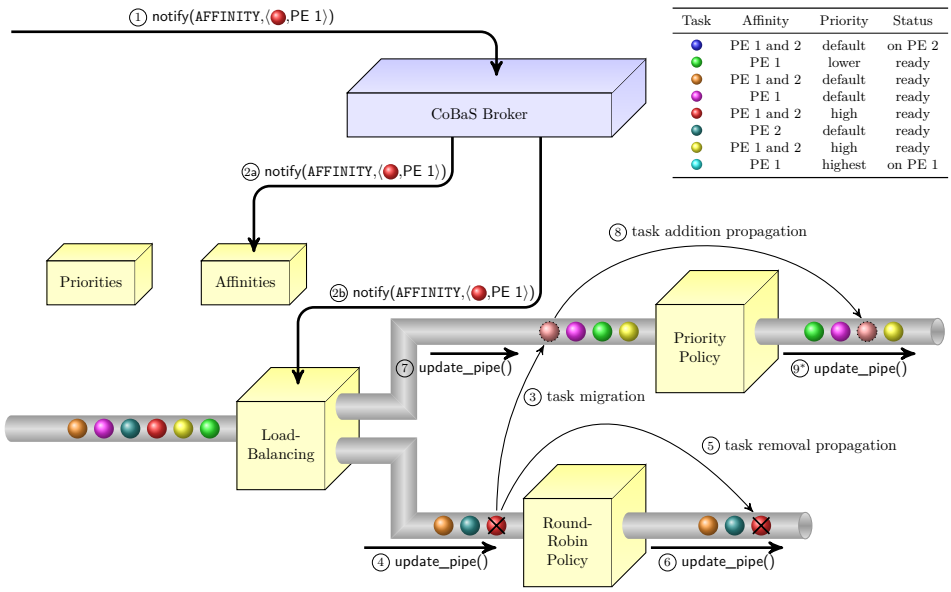


Figure 5.19: Change of the affinity property of a task. Note that the steps for the acquisition of the priority of the task prior to step 9* are not shown. Refer to Section 5.7.2 for details on that process.

from the lower of its output Pipes to the upper one (③ in Fig. 5.19). This action results in a Pipe update to both output Pipes subsequently (④ and ⑦ in Fig. 5.19) that is propagated through the connected Component instances (⑤ and ⑧ in Fig. 5.19). The Round-Robin Policy Component simply removes the task from its output Pipe and triggers an update on that Pipe (⑥ in Fig. 5.19). However, the Priority Policy Component needs to query for the priority of the task, which works analogously to the request of the affinity from the Load-Balancing Component presented in Section 5.7.2. As the process was already described, it is not depicted in Fig. 5.19, but only the resulting Pipe update is shown (⑨* in Fig. 5.19).

5.8 Composition

To obtain a working and sensible scheduler, CoBaS Components cannot be combined arbitrarily. Sometimes, those Components cannot be used together in a Topology at all; sometimes, they can only be utilized in a certain manner. Take for example a Component that enforces a FCFS policy and another Component that applies the Last-Come, First-Served (LCFS) policy. In a single-core system, these two Components cannot be combined in a meaningful way, as an instance of one of these Components will always change the order in a way that does not comply with the policy of the other Component. However, in a multi-core system, instances of these two Components can co-exist in a sensible manner, when used on different cores.

This section first discusses general considerations regarding optimization goals for scheduling and the composition of partial solutions for those aims. The second part shows how CoBaS Components can be classified and composed based on these considerations.

5.8.1 Composability

Composition is a technique that allows finding a solution for a problem by solving distinct aspects of that problem and combining those solvers to find the best overall option. The basis for this approach is the assumption that it is easier and more efficient to concentrate on different aspects of the whole problem in each component and optimize the partial solution. Looking only at one aspect at a time allows a better comprehension and therefore a potentially better solution.

However, applying this technique requires certain degrees of freedom to solve the problem and can be considered a combinatorial problem. If two components use the same degree of freedom, they can come to contradicting decisions how to use it, making a composition suboptimal or even impossible. Take the following abstract scheduling example: A system requires the optimization towards two goals $\mathfrak{X}$ and $\mathfrak{Y}$. This could be, for example, responsiveness and energy efficiency. Assume a certain task set $\tau_0 \dots \tau_n$ with $n \in \mathbb{N}$ to be scheduled. Furthermore, two value functions $|\tau_i|_{\mathfrak{X}}$ and $|\tau_i|_{\mathfrak{Y}}$ that

calculate a value for each task regarding the goals are given. The parameter p_i is the position of task i in the computed schedule, e.g., for the next task $p_i = 0$, for the second next task $p_i = 1$, and so on. Moreover, a task can be selected to be never scheduled, resulting in $p_i = \varnothing$:

$$|\tau_i|_{\mathfrak{X}} = \begin{cases} i \cdot 2p_i & \text{if } p_i \neq \varnothing \\ -\infty & \text{if } p_i = \varnothing \end{cases} \qquad \text{and} \qquad |\tau_i|_{\mathfrak{Y}} = \begin{cases} -i \cdot p_i & \text{if } p_i \neq \varnothing \\ -\infty & \text{if } p_i = \varnothing \end{cases}$$

The total value functions of the computed schedule regarding the goals $\mathfrak{X}$ and $\mathfrak{Y}$ for this example are defined as:

$$|\mathfrak{X}| = \sum_{i=0}^{n} |\tau_i|_{\mathfrak{X}} \qquad \text{and} \qquad |\mathfrak{Y}| = \sum_{i=0}^{n} |\tau_i|_{\mathfrak{Y}}$$

The value function for the whole system regarding the computed schedules is defined in this example as $\Pi = |\mathfrak{X}| + |\mathfrak{Y}|$ and the optimization goal is $\max\{\Pi\}$. The optimal schedule for this problem is $\tau_0 \to \tau_1 \to \ldots \to \tau_{n-1} \to \tau_n$.

Now let us assume the problem gets decomposed in two components, one finding the optimal schedule for $\mathfrak{X}$ and one for $\mathfrak{Y}$ so that $\max\{|\mathfrak{X}|\}$ and $\max\{|\mathfrak{Y}|\}$ holds. For $\mathfrak{X}$, this is the same solution as above, however, for $\mathfrak{Y}$ the optimal solution would be $\tau_n \to \tau_{n-1} \to \ldots \to \tau_1 \to \tau_0$. Because the two results are completely contradicting, it is not possible to compose these two solutions. Looking closer at the two value functions for the optimization goals, it can be seen that the value function for $\mathfrak{Y}$ is strictly dominated by the value function of $\mathfrak{X}$, therefore, an optimization towards $\mathfrak{Y}$ is in general not possible. This example is one extreme of the composition problem. The other extreme would be that both optimizations come to the same result, which is, for example, the case when the value function for $\mathfrak{Y}$ is changed to $\mathfrak{Z}$ as following:

$$|\tau_i|_{\mathfrak{Z}} = \begin{cases} i \cdot p_i & \text{if } p_i \neq \varnothing \\ -\infty & \text{if } p_i = \varnothing \end{cases}$$

The value functions $|\mathfrak{X}|$ and $|\mathfrak{Z}|$ are only a linear combination of each other as $|\mathfrak{X}| = 2 \cdot |\mathfrak{Z}|$. In such a case, the composition of two components optimizing for each goal separately is trivial as they should come to the same solution. The following paragraphs discuss the cases that lie between these two border cases in more detail.

Absolute Decisions

Some optimization goals may require that tasks are never scheduled, as, for example, a task is not suited to run on a specific PE or the energy budget does not allow the execution right now, or that a task needs to be executed at a certain point, e.g., next, to improve responsiveness of the system. Take the following example that depicts the former of the two situations above. In the task set $\tau_0 \ldots \tau_n$ with $n \in \mathbb{N}$, the subset $\tau_k \ldots \tau_n$ with $k \in \mathbb{N} \wedge k < n$ is currently not suited to be executed. It shall be modeled through the optimization goal $\mathfrak{A}$ with the following value function:

$$|\tau_i|_{\mathfrak{A}} = \left\{ \begin{array}{ll} 0 & \text{if } p_i \neq \varnothing \\ -\infty & \text{if } p_i = \varnothing \end{array} \right\} \forall i < k \qquad \text{and} \qquad |\tau_i|_{\mathfrak{A}} = \left\{ \begin{array}{ll} -\infty & \text{if } p_i \neq \varnothing \\ 0 & \text{if } p_i = \varnothing \end{array} \right\} \forall i \geq k$$

The optimization goal $\mathfrak{A}$ is indifferent regarding the schedule for task τ_0 through τ_{k-1}, however, it requires tasks τ_k through τ_n never to be scheduled. A component optimizing for this goal requires a solution that simply excludes tasks τ_k through τ_n and does not have to care about the order of the other tasks.

Now assume that schedule shall not only be optimized regarding the goal $\mathfrak{A}$, but also regarding $\mathfrak{X}$ from above. Having two distinct components, each optimizing regarding one of those two goals can be composed without any restriction in a subsequent order to achieve the optimal schedule.

Degrees of Freedom

As discussed above, the composition also depends on the degrees of freedom. Again, the impact is explained via an example. Take a system that consists of two PEs 0 and 1 and a new optimization goal $\mathfrak{M}$ that depends on the assigned PE c_i because the system might, e.g., be heterogeneous:

$$|\tau_i|_{\mathfrak{M}} = \left\{ \begin{array}{ll} 1 & \text{if } c_i = 0 \\ 0 & \text{if } c_i = 1 \end{array} \right\} \forall i \in \{x | x \pmod 2 = 0\}$$

$$|\tau_i|_{\mathfrak{M}} = \left\{ \begin{array}{ll} 0 & \text{if } c_i = 0 \\ 1 & \text{if } c_i = 1 \end{array} \right\} \forall i \in \{x | x \pmod 2 = 1\}$$

Again as above, a component that optimizes for this goal can be combined with a component that optimizes for goal $\mathfrak{X}$. In this example, the reason lies in the fact that the criterion for optimality only depends on the degree of freedom regarding the assignment to a PE or the order respectively. This allows a sequential composition of those two components. The order is, again, not relevant as long as the component only touches one of the degrees of freedom, i.e., does not change the task order when it assigns tasks to a PE.

In current multi- and many-core systems, only two degrees of freedom exist: task order and assignment to PE. However, by the wider introduction of FPGA technology, additional degrees of freedom can emerge regarding the scheduling (cf. Section 2.2.4).

Decision Refinement

Certain situations exist that allow a composition even though two optimization goals are bearing the same degree of freedom. An example is a priority based scheduling like the ULE scheduler [134]. It consists of two aspects: Each task is assigned to a priority group that is scheduled subsequently, beginning with the highest priority. Inside the priority groups, another scheduling scheme is applied regarding a specific optimization goal. This would result in the following model with the target $\mathfrak{P}$ representing the priority and the goal $\mathfrak{I}$ representing the scheduling inside the groups:

$$|\tau_i|_{\mathfrak{P}} = \left\{ \begin{array}{ll} (i \bmod 3) \cdot p_i & \text{if } p_i \neq \varnothing \\ -\infty & \text{if } p_i = \varnothing \end{array} \right. \quad \text{and} \quad |\tau_i|_{\mathfrak{I}} = \left\{ \begin{array}{ll} -i \cdot p_i & \text{if } p_i \neq \varnothing \\ -\infty & \text{if } p_i = \varnothing \end{array} \right.$$

To achieve the scheduling described above, a particular composition is required. First, the component that enforces $\mathfrak{P}$ has to be applied to the task set. It will group the incoming tasks in three groups:

$$\tau_0 \to \cdots \to \tau_{\lfloor \frac{n}{3} \rfloor \cdot 3i} \to \tau_1 \to \cdots \to \tau_{\lfloor \frac{n}{3} \rfloor \cdot 3j+1} \to \tau_2 \to \cdots \to \tau_{\lfloor \frac{n}{3} \rfloor \cdot 3k+2}$$

$$\forall i \in \mathbb{N} \wedge \lfloor \tfrac{n}{3} \rfloor \cdot 3i \leq n \;\text{ and }\; \forall j \in \mathbb{N} \wedge \lfloor \tfrac{n}{3} \rfloor \cdot 3j+1 \leq n \;\text{ and }\; \forall k \in \mathbb{N} \wedge \lfloor \tfrac{n}{3} \rfloor \cdot 3k+2 \leq n$$

Because the goal $\mathfrak{P}$ is indifferent to the order of tasks inside the groups, the optimization of $\mathfrak{I}$ can be applied to each cluster:

$$\underbrace{\tau_0 \to \cdots \to \tau_{\lfloor \frac{n}{3} \rfloor \cdot 3i}}_{\max\{|\mathfrak{I}|\}} \to \underbrace{\tau_1 \to \cdots \to \tau_{\lfloor \frac{n}{3} \rfloor \cdot 3j+1}}_{\max\{|\mathfrak{I}|\}} \to \underbrace{\tau_2 \to \cdots \to \tau_{\lfloor \frac{n}{3} \rfloor \cdot 3k+2}}_{\max\{|\mathfrak{I}|\}}$$

$$\overbrace{\tau_{\lfloor \frac{n}{3} \rfloor \cdot 3i} \to \cdots \to \tau_0}^{\max\{|\mathfrak{I}|\}} \to \overbrace{\tau_{\lfloor \frac{n}{3} \rfloor \cdot 3j+1} \to \cdots \to \tau_1}^{\max\{|\mathfrak{I}|\}} \to \overbrace{\tau_{\lfloor \frac{n}{3} \rfloor \cdot 3k+2} \to \cdots \to \tau_2}^{\max\{|\mathfrak{I}|\}}$$

5.8.2 Component Classification

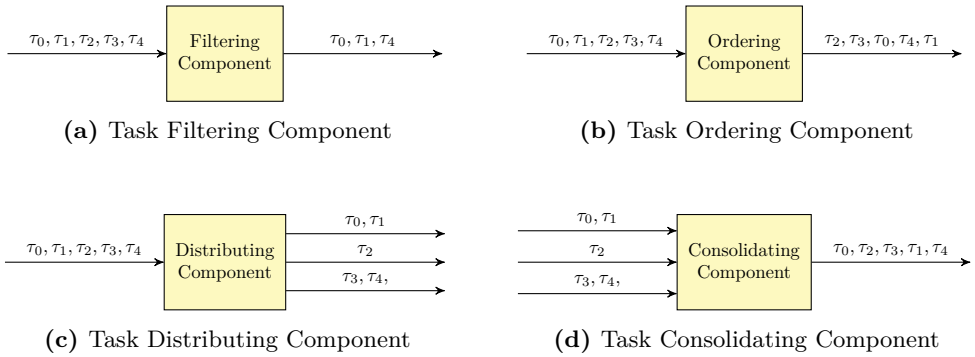
Components in CoBaS can be classified based on their functionality and the considerations of the section above. A Component does not necessarily only belong to one of those classes, but can also have properties of multiple classes at the same time.

Task Filtering Components

Task Filtering Components are enforcing absolute decisions as discussed above. They have one incoming Pipe and one outgoing Pipe. They do not change the order of the incoming tasks but remove tasks from the pipeline (Figure 5.20a). A Task Filtering Component can be integrated into every pipeline without restriction. However, it should be noted that the composition of multiple Task Filtering Components in one pipeline can lead to the removal of all tasks in the current scheduling epoch. The occurrence of such a situation is not caused by the composition itself but results from the fact that the resulting schedule is the one satisfying all specified optimization goals.

Task Ordering Components

Every Component that relies on defining a certain task order to create a solution participates in the class of *Task Ordering* Components. These Components are connected to one input pipe and one output pipe. They apply certain rules to the set of current tasks that result in a certain task order to be submitted to the outgoing pipe or pipes (Figure 5.20b). A task ordering pipe can, for example, apply the FCFS policy to a task set.



(a) Task Filtering Component

(b) Task Ordering Component

(c) Task Distributing Component

(d) Task Consolidating Component

Figure 5.20: Component classes in CoBaS.

Task Distributing Components

Components belonging to the class of *Task Distributing* Components make use of the spatial aspect of process scheduling. They split one incoming Pipe to several outgoing Pipes while maintaining the relative order between Tasks (Figure 5.20c). An example for a Task Distributing Component would be a load balancing implementation that distributes the tasks among PEs or a Component that sorts tasks according to their priority.

Task Consolidating Components

Task Consolidating Components have different functionality from Task Distributing Components. They merge several Pipes (Figure 5.20d). Task Consolidating Components do not only touch the spatial but also the temporal aspect of scheduling as a new task order has to be defined on the outgoing pipes.

Task Annotating Components

Components of the *task annotating* class are not necessarily connected to any Pipe. They store information about tasks that might be relevant to other Components or the rest of the system. The former information could be, for example, the priority of a task or the ISA it can be executed on, the latter information could be the number of task switches since the system start or the current system load.

5.9 Design Rationales

The two primary architectural decisions in the CoBaS architecture are the use of the framework and the component approach. After describing and discussing such an approach in the scope of this dissertation, this section gives more insights about the rationales to use these two methods and discusses them. In particular, they are compared to possible alternatives.

5.9.1 On Frameworks and Libraries

Frameworks as a design pattern first became prominent in the late 1980s [45, 84]. Today, they are widely used for a huge variety of purposes, like application software [S7, S26, S31], web applications [S1, S11, S33], or big data processing [S19], to name just a few.

Tahchiev et al. are defining a framework as follows:

> A framework is a semi-complete application. A framework provides a reusable, common structure to share among applications. Developers incorporate the framework into their application and extend it to meet their specific needs. Frameworks differ from toolkits by providing a coherent structure, rather than a simple set of utility classes.
>
> *Tahchiev et al. [156, p. 4]*

Another definition for frameworks, in general, is given by Gamma et al.:

> The framework dictates the architecture [...] It will define the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate [...] A framework predefines these design parameters so that you, the application designer/implementer, can concentrate on the specifics of your application. The framework captures the design decisions that are common to its application domain. Frameworks thus emphasize design reuse over code reuse, though a framework [...] you can put to work immediately.
>
> *Gamma et al. [64, pp. 26f.]*

By these definitions, it seems feasible that the framework approach is perfectly suited to the challenges addressed by this dissertation. The *application* in this context is the task or process scheduler of an arbitrary runtime system like an operating system. The user of the framework is the developer, implementing a scheduling policy for that runtime system. As it is feasible to apply the framework approach to the scheduling problem, the traits of the framework approach are beneficial to tackle the problem of process scheduling. The usage of a framework will allow a more structured approach to scheduler design and enable an easier reuse of existing scheduler designs in other runtime systems.

Further benefits of the framework approach that are desirable for the purpose of this dissertation are noted by Gamma et al.: “Not only can you build applications faster as a result, but the applications have similar structures. They are easier to maintain, and they seem more consistent to their users” [64, p. 27].

Even though the framework approach appears to be fitting, the alternative has to be considered; that would be a *library*. Libraries, in general, are collections of implementations for certain functionalities that can be used in user applications. A library offers a well-defined interface to fulfill the desired purpose. An example is the C standard library [128] that implements commonly used functions like string handling, mathematical functions, or standard in- and output. For the in- and output example, the C standard library acts as an adapter between the C programs and the operating system. This exemplifies the relation between Libraries and programming patterns. The Library implementation uses the programming pattern to solve the problem. However, an arbitrary implementation of the C standard library does not necessarily address the problem for every environment. For example, there are distinct implementations of the C standard library for the Windows [S6], Linux [S16], and BSD operating systems [S5] that are not interchangeable even though they implement the same functionality and use the same programming pattern regarding the access of C programs to operating system services.

As libraries are mostly a collection of implementations of common algorithms, they do not give the developer any kind of structure on how to solve a problem. Furthermore, they often are very fine grained regarding their functionality. For that reason, they are inferior for the purpose of this dissertation compared to frameworks. For developers, they would only offer algorithms for common problems but no facility to embed their work into. Moreover, it would be difficult to create an adaptive scheduler only based on libraries.

Another perspective regarding the difference between libraries and frameworks is the *inversion of control* [cf. 55]. With libraries, the programmer using the library is in control of the execution flow; the programmer’s code calls the library functions. With a framework, it is usually the other way around. The framework uses and calls the code provided by the programmer. Therefore, the framework is in control of the execution flow. Regarding the proposed CoBaS architecture, this characteristic is not an obvious reason for a decision. The CoBaS approach can be seen from two perspectives: the developer of new scheduling policies and the runtime system itself. From the viewpoint of the scheduler developer, the framework approach seems more suitable as the main goal is to determine the task order. From the perspective of the runtime system, the CoBaS framework itself can be considered as a library as it is invoked by the runtime system and implements a functionality needed by the runtime system. However, as the main focus of this dissertation is the creation of an architecture to create schedulers for future systems, the interest of the scheduling developer is also in focus.

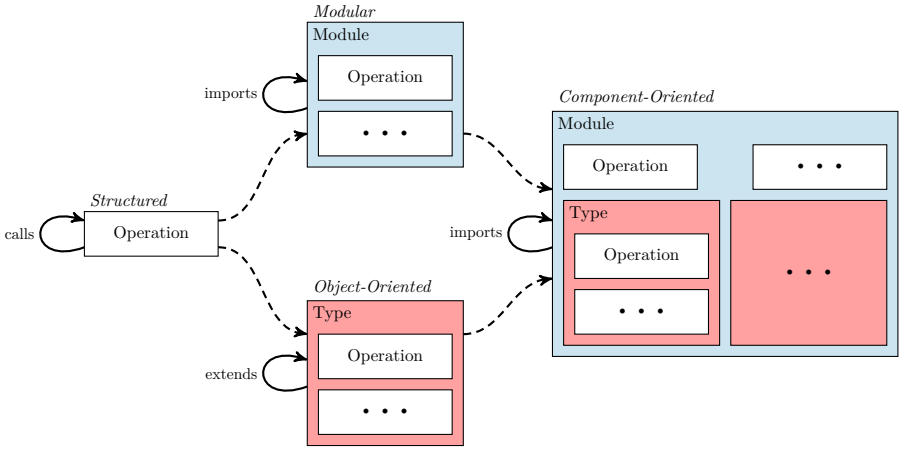


Figure 5.21: Relation between structured, modular, object-oriented, and component-oriented programming [60, p. 19].

5.9.2 On Components, Interfaces, Objects, and Modules

The CoBaS framework as described in this chapter uses a component based approach. The basic idea of Component-Oriented Programming (COP) dates back to McIlroy [105] with a discussion in 1968. He advocated a software design principle that is comparable to industrial mass production, where component producers create generic components that are sold to application developers who compose those pre-made components to full applications for the end-user market. According to Szyperski [153, p. 8], components can be considered the software equivalent to what Integrated Circuits (ICs) are in hardware. There is no general definition for a software component, however, Szyperski and Messerschmitt [154] are listing five properties for a software component:

- multiple-use
- non-context-specific
- composable with other components
- encapsulated (cannot be modified or, typically, even examined)
- a unit of independent deployment and versioning.

In the beginning, COP was considered an alternative approach to Object-Oriented Programming (OOP). An example of this effort is the Objective-C programming language, even though it is mostly considered object-oriented today. The modern view on COP is a higher-level programming paradigm that uses OOP as well as modular programming to construct composable components (cf. Figure 5.21).

The subsequent sections are discussing properties of further programming approaches and why they were discarded for the purpose of this dissertation.

Interface-Based Programming

The idea of interfaces in software development can be traced back to Dijkstra [46] and Parnas [120]. Before the introduction of the concept of OOP, interfaces were used to distinguish a module or software component description from the actual implementation. In many programming languages, this difference became explicit in a way that interface descriptions are placed in separate files with distinct file extensions differentiating between the module description and implementation e.g. `*.ads` and `*.adb` in Ada, `*.def` and `*.mod` in Modula-2, or `*.c` and `*.h` in C. Interfaces enabled reusability of existing implementation since it was unambiguous how to use a certain implementation.

The term *interface-based programming* was coined mainly by Pattison [123] and was thoroughly discussed by Steimann and Mayer [149]. Even though with the introduction of OOP the modularity aspect of interface-based programming was replaced by the class concept, interface-based programming did not become obsolete. On the contrary, it is heavily used in OOP languages and interfaces are even types on their own in languages like Java or C#.

Even though the *interface-based programming* would allow composition in general, it falls short for other aspects necessary for the purpose of this dissertation. For example, it makes no statement regarding the management of the functional implementation itself and therefore how to handle, e.g., different versions of an implementation or how to encapsulate functionality. The last aspects would make it hard to create reusable entities of scheduling policy implementations.

Object-Oriented Programming

OOP is the next evolutionary step in programming languages after procedural programming. Instead of handling data and procedures or processing instructions separately, OOP bundles them in *objects*. The structure of objects is described by *classes*. Therefore, *objects* are instances of *classes*. This structure significantly improves the modularity and reusability of the code. The concept of *inheritance*, which is very common in OOP, creates a hierarchy in the code and increases reusability even further.

The idea of OOP was introduced with the *SIMULA* programming language [74], even though first ideas of objects and object-orientation were already available in the *LISP* programming language [102, 103]. A general overview on design patterns in OOP is given by Gamma et al. [64]. The most popular programming languages today are object-oriented [169].

Even though intended for reusability and modularity, OOP is sometimes criticized for not reaching that goal [36]. Particularly in respect to operating systems, it is also argued that OOP programming is an inferior approach. Raymond [130, pp. 101–103] argues in the UNIX context that OOP has strong incentives for the programmer to introduce thick abstraction layers that destroy transparency. This claim can be backed by the fact that operating systems that offer an object-oriented interface to the userland

use a procedural programming language for the kernel. Examples are Windows NT derivatives [S41], OS X [S10], the Be operating system [S3]. However, several operating systems use an OOP language for the kernel like eCos [S12] or Haiku [S20], which succeeded the Be operating system.

For the purpose of this dissertation, OOP alone falls short to enable composition of different aspects of the scheduler. Furthermore, as OOP focuses on encapsulation of complete entities in classes, this approach is not optimal for the given problem as not all areas of the scheduling can be seen as such an entity.

Modular Programming

The concept of modular programming became prominent with the first, and only, national symposium on modular programming [18]. Today, the idea of modular programming is widely spread and available on most higher-level programming languages. Modular programming introduces the concept of *modules*[10] that offer a certain functionality. In contrast to a simple library, modules offer their service through an interface (cf. Section 5.9.2), therefore hiding details of the data processing inside the module.

Even though the modular approach comes close to fulfilling the requirements necessary for the goals of the dissertation, it lacks the notion of types that would significantly limit an actual implementation of the proposed framework.

[10] In some programming languages e.g. Go [S18], the term *package* is used instead of *module*.

CHAPTER 6

Qualitative Evaluation

The previous chapter described the CoBaS framework and gave an example how the different parts of the framework interact with each other. Furthermore, it discussed the component and framework approaches themselves. This chapter is dedicated to a qualitative evaluation of the features and resulting properties of the CoBaS architecture regarding the challenges researched by this dissertation. This discussion will demonstrate how the design decisions of the architecture support the development and prototyping for various scheduling policies for heterogeneous many-core systems. The initial qualitative considerations in this chapter will be completed by quantitative evaluation in the further course of this dissertation in Chapter 8.

6.1 Schedule Computation and Scalability

Even though the scheduler does not contribute to the progress of the computational tasks, it requires processing time to come to a scheduling decision. On a single-core system, it is evident where this computation is performed. However, on a multi- or even many-core system this decision is not so obvious. As a prototyping and research facility, the CoBaS framework supports as many of the suitable solutions as possible, which is discussed in the following sections.

6.1.1 Decentralized Scheduling

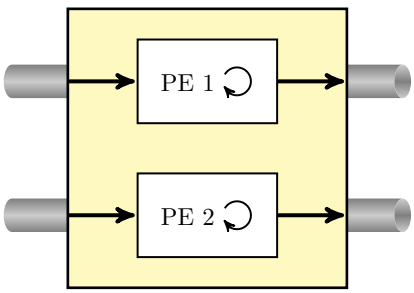
In decentralized scheduling, each PE manages its task set individually with a given policy. This allows a high scalability as every PE is mostly independent of the others and potential bottlenecks are avoided. Yet, load balancing has to be employed to distribute the tasks to the cores, which might result in wasted performance as recently shown by, e.g., Lozi et al. [98]. Such a scheduling scheme can be employed with CoBaS in two ways as described subsequently.

Multiple Pipes

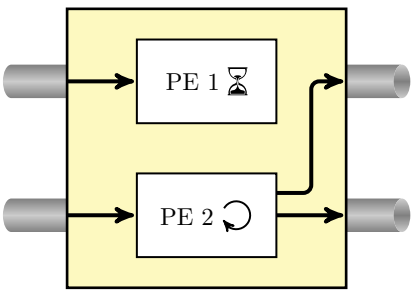
As a CoBaS Component is, in general, not limited by the number of incoming pipes, it can be designed in a way that it has as many incoming pipes as the system has PEs. These pipes can be accessed in parallel without blocking one of the PEs. Depending on the inner workings of the Component, several PEs can execute the code path of the Component without being blocked as well letting them pass the Component quickly (cf. Fig. 6.1a). For, e.g., the load balancing Component, that might be the usual case when there are no significant changes to the load and the re-admitted task gets assigned to its previous PE. However, especially during task migration between different output Pipes, this approach might result in contention (cf. Fig. 6.1b).

Postponed Pipe Updates

Instead of having multiple incoming queues, lock contention can be reduced by decreasing the locking time. This can be achieved by postponing the update of outgoing pipes. As the changes to a *Pipeline* become only relevant when a PE that is dependent on this pipeline requires the assignment of a new task, the update of that pipeline can be postponed. This reduces the time a pipe is locked and therefore decreases the risk of lock contention. It is also possible to combine both approaches – multiple *Pipes* and postponed Pipe updates – to even further reduce the possibility of lock contention.



(a) No contention inside the Component as both PE 1 and 2 can execute the Component code concurrently.



(b) Contention inside the Component during, e.g., task migration between the outgoing Pipes. PE 1 has to wait for PE 2 to finish accessing both the upper and lower outgoing Pipe.

Figure 6.1: Component with multiple incoming and outgoing Pipes. The PE 1 and 2 are executing the Component's code path at the same time.

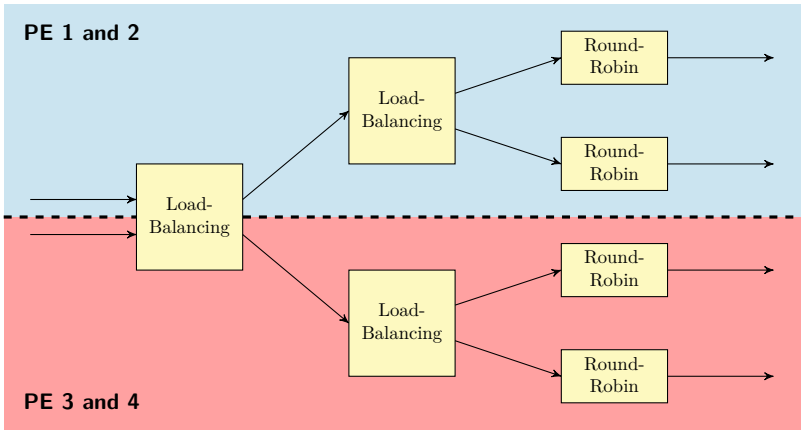


Figure 6.2: A hierarchical scheduler Topology. The upper part of the hierarchy is only accessed by PE 1 and 2, while the lower part is only accessed by PE 3 and 4.

6.1.2 Centralized Scheduling

In contrast to the decentralized scheduling, the whole task set is managed by every PE with the centralized scheduling approach. This method has the advantage that it is not prone to the risk of, e.g., inefficiencies in load balancing. However, it introduces a bottleneck as every PE runs through the same code in the same dataset resulting in possible contention with a rising number of cores on that data set. Still, for a small number of cores, this approach is superior to the decentralized scheduling approach [115].

This scheduling architecture is also supported by CoBaS. For a simple topology with only one Component that has one incoming and several outgoing *Pipes*, this is evident. Even the example given in Section 5.7 resembles a centralized scheduling approach, because in the example, outgoing *Pipelines* from the load balancing Component get updated the moment they are modified.

6.1.3 Hierarchical Scheduling

As both centralized and decentralized scheduling have their advantages and disadvantages, it might be beneficial to employ hierarchical scheduling approaches that combine these two. This scheme is also supported by CoBaS as illustrated in Fig. 6.2. The scheduler implementation can be partitioned in such a way that certain parts will be only used by a certain set of PEs. In the example, the upper part of the Topology is solely executed by PE 1 and 2 and the lower part solely executed by PE 3 and 4.

6.1.4 Foreign Scheduling

For general purpose PEs like, e.g., CPUs, it is common that the PE does not only execute the tasks, but also runs the scheduler logic. This is not possible for processing accelerators like, e.g., GPUs and might also be an inferior solution for many-core systems.

Processing accelerators are optimized for and restricted with regard to a certain kind of computation. This makes them unusable to enforce a scheduling policy most of the time. The schedule for this sort of PE is computed on a different, general purpose PE that is capable of executing the scheduler logic. Tasks are then deployed from that foreign PE to the special purpose PE.

In the case of many-core systems having hundreds or thousands of cores, this approach might also become feasible for general purpose PEs. In such a system, the computational power of one PE might become negligible compared to the whole system. Therefore, it might be possible to dedicate a single core just to the scheduler logic. This has several benefits. It reduces the overhead on the other cores as they do not have to be interrupted periodically.[11]

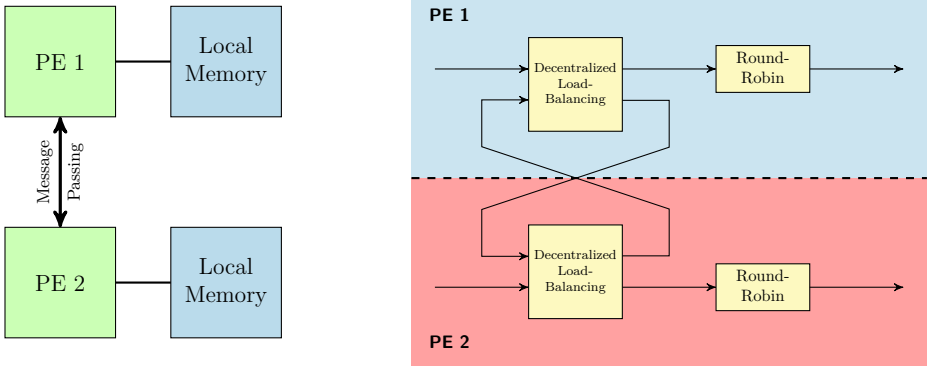
6.2 Non-Cache-Coherent or Distributed Memory

The challenges of none-cache-coherent systems and systems with distributed memory were discussed in Section 2.2. The CoBaS architecture can support such systems in several ways. The first and straight forward way is when the foreign scheduling scheme described in Section 6.1.4 is employed. As the whole scheduler logic would run on one distinct PE, the lack of cache coherency would not involve the CoBaS framework at all or only the *Runtime System Adapter*. However, as already discussed in Section 6.1.4, this approach is only feasible for systems within which the number of PEs exceeds the number of tasks or when the scheduler has to be invoked only rarely.

Also, a distributed scheduling for a non-cache-coherent or distributed memory system can be employed. Both the notification and Pipe system can easily be implemented on a message based semantic, making cache coherency and shared memory not mandatory. As notifications are messages by definition, it is evident that they can be transported through, e.g., a NoC. However, it is important to mention that the payload of the notification has to be accessible in the receiving core, i.e., pointers that are only valid on the remote core have to be avoided. As for *Pipes*, it is not obvious, but also feasible. All operations on the Pipes can be restricted to a certain set, which can then be used for a proxy pattern so that the main data can be held on the remote core.

Special care has also to be taken regarding the Topologies. As Component instances may have private data, single Component instances cannot be accessed by cores that do not have shared access to that data. Assume, e.g., a system every core of which has its

[11] A similar approach is already employed in current operating systems when only one task is assigned to a core [116].



(a) Example for a distributed memory system with two PEs.

(b) Topology for a distributed memory system with two PEs.

Figure 6.3: Example for a CoBaS based scheduler with distributed memory.

private memory and data sharing outside this scope is realized only through message passing. In that system, every Component instance has to be assigned to exactly one core. As a consequence, that would mean that the Load-Balancing Component used in the example in Section 5.7 would not be possible and that the whole Topology would need to be decentralized.

An example for coping with this challenge in CoBaS is outlined in Fig. 6.3. Assume a simple system consisting of two cores each with private memory and connection only through message passing (Figure 6.3a). To avoid data sharing in Components among the two cores, each Component instance is only accessed by one core. In the example, each core is assigned a *Decentralized Load-Balancing Component* that ensures the load-balancing among the two cores and a *Round-Robin Component* that enforces a round-robin scheduling policy on each core (Figure 6.3b). Over- or underload situations could be signaled by the load-balancing Components via a notification. In an overload situation, the load-balancing Component can submit tasks to the Pipe that is connected to the remote load-balancing Component causing the task to be dispatched on the remote core the next time it is scheduled. Note that the migration of the task context is not within the scope of the CoBaS framework.

6.3 Adaptability

It was already pointed out in Section 5.6.3 how the CoBaS architecture reacts to changes in the hardware configuration through adaptive Topologies. This section shows in more detail how this goal is accomplished by the properties and features of the CoBaS framework on a sophisticated example. Take for example a CPU that has an additional FPGA fabric on die like the Xilinx Zynq [180] or the upcoming Intel Broadwell EP Xeon

processors with Altera Arria 10 GX FPGA on package [69]. Assume further that the FPGA part of such a CPU is programmed with two additional cores that can execute different ISAs α and β to, e.g., execute proprietary legacy binaries that are not available for the ISA of the hard wired cores. Also, the system is running an operating system supporting several ISAs like, e.g., Popcorn Linux [17].

A possible scheduler layout for such a system with CoBaS is depicted in Fig. 6.4. The figure follows the same notion as in Section 5.7; however, not every task has its distinct color but the colors represent the ISA a task belongs to:

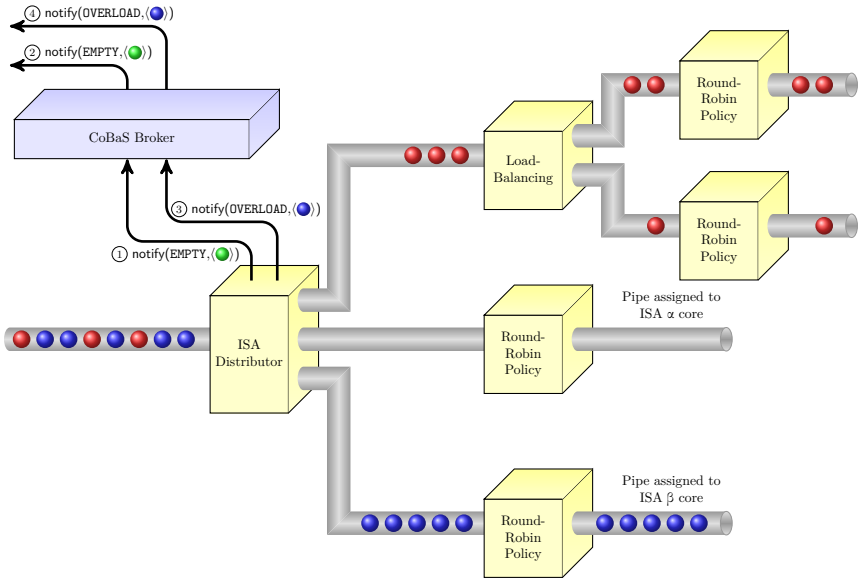
- Tasks that can only be execute on the native cores: 🔴.
- Tasks that can only be executed on an ISA α core: 🟢.
- Tasks that can only be executed on an ISA β core: 🔵.

In the current situation, the FPGA is configured with one core for ISA α and β each and the system has two native cores. The scheduler consists of an *ISA Distributor* that distributes the tasks according to their ISA: the native tasks in the upper output Pipe of the ISA Distributor, the ISA α tasks in the middle Pipe, and the ISA β tasks in the lower Pipe. The native tasks are balanced among the two native cores and scheduled in a round-robin manner, whereas the none-native tasks are also scheduled in a round-robin manner but need no load-balancing as there is only one core for each.

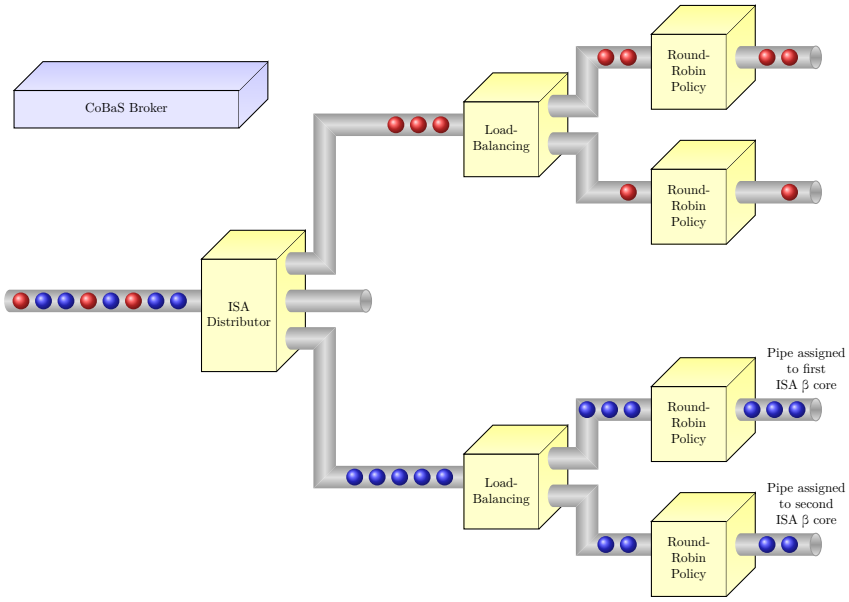
As there are no tasks for ISA α in the current situation, the respective core is underutilized. The underutilization causes the ISA Distributor to create a notification (① in Fig. 6.4a) that is forwarded to the Runtime System Adapter by the CoBaS Broker (② in Fig. 6.4a). Furthermore, the ISA Distributor detects an overload situation at the core for ISA β that causes it to generate a notification as well (③ in Fig. 6.4a) that is also forwarded to the Runtime System Adapter (④ in Fig. 6.4a).

Depending on how the FPGA is managed by the system, several possibilities exist to react to this situation. The FPGA could be managed by the runtime system itself, the Runtime System Adapter, or even by a Component. Regardless of this, we assume that the FPGA will be reconfigured based on the notifications of the ISA Distributor in a way that it no longer has one core for ISA α and β each but only two cores for ISA β. This change can be announced to an Adaptive Topology that will modify the scheduler layout as depicted in Fig. 6.4b. As for ISA β now also two cores are available, it will create a new instance of the same Load-Balancing Component as for the native cores and a second Round-Robin Component instance. It can also remove the Round-Robin instance responsible for the ISA α core as it is currently not needed.

This example shows how the characteristics of reusability and adaptability in CoBaS can realize a reaction to changes to the underlying hardware architecture. Note that this is only one example for handling the situation with CoBaS. Among other possible changes, the ISA Distribution Component instance could also have been replaced by one only differentiating between the currently available ISAs.



(a) An unbalanced system situation where the core for ISA α is underused and the core for ISA β is overloaded. The ISA distributor notifies the system about the two occurrences.



(b) Modified scheduler to counteract the unbalanced situation. Based on the notification of the ISA Distributor, the system was reconfigured and the scheduler layout was modified by a Dynamic or Adaptive Topology.

Figure 6.4: Scheduler modification caused by hardware changes.

6.4 Hardware Assisted Scheduling Acceleration

So far, this dissertation considered FPGAs only as a processing resource that has to be managed by the scheduler. However, due to its versatile nature, it can be used to accelerate the scheduler execution itself and several hardware schedulers or hardware accelerated schedulers have already been proposed [3, 31, 63, 71, 72]. With the broader introduction of FPGA technology into commodity hardware, it can be expected that this approach might become widespread. Therefore, the support for this kind of technology is desirable for the CoBaS architecture.

The Component-Based approach already has a functional separation and encapsulation of certain aspects of the scheduler implementation. With its well-defined interface, a CoBaS Component can be expected to be easily implemented in hardware with a very thin software layer. Take again, for example, the Zynq All Programmable System on Chip (SoC) [180] that combines two general purpose processing cores with a FPGA fabric and a shared memory between the cores and the FPGA. Using such an architecture, it is plausible that a hardware accelerated Component can be constructed. An example for such a Component is depicted in Fig. 6.5. The task list of the incoming Pipe is mapped to a memory region that is accessible by the FPGA. The hardware logic can check this memory for changes. Alternatively, the FPGA could also be triggered by a Pipe update through a thin software layer. The scheduling algorithm can then be performed in hardware by the FPGA. However, submitting the changes to the task set to the outgoing Pipe is not as straight forward as with a software based Component as the FPGA cannot execute the framework API. This means that the FPGA either needs to create the necessary data structures in memory by itself, which could be provided via a hardware macro, or it requires the assistance of a general purpose PE to update the outgoing pipe.

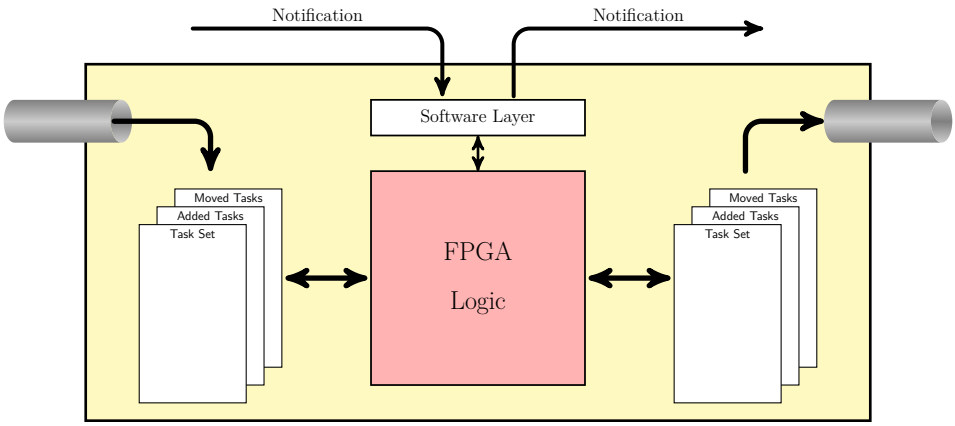


Figure 6.5: Hardware accelerated CoBaS Component.

Accessing the notification system of CoBaS needs more software support. The access has two aspects, outgoing notifications and requests and incoming notifications and replies. The incoming data flow can be handled by a thin software layer on its own. When the Broker delivers a notification or reply, it can invoke the software layer that, e.g., modifies the state of an incoming line of the hardware Component or a memory address that is monitored by the Component. The realization of the outgoing data flow is more complicated. As the Broker needs to be executed to distribute the events, it needs a general purpose PE to do so. To achieve this goal, the hardware Component could assemble the notification or request in memory and trigger an inter-processor interrupt on a general purpose PE that submits the notification or request to the Broker.

Having a Component constructed in Hardware has some similarities to implementing the Components in a programming language different to that of the framework itself. This problem is discussed in more detail during the quantitative evaluation, specifically in Section 8.9.

CHAPTER 7

CoBaS Implementation

Chapter 5 described and discussed the design and architecture of the CoBaS framework. Even though the approach is reasonable from an architectural point of view, it remains questionable whether it can be built that way and is suited for real-world scenarios. To analyze the properties of the CoBaS architecture, a prototype was implemented that is described in detail in this chapter.

Before the implementation of the CoBaS prototype could begin, a programming language had to be selected to implement it. Because the application domain of CoBaS extends widely to operating systems, the considered languages had to have solid support for bare metal programming. This requirement narrowed down the number of possible languages significantly and especially ruled out a significant number of the currently most popular languages like Java, C#, Python, or Go. For the implementation, the languages Ada, C, C++, and Rust were considered as they are most common languages under the remaining suitable languages.

The programming of the CoBaS prototype started in late 2012. At that point, the Rust programming language was still very young, and the definition of the language itself was still in a flow. It only became stable in mid-2015 [162]. Therefore, Rust was ruled out as the main programming language for the prototype even though a reevaluation today might come to a different assessment. The remaining programming languages were considered equally suitable to implement the prototype. However, as CoBaS is intended not only for the domain of operating systems, the integrability of other systems had the next highest priority when choosing the programming language. Regarding that, C and C++ are a better choice compared to Ada as most other programming languages have extensive support to incorporate legacy C code and with that also C++ code (e.g. [155, Annex B.2][S35, Sec. 5.9]). This left the choice between C and C++ only. As most operating system kernels today are implemented in C, the selection for CoBaS was also C over C++ as it promised to make it easier to integrate early stages of the prototype into the existing systems without the additional obstacles of differences in programming languages. Also, C promised to be a good starting point if a more mature framework should be migrated to a more modern language like Rust later on.

Based on this consideration regarding the programming language, the CoBaS prototype was implemented using C. The implementation can roughly be divided into the following parts that are discussed in the subsequent sections:

1. The framework itself that is independent of the underlying runtime system.
2. The runtime adapters that integrate the framework into the runtime system.
3. Topologies that describe a scheduling policy based on CoBaS components.
4. The management of Components in the CoBaS framework.

Moreover, a list of implemented Components for the prototype and their functionality is described in detail in Appendix A.

7.1 Framework Implementation

The framework implementation itself is the heart of the CoBaS prototype. However, discussing every detail of the framework implementation thoroughly would go far beyond the scope of this dissertation. Therefore, this section will focus on the two most important elements of CoBaS: The message broker and the pipe system.

7.1.1 Message Broker

The CoBaS Message Broker manages all topics and subscriptions as described in Section 5.4. As CoBaS is a dynamic system, topics can be defined not only at compile time, but also during runtime. To make CoBaS aware of a new topic, each topic has to be registered with the framework before it can be used. To manage the topics, the framework stores several properties of each topic including:

- A unique identifier to avoid collisions and, if needed, distinguish between versions.
- A simple identifier that serves as a name for the topic.
- Descriptive information about the topic in a human readable form.
- A registration counter that ensures that the topic is not removed from the system while it is still needed.
- The Component instance for the topic that response to it.
- The callback that is invoked for the topic.

Apart from this information, each topic has a concrete structure for the notification payload. However, this structure is opaque to the CoBaS framework and therefore neither announced to the framework nor managed by it.

Every topic is assigned both a simple and a unique identifier. For performance reasons, the topics are stored rather in an array than in a list. The simple identifier is used as the index for that list. As the array has a limited size that will in most use-cases be below 1000 entries, the simple identifier has the same limited value space. This makes the simple identifier unfeasible for a unique identification of the topic. Furthermore, it makes it also impossible to differentiate between different versions of notification payload structures that belong to this topic. The unique identifier allows the framework to run additional checks, whether the published event fits the currently registered topic or not and it prevents the use of the simple identifier for two different topics.

Listing 7.1: Example for a Topic definition in CoBaS.

```
1  #ifndef TOPIC_AFFINITY_H_
2  #define TOPIC_AFFINITY_H_
3
4  #include <fw_cpumask.h>
5
6  #define FW_TOPIC_AFFINITY    16                 /** Topic ID */
7  #define FW_TUUID_AFFINITY    468608183          /** Topic UUID */
8  #define FW_TNAME_AFFINITY    "AFFINITY"         /** Topic Name */
9  #define FW_TDESC_AFFINITY    "Task Affinities"  /** Topic Description */
10
11 typedef struct fw_task fw_task_t;
12
13 typedef struct fw_affinity_msg {
14     fw_task_t *task;        /**< The affected task. */
15     fw_cpumask_t mask;      /**< The tasks (new) affinity. */
16 } fw_affinity_msg_t;
17
18 #endif /* TOPIC_AFFINITY_H_ */
```

Once a topic is registered, Components can subscribe to that topic or register as a unique identifier (cf. Section 5.4). The framework creates a list of subscriptions for every registered topic. Each subscription includes a reference to the subscribing component instance and a reference to the callback function the instance used for notification of events of that topic. When a notification arrives for a particular topic, the framework iterates over the subscription list and invokes every callback that is assigned to that specific subscription. The responder functionality works the same way. However, only one component instance can register as a responder for each topic.

Again, because CoBaS is a dynamic system, topics cannot only be introduced to the framework, but also removed again at runtime. To assure that the system does not become inconsistent by deleting a topic that is still in use, the implementation tracks the components that need the topic and only removes it when no component is left that might use the topic.

An example for a topic definition is given in Listing 7.1. The topic is used to notify the framework about the affinity of a task or to retrieve it through the responder infrastructure. The simple identifier is given in Line 6 and the unique identifier in Line 7. Line 8 defines a human readable string representation of the topic and Line 9 gives a short description of the Topic. The Topic specific argument is defined in Lines 13 to 16. As explained above, this definition is only used by the Components that are related to the Topic and is opaque to the CoBaS framework. In this specific example, the Topic message contains a reference to a task (Line 14) and a CPU mask (Line 15). The shown message is used for both notifications and requests. In the case of a notification, the task field contains the task of which the CPU affinity of which is changed and the affinity field the new affinity. In the case of a request, the requesting instance will set the task reference to the task it wants to retrieve the affinity from. The request responder will set the affinity and send the message back to the requesting instance.

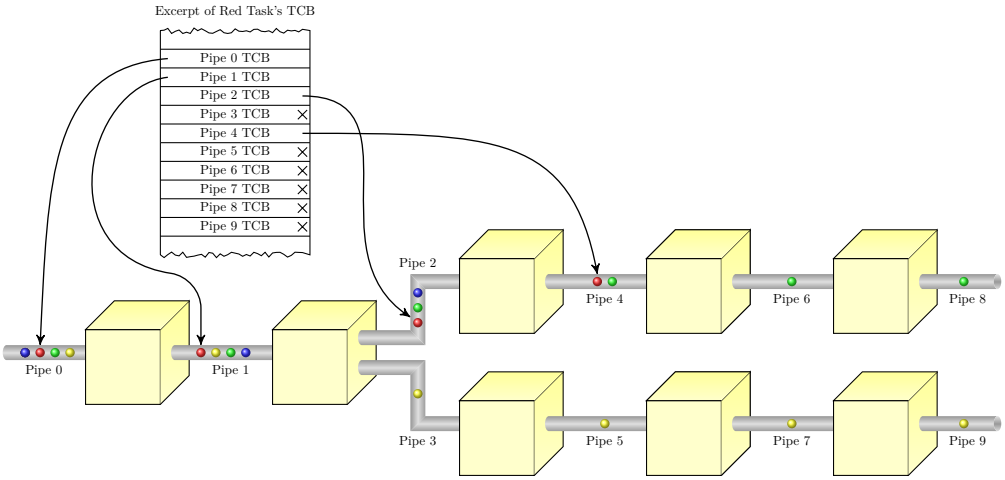


Figure 7.1: Example for referencing tasks in CoBaS pipes. Component instances are depicted as boxes and tasks as colored balls. Each pipe has a dedicated list hook in every task's TCB that is used when the task is referenced. For comprehensibility, only the hooks for the task list are shown. An excerpt of the red tasks' TCB, which is referenced by four pipes, is depicted as an example.

7.1.2 The Pipe System

The pipe concept was introduced in Section 5.3. The first issue that had to be solved is that every task can be referenced by multiple pipes. This matter arises mostly from the limited capabilities of the C language. To avoid the definition of a new list type and the corresponding list operations for every data type that might be stored in a list, CoBaS uses the same generic list approach as it is used, e.g., in the Linux kernel [cf. 28, pp. 87–89]. However, the genericness of the approach comes with the drawback that the linked data structure needs to have a hook for being linked into a list. This is not an issue if an element can only be part of one list at a time, but if it can be part of several lists at a time, it needs as many hooks as there are lists it might be linked into at the same time. For the pipe system, this means every task needs three hooks for every pipe in the system as it might be referenced by every Pipe. For that purpose, the CoBaS Thread Control Block (TCB) has list hooks for every possible pipe. It could be argued that this leads to a big memory consumption as every pipe causes an additional memory consumption of, e.g., 24 Byte in a 64-Bit system for the three list hooks. However, as the number of pipes scales with the size of the system regarding the number of cores, the system will also scale in memory to serve the cores. Therefore, memory consumption will be no issue at that point. Figure 7.1 shows how a task is referenced by several pipes using the hooks in its TCB. For simplification, only the hooks for the task lists are shown.

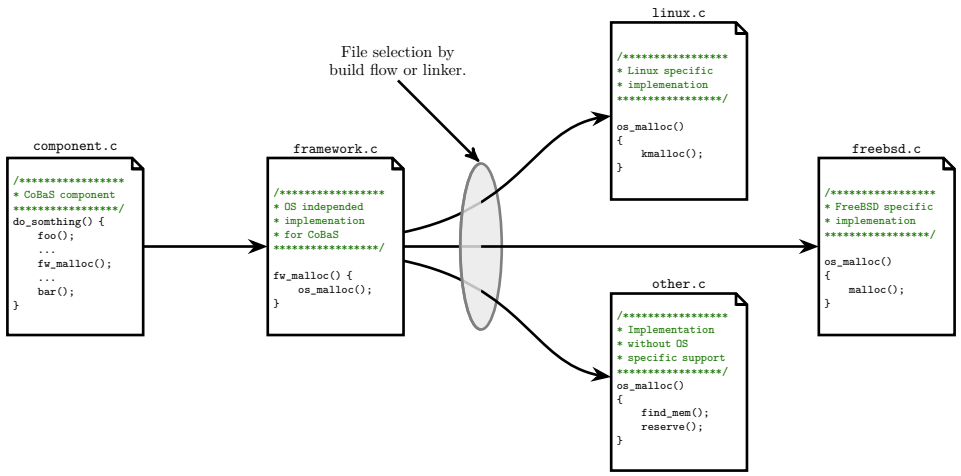


Figure 7.2: General function wrapping in CoBaS on the example of memory allocation. Function parameters and return values are neglected for better readability. This figure only illustrates the idea, the actual wrapper functions are more comprehensive.

Apart from the task sets that are relevant to the connected component instances, every pipe consists of further elements to ensure the functionality of the CoBaS framework. They hold references to the components they are connected to. This allows an easier management of the system, especially when component instances are dynamically added or removed.

7.2 Runtime System Adapters Implementation

The CoBaS framework itself is independent of the runtime system. However, to be usable in an actual runtime system, a *Runtime System Adapter*, as discussed in Section 5.5, is needed. This section discusses the technical details and challenges that occurred when implementing these Runtime System Adapters. The implementation details presented in this section were previously published in parts in Busse et al. [32].

7.2.1 Adapting CoBaS to the Runtime System

To adapt CoBaS to the runtime system, the concepts of wrapper functions and libraries are used, which are similar to the *adapter* and *façade* pattern in object-oriented programming – hence the name Runtime System Adapter. The idea of the pattern is outlined with an example for CoBaS in Fig. 7.2. All runtime independent functions of the CoBaS prototype are decorated with `fw_`, where all runtime dependent functions are decorated

Listing 7.2: Example of a function definition for runtime system dependent calls. The definition is only used when not superseded by a preprocessor macro defined for the target system. In addition, it serves as documentation for the function and its type signature.

```
/**
 * \brief Writes the C string pointed by format to the kernel log.
 *
 * Writes the C string pointed by format to the kernel log. If format includes
 * format specifiers (subsequences beginning with %), the additional arguments
 * following format are formatted and inserted in the resulting string replacing
 * their respective specifiers. Refer to ISO 9899 Section 7.21.6.3 for details.
 *
 * @param format C string that contains the text to be written to the kernel log
 * @param ... (additional arguments) Depending on the format string, the
              function may expect a sequence of additional arguments.
 * @return On success, the total number of characters written is returned.
 */
#ifndef os_printf
    int os_printf(const char* format, ...);
#endif
```

with `os_`. The decorators were also introduced to avoid the pollution of the namespaces as C does not support automatic function name mangling. The latter functions are not intended to be used by the CoBaS user, but only by the framework itself. The example shows how the develper accesses the `fw_malloc` function, which, in this example, is part of the framework library set. The `fw_malloc` then relies on the `os_malloc` function. The `os_malloc` function can and will most likely be different for every operating system. In the example, in, e.g., Linux, it would call the `kmalloc` function and in FreeBSD the `malloc` function respectively. It is not unlikely that certain functionality that is required by CoBaS is not given in the target runtime system. In those cases, the functionality has to be programmed for the target system. For example, in a runtime system that does not have a memory management feature, the functionality might be implemented in the `os_malloc` function itself. During compilation, only the source file fitting to the target runtime system is considered by the build flow, ensuring that only the suitable code paths are incorporated into the binary. The set of all runtime dependent functions represents the interface that needs to be satisfied to use CoBaS.

Some functions do not necessarily need an explicit wrapper. Take for example the printing of kernel messages. This function often has an identical signature as the standard C `printf` function [cf. 78, Sec. 7.21.6.3]. As a result, it is possible to simply link the runtime system dependent function symbol against the function of the runtime system. This linking can be achieved through preprocessor macros. Still, this approach is only feasible for some runtime systems and functions. For that reason, preprocessor constructs such as the one shown as an example in Listing 7.2 are used. It checks if the runtime system dependent function is already defined as preprocessor symbol and, therefore, another existing function to link against. If so, the native function will be linked as described. If not, it will define a function prototype, and a dependent function has to be available at the latest during the final linking. If this function was not

implemented for the target system, the linking will fail by intention as the functions are vital for CoBaS. The construct also serves as documentation. An ordinary preprocessor macro does not include types for the function signature, which would make it difficult to implement the Runtime System Adapter. The checking construct shows the type signature for every interface function and its purpose. This information is indispensable when adapting CoBaS to a new runtime environment.

A similar approach is used for some specific types. A few types have to be mapped to the native types, for example the type that describes the size of memory spaces. In some systems, this is not only specific to the runtime system, but even to the target architecture. Take for example the `size_t` type in Linux. On most 32 Bit architectures, it is defined as an `unsigned int`, while on most 64 Bit architectures it is defined as an `unsigned long`.[12] An exception to that is the SuperH architecture, where the `size_t` is defined as a `long unsigned int`.[13] To keep CoBaS both platform and architecture independent, the CoBaS `size_t` equivalent `fw_size_t` is wrapped to the runtime systems type. With Linux, the preprocessor first replaces every `fw_size_t` declaration with `size_t` and then subsequently with the target architecture specific type, e.g., `unsigned int`.

7.2.2 Adapting the Runtime System for CoBaS

Incorporating CoBaS into a runtime system has two facets: Either it is intended to be used for a whole new system or it is to be integrated into an existing system. Even though the general approach for integration does not differ much, the latter scenario potentially requires more implementation effort as the legacy system needs to be adapted. The reason is that an existing system might have several dependencies regarding the scheduler. For example, in the FreeBSD kernel, the locking of, e.g., drivers is partially realized through the run queue lock, which is not available when using CoBaS. Another example are task dependencies during the bootstrapping of the system. Further problems might occur when the functionality of CoBaS is, especially in an early phase of adaption, not as wide as the original scheduler. In this study, this has been particularly true for the Linux kernel, which has a very sophisticated scheduler.

Also in both scenarios, the task-related information that is typically held in the TCB might not all fall into the responsibility of CoBaS. Examples are information regarding the virtual memory mapping, file system handlers, or debug information unrelated to process scheduling. For that reason, the CoBaS prototype realizes the coexistence of the runtime system's TCB with the CoBaS internal TCB. In order to do that, the CoBaS TCB is linked into the runtime system TCB and vice versa as depicted in Fig. 7.3. The link to the runtime system's TCB is another example of a type that is mapped to the target runtime system's type as described in the previous section. Besides these issues, the only access interface of the runtime system to CoBaS is the publish–subscribe interface and a call to request a new task for a specific PE.

[12] cf. `include/uapi/asm-generic/posix_types.h` in [S24]
[13] cf. `arch/sh/include/uapi/asm/posix_types_64.h` in [S24]

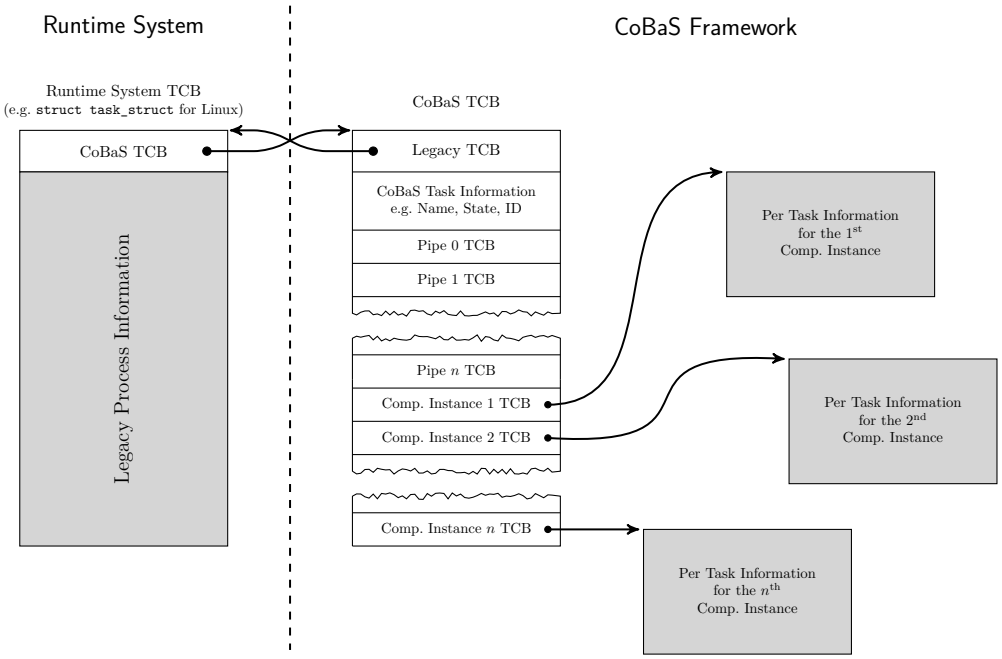


Figure 7.3: Implementation for the coexistence of the CoBaS and runtime system TCB. The CoBaS TCB holds a link to the runtime system TCB and vice versa. The TCB contains links to the Component instance specific TCBs. All grey parts of a data structure are opaque to the CoBaS prototype.

7.3 Topology Implementation

Section 5.6 described why topologies in CoBaS are necessary to obtain a working scheduling policy. Topologies can be divided into three categories: Static topologies, dynamic topologies, and adaptive topologies. This section will describe how the three kinds of topologies are realized in the CoBaS prototype.

7.3.1 Static Topologies

Static topologies are the initial topologies for every system using CoBaS. When a system is started, an initial topology is needed to bootstrap the system. Static topologies can be widely considered as equivalent to the state of the art of implementing scheduling policies. Several static topologies were implemented to test and evaluate the prototype. In the current version of the CoBaS prototype, static topologies are described by explicit programming. However, the definition process of the prototype is designed in a way that a DSL or GUI based definition can be easily developed.

7.3.2 Dynamic Topologies

With a dynamic Topology, the system can react to events in the system as, e.g., the arrival of a new PE as described in the example in Section 5.6.2 on Page 58. The implementation for that scenario can rely on the notification system of the CoBaS framework. The same way Components can subscribe to a Topic, a Topology can subscribe to it. For instance, the example from Section 5.6.2 can be implemented in a way that the Topology subscribes to an event that announces a new PE. When it receives the notification, it can reconfigure the Topology the way it is depicted in Fig. 5.13 on Page 58.

7.3.3 Adaptive Topologies

The current CoBaS prototype supports runtime system controlled adaptive Topologies in a way that it allows the replacement of Component instances through the user space. Two steps are necessary to achieve this. First, the old Component instance has to be removed and, second, the new instance has to be created. To do that in a consistent way, it has to be assured that no Pipe update or notification gets lost. The approach to achieve that for the Pipes is straight forward as every Pipe has a separate lock. Therefore, the Pipes that are concerned with the reconfiguration are locked and a Pipe update is triggered on each to ensure that no Pipe updates are pending. For the notification system, the approach is slightly more complicated. To ensure consistency, a readers–writer lock was employed in the notification system, where the normal operations act as readers and the reconfiguration process acts as a writer. This ensures that an arbitrary number of tasks can use the CoBaS Broker during normal operations, but only one reconfiguration is possible and the reconfiguration does not interfere with normal operations. It has to be noted that with this implementation, during the reconfiguration of the Topology, the whole notification system is stalled. However, this is not an issue as long as the reconfiguration happens not very frequently.

Once the system is in that safe state, the replacement process starts. Besides creating the new instance and removing the old instance, the Component specific parts of the TCB have to be considered. As the framework does not know what the TCB looks, the Component instances have to be involved. Before the old Component instance is removed from the system, the destructor for the Component specific TCB that may be defined by each Component is invoked for each task in the system. For the new Component instance, the same is done for its constructor of its Component specific TCB, if defined by by the new Component. Once these steps are completed, the replacement process is finalized by adding all tasks that are in the ingoing Pipes of the new instance as new tasks, so they are initially processed by the new Component instance and all locks are freed so that normal operations can continue.

7.4 Component Management

The three most interesting aspects of the component system of the prototype are the loading of a component during runtime, the initialization of Components, and the instantiation of component instances. The actual components that were implemented for the prototype are summarized in Appendix A.1.

7.4.1 Component Loading

The loading of components has two general scenarios: Either the component is already statically built into the system, or the component is loaded during runtime. This problem is similar to *loadable kernel modules*. Therefore, some techniques from that area were reused and adapted for CoBaS. The loading of components in the former scenario is trivial, as the components can simply be linked into the final binary. However, the latter scenario requires more effort.

To load a Component, it is first needed in compiled binary format. For the prototype, the Executable and Linking Format (ELF) [164, 170] was chosen as it is a format that is widely used. A framework function was implemented that takes such an ELF binary as an argument together with some meta information like, e.g., the size of the binary blob. To load the Component, memory is allocated in an executable memory region and the executable code is copied from the ELF file into that memory region. However, the code is not executable yet, as the symbols in the code have to be resolved. In order to do so, the addresses of the symbols have to be known to the CoBaS framework. As the framework offers only a limited set of functions to components, the mapping is built into the framework itself. Once the symbols are resolved in the binary code, the component can be initialized and is available to be used in a Topology.

7.4.2 Component Initialization

The initialization of components is necessary so that the framework knows about the existence of and ways to handle the respective components. Contrary to the loading, the initialization of components is trivial for components loaded at runtime as it is evident that the loaded component also has to be initialized. The more challenging part is the initialization of built-in components. As the built-in components are only linked into the final binary, the rest of the CoBaS functions does not necessarily have knowledge of their existence. To avoid changes to the CoBaS framework itself every time a component is added or removed from the linking phase, another technique from the area of *loadable kernel modules* was adapted.

The CoBaS prototype offers a preprocessor macro that takes a function pointer as an argument. The components can specify their distinct initialization function through this macro. The preprocessor macro then creates a variable that holds the address of

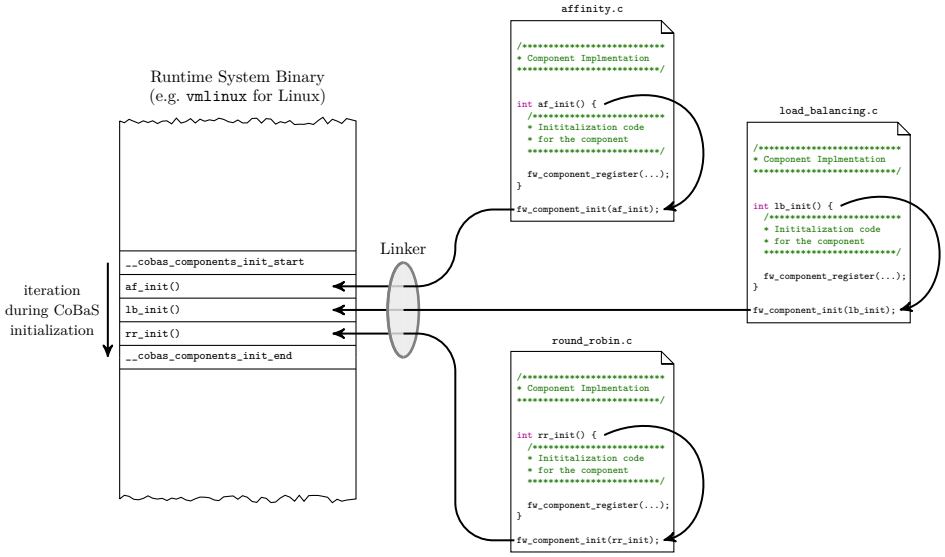


Figure 7.4: Pointers to the initialization functions of built-in components are assembled in an array by the linker during the final linking of the operating system's kernel. The initialization functions are marked by the developer as such.

the initialization function and assigns a special linker attribute to it. During the linking phase of the runtime binary, all those variables are subsequently placed in the data section of the kernel binary, and the begin and end address of this array are assigned to a variable. During the initialization of CoBaS itself, it can iterate over this area using the supplied variables and call every initialization function of every component compiled into the kernel. The whole procedure is summarized in Fig. 7.4.

7.4.3 Component Instantiation

During its initialization, every component announces a constructor function to the CoBaS framework, which can be used to create new instances. These functions take two arguments. An ID that is unique and will identify the instance in the future and a pointer to a data structure. As the framework itself does not necessarily need to know how the component works, the type of the pointer is opaque and can even be empty. However, ordinarily, the pointer will point to a structure that, e.g., includes the pipes that are connected to this component. The constructor of the component checks whether the supplied structure is valid and will initialize and return a pointer to a new instance of that component.

CHAPTER 8

Quantitative Evaluation

After the qualitative evaluation of the CoBaS approach in Chapter 6, the CoBaS prototype presented in the previous section also allows a quantitative evaluation that is presented in this chapter. The goal of the quantitative evaluation is to further substantiate the claims of this dissertation stated in the introduction and refined by the requirements discussed in Chapter 3. For that purpose, several case studies and experiments have been performed. This chapter presents them and substantiates, namely, the following claims in the subsequent sections:

- It is possible to build a process scheduling framework that is suitable for multiple runtime systems. (Section 8.1)
- The maintainability of the CoBaS framework itself is feasible for multiple runtime systems and versions. (Section 8.2)
- Different execution models can be handled by the CoBaS approach. (Section 8.3)
- The CoBaS framework scales for many-core systems. (Section 8.5)
- The overhead introduced by the CoBaS approach is not prohibitive. (Section 8.6)
- The CoBaS approach can be used in real world scenarios. (Section 8.7)
- An adaptable scheduler can improve the system performance. (Section 8.8)
- The component approach allows different programming schemes for the scheduling policy implementation. (Section 8.9)

Aside from the CoBaS related studies and experiments, Section 8.4 introduces a new tracing technique that was developed in order to be able to perform the detailed tracing necessary for the evaluation in Sections 8.5 through 8.9.

8.1 Runtime System Independence

To demonstrate and measure the degree of runtime system independence, CoBaS was integrated as the main process scheduling facility into the Linux and FreeBSD kernel. Several considerations are the foundation for this decision. First, only free and publicly available operating systems were considered. This ensures that the findings of this study can be widely published and reproduced. Second, the operating systems should be relevant to productive use, which shows that CoBaS is feasible for real world systems.

Third, if possible, the operating systems should cover different application domains. Finally, the selected operating systems should differ with regard to the complexity of their scheduler subsystem.

The Linux kernel is by far the most prominent and widely used open source kernel. Its applicability reaches from small embedded systems, over desktops, to large supercomputing environments. Its code base is freely available, and it has over 3500 active developers (cf. Table 1.1 on Page 9).

Even though the FreeBSD kernel is used in far fewer systems and has a much smaller base of active developers, it is still one of the most important open source kernels apart from Linux. This stems mostly from the fact that the licensing of the FreeBSD [cf. 160] kernel is not as restrictive as the licensing of the Linux kernel [cf. 148]. Because of this, the FreeBSD kernel is a popular choice for closed systems like the Playstation 4 [146]. Apart from closed systems, FreeBSD is a favorable choice for storage solutions because it has the best support for ZFS[14] after the proprietary implementation of the Solaris operating system.

8.1.1 Using CoBaS as Process Scheduler for an Operating System

Integrating CoBaS into an existing operating system can in most cases be considered more challenging than using CoBaS for a new operating system.[15] The reason is that the scheduler is often highly integrated into the rest of the system. This is even true for microkernel operating systems, as even there the scheduler is part of the kernel and not placed in the user space as a separate entity.

The approach for integrating CoBaS into an existing operating system can be generalized and was applied to both the Linux and FreeBSD kernels: First, the current scheduler subsystem was thoroughly analyzed regarding its structure and integration into the kernel. This is necessary as the runtime system might have several assumptions concerning the properties of the scheduler implementation that are neither documented nor explicitly stated. Then, the existing scheduler logic was completely removed, leaving only empty function bodies as stubs for functions that are called from outside the scheduler. The same applies to global variables and structures that are used outside the scheduler subsystem.

In an operating system, particular attention has to be paid to the locking. Unlike in user space, the *Coffman conditions* [39] can hold regarding the CPU and, e.g., a lock in kernel space. In user space, only the conditions mutual exclusion, hold and wait, and circular wait hold regarding the CPU as the operating system can preempt every user process through a hardware interrupt, i.e., by a timer. In kernel space, the condition *no preemption* can also hold since the operating system cannot preempt itself

[14] ZFS is a filesystem designed for high performance, reliability, and capacity [118, pp. 24ff.].
[15] Exceptions are operating systems like, e.g., eCos that are designed to support a custom scheduler.

Table 8.1: Size of the Runtime System Adapter for the FreeBSD kernel in lines of code.

Language	Files	Blank	Comment	Code
C	3	192	592	670
C Header	2	50	86	169
$\sum$	5	242	678	839

in certain circumstances. For example, assume that the operating system executes code in a system call triggered from the userland; then, the execution jumps to another part of the code to handle an interrupt. If both contexts try to use the same resource, e.g., a lock, a deadlock situation occurs. The interrupt handler will never give up the CPU as it attempts to acquire the lock, whereas the other kernel code cannot release the lock as it does not assigned the CPU to progress further. The issue is discussed in more detail by Russell [137].

8.1.2 Integrating CoBaS into the FreeBSD Kernel

FreeBSD uses the ULE scheduler [134] as its main process scheduler even tough the traditional BSD scheduler [cf. 101, pp. 16f.] is also still available. The scheduler can be selected during compile time of the kernel. The ULE scheduler implementation consists of approximately 2900 lines of code. The implementation of the ULE scheduler was stripped down to use the CoBaS scheduler framework as main process scheduling facility. The size of the resulting Runtime System Adapter is summarized in Table 8.1. Besides the Runtime System Adapter, some minor changes had to be included in the kernel tree, namely seven lines of code in the linker script and another 50 lines of code in the kernel Makefile to include the CoBaS sources during the build process.

It has to be noted that the support for dynamic Component loading and replacement as described in Section 7.4 is ongoing work for the Runtime System Adapter of the FreeBSD kernel, hence not yet completed and considered in Table 8.1.

8.1.3 Integrating CoBaS into the Linux Kernel

Contrary to FreeBSD, the default Linux process scheduler supports several scheduling policies as already described in Section 4.3.1. Furthermore, it offers much more features than the FreeBSD scheduler, including, i.e., control groups[16] or NUMA aware process scheduling. Because of this, the code base of the Linux scheduler is much bigger than the FreeBSD scheduler. The implementation of the Linux scheduler is split into several

[16] Control groups is a functionality that allows the assignment of resources and the limitation of the use of resources, e.g., the CPU for particular groups of processes [cf. 107].

Table 8.2: Size of the Runtime System Adapter for the Linux kernel in lines of code.

Language	Files	Blank	Comment	Code
C	5	448	1123	1547
C Header	3	155	324	323
make	1	31	12	92
Assembly	1	0	4	8
$\sum$	10	634	1463	1970

files roughly consisting of the `kernel/sched/core.c` file, which contains the basic infrastructure for process scheduling and several files implementing the actual scheduling policies. To replace the vanilla scheduler with CoBaS, the `kernel/sched/core.c` with approximately 8600 lines of code was stripped down and used as a basis for the wrapper logic. The source files that are implementing the actual scheduling policies were dropped and completely ignored for the CoBaS integration. The size of the resulting Runtime System Adapter is summarized in Table 8.2. In addition to theses files, some minor changes were applied to the kernel tree to be able to select either the vanilla scheduler or the CoBaS framework as main scheduling facility.

8.1.4 Discussion

The case study presented in this section shows that it is possible to create a scheduler framework suitable for multiple runtime systems. Furthermore, it gave insights on the necessary steps and the complexity of creating a Runtime System Adapter for a specific runtime system and integrating CoBaS into that runtime system. It can also be assessed that the effort for creating a new Runtime System Adapter strongly depends on the complexity of the runtime system or, to be more precise, on its scheduler subsystem. The Runtime System Adapter for the FreeBSD kernel is much smaller and less complex compared to the one for the Linux kernel. This resembles the complexity of the individual schedulers available for both systems.

8.2 Maintainability

Creating a new feature for an operating system automatically raises the question of maintainability. In general, internal kernel APIs are not necessarily stable between individual revisions, making it often necessary to adapt features to new kernel versions. Even though CoBaS was designed to improve the maintainability of policy implementations, the framework itself has to be maintained to be compatible with new versions of the intended target operating system. For this dissertation, the maintainability of CoBaS was researched using the exemplary implementations for Linux and FreeBSD.

Table 8.3: Number of changes in the Linux kernel's Runtime System Adapter due to new kernel versions.[16]

Linux Kernel Version Delta	Changes		
	Files	Insertions	Deletions
v3.9 → v3.10	1	52	1
v3.10 → v3.11	0	0	0
v3.11 → v3.12	0	0	0
v3.12 → v3.13	2	21	263
v3.13 → v3.14	2	30	4
v3.14 → v3.15	1	0	10
v3.15 → v3.16	2	6	2
v3.16 → v3.17	1	6	0
v3.17 → v3.18	2	33	0
v3.18 → v3.19	2	5	4
v3.19 → v4.0	1	54	24
v4.0 → v4.1	1	1	1
v4.1 → v4.2	2	57	10
v4.2 → v4.3	1	1	1
v4.3 → v4.4	3	245	49

Table 8.4: Number of changes in the FreeBSD kernel's Runtime System Adapter due to new kernel versions.[16]

FreeBSD Kernel Version Delta	Changes		
	Files	Insertions	Deletions
v9.2 → v9.3	2	24	24
v9.3 → v10.0	1	9	9
v10.0 → v10.1	1	9	9
v10.1 → v10.2	0	0	0
v10.2 → v10.3	1	1	0

The development of the CoBaS framework started in 2013 when the Linux kernel v3.9 and FreeBSD kernel v9.2 were the most recent versions. As the study of CoBaS stretched over three years, new and improved kernel versions became available. At the time of writing, this meant v4.4 for Linux as the latest long term support version released on January 10th 2016 and for FreeBSD v10.2 published on March 28th 2016. This made it possible to research the maintainability of CoBaS for 15 kernel revisions for Linux and four kernel revisions for FreeBSD. The summarized code statistics for both adaptations are presented in Tables 8.3 and 8.4 respectively. The Linux example shows that the

[16]The statistics were obtained from the CoBaS source code repository using *git* with the `'git diff --stat [old] [new]'` command.

changes are limited to a few lines of code. Only the moves from v3.12 to v3.13 and from v4.3 to 4.4 are standing out with more than 100 changes. Looking closer at those changes shows that the adaptation for v3.13 made it possible to remove several lines of code from the framework as they were also moved to a different part of the vanilla kernel, making them not necessary anymore in the frameworks code base. The move from v4.3 to v4.4 was more challenging, though, as some scheduler logic was restructured. This is also reflected in the amount of time it took to adapt to the new versions. The stepwise adaptation from v3.9 to v4.3 took approximately 10 working hours in total, whereas the further adaptation for the kernel v4.4 alone took another 30 woking hours.

For FreeBSD, the changes on the Runtime System Adapter are even more marginal. Looking into detail shows that they are almost not existing. Contrary to the Linux kernel, the FreeBSD kernel source of v10.0 is not a direct ascendent of v9.3 as they were developed in parallel. Because of that, v9.3 already included changes not present in v10.0, which made it necessary to revert changes to the Runtime System Adapter. This reversion accounts exclusively for the nine insertions and deletions reflected in Table 8.4. As the changes from v9.3 were later on applied to v10.1, the modification of the Runtime System Adapter had to be introduced yet another time, resulting, again, in the nine insertions and deletions. Migrating the CoBaS framework from FreeBSD v9.2 to v10.3 including the intermediate versions took less than two working hours.

It has to be noted that the dynamic Component loading and replacement as described in Section 7.4 was not yet implemented during the adaptation for newer kernel releases. Still, even with additional work possibly stemming from these code paths, it is safe to say that maintaining CoBaS over several kernel releases is both feasible and possible.

8.3 System Heterogeneity

As previously discussed, current and future systems most likely have to rely on heterogeneity to reach the desired processing performance. To enable the operating system to manage all PEs, no matter how diverse they are, the scheduler subsystem has to be as generic as possible. The case study presented in this section demonstrates how the CoBaS framework handles this heterogeneity. For the remainder of this section, the notion native or native PE will refer to the part of the system that boots it and is the main processing facility. The notion foreign or foreign PE will refer to every processing not conducted by the native part. Such a foreign PE could be, for example, a processing accelerator like a GPU.

In order to demonstrate that and how the CoBaS framework can handle tasks that are not native to the main system architecture, the Topology as depicted in Fig. 8.1 was created similar to the example discussed in Section 5.6.3 on Page 59 (Fig. 5.14). The Topology can be subdivided into two parts: one for the native part of the system and one to support the foreign part. The elements of the Topology for the native part employ a simple decentral multi-core scheduling. It consists of an instance of the *Load Balancing* Component and several *Head Queue* Component instances, one for each native PE.

Furthermore, the Topology has an instance of the *Affinity* Component that stores the affinity of every task for each PE. It is used by the *Load Balancing* Component to determine which outgoing Pipes are suitable for each task. All of the used components are described in more detail in Appendix A.1.

Concerning the foreign part of the system, the Topology includes an *ISA Demux* Component. The *ISA Demux* Component examines the ISA requirements of every ingoing task and assigns it to an outgoing Pipe fitting this requirement. The ISA requirement itself is stored by an instance of the *ISA Tagging* Component. The tasks for the foreign PE are scheduled in a first come first served manner by an instance of the *FCFS* Component. As illustrated by Fig. 8.1, the Topology also includes an instance of a *TCB Entry* Component (cf. Appendix A.1). For this case study, it held the thread context for the foreign PE. Note that it would have also been possible to implement this data structure in the Runtime System Adapter; however, to show examples of the capabilities of the component approach, the realization through a CoBaS Component was chosen.

The foreign PE element was emulated through a Linux kernel module executing the algorithm in Listing 8.1. The kernel module requests a task from the CoBaS framework for the foreign PE (Line 2). If a task is available, the framework will return it, the module informs the framework about the dispatching (Line 4), acquires the task context from the *TCB Entry* Component (Line 5), and starts processing it (Line 6). If no task is available, the module sleeps for one second before retrying (Line 16). After finishing a process run, the module checks if the task is finished. If not, it is resubmitted to the framework (Line 8). If the task is finished, the module checks if it was a standalone task. If this is the case, the task gets completely removed from the system (Line 10), if it is not the case, the ISA requirement is reset (Line 12) and the task is resubmitted to the framework (Line 13). To emulate the processing in a processing accelerator, the kernel module simply uses a busy wait loop. Therefore, the task context is also unpretentious.

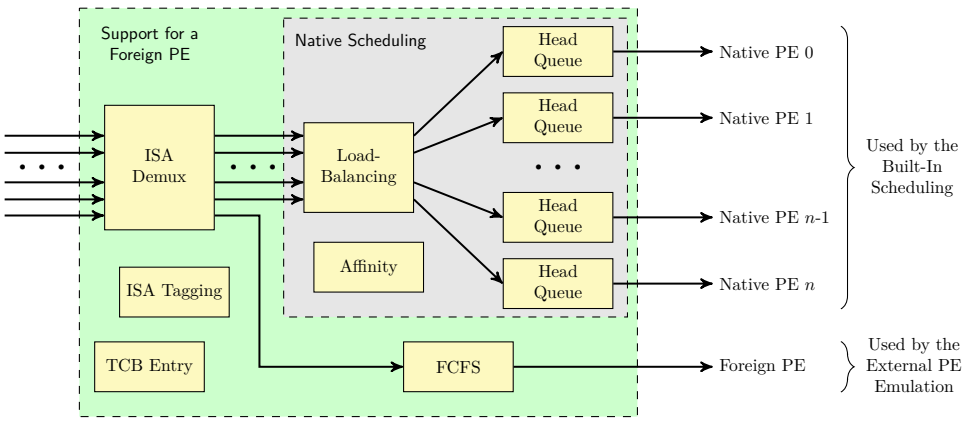


Figure 8.1: Topology supporting a foreign PE besides the native ones. The Topology supports an arbitrary number of native PEs.

Listing 8.1: Algorithm for the external PE emulation.

```
1: while keep processing data do
2:     task ← Schedule(Foreign PE)
3:     if task ≠ ∅ then
4:         Notify(DISPATCH, ⟨task⟩)
5:         context ← Request(TCB, ⟨task⟩)
6:         Process(context)
7:         if context not finished then
8:             Notify(RELINQUISH, ⟨task⟩)
9:         else if task is standalone then
10:            Notify(EXIT, ⟨task⟩)
11:        else
12:            Notify(ISA, ⟨task, native⟩)
13:            Notify(RELINQUISH, ⟨task⟩)
14:        end if
15:    else
16:        Sleep(1000 ms)
17:    end if
18: end while
```

It simply consists of the number of processing iterations, a relative value for the length of the busy wait loop, and the information whether the task is a standalone task or not; therefore, whether it has to be scheduled again on the real PE after finishing or not.

The functionality of this scheduling Topology was validated with two experiments. In the first experiment, several standalone tasks were created by the userland and submitted to the scheduler. The execution of those tasks was verified through the kernel log. In the second experiment, a task was created that assumed the need for the external PE at some point of the execution. Therefore, it changes its ISA requirement at a certain point of execution, yields from the current native PE, and gets processed by the external processing emulation. After that, the task was resubmitted to the native scheduling path and finished its execution.

This case study has shown an example that the CoBaS architecture can handle arbitrary tasks for non-native tasks. The emulated processing can, from a high-level point of view, easily be replaced by an actual processing accelerator. This case study forewent to explore an actual implementation as it is mainly an implementation challenge and promises no new insights from the architectural or scientific points of view.

In combination with the dynamicity aspects of the CoBaS architecture, the capabilities demonstrated in this section would also allow to deliver processing accelerators together with a scheduling implementation. Even though this is already happening today, those schedulers are either realized in user space or as a dedicated subsystem in the kernel. CoBaS would allow to integrate the specific scheduling policy into the main process scheduler subsystem. As a result, the operating system has, again, full control of all PEs and optimizations regarding the scheduling of multiple processing accelerators and PEs might be possible.

8.4 High Precision Kernel Tracing

The remaining case studies presented in this chapter require a detailed analysis of the runtime behavior of the CoBaS prototype. Such an analysis can be performed either in a simulated environment or a real runtime system like e.g. Linux. The former approach has the benefit that it allows a very detailed analysis and every aspect of the execution can be controlled. However, it has the drawback that the results are strongly depending on the model used to create the simulator. In particular, the runtime system simulation approach might fall short to analyze or show certain aspects as the model is not a complete representation of the real system. Furthermore, the results might have a bias also caused by an inaccurate modeling of a real runtime system. To avoid these issues, the experiments presented in the remainder of the chapter use the Linux kernel as runtime system.

Using a real runtime system instead of a simulated poses several other difficulties. The most noteworthy is the acquiring of the tracing data itself. The operating system is usually involved with tracing and profiling of executed code. A common approach when tracing the execution of the operating system is in-system tracing that uses different kinds of hooks in the operating system kernel. The most generic of these approaches is *ftrace*, which is embedded in the upstream kernel. It allows the tracing of arbitrary functions through the same technique used for live patching with *kpatch* and *kGraft* (see Section 4.2.5). When the tracing is active, at the beginning of traced functions, the execution flow is redirected to the *ftrace* infrastructure to log the execution and possibly perform further tracing steps. Other, more sophisticated, tracing frameworks like *perf* [S30], *extended Berkeley Packet Filter (eBPF)* [41], *SystemTap* [52, S39], or *LTTng* [44, S25] exist that allow an even more specific tracing and more extensive analysis. However, all in-system tracing facilities have an intrinsic drawback. They introduce an additional bias as they change the execution timing through their overhead (cf. Weaver [176, 177] and Section 8.5.3). This overhead becomes an issue especially with short functions. The tracing overhead becomes the dominant factor in the execution time, making the results useless for the analysis. Furthermore, the changes in execution time also become an issue for real-time sensitive executions. For example, the scheduler might rely on a periodic timer interrupt to determine the end of a time slice. When the execution of certain functions is prolonged by the tracing, the scheduling behavior might become completely different as the timing relations change. Another issue regarding in-system tracing arises from the amount of tracing data. When tracing the kernel in a stress situation, the trace can become huge in the order of several hundred megabytes per traced core and second. This data has to be stored. As the secondary storage, today, rarely exceeds a throughput of gigabytes per second, the data has to be stored in main memory or some tracing samples have to be dropped to not exceed the throughput of the secondary storage. Relying on main memory for the storage of the data limits the time span that can be traced. Finally, the main issue of in-system tracing, namely needing support by the operating system, cannot be entirely avoided. Therefore, for example, certain functions in the Linux kernel are not traceable by *ftrace*.

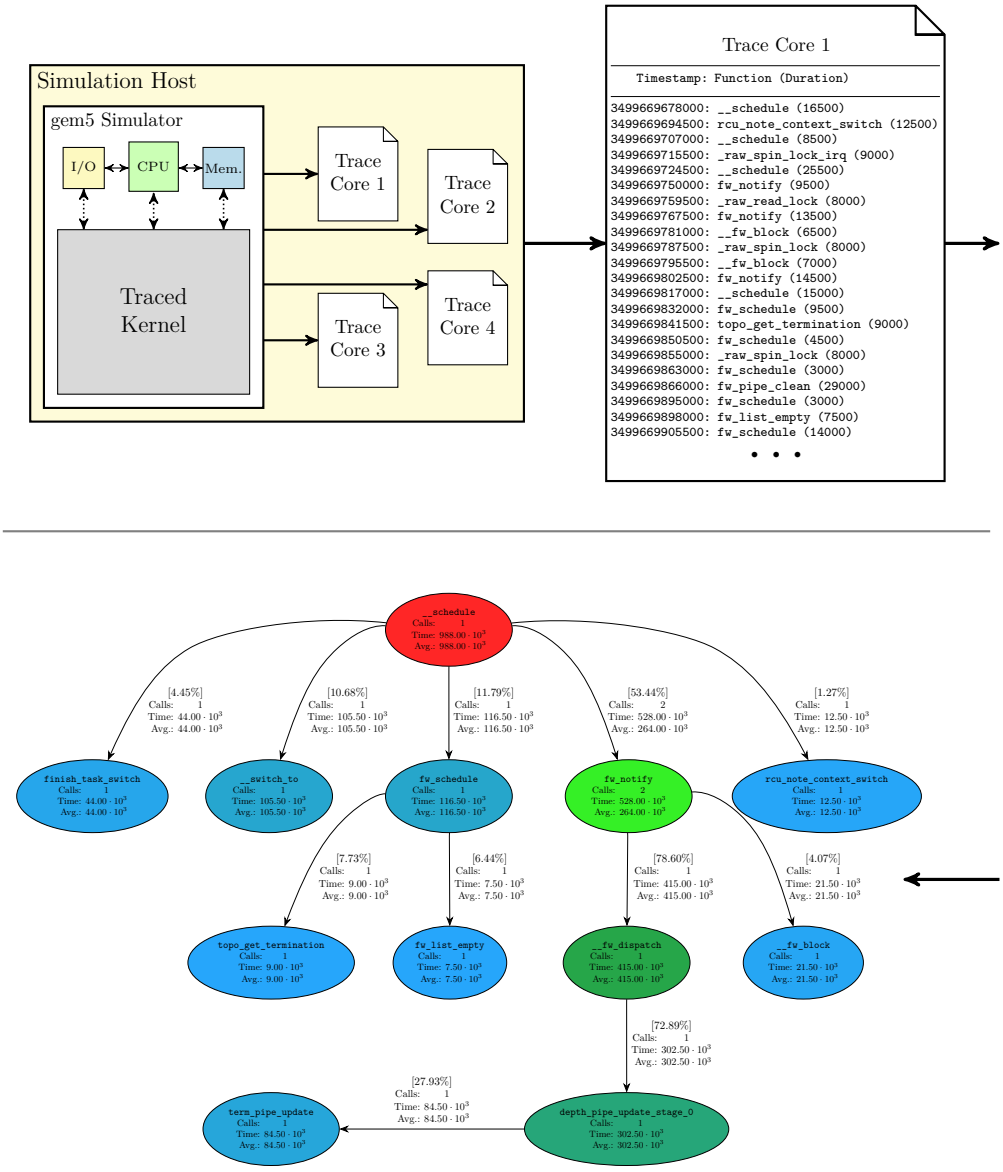


Figure 8.2: Process of the simulation-based kernel tracing. The tracing data is collected by the gem5 simulator per core. The simulation trace is transformed to a call-graph (right page) that can be simplified to reliably identify critical code paths.

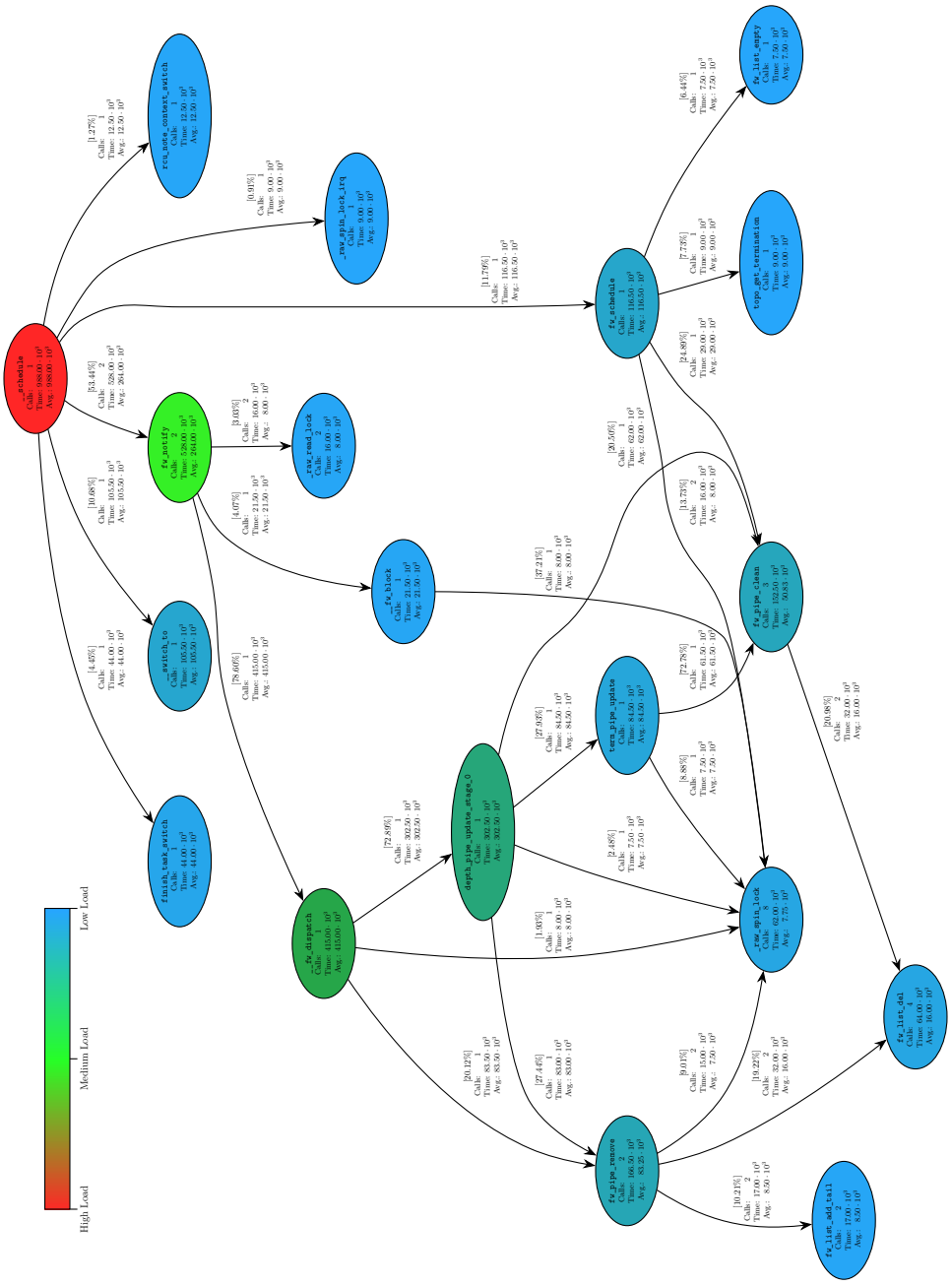


Figure 8.2 continued.

As theses issues would hinder a thorough evaluation of the CoBaS prototype, another approach was developed for this dissertation that combines the benefits of both in-system tracing and a runtime system simulation. This method uses a simulation of an entire system including the CPU, caches, memory, timers, and all other hardware features. The simulator executes the Linux kernel with the CoBaS prototype as main scheduling facility. With the support of the simulator, the execution of the operating system can be traced on an instruction level without interference. The resulting execution trace was then processed to recreate the call-graph. This step is necessary as the simulator has no knowledge about the call relation in the simulated machine since it lacks information about the location of the stack inside the simulated machine. It has to be emphasized that this approach needs a simulated machine and not a virtualized machine. Even though it is possible to track the execution flow in the virtual machine from the outside with, e.g., a debugging facility, this debugging would also influence the timing behavior as the virtualized timers for the virtual machine would still progress.

To implement this tracing and profiling, the *gem5* simulator [24] was used. Two main considerations lead to the use of *gem5*: First, *gem5* is a well-established tool in the research community regarding the low-level simulation of an entire system with extensive tracing and profiling support and, second, *gem5* is free and open-source allowing uncomplicated extension and reuse for further research and researchers. The *gem5* simulator permits the simulation of a full system, therefore, it cannot only provide a simulation environment for an operating system, but also arbitrary bare metal applications. This allows the approach, presented in this section, also to be applied to such applications. The *gem5* simulator allows, furthermore, the simulation of a significant number of cores. The number of cores is conceptually only limited by the simulated hardware. For example, the x86 simulation is limited to 254 cores as the simulated Advanced Programmable Interrupt Controller (APIC) provides only 254 interrupt lines for CPUs. Moreover, it is possible to integrate a SystemC based hardware model into the *gem5* simulator as presented by, e.g., Jung [87]. This may allow the analysis of the scheduler behavior for possibly emerging new hardware architectures.

The *gem5* simulator has the capability to trace the execution of code using debug symbols supplied at simulation start. When tracing is enabled, every time a new instruction is fetched, the simulator checks if the new instruction resides in the same function of the previous function. If this is not the case, it creates a new entry in the trace log. An example of the output is given in Fig. 8.2. Based on this call trace, the call-stack is reconstructed, deducing information about the call relation of the different functions. Once this information together with the timing simulation is collected, a call-graph can be generated as depicted in Fig. 8.2. The nodes of the graph represent the kernel functions and the edges function calls. Besides the number of calls for each function, the total time in simulation ticks to finish each function is shown. The total time incorporates both the time needed to execute the function itself and all functions called by it. Furthermore, the average time in simulation ticks to execute every function is given. Every edge shows the time the outgoing function spent in the target subgraph; therefore, the difference between the time of all outgoing edges and the time given for each function call is the time only spent in that function itself. The colors of the

function nodes indicate the amount of time spent in the function in relation to the total execution time of the functions in the top row of the graph. This makes it easier to identify critical code paths that are worth investigating for further optimization. Some functions in the kernel, especially helper functions, are called very frequently from various functions. As this can influence the comprehensibility of the graph, they are filtered. In the filtered example graph in Fig. 8.2, the functions starting with `fw_pipe_` and `_raw_` were filtered. The runtime of the filtered function is still part of the total runtime of the calling functions. Therefore, the critical code path stays recognizable.

The most significant drawback of the presented approach is the required simulation time. Depending on the number of cores and activity in the simulated system on a current simulation host, the simulation of one real time second can reach from minutes up to days. The situation is aggravated by the fact that the *gem5* simulator in its current implementation only allows a sequential simulation and, therefore, cannot profit from a multi-core system.

8.5 Scalability and Contention

The requirement of scalability was introduced in Section 3.2 and further elaborated in Section 6.1. The goal of the experiment presented in this section is to show that it is possible to build a scheduler with the CoBaS framework that can handle up to hundreds of cores and does not suffer from contention. This claim is proven by quantifying the impact of scalability and contention issues and that it is possible to eliminate them. The measurements are acquired with a topology that is configurable regarding its input pipes and the computation time of the scheduling algorithm. Configuring the topology with a single input Pipe would resemble a system that is completely centralized and, therefore, greatly suffers from contention. Configuring the topology with the same number of input Pipes as PEs resembles a system with minimal potential for contention. It has to be expected that, with a many-core system, the execution of several code paths of the scheduler framework will be slowed down due to the higher degree of contention, whereas this contention should significantly decrease with the fully scalable configuration.

8.5.1 Experimental Setup

The experiment described in this section uses the Topology depicted in Fig. 8.3. It consists of one *Load-Balancing* Component that has p ingoing Pipes, where p is the number of the PEs of the benchmarking machine. Furthermore, one instance of a *Burn* Component is connected to every ingoing Pipe of the Load-Balancing Component instance. The *Burn* Component instances are used to emulate the computational overhead of a real scheduling algorithm. They use the function presented in Listing 8.2 to burn CPU cycles through busy waiting that would, in a real, more sophisticated scheduling algorithm, be used to compute the task order. The number of waiting cycles for the Burn Components is adjustable through a notification. The Topology furthermore

Listing 8.2: The loop used to emulate a workload.

```
for(int c=0 ; c<work_cycles ; c++) {
    __asm__ __volatile__("");
}
```

includes an *Affinity* component that tracks the affinities of every task and is used by the load-balancing component to determine feasible PEs. Finally, the Topology also includes a *Marker* Component that has no functional purpose, but allows to place a mark in the function call-graph of the kernel. All of the used components are described in more detail in Appendix A.1.

The Topology used in this experiment is adaptive in a way that it allows to adjust the number of active ingoing Pipes. The adaptation is triggered by a notification that can set the number of utilized ingoing Pipes to an arbitrary number between 1 and n. That means if the number of ingoing Pipes is set to 1 in the one extreme, all PEs have to share the same Pipe for submitting tasks to the CoBaS framework, whereas, when in the other extreme the number of ingoing Pipes is set to n, every PE has a dedicated Pipe for task submission.

The experiments were conducted in two different environments: a simulated one and a real system. The simulation scenario used the *gem5* simulator with the tracing technique presented in Section 8.4. The experiments with the real system were conducted on the system summarized in Table 8.5. All experiments of this sections used the CoBaS implementation for the Linux kernel v4.4. The real system used a Gentoo userland. The simulated system used a fully automated, *Busybox* based userland that received the experimental parameters through the kernel command line.

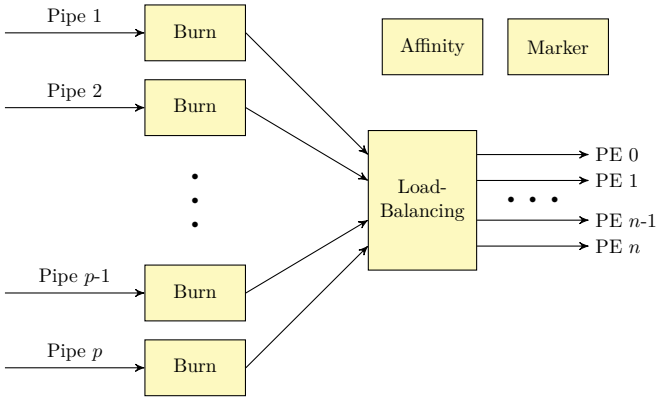


Figure 8.3: The Topology used for the scalability evaluation of the CoBaS framework. It can be adapted to use between 1 and p ingoing Pipes, where p has to be equal or smaller the number of PEs.

Table 8.5: System configuration used in the scalability experiments.

Architecture	x86_64
Microarchitecture	AMD Abu Dhabi
Model	AMD Opteron 6386 SE
Clock Rate	2800 MHz
Sockets	4
NUMA-Nodes	8
PEs	64
Memory	128 GB

8.5.2 Experiment Execution

Figure 8.4 depicts an exemplary call-graph generated from one of the experiments using the technique described in Section 8.4 as an orientation for further analysis. The functions `__schedule` and `activate_task` are part of the Runtime System Adapter for the Linux kernel. They are the main scheduling functions that are called by the kernel. The `activate_task` function is called when a task gets either unblocked or is woken up for the first time. The `__schedule` function is called by the Linux kernel when a new task has to be selected for a specific PE. This can be the case when either a task is blocked or a scheduler tick occurs and a task has to be relinquished. The other functions depicted in Fig. 8.4 are all part of the framework. Most of the functions are self-explaining by their name. The `fw_notify` function is the main interface of the CoBaS Broker as all notifications are submitted via this function. As the call-graph indicated, the notification system is also used by the CoBaS framework internally to trigger task state transitions. Internal functions are prefixed with `__fw`, while functions also available to Components are prefixed with `fw`. Functions that have another prefix are part of a Component, i.e., `burn_pipe_update` is the Pipe update function of the *Burn* Component and `lb_pipe_update` is the Pipe update function of the *Load-Balancing* Component.

For the real system, the function tracer *ftrace* of the Linux kernel was used to acquire the call-trace needed to create the call-graph. Even though *ftrace* is capable of collecting more information about the function calls as it has access to the call stack, it has other downsides compared to the *gem5* traces. The first and most noteworthy difference is that it cannot trace all functions in the Linux kernel as it is part of the kernel itself. This applies especially to the central Linux scheduler function that is excluded from *ftrace*. Therefore, it was not possible to track the whole scheduling code path, but still the relevant parts for this experiment that lie inside the framework. Second, as the measurements generate much information in a very short time, either the measurements have to fit into main memory or certain function call events have to be dropped. As the latter is not acceptable for the purpose of this experiment, the simulation time was very limited. The third disadvantage of *ftrace* is that, caused by the mechanism used for the tracing, every function call takes an additional amount of time as it needs to be

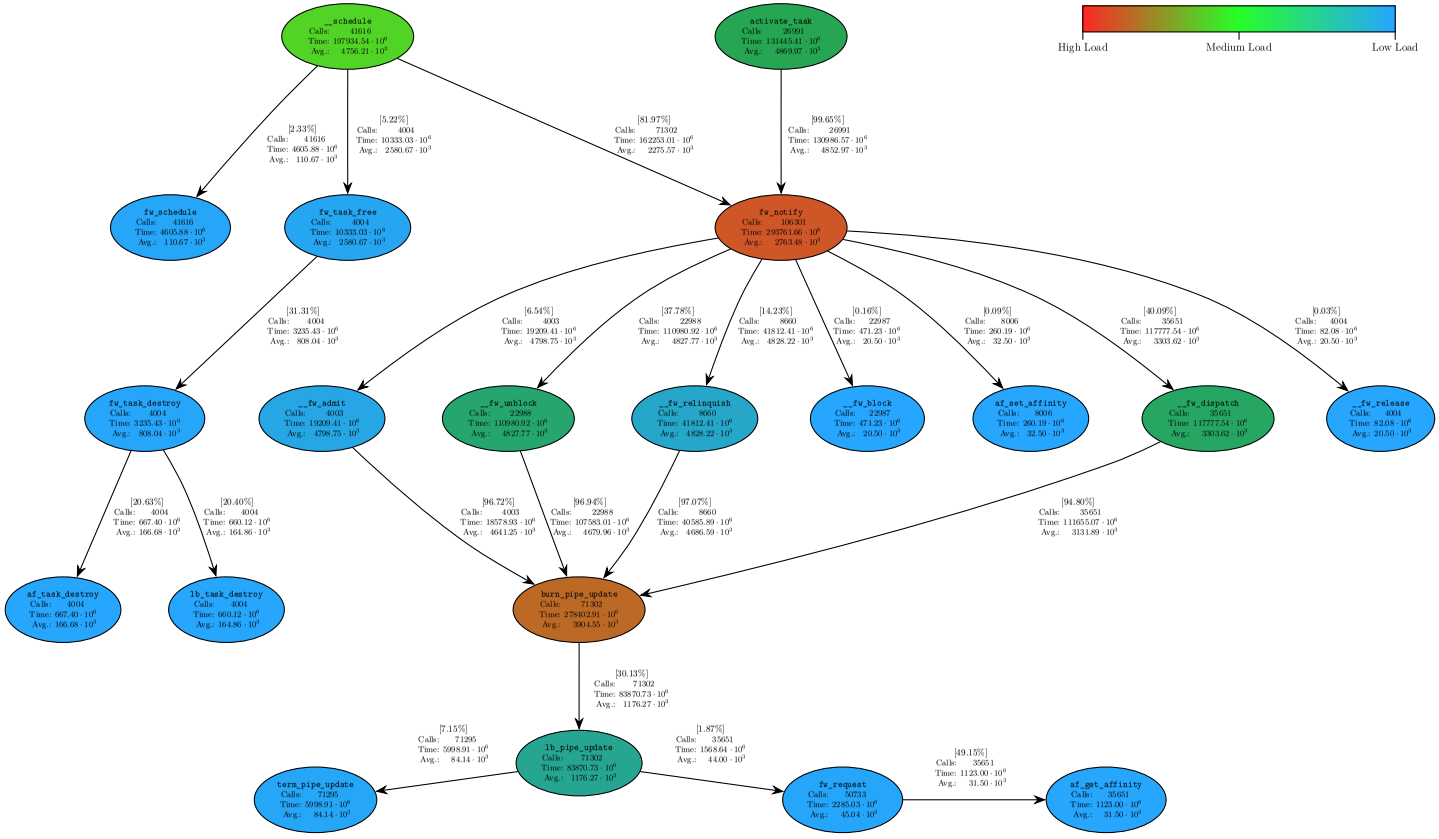


Figure 8.4: Exemplary call-graph of the scalability experiment. The call-graph was generated from the experiment with 16 PEs and the same number of scheduling Pipes.

recorded. This is particularly the case when looking at relatively short functions. This impact can be significant as the recording of the function call can take several times longer than the function execution itself.

With the *gem5* simulator, five different configurations were simulated: a 16 core, a 32 core, a 64 core, a 128 core, and a 254 core x86-64 machine. 254 Cores represent the current limit of the *gem5* simulator as the used APIC only supports 256 interrupt lines, two of which are already reserved. With each simulated machine size, the performance was evaluated for 0, 500, 1000, and 2000 loop iterations in the *Burn* components. Each loop iteration accounts for 2500 simulator ticks, five clock cycles of the simulated CPUs, or 2.5 ns real time. Therefore, the real time delay is 0, 1.25 µs, 2.5 µs, and 5 µs respectively. The same four experiments were conducted with the real machine and traced with *ftrace*.

To determine the scalability of the CoBaS framework, the scheduler subsystem needed to be stressed during the measurement. To stress the scheduler subsystem, the *hackbench* benchmark was used. Note that *hackbench* was only used to generate a high load on the scheduler subsystem and not for using it as benchmark measure. In both environments, the benchmark was run with 100 groups each with 20 senders and 20 receivers communicating via pipes 100 times with each other resulting in 4000 tasks in total. The call trace was marked via the *Marker* Component to indicate when the benchmark started. That allows only to consider the function trace of the time the benchmark ran and allows the analysis of the system in the high-stress situation without being biased by the low load situation during the system boot.

8.5.3 Experimental Results

The results of the experiments for 64 PEs are presented in Fig. 8.5 for the *gem5* experiments and in Fig. 8.6 for the real machine summarized in Table 8.5. The diagrams present the runtime both in terms of simulation ticks (left y-scale of each diagram) and the resulting real time (right y-scale of each diagram). For an improved readability, the results for 16, 32, 128, and 254 PEs are moved to Appendix B.1 depicted in Figs. B.1 to B.3 on Pages 150 to 152. Because of the high computational complexity of the simulation for the 254 PEs experiment, it was not possible to collect results for 2000 loop iterations. The diagrams show the average number of cycles required to process the respective functions. For each measurement, the average values are shown, as well as the confidence interval for a confidence level of 95 % in a Student's t-distribution. As most of the confidence intervals are very small and therefore hard to illustrate, the data collected during the experiments and used to generate the diagrams is summarized in Appendix B.1 as well in the Tables B.1 to B.16 on Pages 154 to 167.

The results show that, as expected, when several cores try to access the same Pipe, the system congests. In fact, the congestion is even so severe that once more than 32 PEs access one Pipe, it was not possible to gather meaningful measurements as most of the time was spent in the scheduler, resulting in almost no progress of the workload. Moreover, the results show that, in all functions depending on Pipe processing (`fw_dispatch`, `fw_relinquish`, and `fw_unblock`), the overhead drops signifi-

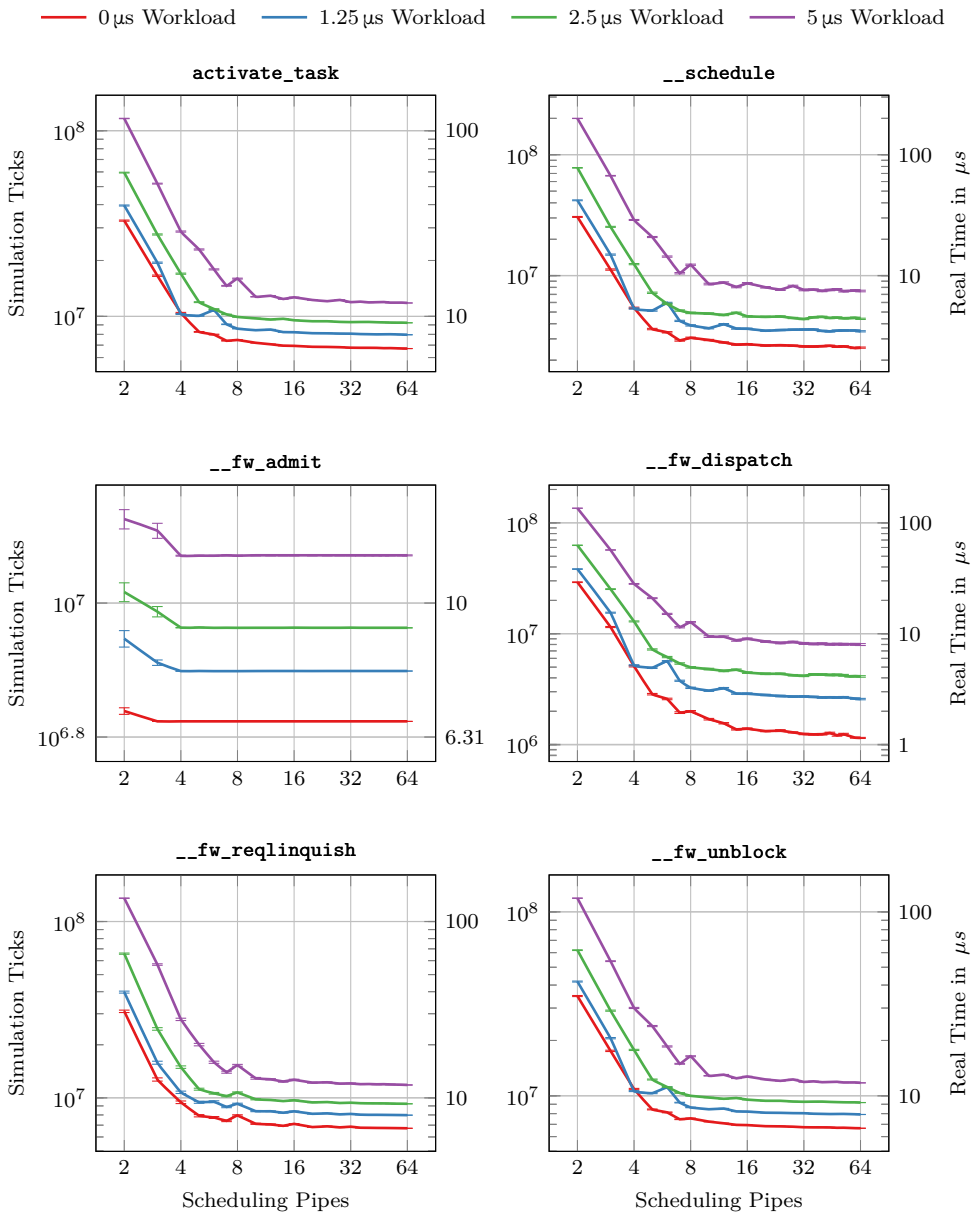


Figure 8.5: Results of the scalability experiments with 64 PEs using the *gem5* Simulator and four different workload emulations. The average runtime of each function is given in simulation ticks (left y-axis) and real time (right y-axis) as a subject to the number of scheduling Pipes.

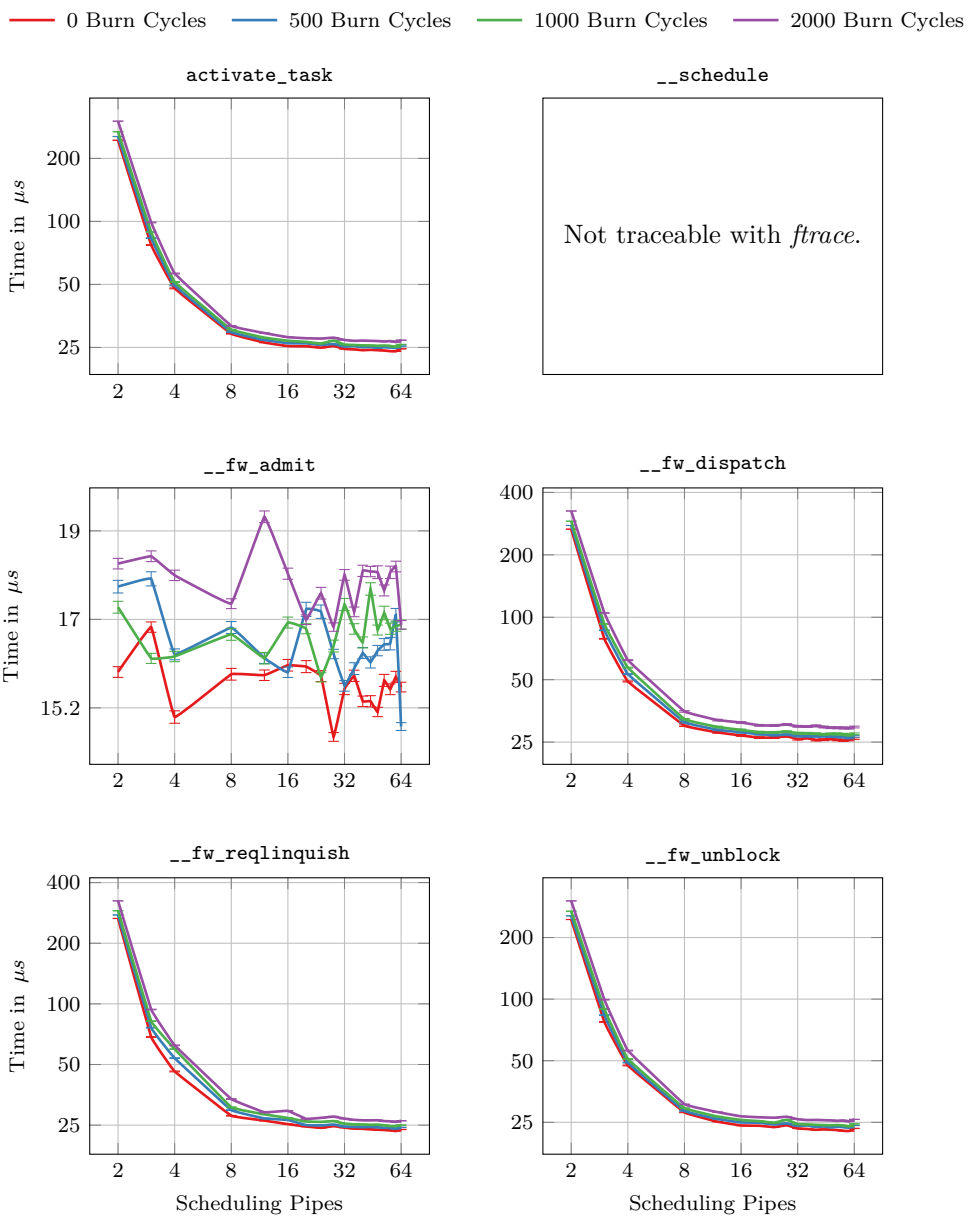


Figure 8.6: Results of the scalability experiments with 64 PEs using the real system as summarized in Table 8.5.

cantly because of the lock contention on the Pipes as soon as the number of Pipes increases. In consequence, the same applies to the Runtime System Adapter functions (`activate_task` and `__schedule`) that depend on those framework functions. The degree of contention also depends on the time that is required for each Pipe update. For example, with zero burn cycles, the contention already reaches a low level when less than 16 PEs have to share one Pipe, whereas with 2000 burn cycles this process starts below 8 PEs per Pipe.

At first sight, the results of the `fw_admit` function do not seem to fit the remaining results. Even though it also depends on the processing inside the Pipeline, it only shows indication of contention when 16 or more PEs are accessing the same Pipe in the *gem5* simulation, while in the real x86 system, the behavior seems to be promiscuous. In the *gem5* case, the results are explained by the inner workings of the *hackbench* benchmark. It forks all children in the very beginning. That means that at that point all other cores of the simulated machine are idle; there is no contention on the scheduler system. Therefore, the task admittance does not compete with other PEs that want to use the scheduler subsystem. The competition and therefore possible congestion starts only after some time, when the benchmark has already spawned most of its children. Besides the tasks spawned by *hackbench*, no significant number of other tasks is spawned that might influence the results. That this reasoning is sound is backed by the fact that not substantially more than 4000 task admittances are recorded in the task trace (cf. Tables B.1 to B.16 in Appendix B.1).

The situation with the results gathered by the *ftrace* tool is different and a result of the biggest drawback of the measurements collected by *ftrace*. Through the way it works, *ftrace* introduces an additional noise into the measurements as it takes extra time for every function to execute and finish. That can be observed when comparing the relative distance of the curves in the diagrams in Figs. 8.5 and 8.6 with each other. The distances for the *ftrace* results are much smaller than for the *gem5* results. This is caused by the *ftrace* overhead that makes the overhead posed by the Pipe functions less significant. Because of that, the results for the `fw_admit` function in Fig. 8.6 show more differences in the processing of the *ftrace* tracking than the differences caused by the CoBaS framework.

The graphs in all but the `fw_admit` functions are not completely monotonic. This can be explained by the fact that the runtime changes are based on lock contention and, furthermore, the lock contention is not completely deterministic. It depends on two or more PEs accessing the same mutual exclusive data or code path, which is a random process. As a result with a small but none zero probability, it is possible that the observed experiment is not close to the average and causes the spikes. However, as the simulator itself is completely deterministic, it will create the same conditions that lead to the results, therefore, repeating the experiment will not result in a different measurement. This reasoning is backed by the results gathered on the real system that does not suffer from that necessitarianism and does not show these severe spikes.

8.5.4 Discussion of the Results

The experiments of this section have shown that a scheduler that suffers from contention cannot be used in a system with dozens or hundreds of cores. With such a scheduler, the system is completely occupied by the scheduling routine and cannot perform any productive work. The situation greatly improves once it is possible for several PEs to execute the scheduling code concurrently. The results indicate that it is possible to build such a scheduler based on the CoBaS framework as the time spent in the scheduler drops more than linear once a scalable topology is used. The experiments of this section have shown that the CoBaS framework is capable of scaling for systems up to 254 cores and they give no indication that a scaling beyond that number of cores is not possible. Therefore, it can be concluded that CoBaS is suited for many-core systems and it is possible to implement scalable schedulers regarding the number of cores with it.

8.6 Composition Overhead

The CoBaS framework allows the construction of a sophisticated scheduling policy from several Components. However, having a processing of a complex algorithm distributed among several Components introduces an overhead as information or, to be more precise, the task orders have to be moved from one component to another. The goal of the experiment presented in this section is to quantify this overhead. The quantification is done by dividing an artificial computation that emulates a schedule computation among several Component instances. The resulting processing time is then compared to the processing time needed to perform the computation in a single Component instance.

8.6.1 Experiment Description and Setup

To determine the overhead of dividing a workload that emulates a scheduling algorithm between several Component instances, a Topology as depicted in Fig. 8.7 was created. To simulate the workload, an approach similar to the *Burn* Component from the previous section was used. However, to be able to distinguish the different stages

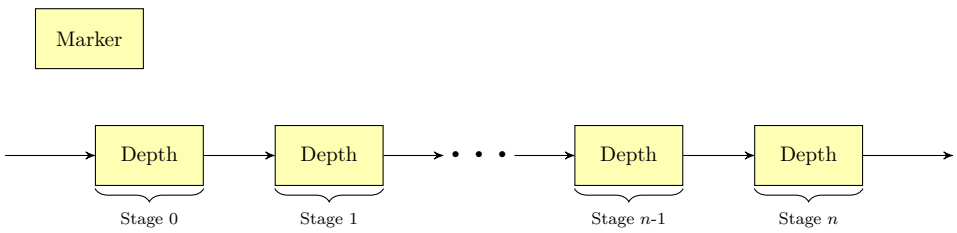


Figure 8.7: Topology used for the evaluation of the composition overhead.

of the Pipeline from one another, the *Burn* Component was extended in a way that it can be configured during instantiation according to the supposed name of the update function. This Component will be denoted as *Depth* Component. The experiment was conducted with the *gem5* simulator and *hackbench* as stress for the scheduler subsystem also similar to the previous section. Pipeline depths reaching from one to six Component instances were evaluated with various computation workloads for the emulated scheduling algorithm. The processing times of the `depth_pipe_update_stage_0` function were collected for all experiments as they do not only include the processing time of the `depth_pipe_update_stage_0` function itself, but also all subsequent component instances. The *Marker* Component was also used in this experiment to narrow the trace down to the time during the high-stress situation of the scheduler subsystem similar to the previous section.

As the differences in processing time were expected to be very small, the experiments were only conducted with the *gem5* simulator as it allows a very precise quantification of how much time was spent in which function. That would not have been the case with in-system tracing approaches with the introduction of a bias. The simulated system consisted of only one core as it is sufficient to determine the overhead. The experiments used the same *Busybox* based userland as in the previous section as well as the *hackbench* benchmark to stress the scheduler subsystem. However, only 25 groups were used, reducing the number of threads to 1000. This was sufficient to stress the scheduler of a single-core machine.

Again, similar to the previous section, an exemplary call-graph is given as an orientation in Fig. 8.8 generated from an experiment with four stages. The upper part of the call-graph represents the framework function as explain in the previous section. The lower part of the call-graph illustrates the chain of Pipe updates of the *Depth* Component instances with their update functions `depth_pipe_update_stage_0` through `depth_pipe_update_stage_4`. Each task is handed through the different Depth Component instances.

8.6.2 Experimental Results

The experimental results are summarized in Fig. 8.9. It shows the ratio between the processing time of the various Pipeline lengths and the processing time with only one Component instance in the Pipeline. The exact measurements are in the appendix in Table B.20 on Page 171. The results in the appendix also show the overhead for no workload, which poses the worst case scenario, as it cannot be illustrated in Fig. 8.9 with a logarithmic scale. Looking only at the Pipeline itself, without workload, the Pipeline with six instances would also take six times the amount of time to complete. However, as the Pipeline is not the only part involved in scheduling, this is not the case for the measurements; therefore, in the worst case scenario with zero work, the ratio is smaller. The experimental results also show that the additional processing time with more instances becomes more and more insignificant the more work has to be completed.

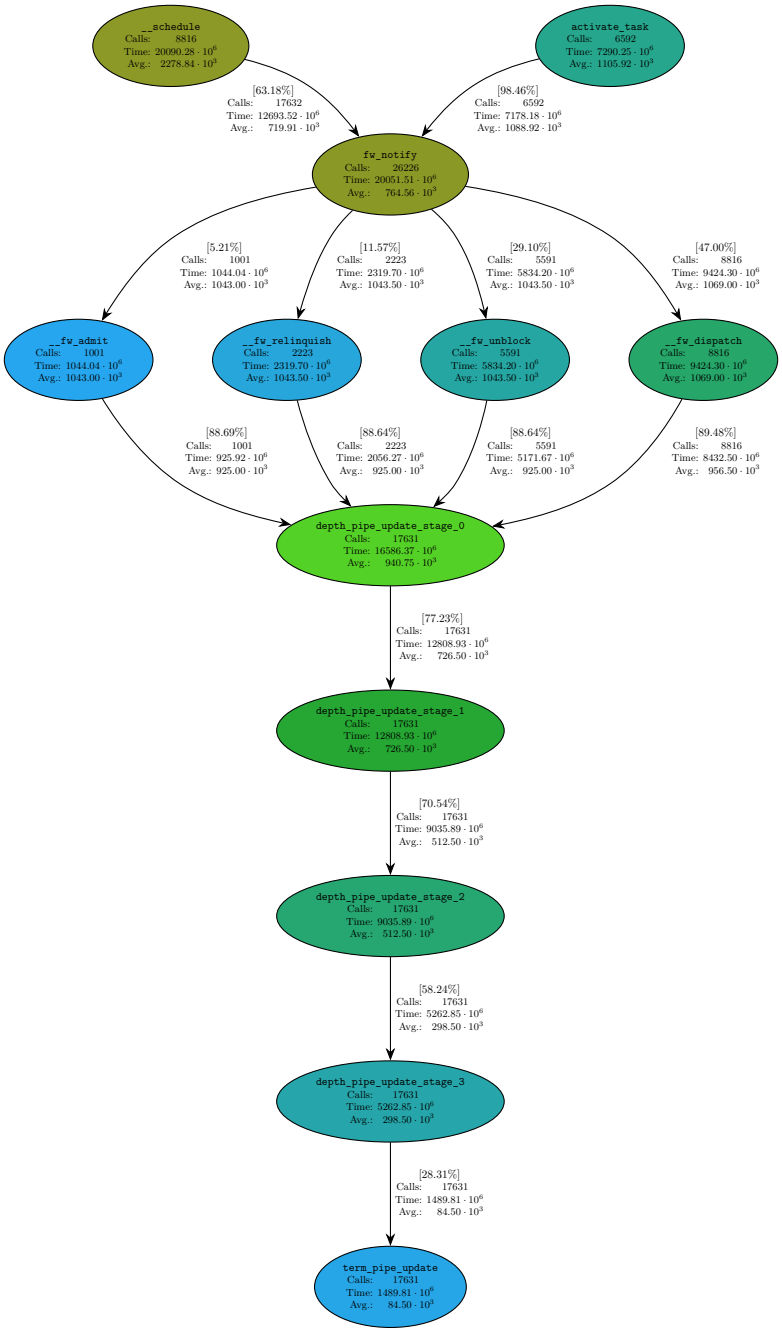


Figure 8.8: Exemplary call-graph of the overhead experiment with four stages.

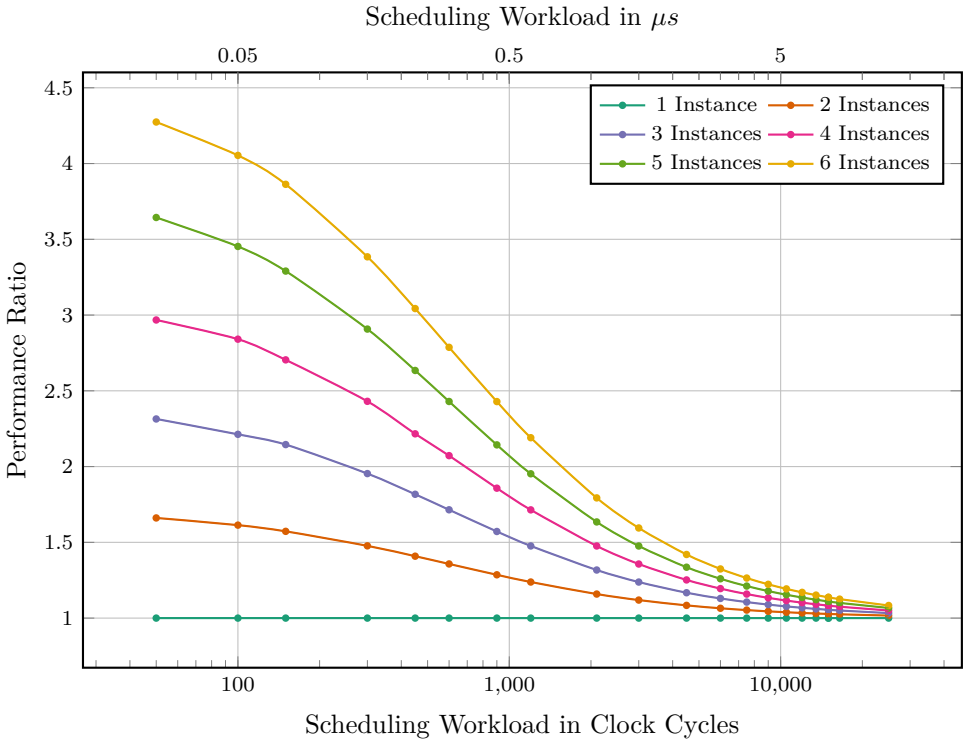


Figure 8.9: Overhead in Pipe processing by dividing the scheduler computation among several Component instances. The processing time of each Pipeline is put into relation to the Pipeline with only one Component instance.

However, even in the worst case, the scheduler overhead is rather small putting it into relation with the total runtime of the experiments. For example, the total runtime of the experiment with zero work and six Pipeline stages was approximate $1.222 \cdot 10^{12}$ simulation ticks. Taking the difference of the values from two processing stages in Fig. 8.9, the composition overhead is the following (all times in simulation ticks):

$$\text{total overhead} \approx \frac{20.62 \cdot 10^9 - 16.80 \cdot 10^9}{1.222 \cdot 10^{12} - (20.62 \cdot 10^9 - 16.80 \cdot 10^9)} \approx 1.66 \cdot 10^{-3} = 1.66\,‰$$

Measuring this small overhead was mainly possible through the simulation of the system. In a real system, the overhead would most likely be smaller than the measurement uncertainty.

8.6.3 Discussion of the Results

Looking at the overhead of dividing the task ordering among several Component instances shows two different pictures. On the one side, for the relative processing time in the scheduler itself, the partition can create a significant overhead. This is especially true when having a scheduling algorithm with a low computational complexity distributed over several components. However, subdividing the algorithm between several components also opens the opportunity of parallel execution that can lead to a reduced contention in multi- and many-core systems as shown in the previous section. Moreover, looking at the overall system, the overhead rather small as the scheduler contributes only in a small fraction to the overall system load. Finally, it should be noted that the used scenario poses a very high-stress situation for the scheduler that is probably very rare in real world scenarios. Therefore, the results of this evaluation can be treated as a worst-case approximation.

8.7 Performance Evaluation

While the previous experiments focused on the properties of the CoBaS framework itself, this section evaluates the impact of CoBaS on real world scenarios. As most benchmarks are sensitive to the actual employed scheduling policy, this section can only give an idea how the CoBaS approach will perform in actual productive systems. Still, it is possible to assess, whether it is feasible to use CoBaS in real world systems.

8.7.1 Experimental Setup

To evaluate the real world performance of CoBaS, a smaller machine compared to Section 8.5 was used. The machine used in Section 8.5 is a module based CPU design[17] with eight NUMA domains. Having only basic scheduling algorithms available for CoBaS today, it can be expected to result in a poor performance that does not reflect

Table 8.6: System configuration used in the performance evaluation.

Architecture:	x86_64
Microarchitecture:	Intel Sandy Bridge
Model:	Intel Core i7-2600
Clock Rate:	3400 MHz
Sockets:	1
NUMA-Nodes:	1
PEs:	8 (4 cores with SMT)
Memory:	8 GB

[17] Refer to Section 1.2.1 for issues regarding the scheduling on this particular design.

shortcomings in the CoBaS architecture but in the used scheduling policy. In order to reduce the impact of the scheduling policy, the machine summarized in Table 8.6 was used. Even though it has Simultaneous Multithreading (SMT), it seems feasible that a simple scheduling policy can manage the machine.

To benchmark the CoBaS framework, the *Numerical Aero Dynamic Simulation (NAS) parallel benchmarks suite* [14, 15, S27] and *hackbench* benchmark were used. The NAS benchmark suite consists of several micro-benchmarks resembling typical computational problems appearing in science (Table 8.7). The benchmarks are available in increasing problem sizes: *S*, *W*, and *A* through *E*. However, not every benchmark is available in the bigger sizes. As they are only intended for test purposes, the small *S* and *W* input sizes were not evaluated. Furthermore, because of the memory requirements of the larger problem sizes, only problem sizes up to *B* for FT and MG, and problem sizes up to *C* for the other benchmarks were measured. For the evaluation, an OpenMP implementation of the benchmark suite was used (cf. Jin et al. [83]). The benchmarks were executed with eight concurrent threads, matching the number of PEs of the machine.

The hackbench benchmark was already used in the previously presented experiments; however, it was only used to create stress on the scheduler subsystem. In this evaluation, it is used to benchmark the scheduler performance. As the benchmark outputs its runtime with an accuracy of only 10 ms, the number of loops was raised to 10 000 from the default of 100 to increase the benchmark's runtime. The number of communication groups varied between 1 and 256 with the number of groups being a power of two. This results in 40 up to 10 240 concurrent threads.

The CoBaS scheduler used the Topology depicted in Fig. 8.10. As comparison, the Completely Fair Scheduler (CFS) was benchmarked. A Gentoo userland was used with a vanilla Linux kernel v4.4 for the CFS benchmarks and the modified Linux kernel v4.4 with the CoBaS prototype implementation for the CoBaS measurements. The execution of each benchmark was repeated 50 times.

Table 8.7: Overview of the used NAS benchmarks.

Name	Description	Computational Challenge
BT	Block Tridiagonal	Floating point performance
CG	Conjugate Gradient	Irregular communication
EP	Embarrassingly Parallel	Floating point performance
FT	Fast Fourier Transform	Long-distance communication
IS	Integer Sort	Integer performance
LU	Lower-Upper symmetric Gauss-Seidel	Regular communication
MG	Multi Grid	Regular communication
SP	Scalar Pentadiagonal	Floating point performance
UA	Unstructured Adaptive	Irregular communication

8.7.2 Experimental Results

The results for the NAS benchmarks are summarized in Fig. 8.11 and the results for the hackbench benchmark in Fig. 8.12. The mean of the absolute runtimes for the benchmarks is summarized in Tables B.21 and B.22 in the appendix on Page 172 together with a 95 % confidence interval in a Student's t-distribution.

The results show a slight speedup of the CoBaS based scheduler compared to the CFS for the LU benchmarks with input size B. For all other evaluated benchmarks in the NAS benchmark suite, a slowdown of up to 8 % can be observed. However, with exception of the BT, CG, and UA benchmarks with input size A, the slowdown is close to or even below 1 %. Looking even closer, the confidence intervals suggest that the difference might not even be statistically relevant for most of the benchmarks. The situation is different for the hackbench benchmark. There, the CoBaS scheduler experiences a slowdown of approximately 15 % only for the smallest group size. For the other group sizes, the CoBaS scheduler has a speedup factor of up to 1.8 compared to CFS.

8.7.3 Discussion of the Results

There is a mixed picture for the results. For the NAS benchmark, the CoBaS based scheduler causes a slight performance degradation, while in the hackbench benchmark it is superior. Based on the results, it can be concluded that it is possible to construct a performant scheduler with the CoBaS architecture. However, the results also show that the performance strongly depends on the actual scheduling policy. For example, the performance degradations in the NAS benchmarks are most likely caused by the inferior consideration of cache affinity of the CoBaS implementation compared to the CFS. In order for a task to have the chance to have remaining data in the cache after re-scheduling, the load-balancer implemented for CoBaS only tries to schedule an arriving

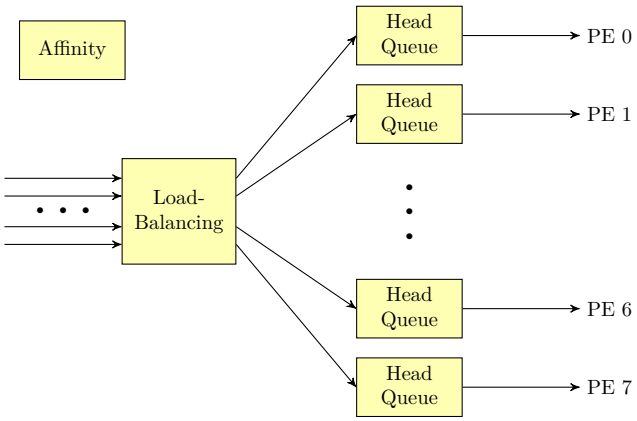


Figure 8.10: Topology used for the CoBaS performance evaluation.

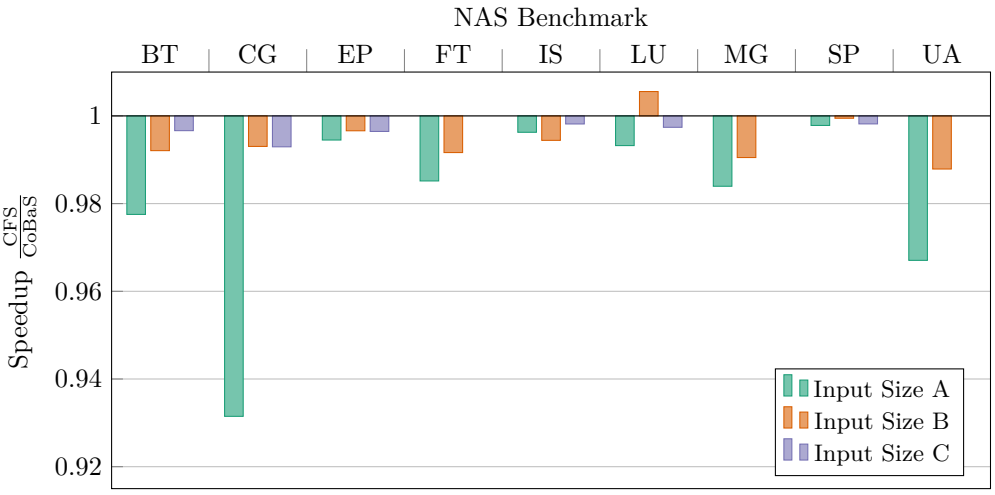


Figure 8.11: Speedup of the CoBaS scheduler in relation to the CFS scheduler for the *NAS benchmark suite*.

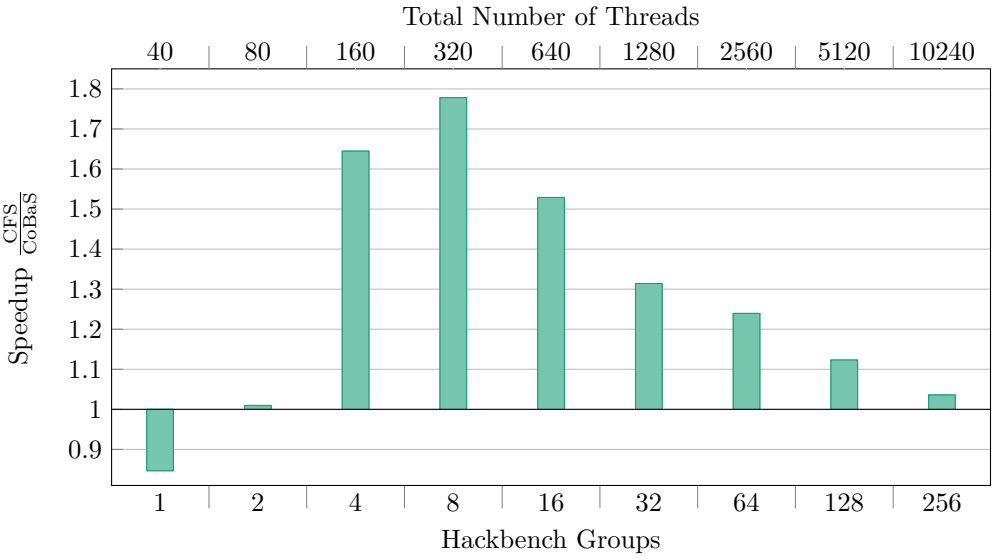


Figure 8.12: Speedup of the CoBaS scheduler in relation to the CFS scheduler for the *hackbench* benchmark. The lower x-axis is labeled by the number of groups, while the labeling of the upper x-axis shows the corresponding number of threads.

task once at the PE it was executed last. The the CFS has a more sophisticated strategy for that purpose that increases the chances to still find data in the cache. The hackbench benchmark, contrary to the NAS benchmarks, has almost no cache dependencies and, based on its simple communication pattern, profits most likely from the round-robin scheduling policy. Furthermore, does the CoBaS scheduler implementation not consider the special requirements for scheduling in an SMT system, where tasks scheduled on a single SMT sibling can influence each other.

8.8 Scheduler Adaptation

The necessity for the scheduler to be adaptive to the system was specified in Section 3.3. The argument primarily focused on the adaptability towards the system architecture. This aspect was already partially discussed earlier in this chapter in Section 8.3. However, the adaptability and reconfiguration of the scheduler also allow it to be adapted to the workload. The goal of the experiment presented in this section is to show how an adaptive scheduler can improve the system performance. The benefits are shown by a workload that reconfigures the scheduling policy according to its needs.

8.8.1 Experiment Description and Setup

The experiment assumes a simple workload that works in two phases through a processing pipeline, where each stage is an individual thread. First, it works through the pipeline forward; the second thread waiting for the first one, the third for the second and so on. In the second phase, the same pipeline is processed backward; the last thread is waiting for the second last, the second last for the third last and so on. The threads synchronize through a spinlock. The runtime of this workload is evaluated with three different scheduling strategies. First, with the vanilla Linux scheduler that implements the CFS strategy. Second, with a simple round-robin scheduling, using the CoBaS framework. Moreover, third, with an adaptive scheduling in CoBaS that first employs a FCFS scheduling strategy and is reconfigured to a LCFS strategy before the second phase begins (cf. Fig. 8.13). The workload is executed on a single core; therefore, the threads of the application are scheduled to the same PE. The experiment was conducted with 1 to 128 threads and the resulting runtime was measured for 100 runs. The experiments were carried out on the machine as the previous experiment (cf. Table 8.6 on Page 123). However, SMT and dynamic frequency scaling were deactivated to minimize the variances between measurements. A Gentoo userland was used with a vanilla Linux kernel v4.4 for the CFS benchmarks and the modified Linux kernel v4.4 with the CoBaS prototype implementation. The threads used the same loop as presented in Listing 8.2 in Section 8.5 on Page 112 to emulate a workload. Three workload scenarios were evaluated: a small workload with 100 loop cycles, a medium workload with 10^6 loop cycles, and a high workload with 10^9 loop cycles.

8.8.2 Experimental Results

The experimental results are summarized in Fig. 8.14. The data acquired during the experiment and used to generate the diagrams is summarized in the appendix in Table B.23 on Page 173. The upper row of the Figure depicts the absolute runtimes of the experiments for the three scheduling strategies and workloads per thread. The lower row illustrates the ratio between the optimized scheduling policy and the two other scheduling policies. Note that for the high workload experiment the results of the vanilla Linux CFS scheduling and CoBaS Round-Robin scheduling are highly overlapping.

For the high-workload scenario, the optimized approach is strictly superior to the other two scheduling approaches. In the two other scenarios, the situation is more diverse. The first thing that is unusual are the results of the CFS policy for one processing thread. For both the low and medium workload scenario, the runtime is several orders of magnitudes lower than the other scheduling policies. Two reasons explain this behavior: First, CFS

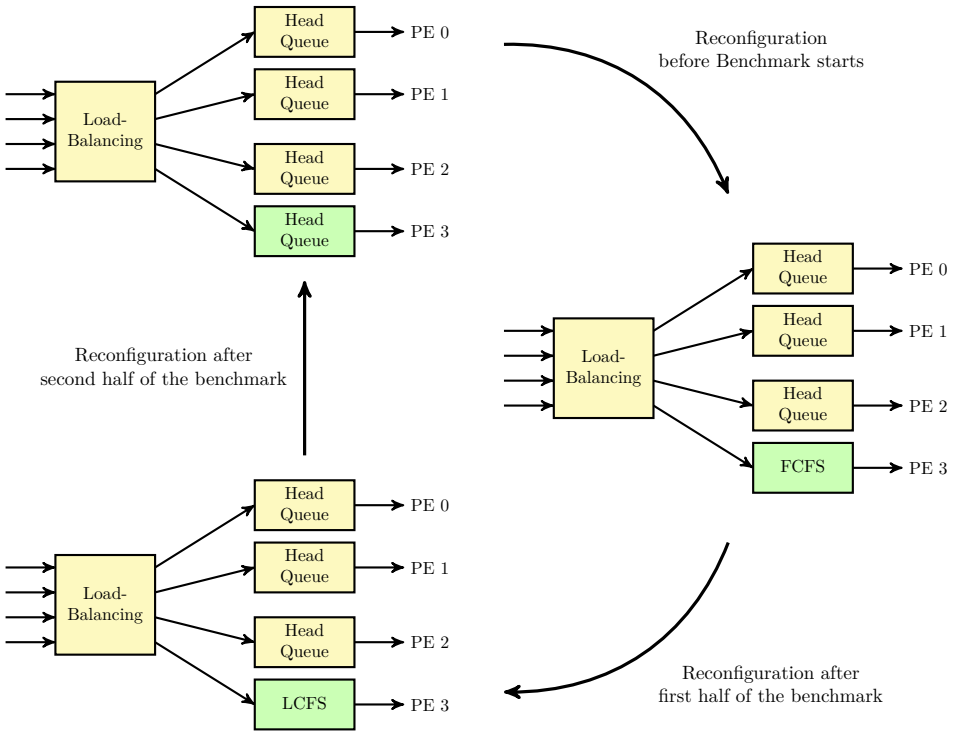


Figure 8.13: Adaptation of the scheduler topology for the Benchmark. The benchmark is exclusively executed on the PE 3 and subject to the scheduling policy of the lower Component instance (green). The scheduling policy is changed from *Head Queue* to *FCFS* to *LCFS* and back to *Head Queue*

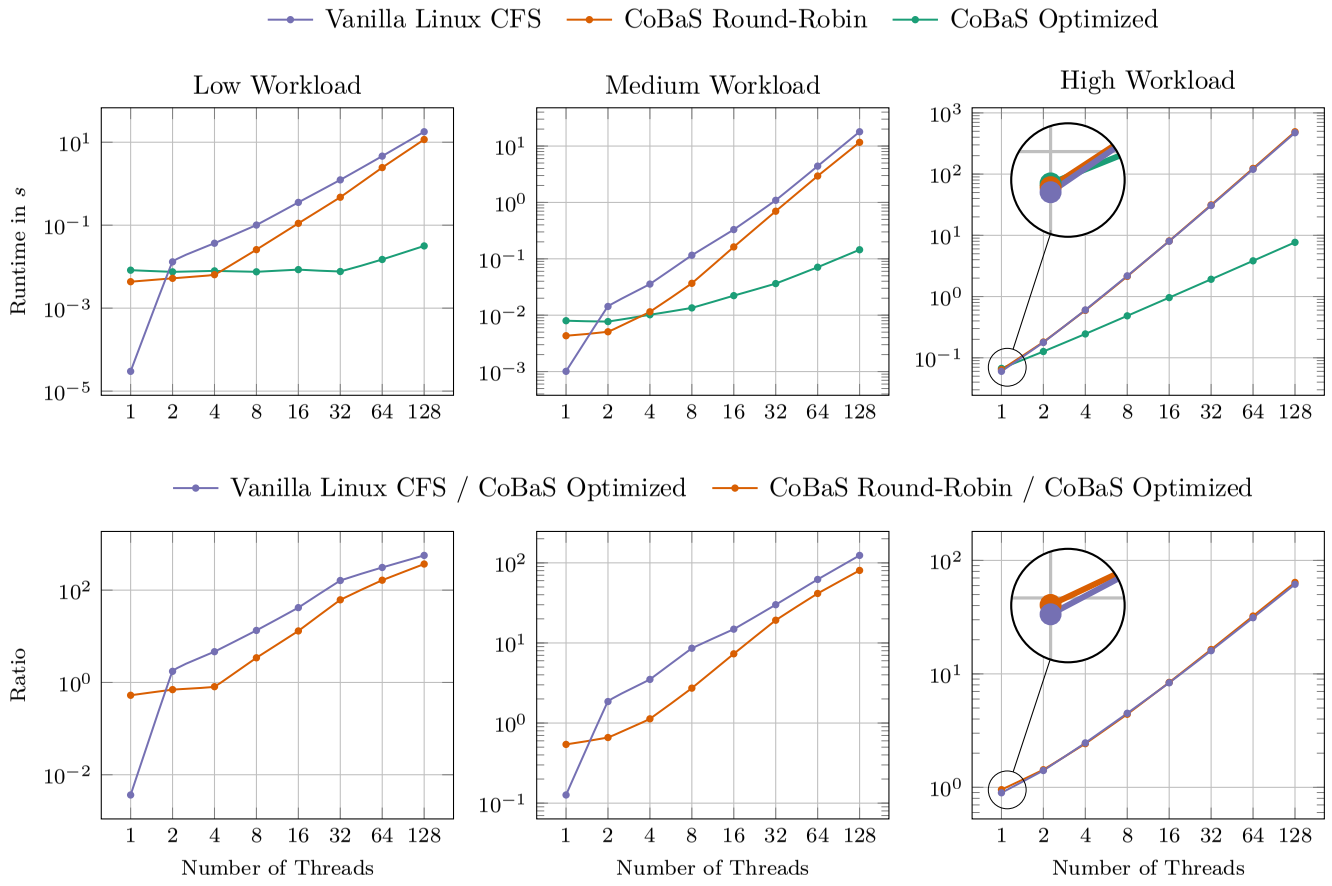


Figure 8.14: Results of the performance evaluation. The upper row shows the absolute runtimes of the experiments, while the lower row illustrates the ratio between the optimized CoBaS strategy and the two other scheduling policies. Note that for the high workload experiment the results of the vanilla Linux CFS scheduling and CoBaS Round-Robin scheduling highly overlap.

is optimized for responsibility; therefore, the newly created task is instantly scheduled to the CPU, whereas no such optimization exists in the CoBaS policies. Second, the task creation process is faster in the vanilla Linux scheduler. As explained in Section 8.1.3, a significant amount of code from the original scheduler, especially regarding task creation, was reused for the CoBaS prototype, therefore adding only additional creation overhead. However, the advantage of CFS vanishes already when considering two pipeline stages. From that point on, CFS is at best equal to the other strategies.

Even though the optimized CoBaS scheduler shares the same task creation overhead with the CoBaS round-robin scheduler, it performs worse for the lowest number of threads both in the low and medium workload scenarios. This behavior can be explained by the additional overhead required to reconfigure the scheduler Topology. In fact, for the low workload scenario, this overhead is prevailing over the processing time up to 32 threads for the low workload scenario as the total program runtime is constant until that point. However, the optimized strategy is better than the two other strategies even with this additional overhead.

8.8.3 Discussion of the Results

This experiment has given an example that shows that CoBaS can vastly improve the processing time of specific workload scenarios. The optimized scheduling strategy makes use of the property of CoBaS to be able to modify the scheduler at runtime. It has to be emphasized that the used modification goes beyond a reconfiguration of the system as the used scheduling strategies would not necessarily have to be already present in the system. They could also have been loaded during the runtime of the system. With this property, it would be possible to create and implement scheduling strategies specifically tailored to concrete problems. It can be expected that this approach can be used in more relevant and complex real-world scenarios that benefit from an optimized scheduling.

8.9 Foreign Language Components

Contrary to most other scheduler systems, CoBaS has a well-defined set of interfaces that should make it easy to adapt it to specific runtime systems. However, the interface set should also make it easier to implement Components in arbitrary languages. In Section 6.4, it was already discussed how hardware-based Components might be integrated into CoBaS with FPGA technology. The main goal of this section is to show the feasibility to implement CoBaS Components with programming concepts different from its main language. This section studies the integration of a Component written in a language other than C or its relatives, namely Rust.

The Rust programming language was already discussed in the decision for the main programming languages of the CoBaS prototype at the beginning of Chapter 7. Even though it was discarded as main programming language there, it is still interesting

as a study object for system programming. The purpose of this section is to research the challenges when implementing a Component in Rust and evaluating the possible overhead during execution. Therefore, Section 8.9.1 starts with an explanation of the main steps to implement and obstacles that occurred when implementing a CoBaS Component in Rust. Section 8.9.2 describes an experiment conducted to evaluate the overhead of a Rust based component. Finally, the last subsection discusses the result of the experiment in detail.

8.9.1 Implementation Details

Even though Rust was designed as system programming language and requires no special runtime system, bare-metal support is limited, though it is steadily increasing. Also, documentation regarding bare-metal implementations are still limited; yet, a simple operating system – Redox [S32] – that was implemented in Rust and an example for Linux kernel module written in Rust [S37] exist. Both examples were helpful to solve technical problems creating a CoBaS Component in Rust like, e.g., how to compile and link the code or how to handle the mapping of primitive variables or a `void` pointer.

The first challenge to implement a Component is the way it can access the functions offered by the framework. As the interface of CoBaS for Components is limited and well defined in the form of header files, it was possible to almost entirely automatically generate the interface with *rust-bindgen* [S36]. One challenge using *rust-bindgen* was that it parses all include files. The way the CoBaS prototype is built would also include the source files of the runtime system. As the Component is supposed to be independent of the runtime system, including an arbitrary one was not an option. Therefore, a new `zero` runtime system target was introduced that maps functions and types to a default value. With the `zero` target, it was possible to generate a binding that needed only minimal manual adaption regarding the module that contains the primitive types.

Another challenge was the dependence of the CoBaS prototype on the C preprocessor. The issue shall be explained on the example of a list. The CoBaS prototype uses an implementation for generic lists in C that is also employed in the Linux kernel as explained in Section 7.1.2. A structure is linked into a list by adding a *list head* structure that contains the usual pointers one expects from a list element. The list is then constructed by linking those *list heads*. To retrieve the actual data of a list element, a preprocessor macro is used that calculates the address of the list element based on the address of the *list head* and its position in the list element [cf. 28, pp. 87–89]. The offset is computed by the preprocessor at compile time. This implementation of a list is both efficient in performance and code reuse; however, it is not straightforward to be ported to other programming languages. This became an issue in the Rust implementation of a CoBaS Component as this facility is used, e.g., to retrieve the private part of the TCB. There are two general ways to cope with the issue. The first one would be to call a C function when a TCB has to be retrieved; the other one is to hand the offset over during the instantiation of the Rust Component. As the position of the private TCB is not supposed to change, the latter approach was chosen for locks.

8.9.2 Experiment Description and Setup

To evaluate the overhead of a Rust based Component, two Components that perform the minimal functionality of a CoBaS Component, namely moving tasks from an ingoing Pipe to an outgoing Pipe, were implemented both in C and Rust. The C-based component is called *Head Queue* and the Rust based component is called *Rusty*. With those two Components, a simple Topology as depicted in Fig. 8.15 was created that uses a *Head Queue* instance for PEs with an even ID and a *Rusty* instance for PEs with an odd ID. Furthermore, a *Load-Balancing* Component was used in conjunction with an *Affinity* Component to distribute tasks among the PEs. A *Marker* Component was used to mark the begin of the experiment in the call-trace. All of the used components are described in more detail in Appendix A.1.

As the differences in processing time are expected to be very low, similar to Sections 8.5 and 8.6, the *gem5* simulator was used to measure the execution time of the system functions. A system with two cores was simulated resulting in a setup in which the scheduling of the processes for the first core has to run through the *Head Queue* Component and the one for the second core through the *Rusty* Component. To acquire the call-graph, the scheduler subsystem was stressed in the same way as described in Section 8.6.

8.9.3 Experimental Results

The experimental results are depicted as a call-graph in Fig. 8.16. The notation of the graph is the same as described in Section 8.5. The function `hq_pipe_update` is the Pipe update function of the *Head Queue* Component instance and the `rust_pipe_update` function of the *Rusty* Component instance respectively. The results show that the average execution of the Pipe update takes 24.64 % longer in the Rust based Component than in the C based Component even though both Components perform the same task.

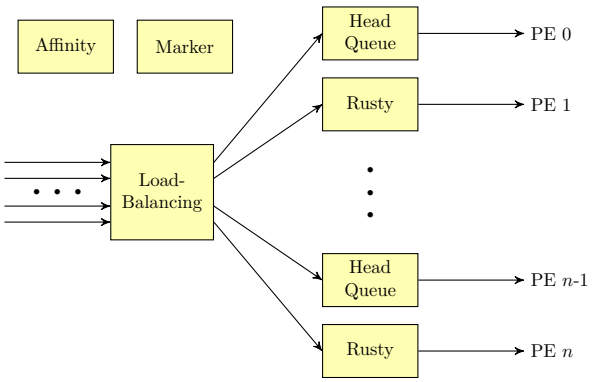


Figure 8.15: Topology used for the evaluation of the Rust based Component.

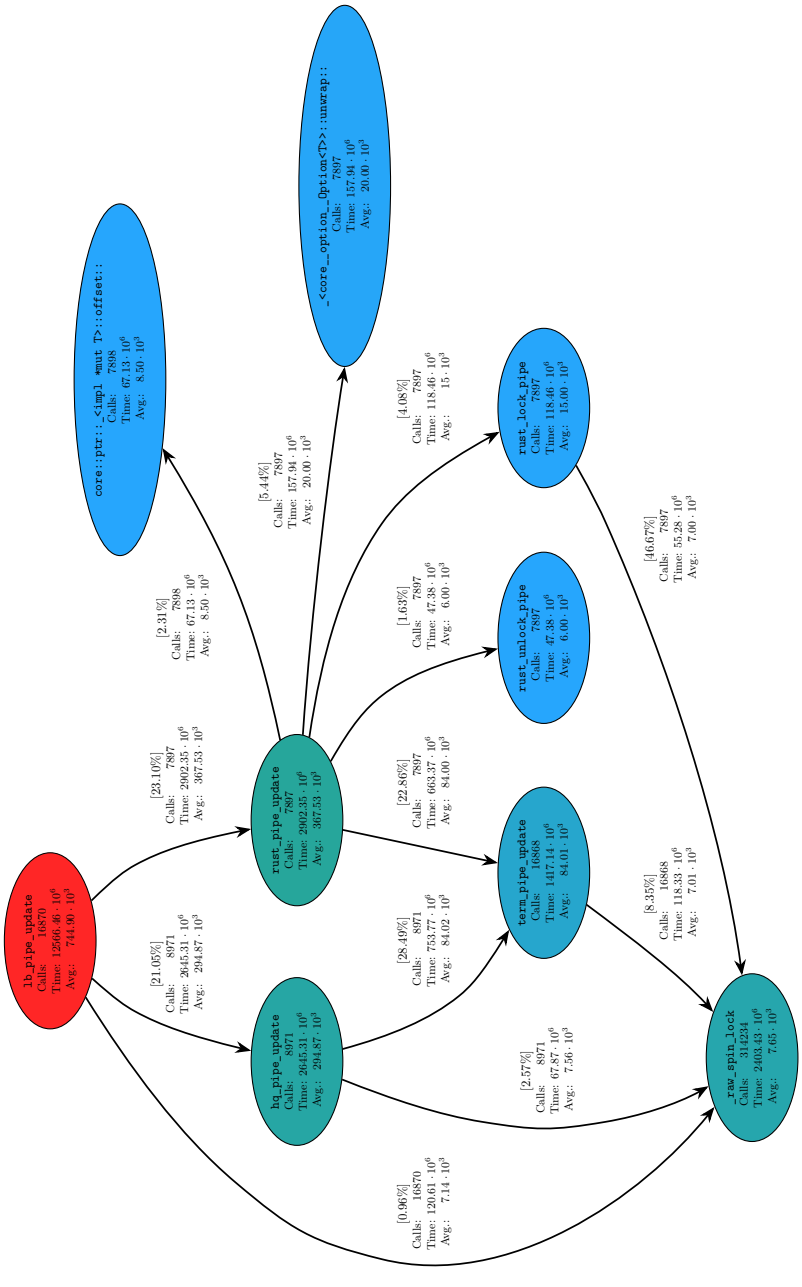


Figure 8.16: Excerpt of the system call-graph, running an instance of a Rust based and a C based CoBaS Component.

Even though this overhead seems very high at first glance, looking at the total times relativizes that. As the total time that is covered by the call-graph is approximately $1.198 \cdot 10^{12}$ simulation ticks, the total overhead regarding the entire system with kernel and user space execution of the `rust_pipe_update` function is rather small and can be quantified as following (all times in simulation ticks):

$$\text{total overhead} \approx \frac{(367.53 \cdot 10^3 - 294.87 \cdot 10^3) \cdot 7897}{1.198 \cdot 10^{12}} \approx 4.78 \cdot 10^{-4} = 0.478\,‰$$

Looking closer at the call-graph, the overhead can be split into several aspects. The first one is the use of the `Option` type [163] that is frequently utilized in the Rust language as return type if the return value is possibly undefined like, e.g., a `NULL` pointer. Accessing the wrapped value requires an explicit `unwrap` function call that accounts for 5.44 % to the execution time of the update function of the *Rusty* Component. The next cause for the additional overhead can be identified by pointer arithmetic. In Rust, a call to the `offset` function is necessary, which accounts, according to the call-graph, for another 2.31 % of the execution time of the update function. Even though in C this would also require an addition or subtraction, it would not take 17 clock cycles compared to the function call in Rust. Finally, the last visible overhead is introduced by an additional function wrapping of a CoBaS call. As discussed above, Rust cannot make use of the preprocessor. Therefore, some functions might have to be wrapped. For this experiment, the lock and unlock functions were wrapped to assess the overhead. The call-graph shows that it takes an additional 8000 simulation ticks or 16 clock cycles to acquire the lock. The overhead for unlocking the spin lock cannot be precisely determined as the unlocking logic was inlined into the wrapper function. However, the execution time of the unlock wrapper takes less time than the call of the locking wrapper. This can be explained by the fact that, through the inlining, the construction of a complete call stack is avoided. The discussed functions only cover approximately half of the observed overhead of the Rust based component. The remaining overhead can be explained by the overhead introduced by building the call stack for the just discussed functions and other Rust internals like a different switch case handling.

8.9.4 Discussion of the Results

The case study has shown how CoBaS allows the use of other programming functions to easily implement partial scheduling decisions. Because of the limited framework interface, it is easier to create a bridge between the framework written in C and a foreign programming language. This separation makes it also plausible that a CoBaS Component might be implemented in a FPGA. Even though the bridging introduces some overhead, it might be worth accepting it as a more modern or domain specific programming language might make the development less error prone, easier to test, and easier to maintain.

CHAPTER 9

Conclusions

This dissertation was set out to explore a novel approach to process scheduler design. The introduction and problem analysis expressed the motivation for that research: Computer architecture is changing dramatically and is becoming more and more heterogeneous, while the pace of that change is increasing drastically. This introduces two challenges at the same time that have contradicting solutions. On the one hand, the scheduler has to evolve faster to keep up with the pace of changes; on the other hand, it has to become more and more sophisticated to cope with complex architectures. These two solutions are contradicting in a way that the development process is losing momentum, when the system becomes bigger and more complex. Without changes to the scheduler architecture as it is common today, it would not be possible to cope with the future challenges.

With the CoBaS framework, this dissertation introduced a new architecture for process schedulers that solves theses issues. The component-based architecture and dynamic properties of the CoBaS framework enables developers and researcher to address the challenges introduced by new hardware architectures and computing concepts in a timely manner. The component-based approach also enables a high degree of reusability that is even further improved by simple means to integrate the framework into existing runtime systems. Using the framework approach makes it easier to use specific scheduler implementations in multiple runtime systems, requiring only the port of the framework and not of each individual scheduler implementation. The claims of the dissertation were substantiated by an extensive qualitative and quantitative evaluation. They have shown that the CoBaS architecture can scale for a high number of cores, has a justifiable overhead regarding the whole system, and can compete in real world scenarios with a native scheduler implementation.

This chapter concludes this dissertation. It starts in Section 9.1 by revisiting the claims initially stated in Section 1.4 and verifying that each of them has been addressed. Sections 9.2 and 9.3 are discussing strengths and weaknesses of the approach of the CoBaS framework respectively. Finally, the last section concludes this dissertation by discussing directions for future research.

9.1 Contributions Revisited

The introduction outlined the four main contributions of this dissertation in Section 1.4. This section revisits each of them and discusses them in the context of the previous chapters. Every subsection addresses one of the contributions with a brief review.

9.1.1 Heterogeneous Many-Core Support

Heterogeneous many-core systems introduce two challenges for the scheduler: on the one hand, the sheer number of cores, on the other hand, the different properties and execution models. Section 8.5 has shown that the CoBaS framework scales even for hundreds of cores. This suggests that it is feasible to implement meaningful scheduling algorithms for many-core systems encapsulated in the CoBaS framework. The Topologies used in the framework enable an easy identification of the information flow and the identification of possible bottlenecks.

The generic task model of the framework keep the architecture inside the boundaries of a specific execution model. This allows the management of arbitrary kinds of PEs. The CoBaS architecture ultimately puts the operating system in a position where it can finally manage all available resources by itself again.

9.1.2 Adaptability

Through its component based approach and its Topologies, the CoBaS framework is highly adaptable to new requirements. The architecture allows to modify the scheduler even at runtime. The benefits of this capability have been demonstrated through an experiment that has shown that the adaptation of the scheduling policy can result in a significant performance improvement. Furthermore, the reusable Components allow the fast creation of a new scheduler through existing partial solutions. In combination with its message broker system, the CoBaS architecture can react to changes both in requirements and in hardware facilities.

9.1.3 Composability

Composability is the foundation of the CoBaS framework with its component based approach. Components in the CoBaS framework have two main facilities that enable composition: Pipes and the Broker. Pipes allow a CoBaS Component to manipulate the order of tasks and the assignment to a PE. Section 5.8 has given a dissection on the properties of Components and a classification that gives a coarse reference whether specific Components are composable with each other or not. It was especially the study in Section 8.8 that has shown how a specific working set can profit from a composable scheduler.

The composability significantly facilitates the reuse of existing implementation parts. Without composability, the reuse of existing implementations for CoBaS would be limited to the whole scheduler implementation and not to smaller parts. The experiments in Chapter 8, for example, greatly profited from the reuse of Components as only a limited set of Components was necessary to conduct the experiments.

9.1.4 Runtime System Independence

The independence of a scheduler architecture from the underlying runtime system was discussed in Section 3.4. The necessity for the independence stems mainly from non-functional challenges like the implementation of a completely new operating systems. This dissertation has shown that the introduced framework enables the use of existing scheduler implementations in multiple operating systems. It was used in the Linux and FreeBSD kernel as main scheduling facility and the adaptation effort was quantified in means of lines of code.

The evaluation in Section 8.9 has also shown that, through its well defined interfaces, the CoBaS framework can support foreign programming languages. This suggests that it is on the one hand possible to create and integrate a DSL for the process scheduler domain; on the other hand, it seems feasible to augment the process scheduler with a hardware based acceleration through, e.g., FPGA technology.

9.2 Strengths of the Approach

The CoBaS architecture was designed to cope with the challenges process scheduler design is facing in the coming years. Its greatest strength is its flexibility and adaptability. As discussed in the introduction on the example of the introduction of the Bulldozer architecture by AMD (cf. Section 1.2.1) and the introduction of the big.LITTLE architecture by ARM (cf. Section 1.2.2), system software developers struggle to keep up with changes to hardware architecture. Today, performance gains are mainly achieved due to an increased number of cores and specialization. Because of the end of *Dennard scaling* and limitations through *Amdahl's law*, the later option will become the vastly dominating factor that enables future performance improvements. In order to use such systems to their fullest potential, the operating system scheduler has to be tailored to such architectures. With the CoBaS framework, the adaptation to such systems can be easily facilitated as shown in the evaluation. A new scheduler can be created by reusing parts of existing solutions. This permits the implementation of changes to the process scheduler in a timely manner. Furthermore, the encapsulation of the scheduler logic into a dedicated framework allows the development and implementation of scheduling policies without extensive knowledge about the specific runtime system. This can make it easier to test new scheduling approaches and compare them amongst each other in research.

The usage of explicit Topologies makes it, especially in the many core scenario, easier to identify potential bottlenecks. The Topologies allow the explicit and visible assignment of specific scheduling strategies to specific PEs. This does not only simplify the development process, but also supports a better understanding of the whole scheduler implementation.

Furthermore, the implementations for the CoBaS framework are portable between different runtime systems that support the CoBaS framework. This makes it easier for smaller development projects to profit from advances in scheduler design and might enable diversity among operating systems. Moreover, the framework approach enforces much cleaner and more precise interfaces than the internal interfaces most current operating systems provide. By decoupling the scheduler implementation from the underlaying runtime system, the interfaces can be stable over a longer period of time. This facilitates the development of new implementations and the comprehension of exiting ones. The evaluation has also shown that the clean interface makes it straight forward to implement the scheduler in a different programming language than that of the runtime system. This paves the way for a scheduler implementation in a DSL or with hardware acceleration.

9.3 Weaknesses of the Approach

The main weakness of CoBaS can clearly be identified in the additional overhead that the framework introduces into the system. The overhead was quantified for the prototypical implementation throughout Chapter 8. Even though it has been shown that the CoBaS architecture scales even for many-core systems, an overhead remains that can, in general, be considered proportional to the number of used processing instances. The overhead is inherent to the system and cannot be eliminated as a separation and the introduction of additional interfaces hinder a tight integration and optimization and always introduces an overhead. Therefore, it remains to be seen whether the benefits like a cleaner design, reuse, and flexibility outweigh the downsides of the additional overhead.

A minor weakness can be identified in the missing support for real-time systems. Guaranties for a deadline and a dynamic system are hard to bring in line as everything has to be deterministic to give such guarantees. However, contrary to the overhead, the dynamics are given by the system but do not necessarily have to be used. Therefore, it might be possible to design a subset of the CoBaS architecture and its features that can guarantee hard real-time.

A final constraint of the CoBaS approach might be the necessity to adapt an existing operating system to it. This dissertation has discussed the necessary steps and quantified the overhead to adapt both the Linux and FreeBSD kernel to use CoBaS as their main scheduling facility. The necessary effort is undeniable, however, this effort is not necessarily higher than implementing a distinguished scheduling algorithm and will pay out as soon as another algorithm shall be used. This dissertation has also shown that the overhead for maintaining the CoBaS integration is manageable.

9.4 Future Directions

The current state of research to employ a general process scheduler architecture for future systems can be seen as a first step towards a general tool for building such schedulers. This section discusses further directions to advance the state of the art in this area.

9.4.1 Communication Topologies

Section 2.2.1 of this dissertation has outlined different communication topologies that are employed in multi- and many-core systems. Even though discussed on a theoretical level in Section 6.2, the prototype built for this work does not yet support advanced inter processor communication facilities. Consequently, the next step of research would be to extend the prototype and employ CoBaS in a mesh based many-core system, potentially with distributed or non-cache coherent memory and evaluate its scalability in such a scenario. In that context, the extension of the single broker system as it is today towards a multi broker system might become indispensable and another interesting research object.

9.4.2 Topology Management

Another field of study regarding the CoBaS architecture can be identified as the management of Topologies. Even though the management was defined and described in detail, certain aspects, for example regarding the combination of existing Topologies, remain to be investigated. This includes the question on how to automatically merge two existing partial topologies, e.g., for different classes of PEs to a holistic scheduling Topology. Furthermore, the definition of scheduler Topologies themselves can be considered an object of research. In the current prototype, Topologies are defined programmatically in C. The definition process might be simplified through a DSL tailored to that problem or even a Graphical User Interface (GUI) based approach that would improve the comprehensibility, especially for complex scheduler layouts.

9.4.3 Scheduler Analysis

The experiments of Section 8.5 have shown that the scheduler can easily be congested in a many-core system if not designed in a scalable way. However, the experiment has also shown that not every PE needs its own code path. In order to design a scheduler that is both scalable and does not use an unnecessary separation of code paths, tools for analysis are necessary. The evaluation of this dissertation already made initial steps towards a contention analysis in the scheduler code path through the call-graphs. Those could be visualized in a Topology editor and make it easier to spot potential bottlenecks in a many-core environment.

9.4.4 Multi-Scheduler Environments

This dissertation has discussed the issue of scheduling in the context of virtual machines. As the guest most of the time employs system software with its own process scheduling, it is interesting to study the interaction between the host and the guest scheduler. Through the communication infrastructure of the CoBaS framework, it might be possible for both schedulers to interact with each other and, thus, create better overall scheduling results. As the generic information distribution system of the CoBaS framework is, as of today, unique to scheduler architectures, new opportunities for optimization might emerge.

9.4.5 Security Aspects

This dissertation has also shown that the execution of certain workloads can be optimized by a custom scheduling policy. However, as this policy is provided by the application level and executed in the kernel context, several security related issues arise. The analysis of these attack vectors and how to improve the CoBaS Component interface would be worth investigating in future work. Different scenarios with different degrees of freedom exist. From the security perspective the simplest solution could be an interface that only allows the application to chooses from a certain set of policies that are put into action through the super-user or the operating system itself. A more critical solution from a security point of view would be that the user can freely provide his or her scheduler implementation. As the code is brought into execution inside the kernel, this opens many security related issues. However, researching the attack vectors might yield a solution between these two extremes.

APPENDIX A

CoBaS Components

This appendix summarizes details on CoBaS Components that were implemented for the prototype and mainly used during the quantitative evaluation in Chapter 8. Appendix A.1 gives an overview of all the implemented Components. Appendix A.2 presents the source code of the Pipe update functions of the *Head Queue* and *Rusty* Components as used in particular in Section 8.9.

A.1 Implemented Components

Several components were implemented to test and evaluate the CoBaS prototype. These components will be described subsequently. Besides a short description, the following Component properties are summarized in an introductory box:

Ingoing Pipes	The number of Pipes that can be connected as an input to a Component instance. Non-Fixed numbers indicate that the number of ingoing Pipes can be configured during instantiation.
Outgoing Pipes	The number of Pipes that are leaving the Component instance. Non-Fixed numbers indicate that the number of outgoing Pipes can be configured during instantiation.
Classification	Classification of the Component according to Section 5.8.2.
Topic Subscriptions	The list of topics the component subscribes to.
Topic Responder	The list of topics the component acts as a Responder to.
Topic Dependencies	The list of responders the component relies on.

Head Queue Component (0x01)			
Ingoing Pipes	1	Topic Subscriptions	*none*
Outgoing Pipes	1	Topic Responder	*none*
Classification	Neutral	Topic Dependencies	*none*

The *Head Queue Component* is the simplest possible CoBaS Component that utilizes the Pipe system. It consist of one ingoing and one outgoing Pipe. Its only functionality is to forward all changes from the ingoing to the outgoing Pipe. It was used primarily for testing and experimentation on the general functionality of the CoBaS framework. It was also used to simulate a round-robin behavior as that is the default behavior of CoBaS when new tasks are submitted to the framework.

First-Come, First-Served Component (0x02)			
Ingoing Pipes	1	Topic Subscriptions	*none*
Outgoing Pipes	1	Topic Responder	*none*
Classification	Ordering	Topic Dependencies	`Lamport`

The *First-Come, First-Served Component* enforces, as the name implies, a *FCFS* policy on the in-going tasks. It requires that a responder for the `Lamport` topic as it uses a *Lamport* time to determine the task order.

Last-Come, First-Served Component (0x03)			
Ingoing Pipes	1	Topic Subscriptions	*none*
Outgoing Pipes	1	Topic Responder	*none*
Classification	Ordering	Topic Dependencies	`Lamport`

The *Last-Come, First-Served Component* enforces, as the name implies, a *LCFS* policy on the in-going tasks. It requires that a responder for the `Lamport` topic as it uses a *Lamport* time to determine the task order.

Burn Component (0x04)			
Ingoing Pipes	1	Topic Subscriptions	`Burn`
Outgoing Pipes	1	Topic Responder	*none*
Classification	Neutral	Topic Dependencies	*none*

The *Burn* Component works similar to the *Head Queue* Component. However, in addition it is possible to assign a number of cycles that are burned in a busy wait loop. This feature was used to simulate more complex scheduling algorithms.
Each Component instance subscribes to the `Burn` topic that contains the number of cycles that will be burned at each update. The Component uses the same busy wait function as outlined in Listing 8.2 on Page 112. In the gem5 simulator, that results in 2500 simulator ticks or five clock cycles of the simulated CPU per burn cycle.

Depth Component (0x05)			
Ingoing Pipes	1	Topic Subscriptions	`Burn`
Outgoing Pipes	1	Topic Responder	*none*
Classification	Neutral	Topic Dependencies	*none*
The *Depth* Component works in a similar way as the *Burn* Component. However, it has an adjustable name of the Pipe update function to allow the differentiation between the update functions of different Component instances.			

Rust Component (0x09)			
Ingoing Pipes	1	Topic Subscriptions	*none*
Outgoing Pipes	1	Topic Responder	*none*
Classification	Neutral	Topic Dependencies	*none*
The *Rust* Component has, by intention, exactly the same functionality as the *Head Queue* Component. However, instead of being implemented in C as the rest of the CoBaS prototype, the *Rust* Component is implemented in Rust.			

Task Distributor Component (0x10)			
Ingoing Pipes	1	Topic Subscriptions	`CPU_STATUS`
Outgoing Pipes	$n \in \mathbb{N}^+$	Topic Responder	*none*
Classification	Filtering, Distributing	Topic Dependencies	`AFFINITY`
A crucial part of a multi-core scheduler is the distribution of tasks to PEs based on their affinity. In some operating systems like, e.g. Linux or FreeBSD, some specific tasks are required to run on a certain CPU or required not to run on a certain CPU. Therefore, it is not possible to activate additional cores in those systems without enforcing the affinity. The *Task Distributor* Component performs that task and can be considered a very minimalistic load balancing facility. It schedules all tasks on the first PE except the task has to run on a specific PE or cannot run on the first PE. The number of output Pipes are configured during the instantiation. The Component uses the `CPU_STATUS` notification to keep track to which output Pipes tasks can be assigned. If a task requires to be scheduled on a specific PE, but has not arrived yet, the Component will withhold this task until the PE in question arrives. Furthermore, the Component depends on a Component instance that can response to the `AFFINITY` topic in order to determine the affinity of each task.			

Task Mux Component (0x11)			
Ingoing Pipes	$m \in \mathbb{N}^+$	Topic Subscriptions	`CPU_STATUS`
Outgoing Pipes	$n \in \mathbb{N}^+$	Topic Responder	*none*
Classification	Filtering, Distributing, Consolidating	Topic Dependencies	`AFFINITY`
The *Task Mux* Component works similarly to the *Task Distributor* Component. However, instead of only one ingoing Pipe, it has a configurable number of ingoing Pipes that are determined during the instantiation.			

Load Balancer Component (0x12)			
Ingoing Pipes	$m \in \mathbb{N}^+$	Topic Subscriptions	`CPU_STATUS`
Outgoing Pipes	$n \in \mathbb{N}^+$	Topic Responder	*none*
Classification	Filtering, Distributing, Consolidating	Topic Dependencies	`AFFINITY`
The *Load Balancer* Component works similarly to the *Task Mux* Component. However, it additionally employs a real load balancing algorithm. The algorithm checks the load of the outgoing Pipelines and tries to achieve an equal distribution of the load among all outgoing Pipes. The balancing algorithm has a complexity of $\mathcal{O}(n)$. The *Load Balancer* Component implicitly assumes that the affinities are mapped 1:1 to its assigned outgoing Pipes.			

Advanced Balancer Component (0x13)			
Ingoing Pipes	$m \in \mathbb{N}^+$	Topic Subscriptions	`CPU_STATUS`
Outgoing Pipes	$n \in \mathbb{N}^+$	Topic Responder	*none*
Classification	Filtering, Distributing, Consolidating	Topic Dependencies	`AFFINITY`
The *Advanced Balancer* Component works similar to the *Balancer* Component. However, it can be configured with a mapping of PE to outgoing Pipes. This feature can, for example, be used for a hierarchical scheduling when several PEs are mapped to one outgoing Pipe. This approach is illustrated in FigureA.1.			

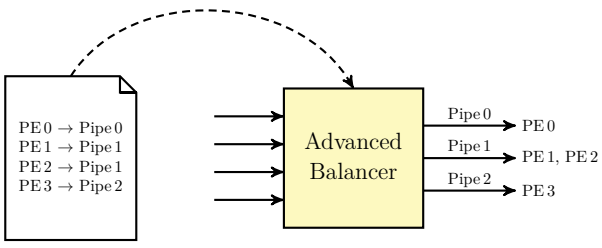


Figure A.1: Mapping in the Advanced Balancer Component.

ISA Demux Component (0x14)			
Ingoing Pipes	$m \in \mathbb{N}^+$	Topic Subscriptions	`ISA`
Outgoing Pipes	$n \in \mathbb{N}^+$	Topic Responder	*none*
Classification	Distributing	Topic Dependencies	`ISA`
The *ISA Demux* Component distributes a stream of incoming tasks to its outgoing Pipes according to their ISA requirements. During instantiation, asides of the outgoing Pipes themselves, the Component is supplied with a mapping which outgoing Pipe supports which ISA. If a task is added, the Component will acquire the ISA requirement of the task via the responder for the `ISA` topic. When the ISA requirement of a task is changed while it is present in one of the outgoing Pipes, the Component will migrate it to a new, fitting outgoing Pipes. If no fitting outgoing Pipe is available for a task to fulfill its ISA requirement, an error will be reported.			

CPU Affinity Component (0x20)			
Ingoing Pipes	*none*	Topic Subscriptions	`AFFINITY`
Outgoing Pipes	*none*	Topic Responder	`AFFINITY`
Classification	Neutral	Topic Dependencies	*none*
The *Affinity* Component can store a PE affinity for every task. The affinity can be retrieved through the responder system.			

Lamport Component (0x21)			
Ingoing Pipes	*none*	Topic Subscriptions	*none*
Outgoing Pipes	*none*	Topic Responder	`LAMPORT`
Classification	Neutral	Topic Dependencies	*none*
The *Lamport* Component creates a Lamport time stamp for every task. Each time a new task is created, the time stamp is increased. Through the timestamp, it is possible to put the task in a relation to the arrival in the system. The time stamp is retrieved through the `LAMPORT` topic request.			

Status Component (0x22)			
Ingoing Pipes	*none*	Topic Subscriptions	*none*
Outgoing Pipes	*none*	Topic Responder	`STATUS`
Classification	Neutral	Topic Dependencies	*none*
The *Status* Component is designed to retrieve several statics about the current state of the scheduler framework, e.g., the load of all or a specific Pipe. It responds to the `STATUS` topic request. The exact kind of information that shall be retrieved has to be specified in the request message.			

TCB Entry Component (0x23)			
Ingoing Pipes	*none*	Topic Subscriptions	TCB
Outgoing Pipes	*none*	Topic Responder	TCB
Classification	Neutral	Topic Dependencies	*none*
The *TCB* Component can store a pointer in an arbitrary data set. The data set is assigned through a notification. The retrieval of the data can either be requested through a shallow copy, which is in general just the return of a reference, or a deep copy, which duplicates the data in the response message.			

Marker Component (0x24)			
Ingoing Pipes	*none*	Topic Subscriptions	TCB
Outgoing Pipes	*none*	Topic Responder	*none*
Classification	Neutral	Topic Dependencies	*none*
The *Marker* Component was designed as a tracking facility to create markings in the call-trace of a system. The triggering of the *Marker* Component through a notification will result in the execution of a code path that can be observed in the call trace of the system. The notification for this component is intended to be generated in user space and has no specific payload.			

Termination Component (0xFD)			
Ingoing Pipes	1	Topic Subscriptions	*none*
Outgoing Pipes	*none*	Topic Responder	*none*
Classification	Neutral	Topic Dependencies	*none*
The *Termination* Component is a mandatory Component and used internally for the framework. It is used as an endpoint for the Pipe update chain and its Pipe is used by the scheduling function of the CoBaS framework to acquire tasks for specific PEs.			

A.2 Source Code Excerpts from the Prototype

Listing A.1: Pipe update function of the Head Queue Component.

```
static void hq_pipe_update(fw_component_inst_t *self, fw_pipe_t *pipe)
{
    fw_task_t *pos, *n;
    hq_private_data_t *pd = inst_data(hq_private_data_t);

    fw_spin_lock(&pipe->lock);

    fw_list_for_each_entry_safe(pos, n, (&pipe->tasks_changed),
                                pipe_pcb[pipe->id].changed) {
        switch (pos->pipe_pcb[pipe->id].change_type) {
            case ADDED:
                fw_pipe_add(pd->out_pipe, pos);
                break;
            case REMOVED:
                fw_pipe_remove(pd->out_pipe, pos);
                break;
            default:
                break;
        }
    }

    fw_pipe_clean(pipe);

    fw_spin_unlock(&pipe->lock);

    pd->out_pipe->out->ops.pipe_update(pd->out_pipe->out, pd->out_pipe);
}
```

Listing A.2: Pipe update function of the Rust based Component.

```
unsafe extern "C" fn rust_pipe_update(_self: *mut fw_component_inst_t,
                                      _pipe: *mut fw_pipe_t) {
    let ref private = *((*_self).inst_data as *mut RustyPrivate);

    rust_lock_pipe((*private).input);

    let start = (*private.input).tasks_changed.next;
    let mut current = start;
    while (*current).next != start {

        let ref mut task = *((current as *mut u8).offset(private.c_offset) as *mut fw_task_t);

        match task.pipe_pcb[(*private.input).id as usize].change_type {
            fw_pipe_change::ADDED => {fw_pipe_add(private.output, task)},
            fw_pipe_change::REMOVED => {fw_pipe_remove(private.output, task)},
            fw_pipe_change::UNDEF => {}
        }

        current = (*current).next;
    }

    fw_pipe_clean(private.input);

    rust_unlock_pipe(private.input);

    (*(*private.output).out).ops.pipe_update.unwrap()((*private.output).out, private.output);
}
```

APPENDIX B

Quantitative Evaluation Data

This appendix records the data gathered in the experiments in Sections 8.5 to 8.8. The columns designated as *runtime* depict the average runtime of the particular experiment or function. The number of data points to create the average is either given in the tables themselves or in the tables' caption. All confidence intervals are given for a confidence level of 95 % in a Student's t-distribution.

B.1 Scalability and Contention

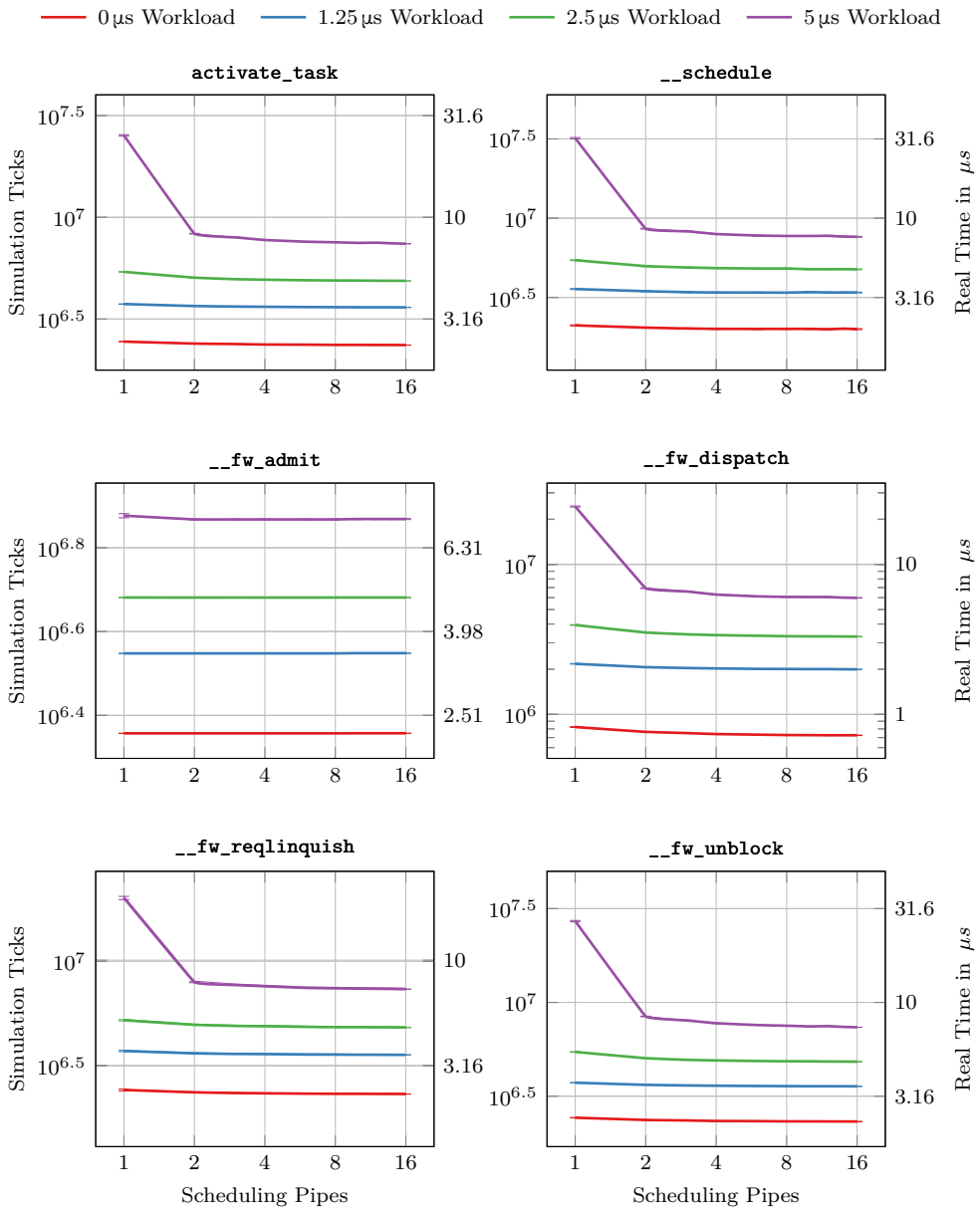


Figure B.1: Results of the scalability experiments with 16 PEs using the *gem5* Simulator and four different workload emulations. The average runtime of each function is given in simulation ticks (left y-axis) and real time (right y-axis) as a subject to the number of scheduling Pipes (cf. Section 8.5.3).

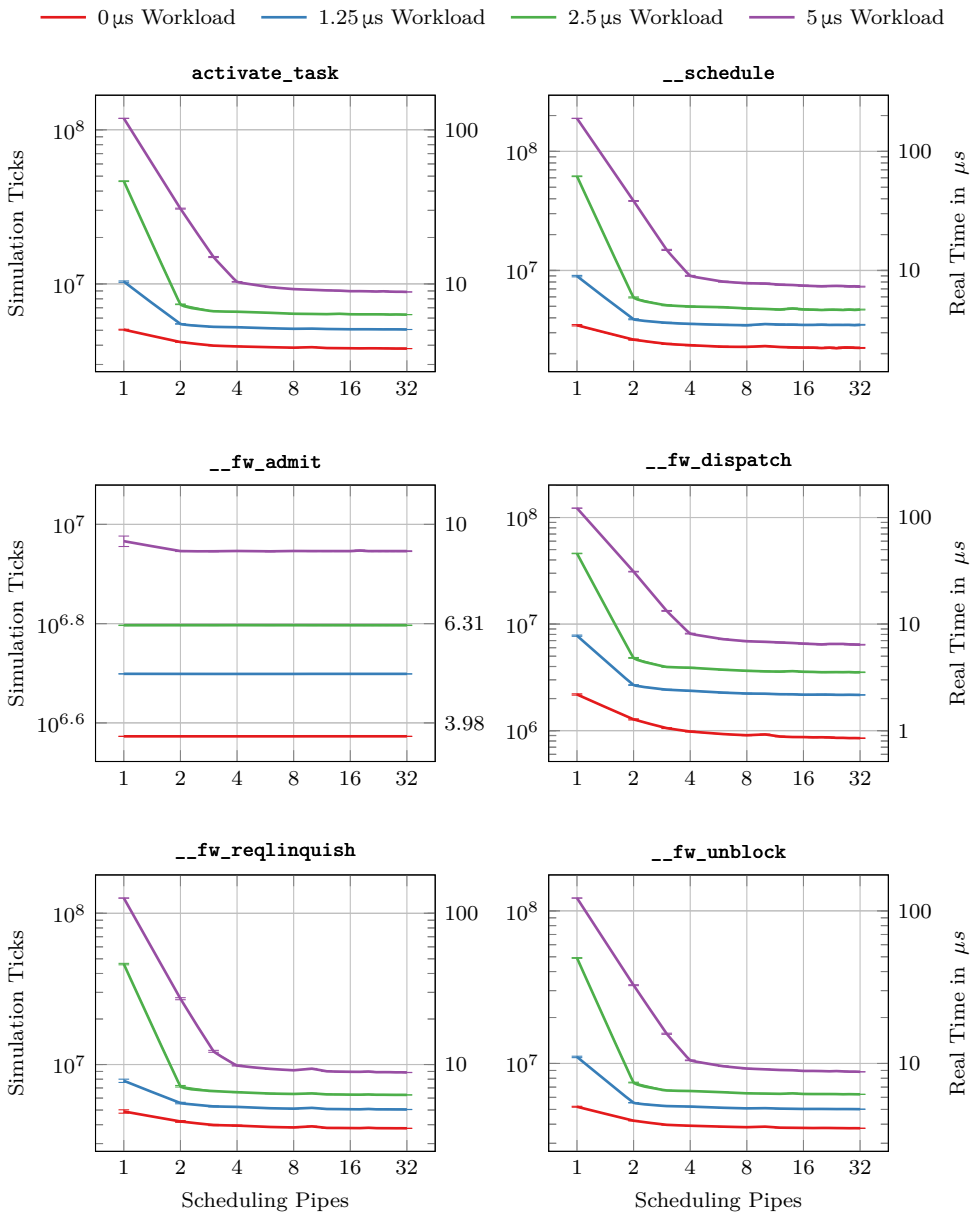


Figure B.2: Results of the scalability experiments with 32 PEs using the *gem5* Simulator and four different workload emulations. The average runtime of each function is given in simulation ticks (left y-axis) and real time (right y-axis) as a subject to the number of scheduling Pipes (cf. Section 8.5.3).

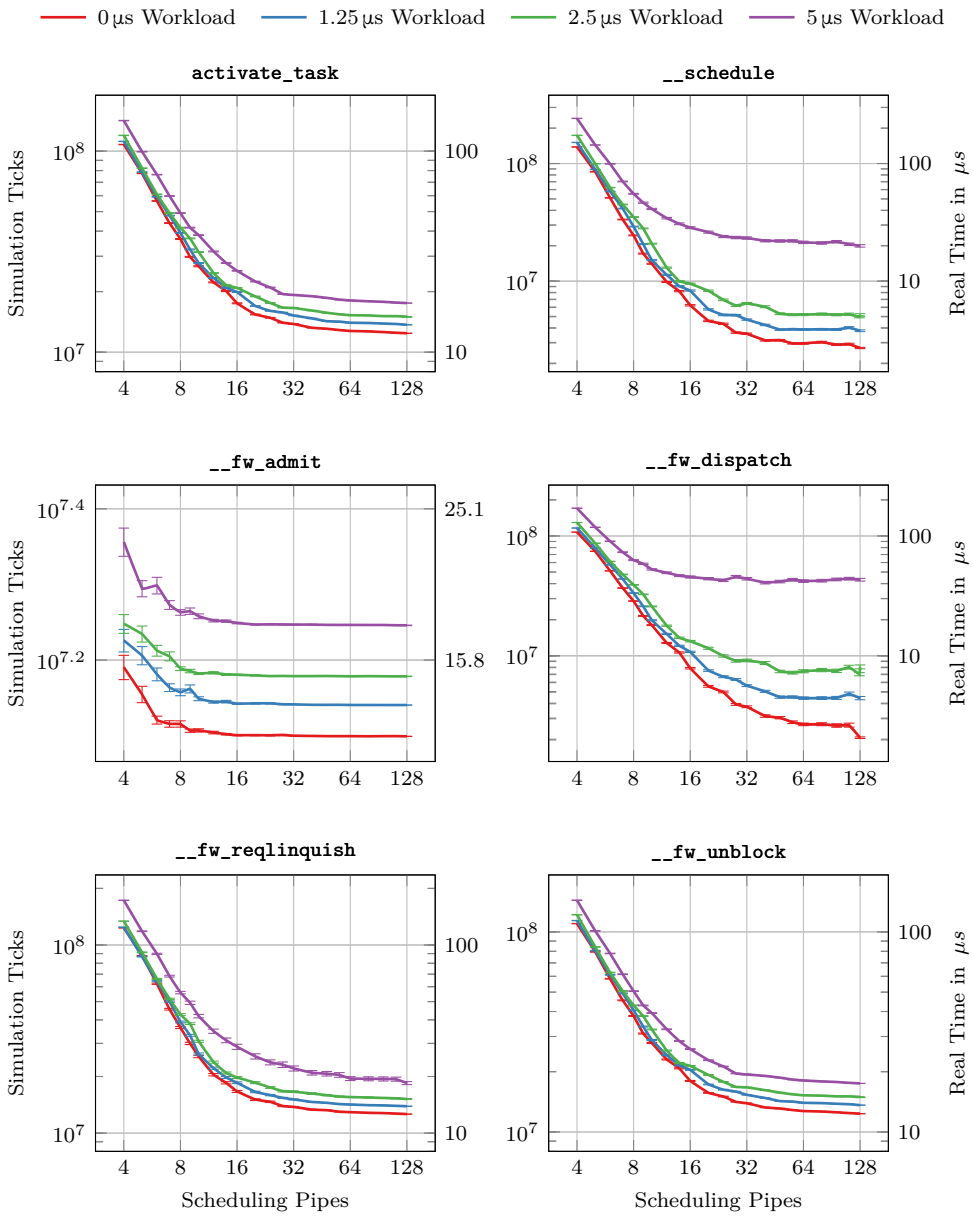


Figure B.3: Results of the scalability experiments with 128 PEs using the *gem5* Simulator and four different workload emulations. The average runtime of each function is given in simulation ticks (left y-axis) and real time (right y-axis) as a subject to the number of scheduling Pipes (cf. Section 8.5.3).

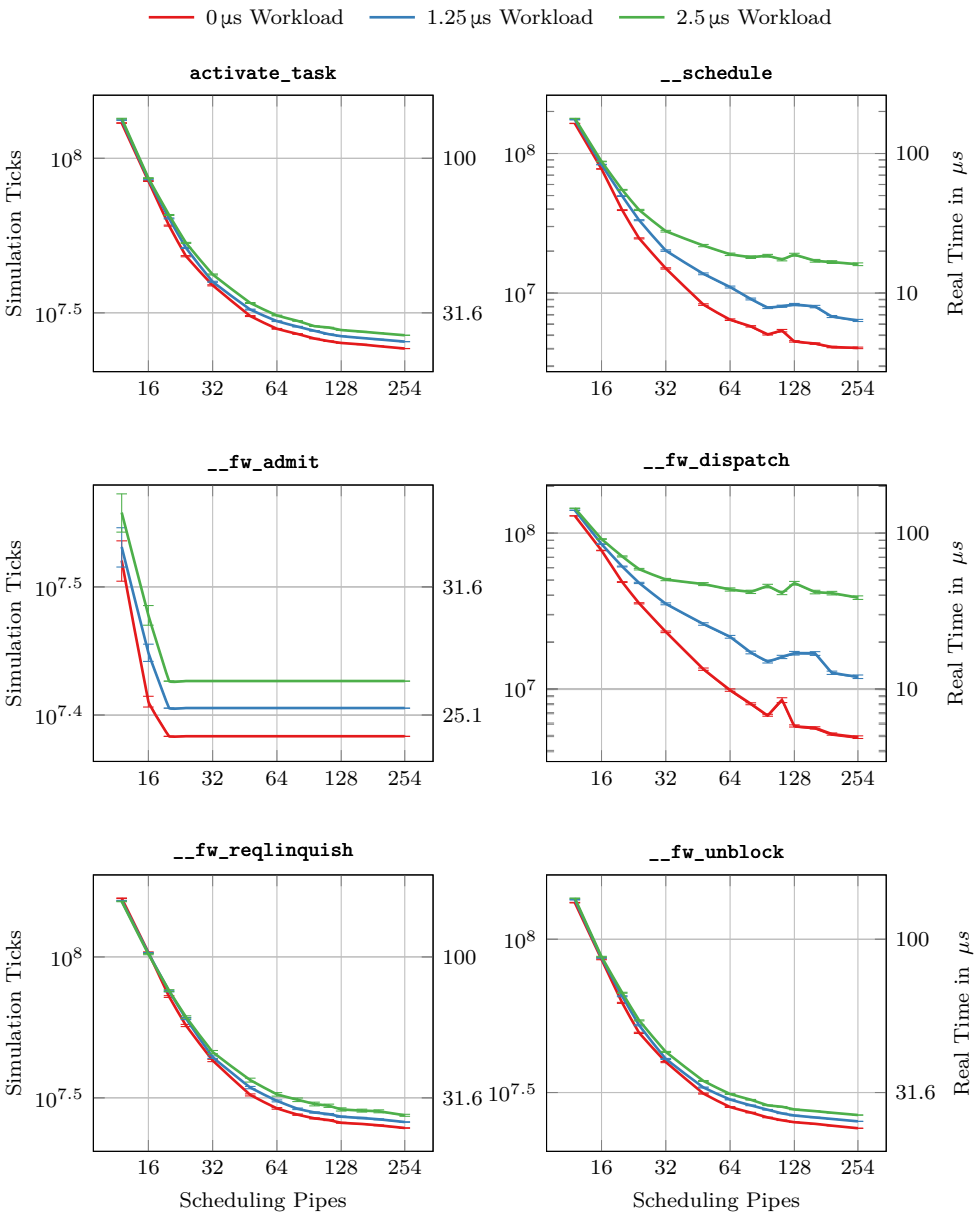


Figure B.4: Results of the scalability experiments with 254 PEs using the *gem5* Simulator and three different workload emulations. The average runtime of each function is given in simulation ticks (left y-axis) and real time (right y-axis) as a subject to the number of scheduling Pipes (cf. Section 8.5.3).

Table B.1: Scalability experiment results with 16 PEs and 0 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	2.449	$\pm 4.83 \cdot 10^{-3}$	25,867	2.107	$\pm 1.66 \cdot 10^{-2}$	40,731	2.275	$\pm 2.89 \cdot 10^{-4}$	4,003
2	2.394	$\pm 2.94 \cdot 10^{-3}$	26,170	2.039	$\pm 1.49 \cdot 10^{-2}$	41,014	2.275	$\pm 2.89 \cdot 10^{-4}$	4,003
3	2.382	$\pm 2.45 \cdot 10^{-3}$	26,554	2.014	$\pm 1.43 \cdot 10^{-2}$	41,431	2.275	$\pm 2.89 \cdot 10^{-4}$	4,003
4	2.369	$\pm 1.88 \cdot 10^{-3}$	26,341	2.003	$\pm 1.40 \cdot 10^{-2}$	41,199	2.275	$\pm 2.89 \cdot 10^{-4}$	4,003
6	2.365	$\pm 1.79 \cdot 10^{-3}$	26,473	2.000	$\pm 1.42 \cdot 10^{-2}$	41,346	2.275	$\pm 2.89 \cdot 10^{-4}$	4,003
8	2.359	$\pm 1.42 \cdot 10^{-3}$	25,796	2.003	$\pm 1.43 \cdot 10^{-2}$	40,670	2.275	$\pm 2.89 \cdot 10^{-4}$	4,003
10	2.359	$\pm 1.31 \cdot 10^{-3}$	26,084	2.003	$\pm 1.40 \cdot 10^{-2}$	40,792	2.276	$\pm 6.28 \cdot 10^{-4}$	4,003
12	2.357	$\pm 1.20 \cdot 10^{-3}$	26,281	1.993	$\pm 1.38 \cdot 10^{-2}$	40,969	2.276	$\pm 6.28 \cdot 10^{-4}$	4,003
14	2.357	$\pm 1.20 \cdot 10^{-3}$	26,036	2.009	$\pm 1.43 \cdot 10^{-2}$	40,673	2.276	$\pm 6.28 \cdot 10^{-4}$	4,003
16	2.355	$\pm 1.03 \cdot 10^{-3}$	26,513	1.996	$\pm 1.41 \cdot 10^{-2}$	41,233	2.276	$\pm 6.28 \cdot 10^{-4}$	4,003

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	0.822	$\pm 5.09 \cdot 10^{-3}$	34,386	2.426	$\pm 2.46 \cdot 10^{-2}$	8,518	2.426	$\pm 5.66 \cdot 10^{-3}$	21,864
2	0.763	$\pm 3.25 \cdot 10^{-3}$	34,697	2.364	$\pm 1.07 \cdot 10^{-2}$	8,527	2.360	$\pm 3.45 \cdot 10^{-3}$	22,167
3	0.749	$\pm 2.63 \cdot 10^{-3}$	35,089	2.345	$\pm 6.02 \cdot 10^{-3}$	8,534	2.346	$\pm 2.87 \cdot 10^{-3}$	22,551
4	0.738	$\pm 2.12 \cdot 10^{-3}$	34,880	2.338	$\pm 4.90 \cdot 10^{-3}$	8,539	2.331	$\pm 2.20 \cdot 10^{-3}$	22,338
6	0.732	$\pm 1.75 \cdot 10^{-3}$	35,010	2.330	$\pm 2.75 \cdot 10^{-3}$	8,537	2.327	$\pm 2.11 \cdot 10^{-3}$	22,472
8	0.727	$\pm 1.65 \cdot 10^{-3}$	34,316	2.326	$\pm 1.96 \cdot 10^{-3}$	8,520	2.319	$\pm 1.66 \cdot 10^{-3}$	21,793
10	0.726	$\pm 1.51 \cdot 10^{-3}$	34,640	2.326	$\pm 1.87 \cdot 10^{-3}$	8,557	2.319	$\pm 1.53 \cdot 10^{-3}$	22,081
12	0.725	$\pm 1.45 \cdot 10^{-3}$	34,814	2.323	$\pm 1.24 \cdot 10^{-3}$	8,534	2.317	$\pm 1.39 \cdot 10^{-3}$	22,278
14	0.725	$\pm 1.61 \cdot 10^{-3}$	34,560	2.323	$\pm 1.17 \cdot 10^{-3}$	8,525	2.317	$\pm 1.42 \cdot 10^{-3}$	22,035
16	0.724	$\pm 1.74 \cdot 10^{-3}$	35,061	2.321	$\pm 1.65 \cdot 10^{-4}$	8,548	2.314	$\pm 1.19 \cdot 10^{-3}$	22,510

Table B.2: Scalability experiment results with 16 PEs and 500 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	3.747	$\pm 5.68 \cdot 10^{-3}$	26,670	3.574	$\pm 2.38 \cdot 10^{-2}$	41,022	3.530	$\pm 2.14 \cdot 10^{-3}$	4,003
2	3.663	$\pm 3.55 \cdot 10^{-3}$	26,174	3.462	$\pm 2.13 \cdot 10^{-2}$	40,546	3.530	$\pm 2.14 \cdot 10^{-3}$	4,003
3	3.641	$\pm 2.81 \cdot 10^{-3}$	26,469	3.413	$\pm 2.05 \cdot 10^{-2}$	40,861	3.530	$\pm 2.14 \cdot 10^{-3}$	4,003
4	3.632	$\pm 2.36 \cdot 10^{-3}$	27,116	3.396	$\pm 2.02 \cdot 10^{-2}$	41,516	3.530	$\pm 2.14 \cdot 10^{-3}$	4,003
6	3.621	$\pm 1.96 \cdot 10^{-3}$	26,238	3.395	$\pm 2.03 \cdot 10^{-2}$	40,619	3.530	$\pm 2.14 \cdot 10^{-3}$	4,003
8	3.616	$\pm 1.82 \cdot 10^{-3}$	26,426	3.389	$\pm 2.02 \cdot 10^{-2}$	40,780	3.530	$\pm 2.14 \cdot 10^{-3}$	4,003
10	3.613	$\pm 1.61 \cdot 10^{-3}$	26,208	3.415	$\pm 2.05 \cdot 10^{-2}$	40,357	3.535	$\pm 2.99 \cdot 10^{-3}$	4,003
12	3.612	$\pm 1.55 \cdot 10^{-3}$	26,702	3.398	$\pm 2.00 \cdot 10^{-2}$	40,878	3.535	$\pm 2.99 \cdot 10^{-3}$	4,003
14	3.612	$\pm 1.36 \cdot 10^{-3}$	26,579	3.397	$\pm 2.01 \cdot 10^{-2}$	40,707	3.535	$\pm 2.99 \cdot 10^{-3}$	4,003
16	3.608	$\pm 1.25 \cdot 10^{-3}$	26,015	3.398	$\pm 2.01 \cdot 10^{-2}$	40,211	3.535	$\pm 2.99 \cdot 10^{-3}$	4,003

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	2.174	$\pm 8.54 \cdot 10^{-3}$	35,287	3.728	$\pm 2.78 \cdot 10^{-2}$	8,617	3.730	$\pm 6.60 \cdot 10^{-3}$	22,667
2	2.065	$\pm 6.12 \cdot 10^{-3}$	34,790	3.632	$\pm 1.08 \cdot 10^{-2}$	8,617	3.633	$\pm 4.13 \cdot 10^{-3}$	22,171
3	2.037	$\pm 5.36 \cdot 10^{-3}$	35,062	3.608	$\pm 7.01 \cdot 10^{-3}$	8,593	3.606	$\pm 3.27 \cdot 10^{-3}$	22,466
4	2.025	$\pm 5.10 \cdot 10^{-3}$	35,711	3.602	$\pm 6.07 \cdot 10^{-3}$	8,595	3.595	$\pm 2.72 \cdot 10^{-3}$	23,113
6	2.011	$\pm 4.79 \cdot 10^{-3}$	34,831	3.585	$\pm 3.31 \cdot 10^{-3}$	8,593	3.582	$\pm 2.27 \cdot 10^{-3}$	22,235
8	2.008	$\pm 5.03 \cdot 10^{-3}$	34,998	3.582	$\pm 3.27 \cdot 10^{-3}$	8,574	3.577	$\pm 2.14 \cdot 10^{-3}$	22,425
10	2.003	$\pm 5.00 \cdot 10^{-3}$	34,805	3.577	$\pm 1.93 \cdot 10^{-3}$	8,597	3.572	$\pm 1.82 \cdot 10^{-3}$	22,205
12	2.004	$\pm 4.99 \cdot 10^{-3}$	35,294	3.576	$\pm 1.82 \cdot 10^{-3}$	8,594	3.571	$\pm 1.79 \cdot 10^{-3}$	22,701
14	1.999	$\pm 4.90 \cdot 10^{-3}$	35,155	3.574	$\pm 1.51 \cdot 10^{-3}$	8,577	3.571	$\pm 1.50 \cdot 10^{-3}$	22,576
16	1.996	$\pm 5.05 \cdot 10^{-3}$	34,636	3.572	$\pm 7.78 \cdot 10^{-4}$	8,622	3.567	$\pm 1.36 \cdot 10^{-3}$	22,012

Table B.3: Scalability experiment results with 16 PEs and 1000 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	5.393	$\pm 1.65 \cdot 10^{-2}$	27,019	5.433	$\pm 3.60 \cdot 10^{-2}$	41,495	4.799	$\pm 6.21 \cdot 10^{-3}$	4,003
2	5.049	$\pm 8.32 \cdot 10^{-3}$	27,110	4.981	$\pm 2.95 \cdot 10^{-2}$	41,713	4.798	$\pm 6.24 \cdot 10^{-3}$	4,003
3	4.963	$\pm 6.09 \cdot 10^{-3}$	26,840	4.887	$\pm 2.84 \cdot 10^{-2}$	41,401	4.798	$\pm 6.28 \cdot 10^{-3}$	4,003
4	4.934	$\pm 4.72 \cdot 10^{-3}$	26,803	4.843	$\pm 2.78 \cdot 10^{-2}$	41,393	4.798	$\pm 6.24 \cdot 10^{-3}$	4,003
6	4.903	$\pm 3.69 \cdot 10^{-3}$	26,999	4.813	$\pm 2.73 \cdot 10^{-2}$	41,578	4.798	$\pm 6.24 \cdot 10^{-3}$	4,003
8	4.892	$\pm 3.54 \cdot 10^{-3}$	26,843	4.812	$\pm 2.73 \cdot 10^{-2}$	41,334	4.799	$\pm 6.21 \cdot 10^{-3}$	4,003
10	4.888	$\pm 3.08 \cdot 10^{-3}$	27,080	4.768	$\pm 2.72 \cdot 10^{-2}$	41,686	4.799	$\pm 6.42 \cdot 10^{-3}$	4,003
12	4.878	$\pm 2.65 \cdot 10^{-3}$	26,775	4.764	$\pm 2.72 \cdot 10^{-2}$	41,366	4.799	$\pm 6.42 \cdot 10^{-3}$	4,003
14	4.876	$\pm 2.61 \cdot 10^{-3}$	26,858	4.771	$\pm 2.73 \cdot 10^{-2}$	41,514	4.799	$\pm 6.42 \cdot 10^{-3}$	4,003
16	4.870	$\pm 2.23 \cdot 10^{-3}$	26,991	4.756	$\pm 2.72 \cdot 10^{-2}$	41,616	4.799	$\pm 6.42 \cdot 10^{-3}$	4,003

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	3.947	$\pm 1.97 \cdot 10^{-2}$	35,743	5.230	$\pm 3.74 \cdot 10^{-2}$	8,724	5.442	$\pm 1.91 \cdot 10^{-2}$	23,016
2	3.516	$\pm 1.23 \cdot 10^{-2}$	35,808	4.971	$\pm 1.66 \cdot 10^{-2}$	8,698	5.038	$\pm 9.63 \cdot 10^{-3}$	23,107
3	3.420	$\pm 1.10 \cdot 10^{-2}$	35,541	4.911	$\pm 1.09 \cdot 10^{-2}$	8,701	4.937	$\pm 7.04 \cdot 10^{-3}$	22,837
4	3.380	$\pm 1.03 \cdot 10^{-2}$	35,515	4.896	$\pm 9.14 \cdot 10^{-3}$	8,712	4.904	$\pm 5.41 \cdot 10^{-3}$	22,800
6	3.346	$\pm 9.53 \cdot 10^{-3}$	35,725	4.860	$\pm 5.90 \cdot 10^{-3}$	8,726	4.867	$\pm 4.18 \cdot 10^{-3}$	22,996
8	3.324	$\pm 8.70 \cdot 10^{-3}$	35,570	4.844	$\pm 4.36 \cdot 10^{-3}$	8,727	4.854	$\pm 4.01 \cdot 10^{-3}$	22,840
10	3.314	$\pm 9.64 \cdot 10^{-3}$	35,765	4.844	$\pm 4.53 \cdot 10^{-3}$	8,685	4.849	$\pm 3.43 \cdot 10^{-3}$	23,077
12	3.313	$\pm 9.65 \cdot 10^{-3}$	35,424	4.838	$\pm 3.85 \cdot 10^{-3}$	8,649	4.837	$\pm 2.90 \cdot 10^{-3}$	22,772
14	3.309	$\pm 9.70 \cdot 10^{-3}$	35,553	4.835	$\pm 3.64 \cdot 10^{-3}$	8,695	4.835	$\pm 2.85 \cdot 10^{-3}$	22,855
16	3.304	$\pm 9.82 \cdot 10^{-3}$	35,651	4.828	$\pm 2.60 \cdot 10^{-3}$	8,660	4.828	$\pm 2.36 \cdot 10^{-3}$	22,988

Table B.4: Scalability experiment results with 16 PEs and 2000 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	25.261	$\pm 2.05 \cdot 10^{-1}$	41,070	31.973	$\pm 3.09 \cdot 10^{-1}$	58,960	7.525	$\pm 8.64 \cdot 10^{-2}$	4,016
2	8.311	$\pm 2.71 \cdot 10^{-2}$	27,621	8.588	$\pm 5.37 \cdot 10^{-2}$	42,712	7.371	$\pm 1.75 \cdot 10^{-2}$	4,003
3	7.970	$\pm 2.07 \cdot 10^{-2}$	28,041	8.270	$\pm 5.04 \cdot 10^{-2}$	43,080	7.371	$\pm 1.75 \cdot 10^{-2}$	4,003
4	7.733	$\pm 1.46 \cdot 10^{-2}$	27,595	7.939	$\pm 4.64 \cdot 10^{-2}$	42,616	7.371	$\pm 1.75 \cdot 10^{-2}$	4,003
6	7.588	$\pm 1.20 \cdot 10^{-2}$	26,788	7.769	$\pm 4.51 \cdot 10^{-2}$	41,822	7.371	$\pm 1.75 \cdot 10^{-2}$	4,003
8	7.536	$\pm 1.14 \cdot 10^{-2}$	27,688	7.721	$\pm 4.45 \cdot 10^{-2}$	42,685	7.371	$\pm 1.75 \cdot 10^{-2}$	4,003
10	7.485	$\pm 8.21 \cdot 10^{-3}$	27,453	7.722	$\pm 4.43 \cdot 10^{-2}$	42,253	7.388	$\pm 1.91 \cdot 10^{-2}$	4,003
12	7.506	$\pm 9.86 \cdot 10^{-3}$	27,863	7.741	$\pm 4.43 \cdot 10^{-2}$	42,655	7.388	$\pm 1.91 \cdot 10^{-2}$	4,003
14	7.445	$\pm 6.97 \cdot 10^{-3}$	27,507	7.658	$\pm 4.35 \cdot 10^{-2}$	42,328	7.388	$\pm 1.91 \cdot 10^{-2}$	4,003
16	7.419	$\pm 5.83 \cdot 10^{-3}$	27,755	7.626	$\pm 4.33 \cdot 10^{-2}$	42,587	7.388	$\pm 1.91 \cdot 10^{-2}$	4,003

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	24.407	$\pm 1.88 \cdot 10^{-1}$	52,855	20.102	$\pm 3.56 \cdot 10^{-1}$	11,785	27.132	$\pm 2.18 \cdot 10^{-1}$	37,054
2	6.952	$\pm 3.23 \cdot 10^{-2}$	36,573	7.948	$\pm 3.90 \cdot 10^{-2}$	8,953	8.416	$\pm 3.11 \cdot 10^{-2}$	23,618
3	6.612	$\pm 2.95 \cdot 10^{-2}$	36,964	7.704	$\pm 2.94 \cdot 10^{-2}$	8,924	8.015	$\pm 2.38 \cdot 10^{-2}$	24,038
4	6.301	$\pm 2.39 \cdot 10^{-2}$	36,504	7.602	$\pm 2.23 \cdot 10^{-2}$	8,911	7.740	$\pm 1.67 \cdot 10^{-2}$	23,592
6	6.132	$\pm 2.16 \cdot 10^{-2}$	35,703	7.463	$\pm 1.53 \cdot 10^{-2}$	8,916	7.571	$\pm 1.37 \cdot 10^{-2}$	22,787
8	6.079	$\pm 2.13 \cdot 10^{-2}$	36,559	7.433	$\pm 1.35 \cdot 10^{-2}$	8,872	7.510	$\pm 1.29 \cdot 10^{-2}$	23,685
10	6.071	$\pm 2.25 \cdot 10^{-2}$	36,328	7.409	$\pm 1.21 \cdot 10^{-2}$	8,876	7.448	$\pm 9.04 \cdot 10^{-3}$	23,450
12	6.068	$\pm 2.22 \cdot 10^{-2}$	36,733	7.394	$\pm 1.09 \cdot 10^{-2}$	8,871	7.471	$\pm 1.10 \cdot 10^{-2}$	23,860
14	6.015	$\pm 2.13 \cdot 10^{-2}$	36,399	7.379	$\pm 9.03 \cdot 10^{-3}$	8,893	7.401	$\pm 7.48 \cdot 10^{-3}$	23,504
16	5.992	$\pm 2.16 \cdot 10^{-2}$	36,623	7.361	$\pm 7.50 \cdot 10^{-3}$	8,870	7.369	$\pm 6.06 \cdot 10^{-3}$	23,754

Table B.5: Scalability experiment results with 32 PEs and 0 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	5.044	$\pm 3.20 \cdot 10^{-2}$	28,566	3.452	$\pm 4.53 \cdot 10^{-2}$	49,512	3.739	$\pm 3.41 \cdot 10^{-4}$	4,017
2	4.192	$\pm 1.36 \cdot 10^{-2}$	27,503	2.611	$\pm 2.80 \cdot 10^{-2}$	48,511	3.739	$\pm 3.41 \cdot 10^{-4}$	4,017
3	3.975	$\pm 8.30 \cdot 10^{-3}$	27,382	2.406	$\pm 2.37 \cdot 10^{-2}$	48,386	3.739	$\pm 3.41 \cdot 10^{-4}$	4,017
4	3.929	$\pm 7.16 \cdot 10^{-3}$	26,505	2.344	$\pm 2.18 \cdot 10^{-2}$	47,471	3.739	$\pm 3.41 \cdot 10^{-4}$	4,017
6	3.883	$\pm 5.77 \cdot 10^{-3}$	26,953	2.278	$\pm 2.04 \cdot 10^{-2}$	47,906	3.739	$\pm 3.41 \cdot 10^{-4}$	4,017
8	3.856	$\pm 4.85 \cdot 10^{-3}$	26,830	2.270	$\pm 2.07 \cdot 10^{-2}$	47,833	3.739	$\pm 3.41 \cdot 10^{-4}$	4,017
10	3.883	$\pm 6.18 \cdot 10^{-3}$	26,593	2.306	$\pm 2.11 \cdot 10^{-2}$	47,688	3.740	$\pm 4.31 \cdot 10^{-4}$	4,017
12	3.833	$\pm 4.09 \cdot 10^{-3}$	26,803	2.264	$\pm 2.05 \cdot 10^{-2}$	47,969	3.740	$\pm 4.31 \cdot 10^{-4}$	4,017
14	3.827	$\pm 3.80 \cdot 10^{-3}$	27,042	2.249	$\pm 2.07 \cdot 10^{-2}$	48,169	3.740	$\pm 4.31 \cdot 10^{-4}$	4,017
16	3.821	$\pm 3.41 \cdot 10^{-3}$	27,318	2.239	$\pm 2.05 \cdot 10^{-2}$	48,444	3.740	$\pm 4.31 \cdot 10^{-4}$	4,017
18	3.815	$\pm 3.39 \cdot 10^{-3}$	26,543	2.237	$\pm 2.04 \cdot 10^{-2}$	47,633	3.740	$\pm 4.31 \cdot 10^{-4}$	4,017
20	3.819	$\pm 3.53 \cdot 10^{-3}$	27,509	2.219	$\pm 1.96 \cdot 10^{-2}$	48,586	3.740	$\pm 4.31 \cdot 10^{-4}$	4,017
22	3.819	$\pm 3.30 \cdot 10^{-3}$	27,278	2.236	$\pm 2.03 \cdot 10^{-2}$	48,390	3.740	$\pm 4.31 \cdot 10^{-4}$	4,017
24	3.813	$\pm 3.24 \cdot 10^{-3}$	27,258	2.215	$\pm 1.98 \cdot 10^{-2}$	48,398	3.740	$\pm 4.31 \cdot 10^{-4}$	4,017
26	3.810	$\pm 3.12 \cdot 10^{-3}$	26,498	2.241	$\pm 2.07 \cdot 10^{-2}$	47,651	3.740	$\pm 4.31 \cdot 10^{-4}$	4,017
28	3.806	$\pm 3.06 \cdot 10^{-3}$	26,430	2.242	$\pm 2.08 \cdot 10^{-2}$	47,535	3.740	$\pm 4.31 \cdot 10^{-4}$	4,017
30	3.807	$\pm 2.90 \cdot 10^{-3}$	26,669	2.230	$\pm 2.03 \cdot 10^{-2}$	47,761	3.740	$\pm 4.31 \cdot 10^{-4}$	4,017
32	3.804	$\pm 2.80 \cdot 10^{-3}$	26,620	2.224	$\pm 1.99 \cdot 10^{-2}$	47,662	3.740	$\pm 4.31 \cdot 10^{-4}$	4,017

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	2.181	$\pm 3.07 \cdot 10^{-2}$	37,308	4.893	$\pm 1.26 \cdot 10^{-1}$	8,742	5.204	$\pm 3.67 \cdot 10^{-2}$	24,549
2	1.273	$\pm 1.58 \cdot 10^{-2}$	36,167	4.203	$\pm 5.69 \cdot 10^{-2}$	8,665	4.215	$\pm 1.58 \cdot 10^{-2}$	23,486
3	1.054	$\pm 1.02 \cdot 10^{-2}$	36,067	3.980	$\pm 3.60 \cdot 10^{-2}$	8,686	3.961	$\pm 9.66 \cdot 10^{-3}$	23,365
4	0.979	$\pm 8.31 \cdot 10^{-3}$	35,144	3.946	$\pm 2.58 \cdot 10^{-2}$	8,640	3.908	$\pm 8.40 \cdot 10^{-3}$	22,488
6	0.926	$\pm 6.71 \cdot 10^{-3}$	35,578	3.864	$\pm 1.48 \cdot 10^{-2}$	8,625	3.853	$\pm 6.75 \cdot 10^{-3}$	22,936
8	0.903	$\pm 7.47 \cdot 10^{-3}$	35,468	3.837	$\pm 1.03 \cdot 10^{-2}$	8,638	3.822	$\pm 5.68 \cdot 10^{-3}$	22,813
10	0.919	$\pm 6.69 \cdot 10^{-3}$	35,197	3.901	$\pm 1.25 \cdot 10^{-2}$	8,604	3.854	$\pm 7.25 \cdot 10^{-3}$	22,576
12	0.879	$\pm 5.17 \cdot 10^{-3}$	35,422	3.809	$\pm 5.85 \cdot 10^{-3}$	8,621	3.795	$\pm 4.80 \cdot 10^{-3}$	22,786
14	0.869	$\pm 5.12 \cdot 10^{-3}$	35,670	3.807	$\pm 5.16 \cdot 10^{-3}$	8,628	3.788	$\pm 4.45 \cdot 10^{-3}$	23,025
16	0.868	$\pm 5.46 \cdot 10^{-3}$	35,945	3.801	$\pm 4.16 \cdot 10^{-3}$	8,627	3.781	$\pm 3.99 \cdot 10^{-3}$	23,301
18	0.863	$\pm 5.45 \cdot 10^{-3}$	35,125	3.797	$\pm 3.49 \cdot 10^{-3}$	8,582	3.774	$\pm 3.99 \cdot 10^{-3}$	22,526
20	0.865	$\pm 4.81 \cdot 10^{-3}$	36,108	3.817	$\pm 5.23 \cdot 10^{-3}$	8,600	3.779	$\pm 4.12 \cdot 10^{-3}$	23,492
22	0.862	$\pm 5.44 \cdot 10^{-3}$	35,878	3.795	$\pm 3.29 \cdot 10^{-3}$	8,600	3.778	$\pm 3.86 \cdot 10^{-3}$	23,261
24	0.854	$\pm 4.31 \cdot 10^{-3}$	35,849	3.793	$\pm 2.88 \cdot 10^{-3}$	8,592	3.771	$\pm 3.80 \cdot 10^{-3}$	23,241
26	0.854	$\pm 4.76 \cdot 10^{-3}$	35,127	3.791	$\pm 2.43 \cdot 10^{-3}$	8,629	3.768	$\pm 3.67 \cdot 10^{-3}$	22,481
28	0.849	$\pm 4.67 \cdot 10^{-3}$	35,028	3.788	$\pm 1.95 \cdot 10^{-3}$	8,599	3.763	$\pm 3.61 \cdot 10^{-3}$	22,413
30	0.850	$\pm 4.58 \cdot 10^{-3}$	35,256	3.790	$\pm 2.27 \cdot 10^{-3}$	8,588	3.765	$\pm 3.41 \cdot 10^{-3}$	22,652
32	0.847	$\pm 5.08 \cdot 10^{-3}$	35,212	3.785	$\pm 2.12 \cdot 10^{-4}$	8,592	3.761	$\pm 3.30 \cdot 10^{-3}$	22,603

Table B.6: Scalability experiment results with 32 PEs and 500 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	`activate_task`			`__schedule`			`__fw_admit`		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	10.342	$\pm 1.08 \cdot 10^{-1}$	32,674	8.940	$\pm 1.26 \cdot 10^{-1}$	55,308	4.997	$\pm 2.78 \cdot 10^{-3}$	4,017
2	5.498	$\pm 1.46 \cdot 10^{-2}$	27,499	3.877	$\pm 3.69 \cdot 10^{-2}$	49,646	4.995	$\pm 2.43 \cdot 10^{-3}$	4,017
3	5.263	$\pm 9.13 \cdot 10^{-3}$	26,926	3.618	$\pm 3.28 \cdot 10^{-2}$	49,134	4.995	$\pm 2.21 \cdot 10^{-3}$	4,017
4	5.232	$\pm 8.56 \cdot 10^{-3}$	27,563	3.544	$\pm 3.02 \cdot 10^{-2}$	49,778	4.995	$\pm 2.21 \cdot 10^{-3}$	4,017
6	5.151	$\pm 6.06 \cdot 10^{-3}$	27,681	3.479	$\pm 2.92 \cdot 10^{-2}$	49,869	4.995	$\pm 2.21 \cdot 10^{-3}$	4,017
8	5.111	$\pm 4.96 \cdot 10^{-3}$	27,174	3.452	$\pm 2.92 \cdot 10^{-2}$	49,323	4.995	$\pm 2.21 \cdot 10^{-3}$	4,017
10	5.125	$\pm 5.24 \cdot 10^{-3}$	27,570	3.531	$\pm 2.86 \cdot 10^{-2}$	48,318	4.996	$\pm 2.50 \cdot 10^{-3}$	4,017
12	5.096	$\pm 4.26 \cdot 10^{-3}$	26,996	3.497	$\pm 2.77 \cdot 10^{-2}$	47,612	4.996	$\pm 2.50 \cdot 10^{-3}$	4,017
14	5.085	$\pm 3.89 \cdot 10^{-3}$	27,467	3.492	$\pm 2.80 \cdot 10^{-2}$	48,216	4.996	$\pm 2.50 \cdot 10^{-3}$	4,017
16	5.074	$\pm 3.44 \cdot 10^{-3}$	27,055	3.472	$\pm 2.75 \cdot 10^{-2}$	47,803	4.996	$\pm 2.50 \cdot 10^{-3}$	4,017
18	5.076	$\pm 3.61 \cdot 10^{-3}$	26,992	3.474	$\pm 2.80 \cdot 10^{-2}$	47,625	4.996	$\pm 2.50 \cdot 10^{-3}$	4,017
20	5.075	$\pm 3.75 \cdot 10^{-3}$	26,732	3.488	$\pm 2.80 \cdot 10^{-2}$	47,425	4.996	$\pm 2.50 \cdot 10^{-3}$	4,017
22	5.071	$\pm 3.44 \cdot 10^{-3}$	26,704	3.472	$\pm 2.78 \cdot 10^{-2}$	47,505	4.996	$\pm 2.50 \cdot 10^{-3}$	4,017
24	5.066	$\pm 3.21 \cdot 10^{-3}$	27,003	3.476	$\pm 2.77 \cdot 10^{-2}$	47,652	4.996	$\pm 2.50 \cdot 10^{-3}$	4,017
26	5.067	$\pm 3.13 \cdot 10^{-3}$	26,503	3.473	$\pm 2.77 \cdot 10^{-2}$	47,164	4.996	$\pm 2.50 \cdot 10^{-3}$	4,017
28	5.066	$\pm 3.12 \cdot 10^{-3}$	27,673	3.477	$\pm 2.74 \cdot 10^{-2}$	48,374	4.996	$\pm 2.50 \cdot 10^{-3}$	4,017
30	5.064	$\pm 3.02 \cdot 10^{-3}$	27,409	3.458	$\pm 2.74 \cdot 10^{-2}$	48,103	4.996	$\pm 2.50 \cdot 10^{-3}$	4,017
32	5.058	$\pm 2.92 \cdot 10^{-3}$	26,926	3.481	$\pm 2.83 \cdot 10^{-2}$	47,695	4.996	$\pm 2.50 \cdot 10^{-3}$	4,017

Pipes	`__fw_dispatch`			`__fw_relinquish`			`__fw_unblock`		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	7.777	$\pm 1.03 \cdot 10^{-1}$	41,850	7.810	$\pm 1.93 \cdot 10^{-1}$	9,176	11.039	$\pm 1.20 \cdot 10^{-1}$	28,657
2	2.675	$\pm 1.95 \cdot 10^{-2}$	36,265	5.564	$\pm 6.30 \cdot 10^{-2}$	8,766	5.530	$\pm 1.69 \cdot 10^{-2}$	23,482
3	2.420	$\pm 1.72 \cdot 10^{-2}$	35,636	5.286	$\pm 3.95 \cdot 10^{-2}$	8,710	5.256	$\pm 1.06 \cdot 10^{-2}$	22,909
4	2.362	$\pm 1.58 \cdot 10^{-2}$	36,311	5.246	$\pm 2.96 \cdot 10^{-2}$	8,749	5.218	$\pm 9.96 \cdot 10^{-3}$	23,546
6	2.269	$\pm 1.43 \cdot 10^{-2}$	36,385	5.138	$\pm 1.77 \cdot 10^{-2}$	8,704	5.123	$\pm 7.05 \cdot 10^{-3}$	23,664
8	2.226	$\pm 1.42 \cdot 10^{-2}$	35,875	5.106	$\pm 1.37 \cdot 10^{-2}$	8,702	5.076	$\pm 5.79 \cdot 10^{-3}$	23,157
10	2.216	$\pm 1.33 \cdot 10^{-2}$	36,296	5.169	$\pm 1.74 \cdot 10^{-2}$	8,726	5.092	$\pm 6.10 \cdot 10^{-3}$	23,553
12	2.191	$\pm 1.22 \cdot 10^{-2}$	35,703	5.079	$\pm 1.04 \cdot 10^{-2}$	8,708	5.058	$\pm 4.98 \cdot 10^{-3}$	22,979
14	2.188	$\pm 1.27 \cdot 10^{-2}$	36,177	5.072	$\pm 9.89 \cdot 10^{-3}$	8,711	5.045	$\pm 4.53 \cdot 10^{-3}$	23,450
16	2.171	$\pm 1.25 \cdot 10^{-2}$	35,772	5.061	$\pm 8.99 \cdot 10^{-3}$	8,718	5.033	$\pm 4.01 \cdot 10^{-3}$	23,038
18	2.172	$\pm 1.30 \cdot 10^{-2}$	35,663	5.056	$\pm 8.23 \cdot 10^{-3}$	8,674	5.034	$\pm 4.26 \cdot 10^{-3}$	22,977
20	2.176	$\pm 1.25 \cdot 10^{-2}$	35,413	5.082	$\pm 8.72 \cdot 10^{-3}$	8,681	5.034	$\pm 4.39 \cdot 10^{-3}$	22,715
22	2.173	$\pm 1.37 \cdot 10^{-2}$	35,435	5.053	$\pm 8.20 \cdot 10^{-3}$	8,732	5.030	$\pm 4.02 \cdot 10^{-3}$	22,687
24	2.164	$\pm 1.21 \cdot 10^{-2}$	35,693	5.052	$\pm 8.26 \cdot 10^{-3}$	8,690	5.024	$\pm 3.74 \cdot 10^{-3}$	22,986
26	2.163	$\pm 1.27 \cdot 10^{-2}$	35,193	5.051	$\pm 8.17 \cdot 10^{-3}$	8,690	5.025	$\pm 3.66 \cdot 10^{-3}$	22,486
28	2.167	$\pm 1.29 \cdot 10^{-2}$	36,410	5.048	$\pm 7.92 \cdot 10^{-3}$	8,738	5.024	$\pm 3.62 \cdot 10^{-3}$	23,656
30	2.163	$\pm 1.33 \cdot 10^{-2}$	36,097	5.048	$\pm 7.97 \cdot 10^{-3}$	8,688	5.021	$\pm 3.51 \cdot 10^{-3}$	23,392
32	2.160	$\pm 1.36 \cdot 10^{-2}$	35,625	5.042	$\pm 7.65 \cdot 10^{-3}$	8,700	5.014	$\pm 3.40 \cdot 10^{-3}$	22,909

Table B.7: Scalability experiment results with 32 PEs and 1000 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	46.418	$\pm 2.70 \cdot 10^{-1}$	62,307	61.809	$\pm 4.30 \cdot 10^{-1}$	94,469	6.260	$\pm 5.91 \cdot 10^{-3}$	4,017
2	7.371	$\pm 3.53 \cdot 10^{-2}$	28,325	5.916	$\pm 6.08 \cdot 10^{-2}$	49,705	6.261	$\pm 6.12 \cdot 10^{-3}$	4,017
3	6.642	$\pm 1.31 \cdot 10^{-2}$	27,103	5.082	$\pm 4.35 \cdot 10^{-2}$	48,411	6.261	$\pm 6.10 \cdot 10^{-3}$	4,017
4	6.597	$\pm 1.30 \cdot 10^{-2}$	27,842	4.959	$\pm 4.19 \cdot 10^{-2}$	49,796	6.263	$\pm 6.28 \cdot 10^{-3}$	4,017
6	6.491	$\pm 9.96 \cdot 10^{-3}$	27,657	4.892	$\pm 3.95 \cdot 10^{-2}$	48,828	6.261	$\pm 6.10 \cdot 10^{-3}$	4,017
8	6.400	$\pm 6.59 \cdot 10^{-3}$	27,281	4.781	$\pm 3.85 \cdot 10^{-2}$	48,598	6.261	$\pm 6.12 \cdot 10^{-3}$	4,017
10	6.381	$\pm 5.82 \cdot 10^{-3}$	26,467	4.733	$\pm 3.86 \cdot 10^{-2}$	48,202	6.261	$\pm 6.17 \cdot 10^{-3}$	4,017
12	6.361	$\pm 5.08 \cdot 10^{-3}$	27,179	4.679	$\pm 3.70 \cdot 10^{-2}$	48,981	6.261	$\pm 6.17 \cdot 10^{-3}$	4,017
14	6.398	$\pm 6.73 \cdot 10^{-3}$	28,727	4.776	$\pm 3.69 \cdot 10^{-2}$	50,506	6.261	$\pm 6.17 \cdot 10^{-3}$	4,017
16	6.350	$\pm 4.66 \cdot 10^{-3}$	27,382	4.690	$\pm 3.71 \cdot 10^{-2}$	48,851	6.261	$\pm 5.98 \cdot 10^{-3}$	4,017
18	6.344	$\pm 4.51 \cdot 10^{-3}$	27,504	4.668	$\pm 3.67 \cdot 10^{-2}$	49,319	6.261	$\pm 6.17 \cdot 10^{-3}$	4,017
20	6.344	$\pm 5.17 \cdot 10^{-3}$	27,349	4.637	$\pm 3.60 \cdot 10^{-2}$	49,176	6.261	$\pm 6.17 \cdot 10^{-3}$	4,017
22	6.343	$\pm 4.23 \cdot 10^{-3}$	27,011	4.663	$\pm 3.70 \cdot 10^{-2}$	48,773	6.261	$\pm 6.17 \cdot 10^{-3}$	4,017
24	6.339	$\pm 5.03 \cdot 10^{-3}$	28,056	4.661	$\pm 3.59 \cdot 10^{-2}$	49,782	6.261	$\pm 6.17 \cdot 10^{-3}$	4,017
26	6.324	$\pm 3.83 \cdot 10^{-3}$	26,910	4.639	$\pm 3.68 \cdot 10^{-2}$	48,667	6.261	$\pm 6.17 \cdot 10^{-3}$	4,017
28	6.333	$\pm 3.87 \cdot 10^{-3}$	27,753	4.676	$\pm 3.68 \cdot 10^{-2}$	49,623	6.261	$\pm 6.17 \cdot 10^{-3}$	4,017
30	6.326	$\pm 3.54 \cdot 10^{-3}$	27,329	4.642	$\pm 3.66 \cdot 10^{-2}$	49,140	6.261	$\pm 6.17 \cdot 10^{-3}$	4,017
32	6.320	$\pm 3.44 \cdot 10^{-3}$	27,265	4.676	$\pm 3.70 \cdot 10^{-2}$	48,725	6.261	$\pm 5.98 \cdot 10^{-3}$	4,017

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	46.008	$\pm 2.43 \cdot 10^{-1}$	81,361	46.186	$\pm 5.13 \cdot 10^{-1}$	19,055	49.136	$\pm 2.75 \cdot 10^{-1}$	58,290
2	4.806	$\pm 4.55 \cdot 10^{-2}$	37,219	7.186	$\pm 8.52 \cdot 10^{-2}$	8,894	7.501	$\pm 4.07 \cdot 10^{-2}$	24,308
3	3.954	$\pm 2.71 \cdot 10^{-2}$	35,887	6.676	$\pm 4.68 \cdot 10^{-2}$	8,785	6.654	$\pm 1.52 \cdot 10^{-2}$	23,086
4	3.888	$\pm 2.83 \cdot 10^{-2}$	36,672	6.561	$\pm 3.60 \cdot 10^{-2}$	8,832	6.598	$\pm 1.51 \cdot 10^{-2}$	23,827
6	3.725	$\pm 2.47 \cdot 10^{-2}$	36,422	6.427	$\pm 1.98 \cdot 10^{-2}$	8,765	6.475	$\pm 1.16 \cdot 10^{-2}$	23,640
8	3.639	$\pm 2.53 \cdot 10^{-2}$	36,072	6.389	$\pm 1.47 \cdot 10^{-2}$	8,791	6.370	$\pm 7.63 \cdot 10^{-3}$	23,264
10	3.593	$\pm 2.41 \cdot 10^{-2}$	35,248	6.441	$\pm 1.39 \cdot 10^{-2}$	8,781	6.348	$\pm 6.76 \cdot 10^{-3}$	22,450
12	3.576	$\pm 2.34 \cdot 10^{-2}$	35,963	6.347	$\pm 1.04 \cdot 10^{-2}$	8,786	6.324	$\pm 5.90 \cdot 10^{-3}$	23,164
14	3.610	$\pm 2.25 \cdot 10^{-2}$	37,540	6.337	$\pm 7.77 \cdot 10^{-3}$	8,814	6.367	$\pm 7.74 \cdot 10^{-3}$	24,710
16	3.563	$\pm 2.41 \cdot 10^{-2}$	36,152	6.325	$\pm 6.36 \cdot 10^{-3}$	8,770	6.311	$\pm 5.35 \cdot 10^{-3}$	23,365
18	3.551	$\pm 2.32 \cdot 10^{-2}$	36,307	6.320	$\pm 5.51 \cdot 10^{-3}$	8,805	6.304	$\pm 5.17 \cdot 10^{-3}$	23,487
20	3.527	$\pm 2.17 \cdot 10^{-2}$	36,117	6.338	$\pm 6.93 \cdot 10^{-3}$	8,769	6.303	$\pm 5.96 \cdot 10^{-3}$	23,332
22	3.531	$\pm 2.36 \cdot 10^{-2}$	35,785	6.312	$\pm 5.88 \cdot 10^{-3}$	8,775	6.302	$\pm 4.85 \cdot 10^{-3}$	22,994
24	3.535	$\pm 2.20 \cdot 10^{-2}$	36,858	6.312	$\pm 5.50 \cdot 10^{-3}$	8,803	6.298	$\pm 5.77 \cdot 10^{-3}$	24,039
26	3.530	$\pm 2.37 \cdot 10^{-2}$	35,657	6.305	$\pm 4.58 \cdot 10^{-3}$	8,747	6.280	$\pm 4.37 \cdot 10^{-3}$	22,893
28	3.539	$\pm 2.38 \cdot 10^{-2}$	36,570	6.300	$\pm 3.65 \cdot 10^{-3}$	8,817	6.291	$\pm 4.40 \cdot 10^{-3}$	23,736
30	3.520	$\pm 2.37 \cdot 10^{-2}$	36,150	6.302	$\pm 3.76 \cdot 10^{-3}$	8,823	6.282	$\pm 4.08 \cdot 10^{-3}$	23,314
32	3.527	$\pm 2.42 \cdot 10^{-2}$	36,033	6.297	$\pm 4.15 \cdot 10^{-3}$	8,768	6.276	$\pm 3.90 \cdot 10^{-3}$	23,248

Table B.8: Scalability experiment results with 32 PEs and 2000 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	118.633	$\pm 1.78 \cdot 10^{-1}$	162,073	189.612	$\pm 3.96 \cdot 10^{-1}$	268,897	9.248	$\pm 2.24 \cdot 10^{-1}$	4,030
2	30.794	$\pm 2.20 \cdot 10^{-1}$	51,646	38.308	$\pm 3.23 \cdot 10^{-1}$	78,715	8.833	$\pm 1.76 \cdot 10^{-2}$	4,017
3	14.941	$\pm 1.20 \cdot 10^{-1}$	36,001	14.846	$\pm 1.58 \cdot 10^{-1}$	59,136	8.825	$\pm 1.70 \cdot 10^{-2}$	4,017
4	10.307	$\pm 4.56 \cdot 10^{-2}$	29,485	8.982	$\pm 7.93 \cdot 10^{-2}$	51,506	8.837	$\pm 1.74 \cdot 10^{-2}$	4,017
6	9.525	$\pm 2.96 \cdot 10^{-2}$	28,187	8.028	$\pm 6.97 \cdot 10^{-2}$	50,437	8.823	$\pm 1.68 \cdot 10^{-2}$	4,017
8	9.247	$\pm 1.85 \cdot 10^{-2}$	28,495	7.793	$\pm 6.40 \cdot 10^{-2}$	50,387	8.837	$\pm 1.74 \cdot 10^{-2}$	4,017
10	9.154	$\pm 1.63 \cdot 10^{-2}$	28,327	7.739	$\pm 6.59 \cdot 10^{-2}$	50,302	8.833	$\pm 1.68 \cdot 10^{-2}$	4,017
12	9.074	$\pm 1.41 \cdot 10^{-2}$	27,825	7.578	$\pm 6.41 \cdot 10^{-2}$	49,822	8.833	$\pm 1.68 \cdot 10^{-2}$	4,017
14	9.042	$\pm 1.43 \cdot 10^{-2}$	27,840	7.524	$\pm 6.35 \cdot 10^{-2}$	49,686	8.833	$\pm 1.68 \cdot 10^{-2}$	4,017
16	8.965	$\pm 1.07 \cdot 10^{-2}$	27,576	7.432	$\pm 6.29 \cdot 10^{-2}$	49,490	8.833	$\pm 1.68 \cdot 10^{-2}$	4,017
18	8.968	$\pm 1.01 \cdot 10^{-2}$	28,123	7.380	$\pm 6.03 \cdot 10^{-2}$	49,985	8.856	$\pm 1.98 \cdot 10^{-2}$	4,017
20	8.948	$\pm 8.90 \cdot 10^{-3}$	27,560	7.340	$\pm 5.97 \cdot 10^{-2}$	49,427	8.833	$\pm 1.68 \cdot 10^{-2}$	4,017
22	8.921	$\pm 9.23 \cdot 10^{-3}$	28,407	7.377	$\pm 6.08 \cdot 10^{-2}$	50,299	8.833	$\pm 1.68 \cdot 10^{-2}$	4,017
24	8.950	$\pm 1.06 \cdot 10^{-2}$	28,491	7.395	$\pm 6.06 \cdot 10^{-2}$	50,399	8.833	$\pm 1.68 \cdot 10^{-2}$	4,017
26	8.910	$\pm 9.55 \cdot 10^{-3}$	27,573	7.373	$\pm 6.18 \cdot 10^{-2}$	49,473	8.833	$\pm 1.68 \cdot 10^{-2}$	4,017
28	8.889	$\pm 8.41 \cdot 10^{-3}$	27,874	7.325	$\pm 6.10 \cdot 10^{-2}$	49,821	8.833	$\pm 1.68 \cdot 10^{-2}$	4,017
30	8.889	$\pm 8.72 \cdot 10^{-3}$	28,378	7.343	$\pm 6.04 \cdot 10^{-2}$	50,185	8.833	$\pm 1.68 \cdot 10^{-2}$	4,017
32	8.867	$\pm 7.82 \cdot 10^{-3}$	28,042	7.288	$\pm 6.04 \cdot 10^{-2}$	49,957	8.833	$\pm 1.68 \cdot 10^{-2}$	4,017

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
1	122.481	$\pm 1.24 \cdot 10^{-1}$	255,005	126.184	$\pm 1.91 \cdot 10^{-1}$	92,932	121.375	$\pm 1.60 \cdot 10^{-1}$	158,043
2	31.039	$\pm 2.15 \cdot 10^{-1}$	65,589	27.263	$\pm 4.22 \cdot 10^{-1}$	13,944	32.596	$\pm 2.31 \cdot 10^{-1}$	47,629
3	13.282	$\pm 1.26 \cdot 10^{-1}$	45,982	12.231	$\pm 1.81 \cdot 10^{-1}$	9,982	15.657	$\pm 1.33 \cdot 10^{-1}$	31,984
4	8.126	$\pm 6.55 \cdot 10^{-2}$	38,617	9.844	$\pm 6.39 \cdot 10^{-2}$	9,133	10.485	$\pm 5.22 \cdot 10^{-2}$	25,468
6	7.207	$\pm 5.64 \cdot 10^{-2}$	37,259	9.357	$\pm 4.96 \cdot 10^{-2}$	9,073	9.588	$\pm 3.42 \cdot 10^{-2}$	24,170
8	6.895	$\pm 5.12 \cdot 10^{-2}$	37,520	9.164	$\pm 3.21 \cdot 10^{-2}$	9,025	9.260	$\pm 2.12 \cdot 10^{-2}$	24,478
10	6.805	$\pm 5.48 \cdot 10^{-2}$	37,349	9.385	$\pm 3.61 \cdot 10^{-2}$	9,022	9.153	$\pm 1.87 \cdot 10^{-2}$	24,310
12	6.712	$\pm 5.38 \cdot 10^{-2}$	36,840	9.020	$\pm 2.71 \cdot 10^{-2}$	9,015	9.061	$\pm 1.62 \cdot 10^{-2}$	23,808
14	6.636	$\pm 5.32 \cdot 10^{-2}$	36,850	8.974	$\pm 2.48 \cdot 10^{-2}$	9,010	9.023	$\pm 1.64 \cdot 10^{-2}$	23,823
16	6.546	$\pm 5.27 \cdot 10^{-2}$	36,559	8.954	$\pm 2.31 \cdot 10^{-2}$	8,983	8.933	$\pm 1.21 \cdot 10^{-2}$	23,559
18	6.492	$\pm 4.89 \cdot 10^{-2}$	37,097	8.946	$\pm 2.44 \cdot 10^{-2}$	8,974	8.932	$\pm 1.13 \cdot 10^{-2}$	24,106
20	6.433	$\pm 4.73 \cdot 10^{-2}$	36,560	8.990	$\pm 1.86 \cdot 10^{-2}$	9,001	8.914	$\pm 1.00 \cdot 10^{-2}$	23,543
22	6.502	$\pm 5.05 \cdot 10^{-2}$	37,383	8.901	$\pm 1.52 \cdot 10^{-2}$	8,976	8.882	$\pm 1.04 \cdot 10^{-2}$	24,390
24	6.500	$\pm 4.95 \cdot 10^{-2}$	37,464	8.905	$\pm 1.85 \cdot 10^{-2}$	8,973	8.915	$\pm 1.21 \cdot 10^{-2}$	24,474
26	6.503	$\pm 5.19 \cdot 10^{-2}$	36,562	8.901	$\pm 1.59 \cdot 10^{-2}$	8,990	8.869	$\pm 1.08 \cdot 10^{-2}$	23,556
28	6.443	$\pm 5.04 \cdot 10^{-2}$	36,865	8.877	$\pm 1.54 \cdot 10^{-2}$	8,991	8.845	$\pm 9.41 \cdot 10^{-3}$	23,857
30	6.424	$\pm 4.96 \cdot 10^{-2}$	37,363	8.882	$\pm 1.58 \cdot 10^{-2}$	8,985	8.845	$\pm 9.77 \cdot 10^{-3}$	24,361
32	6.393	$\pm 4.98 \cdot 10^{-2}$	37,007	8.860	$\pm 1.41 \cdot 10^{-2}$	8,967	8.818	$\pm 8.74 \cdot 10^{-3}$	24,027

Table B.9: Scalability experiment results with 64 PEs and 0 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
2	32.816	$\pm 2.31 \cdot 10^{-1}$	53,824	30.529	$\pm 2.82 \cdot 10^{-1}$	99,799	6.899	$\pm 7.75 \cdot 10^{-2}$	4,063
3	16.565	$\pm 1.36 \cdot 10^{-1}$	42,072	11.192	$\pm 1.38 \cdot 10^{-1}$	83,486	6.658	$\pm 4.07 \cdot 10^{-4}$	4,003
4	10.430	$\pm 7.79 \cdot 10^{-2}$	33,685	5.391	$\pm 7.28 \cdot 10^{-2}$	74,091	6.658	$\pm 4.07 \cdot 10^{-4}$	4,003
5	8.247	$\pm 4.11 \cdot 10^{-2}$	30,797	3.599	$\pm 4.76 \cdot 10^{-2}$	70,801	6.658	$\pm 4.07 \cdot 10^{-4}$	4,003
6	7.981	$\pm 3.66 \cdot 10^{-2}$	30,907	3.383	$\pm 4.39 \cdot 10^{-2}$	70,626	6.658	$\pm 4.07 \cdot 10^{-4}$	4,003
7	7.386	$\pm 2.45 \cdot 10^{-2}$	28,323	2.895	$\pm 3.87 \cdot 10^{-2}$	68,681	6.658	$\pm 4.07 \cdot 10^{-4}$	4,003
8	7.463	$\pm 2.56 \cdot 10^{-2}$	29,416	3.045	$\pm 3.81 \cdot 10^{-2}$	69,325	6.658	$\pm 4.07 \cdot 10^{-4}$	4,003
10	7.203	$\pm 2.14 \cdot 10^{-2}$	29,029	2.919	$\pm 3.60 \cdot 10^{-2}$	64,996	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003
12	7.074	$\pm 1.66 \cdot 10^{-2}$	29,151	2.787	$\pm 3.17 \cdot 10^{-2}$	65,044	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003
14	6.953	$\pm 1.37 \cdot 10^{-2}$	27,943	2.674	$\pm 3.19 \cdot 10^{-2}$	64,224	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003
16	6.935	$\pm 1.34 \cdot 10^{-2}$	27,830	2.695	$\pm 3.36 \cdot 10^{-2}$	64,613	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003
20	6.855	$\pm 1.08 \cdot 10^{-2}$	28,449	2.628	$\pm 2.98 \cdot 10^{-2}$	64,197	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003
24	6.844	$\pm 1.03 \cdot 10^{-2}$	28,438	2.645	$\pm 3.26 \cdot 10^{-2}$	64,394	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003
28	6.820	$\pm 9.27 \cdot 10^{-3}$	28,961	2.622	$\pm 2.93 \cdot 10^{-2}$	64,726	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003
32	6.779	$\pm 7.94 \cdot 10^{-3}$	28,766	2.584	$\pm 2.98 \cdot 10^{-2}$	64,762	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003
36	6.770	$\pm 7.78 \cdot 10^{-3}$	28,268	2.580	$\pm 3.17 \cdot 10^{-2}$	64,890	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003
40	6.769	$\pm 7.95 \cdot 10^{-3}$	28,310	2.597	$\pm 3.09 \cdot 10^{-2}$	64,249	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003
44	6.765	$\pm 7.46 \cdot 10^{-3}$	29,199	2.615	$\pm 3.10 \cdot 10^{-2}$	65,116	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003
48	6.740	$\pm 6.89 \cdot 10^{-3}$	27,991	2.569	$\pm 3.22 \cdot 10^{-2}$	64,530	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003
52	6.744	$\pm 6.94 \cdot 10^{-3}$	28,344	2.588	$\pm 3.15 \cdot 10^{-2}$	64,326	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003
60	6.713	$\pm 5.43 \cdot 10^{-3}$	28,228	2.505	$\pm 2.89 \cdot 10^{-2}$	64,982	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003
64	6.707	$\pm 5.33 \cdot 10^{-3}$	28,118	2.525	$\pm 2.88 \cdot 10^{-2}$	64,345	6.658	$\pm 3.44 \cdot 10^{-4}$	4,003

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
2	29.136	$\pm 2.14 \cdot 10^{-1}$	67,911	30.977	$\pm 4.88 \cdot 10^{-1}$	14,088	34.882	$\pm 2.40 \cdot 10^{-1}$	49,761
3	11.469	$\pm 1.31 \cdot 10^{-1}$	52,233	12.712	$\pm 2.68 \cdot 10^{-1}$	10,162	17.555	$\pm 1.47 \cdot 10^{-1}$	38,071
4	5.079	$\pm 7.58 \cdot 10^{-2}$	42,888	9.437	$\pm 1.61 \cdot 10^{-1}$	9,203	10.886	$\pm 8.68 \cdot 10^{-2}$	29,682
5	2.839	$\pm 4.42 \cdot 10^{-2}$	39,738	7.916	$\pm 1.16 \cdot 10^{-1}$	8,941	8.431	$\pm 4.66 \cdot 10^{-2}$	26,794
6	2.570	$\pm 4.07 \cdot 10^{-2}$	39,825	7.737	$\pm 9.47 \cdot 10^{-2}$	8,918	8.124	$\pm 4.15 \cdot 10^{-2}$	26,904
7	1.934	$\pm 3.41 \cdot 10^{-2}$	37,133	7.404	$\pm 7.50 \cdot 10^{-2}$	8,810	7.452	$\pm 2.83 \cdot 10^{-2}$	24,320
8	1.990	$\pm 3.02 \cdot 10^{-2}$	38,275	7.942	$\pm 7.30 \cdot 10^{-2}$	8,859	7.535	$\pm 2.94 \cdot 10^{-2}$	25,413
10	1.688	$\pm 2.74 \cdot 10^{-2}$	37,910	7.121	$\pm 5.06 \cdot 10^{-2}$	8,882	7.236	$\pm 2.47 \cdot 10^{-2}$	25,026
12	1.551	$\pm 2.15 \cdot 10^{-2}$	37,981	7.067	$\pm 4.11 \cdot 10^{-2}$	8,831	7.086	$\pm 1.92 \cdot 10^{-2}$	25,148
14	1.364	$\pm 1.68 \cdot 10^{-2}$	36,719	6.936	$\pm 3.09 \cdot 10^{-2}$	8,776	6.948	$\pm 1.59 \cdot 10^{-2}$	23,940
16	1.385	$\pm 2.21 \cdot 10^{-2}$	36,622	7.115	$\pm 3.32 \cdot 10^{-2}$	8,792	6.927	$\pm 1.56 \cdot 10^{-2}$	23,827
20	1.317	$\pm 1.66 \cdot 10^{-2}$	37,234	6.826	$\pm 1.93 \cdot 10^{-2}$	8,785	6.833	$\pm 1.25 \cdot 10^{-2}$	24,446
24	1.331	$\pm 2.59 \cdot 10^{-2}$	37,233	6.893	$\pm 2.09 \cdot 10^{-2}$	8,795	6.820	$\pm 1.19 \cdot 10^{-2}$	24,435
28	1.283	$\pm 1.55 \cdot 10^{-2}$	37,756	6.798	$\pm 1.47 \cdot 10^{-2}$	8,795	6.791	$\pm 1.07 \cdot 10^{-2}$	24,958
32	1.240	$\pm 1.72 \cdot 10^{-2}$	37,560	6.858	$\pm 1.80 \cdot 10^{-2}$	8,794	6.744	$\pm 9.21 \cdot 10^{-3}$	24,763
36	1.231	$\pm 1.96 \cdot 10^{-2}$	37,059	6.767	$\pm 1.13 \cdot 10^{-2}$	8,792	6.735	$\pm 9.06 \cdot 10^{-3}$	24,265
40	1.234	$\pm 1.95 \cdot 10^{-2}$	37,109	6.759	$\pm 1.03 \cdot 10^{-2}$	8,799	6.734	$\pm 9.25 \cdot 10^{-3}$	24,307
44	1.267	$\pm 2.06 \cdot 10^{-2}$	37,983	6.749	$\pm 9.44 \cdot 10^{-3}$	8,785	6.728	$\pm 8.64 \cdot 10^{-3}$	25,196
48	1.207	$\pm 2.15 \cdot 10^{-2}$	36,767	6.744	$\pm 8.43 \cdot 10^{-3}$	8,776	6.699	$\pm 8.03 \cdot 10^{-3}$	23,988
52	1.235	$\pm 2.32 \cdot 10^{-2}$	37,126	6.738	$\pm 8.08 \cdot 10^{-3}$	8,783	6.704	$\pm 8.08 \cdot 10^{-3}$	24,341
60	1.148	$\pm 1.15 \cdot 10^{-2}$	37,012	6.725	$\pm 6.01 \cdot 10^{-3}$	8,785	6.668	$\pm 6.33 \cdot 10^{-3}$	24,225
64	1.143	$\pm 1.29 \cdot 10^{-2}$	36,914	6.718	$\pm 4.43 \cdot 10^{-3}$	8,796	6.660	$\pm 6.21 \cdot 10^{-3}$	24,115

Table B.10: Scalability experiment results with 64 PEs and 500 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
2	39.596	$\pm 2.53 \cdot 10^{-1}$	58,823	41.931	$\pm 3.42 \cdot 10^{-1}$	105,620	8.846	$\pm 2.50 \cdot 10^{-1}$	4,063
3	19.467	$\pm 1.53 \cdot 10^{-1}$	43,541	14.880	$\pm 1.71 \cdot 10^{-1}$	85,913	8.147	$\pm 7.59 \cdot 10^{-2}$	4,063
4	10.291	$\pm 5.64 \cdot 10^{-2}$	29,937	5.326	$\pm 6.93 \cdot 10^{-2}$	68,938	7.914	$\pm 2.34 \cdot 10^{-3}$	4,003
5	10.069	$\pm 5.33 \cdot 10^{-2}$	31,350	5.121	$\pm 6.50 \cdot 10^{-2}$	71,015	7.917	$\pm 2.79 \cdot 10^{-3}$	4,003
6	10.794	$\pm 6.14 \cdot 10^{-2}$	35,599	5.923	$\pm 7.08 \cdot 10^{-2}$	75,611	7.913	$\pm 2.22 \cdot 10^{-3}$	4,003
7	9.064	$\pm 3.31 \cdot 10^{-2}$	30,619	4.204	$\pm 4.90 \cdot 10^{-2}$	70,540	7.913	$\pm 2.23 \cdot 10^{-3}$	4,003
8	8.582	$\pm 2.28 \cdot 10^{-2}$	28,047	3.859	$\pm 4.83 \cdot 10^{-2}$	68,140	7.913	$\pm 2.22 \cdot 10^{-3}$	4,003
10	8.421	$\pm 1.85 \cdot 10^{-2}$	29,072	3.654	$\pm 4.34 \cdot 10^{-2}$	69,095	7.916	$\pm 3.61 \cdot 10^{-3}$	4,003
12	8.488	$\pm 2.11 \cdot 10^{-2}$	30,115	3.949	$\pm 4.76 \cdot 10^{-2}$	67,964	7.917	$\pm 2.96 \cdot 10^{-3}$	4,003
14	8.227	$\pm 1.41 \cdot 10^{-2}$	28,597	3.631	$\pm 4.20 \cdot 10^{-2}$	66,905	7.917	$\pm 2.96 \cdot 10^{-3}$	4,003
16	8.212	$\pm 1.32 \cdot 10^{-2}$	28,640	3.619	$\pm 4.49 \cdot 10^{-2}$	67,471	7.917	$\pm 2.96 \cdot 10^{-3}$	4,003
20	8.114	$\pm 1.02 \cdot 10^{-2}$	29,020	3.485	$\pm 4.09 \cdot 10^{-2}$	69,547	7.916	$\pm 3.61 \cdot 10^{-3}$	4,003
24	8.101	$\pm 1.11 \cdot 10^{-2}$	28,009	3.531	$\pm 4.32 \cdot 10^{-2}$	66,665	7.917	$\pm 2.96 \cdot 10^{-3}$	4,003
28	8.076	$\pm 9.25 \cdot 10^{-3}$	29,258	3.557	$\pm 4.15 \cdot 10^{-2}$	67,443	7.917	$\pm 2.96 \cdot 10^{-3}$	4,003
32	8.068	$\pm 9.10 \cdot 10^{-3}$	29,455	3.567	$\pm 4.06 \cdot 10^{-2}$	67,559	7.917	$\pm 2.96 \cdot 10^{-3}$	4,003
36	8.027	$\pm 7.46 \cdot 10^{-3}$	29,578	3.572	$\pm 4.15 \cdot 10^{-2}$	68,102	7.917	$\pm 2.96 \cdot 10^{-3}$	4,003
40	8.012	$\pm 7.72 \cdot 10^{-3}$	28,758	3.472	$\pm 4.26 \cdot 10^{-2}$	67,629	7.917	$\pm 2.96 \cdot 10^{-3}$	4,003
44	7.997	$\pm 6.54 \cdot 10^{-3}$	28,690	3.444	$\pm 4.16 \cdot 10^{-2}$	67,698	7.917	$\pm 2.96 \cdot 10^{-3}$	4,003
48	8.004	$\pm 6.83 \cdot 10^{-3}$	28,711	3.491	$\pm 4.23 \cdot 10^{-2}$	67,240	7.917	$\pm 2.96 \cdot 10^{-3}$	4,003
52	8.009	$\pm 7.22 \cdot 10^{-3}$	29,577	3.504	$\pm 4.13 \cdot 10^{-2}$	67,514	7.917	$\pm 2.96 \cdot 10^{-3}$	4,003
60	7.981	$\pm 5.86 \cdot 10^{-3}$	29,472	3.504	$\pm 3.92 \cdot 10^{-2}$	67,153	7.917	$\pm 2.96 \cdot 10^{-3}$	4,003
64	7.962	$\pm 6.08 \cdot 10^{-3}$	27,389	3.446	$\pm 4.03 \cdot 10^{-2}$	65,609	7.917	$\pm 2.96 \cdot 10^{-3}$	4,003

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
2	38.286	$\pm 2.44 \cdot 10^{-1}$	75,406	39.819	$\pm 5.09 \cdot 10^{-1}$	16,583	41.828	$\pm 2.61 \cdot 10^{-1}$	54,760
3	15.454	$\pm 1.57 \cdot 10^{-1}$	54,340	15.829	$\pm 3.10 \cdot 10^{-1}$	10,799	20.581	$\pm 1.64 \cdot 10^{-1}$	39,478
4	5.173	$\pm 6.71 \cdot 10^{-2}$	39,069	10.751	$\pm 1.66 \cdot 10^{-1}$	9,132	10.604	$\pm 6.40 \cdot 10^{-2}$	25,934
5	4.913	$\pm 6.50 \cdot 10^{-2}$	40,473	9.436	$\pm 1.23 \cdot 10^{-1}$	9,124	10.331	$\pm 6.02 \cdot 10^{-2}$	27,347
6	5.639	$\pm 6.90 \cdot 10^{-2}$	44,945	9.509	$\pm 1.14 \cdot 10^{-1}$	9,347	11.107	$\pm 6.82 \cdot 10^{-2}$	31,596
7	3.753	$\pm 4.53 \cdot 10^{-2}$	39,658	8.876	$\pm 8.47 \cdot 10^{-2}$	9,039	9.184	$\pm 3.77 \cdot 10^{-2}$	26,616
8	3.239	$\pm 4.50 \cdot 10^{-2}$	36,975	9.248	$\pm 7.66 \cdot 10^{-2}$	8,928	8.639	$\pm 2.64 \cdot 10^{-2}$	24,044
10	3.052	$\pm 4.21 \cdot 10^{-2}$	37,978	8.396	$\pm 4.99 \cdot 10^{-2}$	8,907	8.448	$\pm 2.13 \cdot 10^{-2}$	25,069
12	3.214	$\pm 4.89 \cdot 10^{-2}$	39,066	8.391	$\pm 4.43 \cdot 10^{-2}$	8,951	8.521	$\pm 2.42 \cdot 10^{-2}$	26,112
14	2.872	$\pm 3.90 \cdot 10^{-2}$	37,491	8.230	$\pm 3.81 \cdot 10^{-2}$	8,894	8.223	$\pm 1.63 \cdot 10^{-2}$	24,594
16	2.867	$\pm 4.83 \cdot 10^{-2}$	37,556	8.382	$\pm 3.31 \cdot 10^{-2}$	8,916	8.206	$\pm 1.52 \cdot 10^{-2}$	24,637
20	2.790	$\pm 4.18 \cdot 10^{-2}$	37,927	8.118	$\pm 2.25 \cdot 10^{-2}$	8,907	8.091	$\pm 1.18 \cdot 10^{-2}$	25,017
24	2.732	$\pm 4.43 \cdot 10^{-2}$	36,908	8.162	$\pm 2.15 \cdot 10^{-2}$	8,899	8.077	$\pm 1.29 \cdot 10^{-2}$	24,006
28	2.701	$\pm 3.91 \cdot 10^{-2}$	38,159	8.057	$\pm 1.59 \cdot 10^{-2}$	8,903	8.047	$\pm 1.07 \cdot 10^{-2}$	25,255
32	2.707	$\pm 4.01 \cdot 10^{-2}$	38,384	8.103	$\pm 1.71 \cdot 10^{-2}$	8,929	8.038	$\pm 1.05 \cdot 10^{-2}$	25,452
36	2.682	$\pm 4.00 \cdot 10^{-2}$	38,500	8.026	$\pm 1.13 \cdot 10^{-2}$	8,922	7.990	$\pm 8.61 \cdot 10^{-3}$	25,575
40	2.653	$\pm 4.43 \cdot 10^{-2}$	37,631	8.009	$\pm 8.92 \cdot 10^{-3}$	8,873	7.973	$\pm 8.95 \cdot 10^{-3}$	24,755
44	2.660	$\pm 4.45 \cdot 10^{-2}$	37,566	8.010	$\pm 9.65 \cdot 10^{-3}$	8,876	7.956	$\pm 7.58 \cdot 10^{-3}$	24,687
48	2.649	$\pm 4.39 \cdot 10^{-2}$	37,631	8.003	$\pm 8.73 \cdot 10^{-3}$	8,921	7.964	$\pm 7.92 \cdot 10^{-3}$	24,708
52	2.662	$\pm 4.32 \cdot 10^{-2}$	38,472	7.997	$\pm 8.15 \cdot 10^{-3}$	8,896	7.970	$\pm 8.33 \cdot 10^{-3}$	25,574
60	2.589	$\pm 3.52 \cdot 10^{-2}$	38,392	7.984	$\pm 6.28 \cdot 10^{-3}$	8,920	7.937	$\pm 6.76 \cdot 10^{-3}$	25,469
64	2.564	$\pm 3.79 \cdot 10^{-2}$	36,250	7.976	$\pm 4.73 \cdot 10^{-3}$	8,863	7.915	$\pm 7.11 \cdot 10^{-3}$	23,386

Table B.11: Scalability experiment results with 64 PEs and 1000 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
2	59.549	$\pm 2.88 \cdot 10^{-1}$	84,740	77.905	$\pm 4.40 \cdot 10^{-1}$	140,680	10.384	$\pm 3.36 \cdot 10^{-1}$	4,062
3	27.634	$\pm 1.97 \cdot 10^{-1}$	53,809	25.272	$\pm 2.44 \cdot 10^{-1}$	100,850	9.708	$\pm 1.72 \cdot 10^{-1}$	4,062
4	16.989	$\pm 1.22 \cdot 10^{-1}$	41,665	12.413	$\pm 1.44 \cdot 10^{-1}$	82,638	9.188	$\pm 7.16 \cdot 10^{-3}$	4,003
5	11.931	$\pm 6.59 \cdot 10^{-2}$	32,472	7.189	$\pm 9.09 \cdot 10^{-2}$	68,585	9.194	$\pm 7.90 \cdot 10^{-3}$	4,003
6	10.954	$\pm 4.25 \cdot 10^{-2}$	32,234	5.813	$\pm 7.29 \cdot 10^{-2}$	74,846	9.182	$\pm 6.33 \cdot 10^{-3}$	4,062
7	10.248	$\pm 3.34 \cdot 10^{-2}$	30,298	5.120	$\pm 6.71 \cdot 10^{-2}$	72,853	9.185	$\pm 6.71 \cdot 10^{-3}$	4,062
8	9.926	$\pm 2.63 \cdot 10^{-2}$	28,878	4.895	$\pm 6.62 \cdot 10^{-2}$	71,860	9.182	$\pm 6.35 \cdot 10^{-3}$	4,062
10	9.752	$\pm 2.09 \cdot 10^{-2}$	29,155	4.835	$\pm 6.52 \cdot 10^{-2}$	69,247	9.183	$\pm 6.87 \cdot 10^{-3}$	4,003
12	9.615	$\pm 1.85 \cdot 10^{-2}$	28,621	4.705	$\pm 6.26 \cdot 10^{-2}$	69,106	9.183	$\pm 6.87 \cdot 10^{-3}$	4,003
14	9.705	$\pm 2.05 \cdot 10^{-2}$	31,179	4.911	$\pm 6.04 \cdot 10^{-2}$	70,935	9.183	$\pm 6.87 \cdot 10^{-3}$	4,003
16	9.527	$\pm 1.55 \cdot 10^{-2}$	29,172	4.586	$\pm 5.81 \cdot 10^{-2}$	69,840	9.187	$\pm 7.04 \cdot 10^{-3}$	4,003
20	9.412	$\pm 1.29 \cdot 10^{-2}$	28,891	4.540	$\pm 5.74 \cdot 10^{-2}$	69,121	9.183	$\pm 6.87 \cdot 10^{-3}$	4,003
24	9.410	$\pm 1.19 \cdot 10^{-2}$	29,987	4.573	$\pm 5.61 \cdot 10^{-2}$	69,946	9.183	$\pm 6.87 \cdot 10^{-3}$	4,003
28	9.336	$\pm 1.19 \cdot 10^{-2}$	29,098	4.443	$\pm 5.38 \cdot 10^{-2}$	69,079	9.183	$\pm 6.87 \cdot 10^{-3}$	4,003
32	9.312	$\pm 8.64 \cdot 10^{-3}$	28,916	4.356	$\pm 5.28 \cdot 10^{-2}$	69,752	9.187	$\pm 7.04 \cdot 10^{-3}$	4,003
36	9.326	$\pm 9.93 \cdot 10^{-3}$	29,462	4.504	$\pm 5.79 \cdot 10^{-2}$	69,360	9.183	$\pm 6.87 \cdot 10^{-3}$	4,003
40	9.329	$\pm 9.83 \cdot 10^{-3}$	30,706	4.541	$\pm 5.59 \cdot 10^{-2}$	70,945	9.183	$\pm 6.87 \cdot 10^{-3}$	4,003
44	9.298	$\pm 8.94 \cdot 10^{-3}$	29,575	4.447	$\pm 5.66 \cdot 10^{-2}$	69,813	9.183	$\pm 6.87 \cdot 10^{-3}$	4,003
48	9.277	$\pm 9.50 \cdot 10^{-3}$	29,465	4.480	$\pm 5.77 \cdot 10^{-2}$	69,611	9.183	$\pm 6.87 \cdot 10^{-3}$	4,003
52	9.260	$\pm 9.57 \cdot 10^{-3}$	28,572	4.422	$\pm 5.78 \cdot 10^{-2}$	68,740	9.183	$\pm 6.87 \cdot 10^{-3}$	4,003
56	9.253	$\pm 7.20 \cdot 10^{-3}$	30,266	4.434	$\pm 5.21 \cdot 10^{-2}$	70,367	9.183	$\pm 6.87 \cdot 10^{-3}$	4,003
60	9.253	$\pm 7.03 \cdot 10^{-3}$	29,451	4.464	$\pm 5.29 \cdot 10^{-2}$	69,140	9.183	$\pm 6.87 \cdot 10^{-3}$	4,003
64	9.234	$\pm 7.01 \cdot 10^{-3}$	28,437	4.374	$\pm 5.34 \cdot 10^{-2}$	68,425	9.183	$\pm 6.87 \cdot 10^{-3}$	4,003

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
2	62.783	$\pm 2.77 \cdot 10^{-1}$	112,575	65.813	$\pm 5.32 \cdot 10^{-1}$	27,834	61.975	$\pm 2.92 \cdot 10^{-1}$	80,678
3	25.232	$\pm 2.06 \cdot 10^{-1}$	67,015	24.607	$\pm 4.13 \cdot 10^{-1}$	13,205	29.047	$\pm 2.07 \cdot 10^{-1}$	49,747
4	12.890	$\pm 1.37 \cdot 10^{-1}$	52,007	14.942	$\pm 2.42 \cdot 10^{-1}$	10,342	17.767	$\pm 1.33 \cdot 10^{-1}$	37,662
5	7.197	$\pm 9.33 \cdot 10^{-2}$	41,854	11.198	$\pm 1.43 \cdot 10^{-1}$	9,381	12.263	$\pm 7.42 \cdot 10^{-2}$	28,469
6	6.142	$\pm 8.00 \cdot 10^{-2}$	41,445	10.615	$\pm 1.17 \cdot 10^{-1}$	9,210	11.156	$\pm 4.80 \cdot 10^{-2}$	28,172
7	5.371	$\pm 8.04 \cdot 10^{-2}$	39,416	10.226	$\pm 9.27 \cdot 10^{-2}$	9,117	10.359	$\pm 3.82 \cdot 10^{-2}$	26,236
8	4.959	$\pm 7.95 \cdot 10^{-2}$	37,927	10.719	$\pm 8.94 \cdot 10^{-2}$	9,048	9.994	$\pm 3.03 \cdot 10^{-2}$	24,816
10	4.785	$\pm 8.05 \cdot 10^{-2}$	38,179	9.807	$\pm 6.64 \cdot 10^{-2}$	9,025	9.788	$\pm 2.41 \cdot 10^{-2}$	25,152
12	4.608	$\pm 7.83 \cdot 10^{-2}$	37,653	9.723	$\pm 5.59 \cdot 10^{-2}$	9,032	9.632	$\pm 2.14 \cdot 10^{-2}$	24,618
14	4.720	$\pm 7.14 \cdot 10^{-2}$	40,310	9.584	$\pm 4.28 \cdot 10^{-2}$	9,131	9.728	$\pm 2.34 \cdot 10^{-2}$	27,176
16	4.444	$\pm 7.25 \cdot 10^{-2}$	38,186	9.683	$\pm 3.53 \cdot 10^{-2}$	9,014	9.527	$\pm 1.78 \cdot 10^{-2}$	25,169
20	4.338	$\pm 7.13 \cdot 10^{-2}$	37,890	9.420	$\pm 2.69 \cdot 10^{-2}$	9,000	9.394	$\pm 1.49 \cdot 10^{-2}$	24,888
24	4.339	$\pm 6.92 \cdot 10^{-2}$	39,046	9.460	$\pm 3.35 \cdot 10^{-2}$	9,059	9.391	$\pm 1.37 \cdot 10^{-2}$	25,984
28	4.208	$\pm 6.54 \cdot 10^{-2}$	38,114	9.346	$\pm 2.12 \cdot 10^{-2}$	9,017	9.307	$\pm 1.37 \cdot 10^{-2}$	25,095
32	4.157	$\pm 6.59 \cdot 10^{-2}$	37,903	9.384	$\pm 1.85 \cdot 10^{-2}$	8,987	9.278	$\pm 9.95 \cdot 10^{-3}$	24,913
36	4.285	$\pm 7.58 \cdot 10^{-2}$	38,500	9.320	$\pm 1.98 \cdot 10^{-2}$	9,038	9.295	$\pm 1.14 \cdot 10^{-2}$	25,459
40	4.272	$\pm 7.04 \cdot 10^{-2}$	39,741	9.304	$\pm 1.91 \cdot 10^{-2}$	9,035	9.298	$\pm 1.12 \cdot 10^{-2}$	26,703
44	4.235	$\pm 7.37 \cdot 10^{-2}$	38,583	9.299	$\pm 1.81 \cdot 10^{-2}$	9,009	9.262	$\pm 1.03 \cdot 10^{-2}$	25,572
48	4.241	$\pm 7.51 \cdot 10^{-2}$	38,452	9.296	$\pm 1.99 \cdot 10^{-2}$	8,988	9.238	$\pm 1.09 \cdot 10^{-2}$	25,462
52	4.207	$\pm 7.60 \cdot 10^{-2}$	37,563	9.281	$\pm 1.78 \cdot 10^{-2}$	8,991	9.219	$\pm 1.11 \cdot 10^{-2}$	24,569
56	4.111	$\pm 6.18 \cdot 10^{-2}$	39,284	9.265	$\pm 1.54 \cdot 10^{-2}$	9,019	9.210	$\pm 8.23 \cdot 10^{-3}$	26,263
60	4.130	$\pm 6.33 \cdot 10^{-2}$	38,479	9.260	$\pm 1.53 \cdot 10^{-2}$	9,029	9.210	$\pm 8.06 \cdot 10^{-3}$	25,448
64	4.105	$\pm 6.63 \cdot 10^{-2}$	37,428	9.251	$\pm 1.49 \cdot 10^{-2}$	8,991	9.188	$\pm 8.08 \cdot 10^{-3}$	24,434

Table B.12: Scalability experiment results with 64 PEs and 2000 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
2	116.563	$\pm 2.95 \cdot 10^{-1}$	185,840	199.980	$\pm 4.65 \cdot 10^{-1}$	330,585	13.344	$\pm 4.43 \cdot 10^{-1}$	4,062
3	51.938	$\pm 2.76 \cdot 10^{-1}$	79,307	66.967	$\pm 4.10 \cdot 10^{-1}$	132,001	12.823	$\pm 3.31 \cdot 10^{-1}$	4,062
4	28.655	$\pm 2.04 \cdot 10^{-1}$	50,911	28.821	$\pm 2.72 \cdot 10^{-1}$	91,771	11.767	$\pm 1.82 \cdot 10^{-2}$	4,003
5	22.940	$\pm 1.61 \cdot 10^{-1}$	46,082	20.813	$\pm 2.14 \cdot 10^{-1}$	85,021	11.767	$\pm 1.82 \cdot 10^{-2}$	4,003
6	17.918	$\pm 1.20 \cdot 10^{-1}$	39,852	14.332	$\pm 1.69 \cdot 10^{-1}$	77,436	11.764	$\pm 1.80 \cdot 10^{-2}$	4,003
7	14.612	$\pm 7.60 \cdot 10^{-2}$	34,641	10.434	$\pm 1.38 \cdot 10^{-1}$	71,298	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
8	15.981	$\pm 9.16 \cdot 10^{-2}$	38,693	12.268	$\pm 1.46 \cdot 10^{-1}$	75,886	11.767	$\pm 1.82 \cdot 10^{-2}$	4,003
10	12.755	$\pm 3.94 \cdot 10^{-2}$	30,329	8.492	$\pm 1.29 \cdot 10^{-1}$	66,716	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
12	12.916	$\pm 3.89 \cdot 10^{-2}$	32,327	8.735	$\pm 1.21 \cdot 10^{-1}$	68,460	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
14	12.408	$\pm 3.24 \cdot 10^{-2}$	30,991	8.028	$\pm 1.18 \cdot 10^{-1}$	67,262	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
16	12.667	$\pm 3.32 \cdot 10^{-2}$	34,709	8.617	$\pm 1.16 \cdot 10^{-1}$	71,074	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
20	12.255	$\pm 2.59 \cdot 10^{-2}$	31,947	7.982	$\pm 1.14 \cdot 10^{-1}$	67,982	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
24	12.070	$\pm 2.06 \cdot 10^{-2}$	30,548	7.625	$\pm 1.10 \cdot 10^{-1}$	66,927	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
28	12.261	$\pm 2.40 \cdot 10^{-2}$	35,633	8.191	$\pm 1.09 \cdot 10^{-1}$	71,917	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
32	11.927	$\pm 2.06 \cdot 10^{-2}$	29,669	7.584	$\pm 1.16 \cdot 10^{-1}$	65,793	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
36	12.002	$\pm 2.33 \cdot 10^{-2}$	31,612	7.637	$\pm 1.08 \cdot 10^{-1}$	67,700	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
40	11.892	$\pm 2.08 \cdot 10^{-2}$	29,567	7.450	$\pm 1.14 \cdot 10^{-1}$	65,877	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
44	11.921	$\pm 1.89 \cdot 10^{-2}$	31,568	7.585	$\pm 1.11 \cdot 10^{-1}$	67,752	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
48	11.946	$\pm 1.78 \cdot 10^{-2}$	32,323	7.679	$\pm 1.10 \cdot 10^{-1}$	68,393	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
52	11.859	$\pm 2.07 \cdot 10^{-2}$	29,843	7.408	$\pm 1.12 \cdot 10^{-1}$	66,329	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
60	11.861	$\pm 1.81 \cdot 10^{-2}$	30,923	7.511	$\pm 1.13 \cdot 10^{-1}$	66,931	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003
64	11.810	$\pm 1.75 \cdot 10^{-2}$	30,494	7.434	$\pm 1.13 \cdot 10^{-1}$	66,803	11.781	$\pm 1.99 \cdot 10^{-2}$	4,003

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
2	135.658	$\pm 2.36 \cdot 10^{-1}$	301,314	135.829	$\pm 3.63 \cdot 10^{-1}$	115,474	118.822	$\pm 2.92 \cdot 10^{-1}$	181,778
3	56.932	$\pm 2.84 \cdot 10^{-1}$	103,491	57.085	$\pm 5.61 \cdot 10^{-1}$	24,184	54.001	$\pm 2.83 \cdot 10^{-1}$	75,245
4	28.119	$\pm 2.33 \cdot 10^{-1}$	64,300	27.879	$\pm 4.25 \cdot 10^{-1}$	13,389	30.046	$\pm 2.16 \cdot 10^{-1}$	46,908
5	20.904	$\pm 1.95 \cdot 10^{-1}$	57,947	20.052	$\pm 3.19 \cdot 10^{-1}$	11,865	23.952	$\pm 1.72 \cdot 10^{-1}$	42,079
6	15.070	$\pm 1.78 \cdot 10^{-1}$	50,419	15.922	$\pm 2.25 \cdot 10^{-1}$	10,567	18.553	$\pm 1.31 \cdot 10^{-1}$	35,849
7	11.435	$\pm 1.65 \cdot 10^{-1}$	44,360	13.980	$\pm 1.83 \cdot 10^{-1}$	9,719	14.929	$\pm 8.51 \cdot 10^{-2}$	30,638
8	12.688	$\pm 1.58 \cdot 10^{-1}$	48,825	15.299	$\pm 1.87 \cdot 10^{-1}$	10,133	16.416	$\pm 1.01 \cdot 10^{-1}$	34,690
10	9.407	$\pm 1.71 \cdot 10^{-1}$	39,722	12.909	$\pm 1.47 \cdot 10^{-1}$	9,393	12.849	$\pm 4.51 \cdot 10^{-2}$	26,326
12	9.380	$\pm 1.57 \cdot 10^{-1}$	41,839	12.746	$\pm 1.05 \cdot 10^{-1}$	9,512	13.023	$\pm 4.40 \cdot 10^{-2}$	28,324
14	8.696	$\pm 1.60 \cdot 10^{-1}$	40,351	12.373	$\pm 9.59 \cdot 10^{-2}$	9,360	12.448	$\pm 3.69 \cdot 10^{-2}$	26,988
16	8.988	$\pm 1.48 \cdot 10^{-1}$	44,251	12.660	$\pm 9.70 \cdot 10^{-2}$	9,543	12.730	$\pm 3.73 \cdot 10^{-2}$	30,706
20	8.485	$\pm 1.53 \cdot 10^{-1}$	41,324	12.195	$\pm 7.51 \cdot 10^{-2}$	9,378	12.270	$\pm 2.94 \cdot 10^{-2}$	27,944
24	8.236	$\pm 1.49 \cdot 10^{-1}$	39,882	12.251	$\pm 7.81 \cdot 10^{-2}$	9,334	12.060	$\pm 2.34 \cdot 10^{-2}$	26,545
28	8.402	$\pm 1.42 \cdot 10^{-1}$	45,103	12.038	$\pm 5.30 \cdot 10^{-2}$	9,471	12.269	$\pm 2.69 \cdot 10^{-2}$	31,630
32	8.156	$\pm 1.64 \cdot 10^{-1}$	38,931	12.071	$\pm 6.15 \cdot 10^{-2}$	9,262	11.896	$\pm 2.36 \cdot 10^{-2}$	25,666
36	8.071	$\pm 1.47 \cdot 10^{-1}$	40,946	11.955	$\pm 5.08 \cdot 10^{-2}$	9,334	11.981	$\pm 2.65 \cdot 10^{-2}$	27,609
40	8.103	$\pm 1.62 \cdot 10^{-1}$	38,829	11.974	$\pm 6.36 \cdot 10^{-2}$	9,262	11.855	$\pm 2.39 \cdot 10^{-2}$	25,564
44	8.013	$\pm 1.52 \cdot 10^{-1}$	40,943	11.933	$\pm 5.73 \cdot 10^{-2}$	9,376	11.888	$\pm 2.15 \cdot 10^{-2}$	27,565
48	8.034	$\pm 1.49 \cdot 10^{-1}$	41,708	11.951	$\pm 5.94 \cdot 10^{-2}$	9,386	11.916	$\pm 2.01 \cdot 10^{-2}$	28,320
52	8.015	$\pm 1.58 \cdot 10^{-1}$	39,124	11.909	$\pm 5.12 \cdot 10^{-2}$	9,281	11.817	$\pm 2.37 \cdot 10^{-2}$	25,840
60	8.008	$\pm 1.59 \cdot 10^{-1}$	40,244	11.866	$\pm 4.46 \cdot 10^{-2}$	9,321	11.819	$\pm 2.06 \cdot 10^{-2}$	26,920
64	7.968	$\pm 1.60 \cdot 10^{-1}$	39,788	11.831	$\pm 4.25 \cdot 10^{-2}$	9,294	11.761	$\pm 1.99 \cdot 10^{-2}$	26,491

Table B.13: Scalability experiment results with 128 PEs and 0 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
4	107.985	$\pm 2.58 \cdot 10^{-1}$	178,505	138.345	$\pm 4.10 \cdot 10^{-1}$	349,629	15.505	$\pm 5.73 \cdot 10^{-1}$	4,127
5	77.607	$\pm 2.36 \cdot 10^{-1}$	137,027	85.081	$\pm 3.46 \cdot 10^{-1}$	262,230	14.275	$\pm 3.56 \cdot 10^{-1}$	4,127
6	56.512	$\pm 2.30 \cdot 10^{-1}$	99,495	51.003	$\pm 2.88 \cdot 10^{-1}$	202,477	13.200	$\pm 1.59 \cdot 10^{-1}$	4,126
7	43.899	$\pm 2.10 \cdot 10^{-1}$	80,109	33.358	$\pm 2.32 \cdot 10^{-1}$	173,240	13.046	$\pm 1.29 \cdot 10^{-1}$	4,127
8	36.582	$\pm 1.94 \cdot 10^{-1}$	69,022	24.452	$\pm 1.99 \cdot 10^{-1}$	158,171	13.044	$\pm 1.25 \cdot 10^{-1}$	4,127
9	29.806	$\pm 1.73 \cdot 10^{-1}$	58,498	17.103	$\pm 1.63 \cdot 10^{-1}$	144,510	12.783	$\pm 6.53 \cdot 10^{-2}$	4,127
10	26.763	$\pm 1.60 \cdot 10^{-1}$	54,644	14.018	$\pm 1.43 \cdot 10^{-1}$	140,409	12.791	$\pm 6.49 \cdot 10^{-2}$	4,127
12	22.249	$\pm 1.27 \cdot 10^{-1}$	50,895	9.811	$\pm 1.06 \cdot 10^{-1}$	134,998	12.707	$\pm 5.03 \cdot 10^{-2}$	4,127
14	20.138	$\pm 1.15 \cdot 10^{-1}$	47,469	8.191	$\pm 9.72 \cdot 10^{-2}$	130,793	12.644	$\pm 2.95 \cdot 10^{-2}$	4,127
16	17.511	$\pm 9.17 \cdot 10^{-2}$	42,206	6.211	$\pm 8.04 \cdot 10^{-2}$	124,088	12.604	$\pm 1.88 \cdot 10^{-2}$	4,127
20	15.399	$\pm 6.79 \cdot 10^{-2}$	38,401	4.571	$\pm 6.28 \cdot 10^{-2}$	120,959	12.605	$\pm 2.06 \cdot 10^{-2}$	4,126
24	14.834	$\pm 5.59 \cdot 10^{-2}$	40,077	4.317	$\pm 5.66 \cdot 10^{-2}$	123,135	12.601	$\pm 2.15 \cdot 10^{-2}$	4,127
28	14.014	$\pm 4.58 \cdot 10^{-2}$	36,956	3.630	$\pm 4.84 \cdot 10^{-2}$	119,135	12.613	$\pm 2.28 \cdot 10^{-2}$	4,126
32	13.814	$\pm 4.28 \cdot 10^{-2}$	36,641	3.557	$\pm 4.59 \cdot 10^{-2}$	118,086	12.584	$\pm 1.32 \cdot 10^{-2}$	4,127
40	13.224	$\pm 3.45 \cdot 10^{-2}$	34,090	3.122	$\pm 4.46 \cdot 10^{-2}$	116,633	12.575	$\pm 9.48 \cdot 10^{-3}$	4,127
48	13.080	$\pm 3.08 \cdot 10^{-2}$	35,057	3.135	$\pm 4.21 \cdot 10^{-2}$	116,934	12.567	$\pm 6.07 \cdot 10^{-3}$	4,126
56	12.880	$\pm 2.75 \cdot 10^{-2}$	34,156	2.942	$\pm 4.15 \cdot 10^{-2}$	116,948	12.566	$\pm 5.29 \cdot 10^{-3}$	4,127
64	12.748	$\pm 2.43 \cdot 10^{-2}$	34,833	2.941	$\pm 4.09 \cdot 10^{-2}$	117,243	12.567	$\pm 4.45 \cdot 10^{-3}$	4,127
80	12.689	$\pm 2.25 \cdot 10^{-2}$	35,561	3.017	$\pm 4.45 \cdot 10^{-2}$	117,734	12.562	$\pm 4.55 \cdot 10^{-4}$	4,126
96	12.588	$\pm 2.15 \cdot 10^{-2}$	33,675	2.867	$\pm 4.54 \cdot 10^{-2}$	115,988	12.568	$\pm 5.34 \cdot 10^{-3}$	4,127
112	12.489	$\pm 1.93 \cdot 10^{-2}$	33,619	2.899	$\pm 4.99 \cdot 10^{-2}$	114,858	12.569	$\pm 5.35 \cdot 10^{-3}$	4,127
128	12.419	$\pm 1.69 \cdot 10^{-2}$	33,926	2.686	$\pm 3.23 \cdot 10^{-2}$	114,390	12.562	$\pm 3.14 \cdot 10^{-4}$	4,127

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
4	107.599	$\pm 2.08 \cdot 10^{-1}$	270,545	123.543	$\pm 3.58 \cdot 10^{-1}$	92,039	110.126	$\pm 2.55 \cdot 10^{-1}$	174,378
5	74.369	$\pm 2.08 \cdot 10^{-1}$	186,781	87.648	$\pm 4.15 \cdot 10^{-1}$	49,753	79.525	$\pm 2.35 \cdot 10^{-1}$	132,900
6	51.082	$\pm 2.13 \cdot 10^{-1}$	128,527	62.282	$\pm 4.71 \cdot 10^{-1}$	29,031	58.337	$\pm 2.33 \cdot 10^{-1}$	95,369
7	36.713	$\pm 1.98 \cdot 10^{-1}$	100,718	45.609	$\pm 4.60 \cdot 10^{-1}$	20,609	45.526	$\pm 2.15 \cdot 10^{-1}$	75,982
8	28.558	$\pm 1.88 \cdot 10^{-1}$	85,862	36.432	$\pm 4.41 \cdot 10^{-1}$	16,840	38.029	$\pm 2.01 \cdot 10^{-1}$	64,895
9	21.442	$\pm 1.74 \cdot 10^{-1}$	72,333	30.003	$\pm 4.12 \cdot 10^{-1}$	13,835	31.048	$\pm 1.82 \cdot 10^{-1}$	54,371
10	17.977	$\pm 1.59 \cdot 10^{-1}$	67,435	25.573	$\pm 3.73 \cdot 10^{-1}$	12,791	27.854	$\pm 1.69 \cdot 10^{-1}$	50,517
12	12.747	$\pm 1.29 \cdot 10^{-1}$	62,296	20.446	$\pm 2.92 \cdot 10^{-1}$	11,401	23.040	$\pm 1.35 \cdot 10^{-1}$	46,768
14	10.614	$\pm 1.28 \cdot 10^{-1}$	58,366	18.530	$\pm 2.51 \cdot 10^{-1}$	10,897	20.800	$\pm 1.24 \cdot 10^{-1}$	43,342
16	7.842	$\pm 1.11 \cdot 10^{-1}$	52,513	16.666	$\pm 2.00 \cdot 10^{-1}$	10,307	17.992	$\pm 1.00 \cdot 10^{-1}$	38,079
20	5.530	$\pm 9.49 \cdot 10^{-2}$	48,175	15.080	$\pm 1.48 \cdot 10^{-1}$	9,773	15.683	$\pm 7.53 \cdot 10^{-2}$	34,275
24	4.960	$\pm 8.35 \cdot 10^{-2}$	49,818	14.682	$\pm 1.24 \cdot 10^{-1}$	9,741	15.039	$\pm 6.18 \cdot 10^{-2}$	35,950
28	3.906	$\pm 7.23 \cdot 10^{-2}$	46,520	13.933	$\pm 9.53 \cdot 10^{-2}$	9,565	14.138	$\pm 5.12 \cdot 10^{-2}$	32,830
32	3.734	$\pm 6.62 \cdot 10^{-2}$	46,217	13.786	$\pm 8.06 \cdot 10^{-2}$	9,577	13.917	$\pm 4.79 \cdot 10^{-2}$	32,514
40	3.121	$\pm 7.44 \cdot 10^{-2}$	43,482	13.339	$\pm 5.81 \cdot 10^{-2}$	9,391	13.260	$\pm 3.92 \cdot 10^{-2}$	29,963
48	3.023	$\pm 6.55 \cdot 10^{-2}$	44,414	13.248	$\pm 5.11 \cdot 10^{-2}$	9,357	13.096	$\pm 3.49 \cdot 10^{-2}$	30,931
56	2.771	$\pm 6.97 \cdot 10^{-2}$	43,478	13.001	$\pm 3.87 \cdot 10^{-2}$	9,322	12.871	$\pm 3.12 \cdot 10^{-2}$	30,029
64	2.659	$\pm 6.71 \cdot 10^{-2}$	44,171	12.928	$\pm 3.17 \cdot 10^{-2}$	9,337	12.720	$\pm 2.75 \cdot 10^{-2}$	30,706
80	2.667	$\pm 7.57 \cdot 10^{-2}$	44,956	12.828	$\pm 2.60 \cdot 10^{-2}$	9,395	12.653	$\pm 2.54 \cdot 10^{-2}$	31,435
96	2.602	$\pm 8.69 \cdot 10^{-2}$	42,976	12.783	$\pm 2.34 \cdot 10^{-2}$	9,302	12.538	$\pm 2.45 \cdot 10^{-2}$	29,548
112	2.621	$\pm 1.01 \cdot 10^{-1}$	42,915	12.707	$\pm 1.76 \cdot 10^{-2}$	9,296	12.425	$\pm 2.20 \cdot 10^{-2}$	29,492
128	2.058	$\pm 3.13 \cdot 10^{-2}$	43,171	12.627	$\pm 6.93 \cdot 10^{-3}$	9,245	12.347	$\pm 1.92 \cdot 10^{-2}$	29,799

Table B.14: Scalability experiment results with 128 PEs and 500 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
4	111.928	$\pm 2.62 \cdot 10^{-1}$	187,279	151.176	$\pm 4.23 \cdot 10^{-1}$	368,393	16.822	$\pm 5.79 \cdot 10^{-1}$	4,127
5	78.766	$\pm 2.43 \cdot 10^{-1}$	136,582	89.038	$\pm 3.61 \cdot 10^{-1}$	266,225	16.066	$\pm 4.45 \cdot 10^{-1}$	4,126
6	59.352	$\pm 2.20 \cdot 10^{-1}$	109,663	57.933	$\pm 3.01 \cdot 10^{-1}$	219,163	15.174	$\pm 2.95 \cdot 10^{-1}$	4,127
7	47.837	$\pm 2.01 \cdot 10^{-1}$	94,247	41.366	$\pm 2.54 \cdot 10^{-1}$	194,407	14.572	$\pm 1.71 \cdot 10^{-1}$	4,127
8	39.247	$\pm 1.86 \cdot 10^{-1}$	78,640	29.043	$\pm 2.14 \cdot 10^{-1}$	172,430	14.364	$\pm 1.46 \cdot 10^{-1}$	4,127
9	32.448	$\pm 1.74 \cdot 10^{-1}$	63,084	20.599	$\pm 1.83 \cdot 10^{-1}$	152,255	14.516	$\pm 1.77 \cdot 10^{-1}$	4,127
10	27.755	$\pm 1.58 \cdot 10^{-1}$	53,900	15.030	$\pm 1.52 \cdot 10^{-1}$	138,454	14.087	$\pm 8.44 \cdot 10^{-2}$	4,127
12	23.414	$\pm 1.29 \cdot 10^{-1}$	49,860	11.278	$\pm 1.28 \cdot 10^{-1}$	133,076	13.945	$\pm 4.55 \cdot 10^{-2}$	4,127
14	20.912	$\pm 1.08 \cdot 10^{-1}$	46,493	9.027	$\pm 1.06 \cdot 10^{-1}$	129,016	13.963	$\pm 5.46 \cdot 10^{-2}$	4,126
16	19.917	$\pm 9.59 \cdot 10^{-2}$	47,218	8.286	$\pm 9.63 \cdot 10^{-2}$	129,257	13.873	$\pm 2.55 \cdot 10^{-2}$	4,127
20	16.976	$\pm 6.81 \cdot 10^{-2}$	39,821	5.726	$\pm 7.47 \cdot 10^{-2}$	121,235	13.889	$\pm 2.88 \cdot 10^{-2}$	4,127
24	16.052	$\pm 5.93 \cdot 10^{-2}$	37,632	5.137	$\pm 7.52 \cdot 10^{-2}$	119,396	13.890	$\pm 2.65 \cdot 10^{-2}$	4,127
28	15.755	$\pm 5.08 \cdot 10^{-2}$	40,553	5.120	$\pm 6.86 \cdot 10^{-2}$	121,468	13.848	$\pm 1.57 \cdot 10^{-2}$	4,127
32	15.225	$\pm 4.23 \cdot 10^{-2}$	38,895	4.687	$\pm 6.28 \cdot 10^{-2}$	120,055	13.844	$\pm 1.51 \cdot 10^{-2}$	4,127
40	14.727	$\pm 3.62 \cdot 10^{-2}$	36,498	4.225	$\pm 5.79 \cdot 10^{-2}$	117,731	13.826	$\pm 7.81 \cdot 10^{-3}$	4,127
48	14.253	$\pm 3.05 \cdot 10^{-2}$	34,110	3.852	$\pm 5.49 \cdot 10^{-2}$	115,971	13.824	$\pm 7.03 \cdot 10^{-3}$	4,127
56	14.190	$\pm 2.72 \cdot 10^{-2}$	35,634	3.886	$\pm 5.46 \cdot 10^{-2}$	117,267	13.826	$\pm 7.55 \cdot 10^{-3}$	4,127
64	14.013	$\pm 2.33 \cdot 10^{-2}$	35,661	3.862	$\pm 5.65 \cdot 10^{-2}$	116,296	13.819	$\pm 2.50 \cdot 10^{-3}$	4,127
80	13.971	$\pm 2.20 \cdot 10^{-2}$	35,751	3.880	$\pm 5.91 \cdot 10^{-2}$	116,734	13.819	$\pm 2.53 \cdot 10^{-3}$	4,127
96	13.880	$\pm 1.95 \cdot 10^{-2}$	36,299	3.859	$\pm 6.35 \cdot 10^{-2}$	117,220	13.818	$\pm 2.33 \cdot 10^{-3}$	4,127
112	13.813	$\pm 1.82 \cdot 10^{-2}$	36,003	4.005	$\pm 7.74 \cdot 10^{-2}$	115,614	13.817	$\pm 2.20 \cdot 10^{-3}$	4,127
128	13.687	$\pm 1.68 \cdot 10^{-2}$	34,630	3.770	$\pm 6.53 \cdot 10^{-2}$	114,858	13.817	$\pm 2.15 \cdot 10^{-3}$	4,127

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
4	116.531	$\pm 2.16 \cdot 10^{-1}$	290,437	124.430	$\pm 3.40 \cdot 10^{-1}$	103,158	114.023	$\pm 2.59 \cdot 10^{-1}$	183,152
5	79.426	$\pm 2.23 \cdot 10^{-1}$	188,055	86.841	$\pm 4.02 \cdot 10^{-1}$	51,472	80.672	$\pm 2.43 \cdot 10^{-1}$	132,456
6	57.406	$\pm 2.18 \cdot 10^{-1}$	142,417	64.257	$\pm 4.35 \cdot 10^{-1}$	32,754	61.031	$\pm 2.22 \cdot 10^{-1}$	105,536
7	43.659	$\pm 2.06 \cdot 10^{-1}$	118,818	50.057	$\pm 4.27 \cdot 10^{-1}$	24,572	49.312	$\pm 2.05 \cdot 10^{-1}$	90,120
8	33.316	$\pm 2.00 \cdot 10^{-1}$	97,472	39.019	$\pm 4.14 \cdot 10^{-1}$	18,832	40.576	$\pm 1.91 \cdot 10^{-1}$	74,513
9	25.814	$\pm 1.97 \cdot 10^{-1}$	78,197	33.141	$\pm 3.98 \cdot 10^{-1}$	15,114	33.653	$\pm 1.82 \cdot 10^{-1}$	58,957
10	19.736	$\pm 1.78 \cdot 10^{-1}$	66,846	26.354	$\pm 3.49 \cdot 10^{-1}$	12,947	28.838	$\pm 1.67 \cdot 10^{-1}$	49,773
12	15.178	$\pm 1.69 \cdot 10^{-1}$	61,759	22.162	$\pm 2.90 \cdot 10^{-1}$	11,899	24.218	$\pm 1.38 \cdot 10^{-1}$	45,733
14	12.061	$\pm 1.46 \cdot 10^{-1}$	57,741	19.770	$\pm 2.36 \cdot 10^{-1}$	11,248	21.537	$\pm 1.17 \cdot 10^{-1}$	42,367
16	10.744	$\pm 1.30 \cdot 10^{-1}$	58,172	18.533	$\pm 2.00 \cdot 10^{-1}$	10,954	20.445	$\pm 1.03 \cdot 10^{-1}$	43,091
20	7.490	$\pm 1.15 \cdot 10^{-1}$	49,945	16.562	$\pm 1.48 \cdot 10^{-1}$	10,125	17.280	$\pm 7.50 \cdot 10^{-2}$	35,696
24	6.660	$\pm 1.30 \cdot 10^{-1}$	47,539	15.938	$\pm 1.29 \cdot 10^{-1}$	9,908	16.265	$\pm 6.60 \cdot 10^{-2}$	33,505
28	6.307	$\pm 1.09 \cdot 10^{-1}$	50,563	15.398	$\pm 1.05 \cdot 10^{-1}$	10,011	15.919	$\pm 5.61 \cdot 10^{-2}$	36,426
32	5.606	$\pm 1.02 \cdot 10^{-1}$	48,828	15.115	$\pm 8.28 \cdot 10^{-2}$	9,933	15.336	$\pm 4.70 \cdot 10^{-2}$	34,768
40	5.020	$\pm 9.87 \cdot 10^{-2}$	46,168	14.651	$\pm 5.94 \cdot 10^{-2}$	9,670	14.789	$\pm 4.06 \cdot 10^{-2}$	32,371
48	4.493	$\pm 9.53 \cdot 10^{-2}$	43,651	14.498	$\pm 4.96 \cdot 10^{-2}$	9,541	14.259	$\pm 3.46 \cdot 10^{-2}$	29,983
56	4.503	$\pm 9.76 \cdot 10^{-2}$	45,244	14.272	$\pm 3.86 \cdot 10^{-2}$	9,610	14.185	$\pm 3.07 \cdot 10^{-2}$	31,507
64	4.405	$\pm 1.04 \cdot 10^{-1}$	45,257	14.196	$\pm 3.33 \cdot 10^{-2}$	9,596	13.986	$\pm 2.63 \cdot 10^{-2}$	31,534
80	4.410	$\pm 1.13 \cdot 10^{-1}$	45,329	14.087	$\pm 2.51 \cdot 10^{-2}$	9,580	13.938	$\pm 2.49 \cdot 10^{-2}$	31,624
96	4.426	$\pm 1.27 \cdot 10^{-1}$	45,870	14.042	$\pm 2.18 \cdot 10^{-2}$	9,571	13.835	$\pm 2.20 \cdot 10^{-2}$	32,172
112	4.782	$\pm 1.67 \cdot 10^{-1}$	45,528	13.971	$\pm 3.93 \cdot 10^{-2}$	9,527	13.760	$\pm 2.06 \cdot 10^{-2}$	31,876
128	4.410	$\pm 1.38 \cdot 10^{-1}$	44,069	13.901	$\pm 3.55 \cdot 10^{-2}$	9,441	13.617	$\pm 1.90 \cdot 10^{-2}$	30,503

Table B.15: Scalability experiment results with 128 PEs and 1000 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
4	119.897	$\pm 2.77 \cdot 10^{-1}$	206,059	173.740	$\pm 4.43 \cdot 10^{-1}$	408,531	17.697	$\pm 5.05 \cdot 10^{-1}$	4,127
5	82.343	$\pm 2.51 \cdot 10^{-1}$	145,835	99.663	$\pm 3.80 \cdot 10^{-1}$	280,852	17.159	$\pm 4.19 \cdot 10^{-1}$	4,126
6	60.989	$\pm 2.30 \cdot 10^{-1}$	108,045	62.399	$\pm 3.20 \cdot 10^{-1}$	214,844	16.308	$\pm 2.69 \cdot 10^{-1}$	4,127
7	49.234	$\pm 2.02 \cdot 10^{-1}$	93,515	44.836	$\pm 2.75 \cdot 10^{-1}$	192,646	16.029	$\pm 2.18 \cdot 10^{-1}$	4,127
8	41.632	$\pm 1.83 \cdot 10^{-1}$	82,304	35.027	$\pm 2.54 \cdot 10^{-1}$	174,114	15.438	$\pm 9.03 \cdot 10^{-2}$	4,127
9	36.832	$\pm 1.70 \cdot 10^{-1}$	76,759	28.021	$\pm 2.14 \cdot 10^{-1}$	167,526	15.327	$\pm 8.18 \cdot 10^{-2}$	4,127
10	31.485	$\pm 1.54 \cdot 10^{-1}$	64,420	20.785	$\pm 1.86 \cdot 10^{-1}$	151,182	15.207	$\pm 4.37 \cdot 10^{-2}$	4,127
12	24.740	$\pm 1.27 \cdot 10^{-1}$	50,807	12.986	$\pm 1.47 \cdot 10^{-1}$	134,034	15.246	$\pm 5.20 \cdot 10^{-2}$	4,127
14	21.545	$\pm 1.00 \cdot 10^{-1}$	45,403	9.930	$\pm 1.26 \cdot 10^{-1}$	127,718	15.165	$\pm 4.12 \cdot 10^{-2}$	4,127
16	20.886	$\pm 9.53 \cdot 10^{-2}$	45,685	9.482	$\pm 1.20 \cdot 10^{-1}$	127,349	15.151	$\pm 3.03 \cdot 10^{-2}$	4,127
20	18.941	$\pm 7.26 \cdot 10^{-2}$	44,514	8.243	$\pm 1.25 \cdot 10^{-1}$	125,917	15.120	$\pm 2.43 \cdot 10^{-2}$	4,127
24	17.574	$\pm 5.96 \cdot 10^{-2}$	39,918	6.911	$\pm 1.16 \cdot 10^{-1}$	121,184	15.090	$\pm 1.01 \cdot 10^{-2}$	4,127
28	16.652	$\pm 4.77 \cdot 10^{-2}$	37,505	6.168	$\pm 1.10 \cdot 10^{-1}$	118,904	15.096	$\pm 1.24 \cdot 10^{-2}$	4,127
32	16.594	$\pm 4.42 \cdot 10^{-2}$	40,148	6.439	$\pm 1.18 \cdot 10^{-1}$	121,166	15.087	$\pm 1.01 \cdot 10^{-2}$	4,126
40	16.111	$\pm 3.65 \cdot 10^{-2}$	38,833	6.037	$\pm 1.18 \cdot 10^{-1}$	120,257	15.090	$\pm 1.03 \cdot 10^{-2}$	4,127
48	15.707	$\pm 3.21 \cdot 10^{-2}$	37,142	5.257	$\pm 8.81 \cdot 10^{-2}$	119,340	15.094	$\pm 1.13 \cdot 10^{-2}$	4,127
56	15.429	$\pm 2.56 \cdot 10^{-2}$	35,320	5.158	$\pm 9.08 \cdot 10^{-2}$	116,341	15.086	$\pm 9.23 \cdot 10^{-3}$	4,127
64	15.276	$\pm 2.61 \cdot 10^{-2}$	35,670	5.170	$\pm 9.92 \cdot 10^{-2}$	117,464	15.080	$\pm 7.07 \cdot 10^{-3}$	4,127
80	15.228	$\pm 2.13 \cdot 10^{-2}$	34,996	5.225	$\pm 1.06 \cdot 10^{-1}$	115,735	15.087	$\pm 9.80 \cdot 10^{-3}$	4,127
96	15.110	$\pm 2.10 \cdot 10^{-2}$	34,779	5.157	$\pm 1.06 \cdot 10^{-1}$	116,289	15.081	$\pm 6.03 \cdot 10^{-3}$	4,127
112	15.107	$\pm 1.79 \cdot 10^{-2}$	32,820	5.202	$\pm 1.21 \cdot 10^{-1}$	116,882	15.078	$\pm 5.06 \cdot 10^{-3}$	4,126
127	15.032	$\pm 1.38 \cdot 10^{-2}$	35,781	4.951	$\pm 9.41 \cdot 10^{-2}$	118,690	15.077	$\pm 4.89 \cdot 10^{-3}$	4,126
128	14.948	$\pm 1.71 \cdot 10^{-2}$	34,516	5.143	$\pm 1.24 \cdot 10^{-1}$	119,252	15.081	$\pm 5.75 \cdot 10^{-3}$	4,126

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
4	129.530	$\pm 2.34 \cdot 10^{-1}$	332,447	134.016	$\pm 3.47 \cdot 10^{-1}$	126,389	121.939	$\pm 2.75 \cdot 10^{-1}$	201,932
5	86.919	$\pm 2.39 \cdot 10^{-1}$	204,405	91.556	$\pm 4.10 \cdot 10^{-1}$	58,573	84.193	$\pm 2.51 \cdot 10^{-1}$	141,709
6	61.201	$\pm 2.37 \cdot 10^{-1}$	142,257	66.103	$\pm 4.44 \cdot 10^{-1}$	34,212	62.715	$\pm 2.32 \cdot 10^{-1}$	103,918
7	47.714	$\pm 2.38 \cdot 10^{-1}$	119,338	51.764	$\pm 4.37 \cdot 10^{-1}$	25,825	50.718	$\pm 2.06 \cdot 10^{-1}$	89,388
8	39.043	$\pm 2.52 \cdot 10^{-1}$	103,302	42.744	$\pm 4.49 \cdot 10^{-1}$	21,000	42.965	$\pm 1.88 \cdot 10^{-1}$	78,179
9	32.495	$\pm 2.12 \cdot 10^{-1}$	94,647	38.225	$\pm 4.15 \cdot 10^{-1}$	17,890	38.005	$\pm 1.76 \cdot 10^{-1}$	72,632
10	25.753	$\pm 2.07 \cdot 10^{-1}$	79,500	30.804	$\pm 3.75 \cdot 10^{-1}$	15,080	32.549	$\pm 1.60 \cdot 10^{-1}$	60,293
12	17.773	$\pm 2.05 \cdot 10^{-1}$	63,151	23.846	$\pm 3.11 \cdot 10^{-1}$	12,344	25.529	$\pm 1.36 \cdot 10^{-1}$	46,680
14	14.072	$\pm 1.96 \cdot 10^{-1}$	56,713	20.896	$\pm 2.57 \cdot 10^{-1}$	11,310	22.132	$\pm 1.08 \cdot 10^{-1}$	41,276
16	13.195	$\pm 1.86 \cdot 10^{-1}$	56,830	19.763	$\pm 2.27 \cdot 10^{-1}$	11,145	21.405	$\pm 1.03 \cdot 10^{-1}$	41,558
20	11.510	$\pm 2.21 \cdot 10^{-1}$	55,325	18.549	$\pm 2.18 \cdot 10^{-1}$	10,812	19.280	$\pm 7.90 \cdot 10^{-2}$	40,387
24	9.929	$\pm 2.23 \cdot 10^{-1}$	50,202	17.496	$\pm 1.87 \cdot 10^{-1}$	10,284	17.809	$\pm 6.58 \cdot 10^{-2}$	35,791
28	9.032	$\pm 2.24 \cdot 10^{-1}$	47,594	16.716	$\pm 1.58 \cdot 10^{-1}$	10,089	16.792	$\pm 5.33 \cdot 10^{-2}$	33,378
32	9.105	$\pm 2.38 \cdot 10^{-1}$	50,347	16.612	$\pm 1.51 \cdot 10^{-1}$	10,199	16.715	$\pm 4.90 \cdot 10^{-2}$	36,022
40	8.666	$\pm 2.48 \cdot 10^{-1}$	48,935	16.241	$\pm 1.58 \cdot 10^{-1}$	10,103	16.180	$\pm 4.07 \cdot 10^{-2}$	34,706
48	7.366	$\pm 1.81 \cdot 10^{-1}$	47,008	15.912	$\pm 1.05 \cdot 10^{-1}$	9,867	15.732	$\pm 3.60 \cdot 10^{-2}$	33,015
56	7.196	$\pm 1.93 \cdot 10^{-1}$	45,141	15.651	$\pm 9.75 \cdot 10^{-2}$	9,822	15.422	$\pm 2.89 \cdot 10^{-2}$	31,193
64	7.391	$\pm 2.19 \cdot 10^{-1}$	45,489	15.555	$\pm 8.64 \cdot 10^{-2}$	9,819	15.249	$\pm 2.95 \cdot 10^{-2}$	31,543
80	7.566	$\pm 2.37 \cdot 10^{-1}$	44,781	15.475	$\pm 1.09 \cdot 10^{-1}$	9,785	15.194	$\pm 2.41 \cdot 10^{-2}$	30,869
96	7.455	$\pm 2.41 \cdot 10^{-1}$	44,531	15.386	$\pm 8.79 \cdot 10^{-2}$	9,752	15.061	$\pm 2.38 \cdot 10^{-2}$	30,652
112	7.998	$\pm 2.95 \cdot 10^{-1}$	42,508	15.317	$\pm 1.09 \cdot 10^{-1}$	9,687	15.058	$\pm 2.04 \cdot 10^{-2}$	28,694
127	6.986	$\pm 2.12 \cdot 10^{-1}$	45,550	15.193	$\pm 5.54 \cdot 10^{-2}$	9,768	14.974	$\pm 1.56 \cdot 10^{-2}$	31,655
128	8.047	$\pm 3.03 \cdot 10^{-1}$	44,281	15.208	$\pm 7.80 \cdot 10^{-2}$	9,766	14.877	$\pm 1.94 \cdot 10^{-2}$	30,390

Table B.16: Scalability experiment results with 128 PEs and 2000 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
4	142.055	$\pm 3.53 \cdot 10^{-1}$	241,676	242.319	$\pm 5.47 \cdot 10^{-1}$	524,549	22.710	$\pm 9.76 \cdot 10^{-1}$	4,127
5	99.394	$\pm 3.04 \cdot 10^{-1}$	174,184	143.854	$\pm 5.11 \cdot 10^{-1}$	343,871	19.700	$\pm 4.83 \cdot 10^{-1}$	4,127
6	76.473	$\pm 2.57 \cdot 10^{-1}$	143,813	99.292	$\pm 4.78 \cdot 10^{-1}$	276,941	19.886	$\pm 5.03 \cdot 10^{-1}$	4,126
7	59.887	$\pm 2.34 \cdot 10^{-1}$	107,229	70.326	$\pm 4.80 \cdot 10^{-1}$	216,215	18.745	$\pm 2.51 \cdot 10^{-1}$	4,126
8	49.232	$\pm 2.06 \cdot 10^{-1}$	94,701	55.227	$\pm 4.83 \cdot 10^{-1}$	195,756	18.324	$\pm 1.56 \cdot 10^{-1}$	4,127
9	41.667	$\pm 1.89 \cdot 10^{-1}$	77,788	46.413	$\pm 5.29 \cdot 10^{-1}$	172,108	18.393	$\pm 1.78 \cdot 10^{-1}$	4,127
10	38.238	$\pm 1.66 \cdot 10^{-1}$	77,431	41.019	$\pm 4.87 \cdot 10^{-1}$	169,194	18.108	$\pm 1.32 \cdot 10^{-1}$	4,126
12	31.716	$\pm 1.46 \cdot 10^{-1}$	63,657	34.261	$\pm 5.11 \cdot 10^{-1}$	151,022	17.870	$\pm 7.27 \cdot 10^{-2}$	4,127
14	27.769	$\pm 1.19 \cdot 10^{-1}$	58,536	30.690	$\pm 5.17 \cdot 10^{-1}$	144,496	17.842	$\pm 6.31 \cdot 10^{-2}$	4,127
16	25.404	$\pm 1.08 \cdot 10^{-1}$	54,996	28.444	$\pm 5.21 \cdot 10^{-1}$	139,644	17.740	$\pm 4.06 \cdot 10^{-2}$	4,127
20	22.469	$\pm 8.77 \cdot 10^{-2}$	51,537	26.054	$\pm 5.26 \cdot 10^{-1}$	135,543	17.655	$\pm 1.98 \cdot 10^{-2}$	4,127
24	20.990	$\pm 7.62 \cdot 10^{-2}$	46,726	23.898	$\pm 5.05 \cdot 10^{-1}$	129,706	17.667	$\pm 2.42 \cdot 10^{-2}$	4,127
28	19.453	$\pm 6.12 \cdot 10^{-2}$	40,375	23.432	$\pm 5.42 \cdot 10^{-1}$	122,863	17.659	$\pm 2.21 \cdot 10^{-2}$	4,126
32	19.262	$\pm 5.63 \cdot 10^{-2}$	42,868	23.206	$\pm 5.28 \cdot 10^{-1}$	125,453	17.652	$\pm 2.01 \cdot 10^{-2}$	4,126
40	18.983	$\pm 4.40 \cdot 10^{-2}$	44,358	22.002	$\pm 4.89 \cdot 10^{-1}$	126,872	17.653	$\pm 2.02 \cdot 10^{-2}$	4,127
48	18.632	$\pm 4.23 \cdot 10^{-2}$	42,325	21.912	$\pm 4.96 \cdot 10^{-1}$	124,702	17.637	$\pm 1.56 \cdot 10^{-2}$	4,127
56	18.258	$\pm 4.03 \cdot 10^{-2}$	40,263	21.947	$\pm 5.13 \cdot 10^{-1}$	122,311	17.635	$\pm 1.61 \cdot 10^{-2}$	4,126
64	18.086	$\pm 4.33 \cdot 10^{-2}$	40,459	21.317	$\pm 4.95 \cdot 10^{-1}$	123,129	17.635	$\pm 1.53 \cdot 10^{-2}$	4,127
80	17.911	$\pm 3.66 \cdot 10^{-2}$	39,086	21.066	$\pm 4.94 \cdot 10^{-1}$	121,682	17.633	$\pm 1.56 \cdot 10^{-2}$	4,127
96	17.806	$\pm 3.90 \cdot 10^{-2}$	39,107	21.594	$\pm 5.10 \cdot 10^{-1}$	121,386	17.633	$\pm 1.53 \cdot 10^{-2}$	4,127
112	17.650	$\pm 1.90 \cdot 10^{-2}$	37,213	20.543	$\pm 4.91 \cdot 10^{-1}$	124,509	17.617	$\pm 1.29 \cdot 10^{-2}$	4,126
128	17.552	$\pm 2.31 \cdot 10^{-2}$	37,753	19.895	$\pm 4.74 \cdot 10^{-1}$	126,983	17.615	$\pm 1.26 \cdot 10^{-2}$	4,126

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
4	170.613	$\pm 3.31 \cdot 10^{-1}$	443,916	173.067	$\pm 4.28 \cdot 10^{-1}$	202,240	144.081	$\pm 3.54 \cdot 10^{-1}$	237,549
5	118.175	$\pm 3.87 \cdot 10^{-1}$	265,193	118.439	$\pm 5.33 \cdot 10^{-1}$	91,009	101.280	$\pm 3.05 \cdot 10^{-1}$	170,057
6	90.329	$\pm 4.30 \cdot 10^{-1}$	200,275	89.866	$\pm 6.25 \cdot 10^{-1}$	56,462	78.097	$\pm 2.59 \cdot 10^{-1}$	139,687
7	73.173	$\pm 5.25 \cdot 10^{-1}$	142,878	68.384	$\pm 7.28 \cdot 10^{-1}$	35,650	61.485	$\pm 2.37 \cdot 10^{-1}$	103,103
8	62.795	$\pm 5.95 \cdot 10^{-1}$	122,187	55.906	$\pm 8.31 \cdot 10^{-1}$	27,487	50.591	$\pm 2.11 \cdot 10^{-1}$	90,574
9	58.506	$\pm 7.36 \cdot 10^{-1}$	100,041	49.366	$\pm 9.95 \cdot 10^{-1}$	22,254	42.921	$\pm 1.95 \cdot 10^{-1}$	73,661
10	52.542	$\pm 6.95 \cdot 10^{-1}$	98,076	41.912	$\pm 8.78 \cdot 10^{-1}$	20,646	39.322	$\pm 1.71 \cdot 10^{-1}$	73,305
12	49.233	$\pm 8.17 \cdot 10^{-1}$	80,903	34.851	$\pm 9.04 \cdot 10^{-1}$	17,247	32.626	$\pm 1.53 \cdot 10^{-1}$	59,530
14	46.585	$\pm 8.73 \cdot 10^{-1}$	74,571	31.161	$\pm 9.01 \cdot 10^{-1}$	16,036	28.472	$\pm 1.26 \cdot 10^{-1}$	54,409
16	45.394	$\pm 9.17 \cdot 10^{-1}$	69,873	28.785	$\pm 9.08 \cdot 10^{-1}$	14,878	25.975	$\pm 1.15 \cdot 10^{-1}$	50,869
20	43.937	$\pm 9.77 \cdot 10^{-1}$	65,544	25.603	$\pm 8.22 \cdot 10^{-1}$	14,008	22.838	$\pm 9.44 \cdot 10^{-2}$	47,410
24	42.566	$\pm 9.87 \cdot 10^{-1}$	59,984	23.804	$\pm 7.30 \cdot 10^{-1}$	13,260	21.261	$\pm 8.29 \cdot 10^{-2}$	42,599
28	45.573	$\pm 1.13 \cdot 10^{0}$	53,172	23.131	$\pm 7.73 \cdot 10^{-1}$	12,799	19.606	$\pm 6.78 \cdot 10^{-2}$	36,249
32	43.889	$\pm 1.09 \cdot 10^{0}$	55,728	22.085	$\pm 6.87 \cdot 10^{-1}$	12,861	19.382	$\pm 6.21 \cdot 10^{-2}$	38,742
40	40.717	$\pm 9.97 \cdot 10^{-1}$	57,231	20.946	$\pm 5.74 \cdot 10^{-1}$	12,874	19.068	$\pm 4.82 \cdot 10^{-2}$	40,231
48	41.574	$\pm 1.03 \cdot 10^{0}$	55,057	20.705	$\pm 5.70 \cdot 10^{-1}$	12,734	18.688	$\pm 4.67 \cdot 10^{-2}$	38,198
56	43.048	$\pm 1.09 \cdot 10^{0}$	52,783	20.406	$\pm 5.83 \cdot 10^{-1}$	12,521	18.277	$\pm 4.48 \cdot 10^{-2}$	36,137
64	41.948	$\pm 1.07 \cdot 10^{0}$	52,975	19.511	$\pm 4.64 \cdot 10^{-1}$	12,517	18.086	$\pm 4.82 \cdot 10^{-2}$	36,332
80	42.264	$\pm 1.08 \cdot 10^{0}$	51,446	19.464	$\pm 4.79 \cdot 10^{-1}$	12,361	17.892	$\pm 4.09 \cdot 10^{-2}$	34,959
96	43.308	$\pm 1.11 \cdot 10^{0}$	51,492	19.422	$\pm 4.85 \cdot 10^{-1}$	12,385	17.775	$\pm 4.36 \cdot 10^{-2}$	34,980
112	43.940	$\pm 1.14 \cdot 10^{0}$	49,512	19.382	$\pm 4.87 \cdot 10^{-1}$	12,299	17.602	$\pm 2.14 \cdot 10^{-2}$	33,087
128	43.071	$\pm 1.12 \cdot 10^{0}$	50,021	18.463	$\pm 3.43 \cdot 10^{-1}$	12,267	17.492	$\pm 2.58 \cdot 10^{-2}$	33,627

Table B.17: Scalability experiment results with 254 PEs and 0 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
12	130.244	$\pm 2.89 \cdot 10^{-1}$	259,055	164.441	$\pm 4.17 \cdot 10^{-1}$	591,294	33.176	$\pm 1.20 \cdot 10^{0}$	4,251
16	84.496	$\pm 2.64 \cdot 10^{-1}$	157,360	77.510	$\pm 3.35 \cdot 10^{-1}$	366,784	25.753	$\pm 2.49 \cdot 10^{-1}$	4,251
20	60.602	$\pm 2.39 \cdot 10^{-1}$	101,226	39.268	$\pm 2.54 \cdot 10^{-1}$	271,648	24.195	$\pm 4.78 \cdot 10^{-4}$	4,003
24	48.240	$\pm 2.15 \cdot 10^{-1}$	76,642	24.622	$\pm 2.19 \cdot 10^{-1}$	233,505	24.195	$\pm 4.78 \cdot 10^{-4}$	4,003
32	38.826	$\pm 1.67 \cdot 10^{-1}$	63,164	14.981	$\pm 1.62 \cdot 10^{-1}$	211,793	24.195	$\pm 4.78 \cdot 10^{-4}$	4,003
48	30.866	$\pm 1.11 \cdot 10^{-1}$	51,427	8.244	$\pm 1.09 \cdot 10^{-1}$	197,011	24.195	$\pm 4.78 \cdot 10^{-4}$	4,003
64	28.063	$\pm 7.91 \cdot 10^{-2}$	49,401	6.425	$\pm 8.37 \cdot 10^{-2}$	192,480	24.195	$\pm 4.78 \cdot 10^{-4}$	4,003
80	27.004	$\pm 6.40 \cdot 10^{-2}$	50,255	5.756	$\pm 6.53 \cdot 10^{-2}$	192,603	24.195	$\pm 4.78 \cdot 10^{-4}$	4,003
96	26.095	$\pm 5.31 \cdot 10^{-2}$	47,710	5.042	$\pm 5.78 \cdot 10^{-2}$	190,520	24.195	$\pm 4.78 \cdot 10^{-4}$	4,003
112	25.593	$\pm 4.56 \cdot 10^{-2}$	47,501	5.357	$\pm 1.10 \cdot 10^{-1}$	194,383	24.195	$\pm 4.67 \cdot 10^{-4}$	4,003
128	25.250	$\pm 3.82 \cdot 10^{-2}$	47,290	4.488	$\pm 5.27 \cdot 10^{-2}$	193,168	24.195	$\pm 4.67 \cdot 10^{-4}$	4,003
160	24.966	$\pm 3.39 \cdot 10^{-2}$	46,065	4.331	$\pm 5.17 \cdot 10^{-2}$	194,433	24.195	$\pm 4.67 \cdot 10^{-4}$	4,003
192	24.653	$\pm 2.68 \cdot 10^{-2}$	46,203	4.088	$\pm 4.82 \cdot 10^{-2}$	193,080	24.195	$\pm 4.67 \cdot 10^{-4}$	4,003
254	24.222	$\pm 1.15 \cdot 10^{-2}$	47,080	4.033	$\pm 4.99 \cdot 10^{-2}$	193,489	24.195	$\pm 4.67 \cdot 10^{-4}$	4,003

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
12	129.181	$\pm 2.31 \cdot 10^{-1}$	430,764	161.673	$\pm 3.71 \cdot 10^{-1}$	171,709	131.818	$\pm 2.89 \cdot 10^{-1}$	254,804
16	77.351	$\pm 2.59 \cdot 10^{-1}$	221,661	103.798	$\pm 4.66 \cdot 10^{-1}$	64,303	86.080	$\pm 2.66 \cdot 10^{-1}$	153,109
20	48.461	$\pm 2.67 \cdot 10^{-1}$	133,793	72.349	$\pm 5.11 \cdot 10^{-1}$	32,568	62.052	$\pm 2.45 \cdot 10^{-1}$	97,223
24	35.361	$\pm 3.11 \cdot 10^{-1}$	98,846	57.100	$\pm 5.15 \cdot 10^{-1}$	22,206	49.516	$\pm 2.23 \cdot 10^{-1}$	72,639
32	23.177	$\pm 2.77 \cdot 10^{-1}$	80,230	42.888	$\pm 4.08 \cdot 10^{-1}$	17,066	39.767	$\pm 1.76 \cdot 10^{-1}$	59,161
48	13.327	$\pm 2.27 \cdot 10^{-1}$	64,780	32.359	$\pm 2.70 \cdot 10^{-1}$	13,353	31.378	$\pm 1.19 \cdot 10^{-1}$	47,424
64	9.775	$\pm 1.73 \cdot 10^{-1}$	61,743	28.982	$\pm 1.91 \cdot 10^{-1}$	12,342	28.354	$\pm 8.54 \cdot 10^{-2}$	45,398
80	7.982	$\pm 1.13 \cdot 10^{-1}$	62,425	27.524	$\pm 1.49 \cdot 10^{-1}$	12,170	27.196	$\pm 6.91 \cdot 10^{-2}$	46,252
96	6.733	$\pm 9.69 \cdot 10^{-2}$	59,435	26.702	$\pm 1.25 \cdot 10^{-1}$	11,726	26.219	$\pm 5.77 \cdot 10^{-2}$	43,707
112	8.423	$\pm 3.03 \cdot 10^{-1}$	59,295	26.338	$\pm 1.16 \cdot 10^{-1}$	11,794	25.670	$\pm 4.96 \cdot 10^{-2}$	43,498
128	5.746	$\pm 9.15 \cdot 10^{-2}$	58,836	25.756	$\pm 9.80 \cdot 10^{-2}$	11,547	25.297	$\pm 4.16 \cdot 10^{-2}$	43,287
160	5.583	$\pm 9.22 \cdot 10^{-2}$	57,506	25.511	$\pm 9.25 \cdot 10^{-2}$	11,441	24.989	$\pm 3.71 \cdot 10^{-2}$	42,062
192	5.094	$\pm 8.26 \cdot 10^{-2}$	57,464	25.224	$\pm 8.39 \cdot 10^{-2}$	11,261	24.645	$\pm 2.93 \cdot 10^{-2}$	42,200
254	4.866	$\pm 9.48 \cdot 10^{-2}$	58,335	24.683	$\pm 6.26 \cdot 10^{-2}$	11,255	24.174	$\pm 1.26 \cdot 10^{-2}$	43,077

Table B.18: Scalability experiment results with 254 PEs and 500 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
12	133.150	$\pm 2.98 \cdot 10^{-1}$	262,991	174.803	$\pm 4.40 \cdot 10^{-1}$	611,967	33.998	$\pm 1.20 \cdot 10^{0}$	4,251
16	85.598	$\pm 2.60 \cdot 10^{-1}$	165,651	83.327	$\pm 3.52 \cdot 10^{-1}$	388,262	28.118	$\pm 4.35 \cdot 10^{-1}$	4,251
20	63.940	$\pm 2.29 \cdot 10^{-1}$	119,551	49.497	$\pm 3.11 \cdot 10^{-1}$	305,387	25.452	$\pm 2.76 \cdot 10^{-3}$	4,003
24	51.301	$\pm 2.05 \cdot 10^{-1}$	92,094	33.241	$\pm 2.85 \cdot 10^{-1}$	261,408	25.452	$\pm 2.76 \cdot 10^{-3}$	4,003
32	39.857	$\pm 1.60 \cdot 10^{-1}$	68,021	20.113	$\pm 2.45 \cdot 10^{-1}$	225,537	25.452	$\pm 2.76 \cdot 10^{-3}$	4,003
48	32.394	$\pm 1.08 \cdot 10^{-1}$	60,164	13.703	$\pm 2.04 \cdot 10^{-1}$	212,695	25.452	$\pm 2.76 \cdot 10^{-3}$	4,003
64	29.711	$\pm 8.10 \cdot 10^{-2}$	54,313	11.015	$\pm 1.75 \cdot 10^{-1}$	204,129	25.452	$\pm 2.76 \cdot 10^{-3}$	4,003
80	28.517	$\pm 6.88 \cdot 10^{-2}$	53,278	9.042	$\pm 1.35 \cdot 10^{-1}$	203,322	25.452	$\pm 2.76 \cdot 10^{-3}$	4,003
96	27.638	$\pm 5.83 \cdot 10^{-2}$	50,310	7.842	$\pm 1.22 \cdot 10^{-1}$	201,153	25.452	$\pm 2.76 \cdot 10^{-3}$	4,003
112	26.953	$\pm 4.69 \cdot 10^{-2}$	49,597	8.021	$\pm 1.40 \cdot 10^{-1}$	201,565	25.452	$\pm 2.55 \cdot 10^{-3}$	4,003
128	26.558	$\pm 4.46 \cdot 10^{-2}$	50,212	8.243	$\pm 1.59 \cdot 10^{-1}$	203,051	25.452	$\pm 2.55 \cdot 10^{-3}$	4,003
160	26.200	$\pm 3.82 \cdot 10^{-2}$	47,771	7.978	$\pm 1.59 \cdot 10^{-1}$	199,577	25.452	$\pm 2.55 \cdot 10^{-3}$	4,003
192	25.929	$\pm 3.08 \cdot 10^{-2}$	49,460	6.767	$\pm 1.14 \cdot 10^{-1}$	200,747	25.452	$\pm 2.55 \cdot 10^{-3}$	4,003
254	25.507	$\pm 2.01 \cdot 10^{-2}$	48,125	6.344	$\pm 1.11 \cdot 10^{-1}$	199,190	25.452	$\pm 2.55 \cdot 10^{-3}$	4,003

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
12	140.384	$\pm 2.73 \cdot 10^{-1}$	447,138	158.481	$\pm 3.58 \cdot 10^{-1}$	184,150	134.732	$\pm 2.98 \cdot 10^{-1}$	258,740
16	85.036	$\pm 3.06 \cdot 10^{-1}$	235,759	103.246	$\pm 4.48 \cdot 10^{-1}$	70,109	87.064	$\pm 2.63 \cdot 10^{-1}$	161,400
20	60.990	$\pm 3.81 \cdot 10^{-1}$	159,199	75.659	$\pm 4.98 \cdot 10^{-1}$	39,648	65.226	$\pm 2.33 \cdot 10^{-1}$	115,548
24	47.815	$\pm 4.43 \cdot 10^{-1}$	119,483	60.444	$\pm 5.21 \cdot 10^{-1}$	27,391	52.427	$\pm 2.11 \cdot 10^{-1}$	88,091
32	35.105	$\pm 5.00 \cdot 10^{-1}$	87,047	44.079	$\pm 4.55 \cdot 10^{-1}$	19,026	40.709	$\pm 1.67 \cdot 10^{-1}$	64,018
48	26.025	$\pm 4.80 \cdot 10^{-1}$	75,556	34.360	$\pm 3.30 \cdot 10^{-1}$	15,392	32.839	$\pm 1.14 \cdot 10^{-1}$	56,161
64	21.527	$\pm 4.40 \cdot 10^{-1}$	68,445	30.813	$\pm 2.42 \cdot 10^{-1}$	14,132	29.999	$\pm 8.68 \cdot 10^{-2}$	50,310
80	17.062	$\pm 3.39 \cdot 10^{-1}$	66,727	28.878	$\pm 1.82 \cdot 10^{-1}$	13,451	28.716	$\pm 7.40 \cdot 10^{-2}$	49,275
96	14.920	$\pm 3.19 \cdot 10^{-1}$	63,118	28.025	$\pm 1.46 \cdot 10^{-1}$	12,808	27.777	$\pm 6.30 \cdot 10^{-2}$	46,307
112	15.974	$\pm 3.88 \cdot 10^{-1}$	62,349	27.644	$\pm 1.51 \cdot 10^{-1}$	12,753	27.034	$\pm 5.08 \cdot 10^{-2}$	45,594
128	16.878	$\pm 4.53 \cdot 10^{-1}$	63,058	27.075	$\pm 1.29 \cdot 10^{-1}$	12,846	26.604	$\pm 4.84 \cdot 10^{-2}$	46,209
160	16.807	$\pm 4.67 \cdot 10^{-1}$	60,326	26.783	$\pm 1.43 \cdot 10^{-1}$	12,555	26.218	$\pm 4.16 \cdot 10^{-2}$	43,768
192	12.634	$\pm 3.10 \cdot 10^{-1}$	61,817	26.500	$\pm 1.06 \cdot 10^{-1}$	12,358	25.920	$\pm 3.35 \cdot 10^{-2}$	45,457
254	11.911	$\pm 3.11 \cdot 10^{-1}$	60,134	25.917	$\pm 7.05 \cdot 10^{-2}$	12,009	25.461	$\pm 2.19 \cdot 10^{-2}$	44,122

Table B.19: Scalability experiment results with 254 PEs and 1000 µs Workload. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.5.3).

Pipes	activate_task			__schedule			__fw_admit		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
12	134.654	$\pm 2.98 \cdot 10^{-1}$	262,828	178.581	$\pm 4.54 \cdot 10^{-1}$	618,115	36.163	$\pm 1.24 \cdot 10^{0}$	4,251
16	86.392	$\pm 2.56 \cdot 10^{-1}$	169,428	87.694	$\pm 4.04 \cdot 10^{-1}$	399,412	30.074	$\pm 5.32 \cdot 10^{-1}$	4,251
20	65.598	$\pm 2.34 \cdot 10^{-1}$	114,754	54.872	$\pm 3.96 \cdot 10^{-1}$	302,979	26.718	$\pm 6.36 \cdot 10^{-3}$	4,003
24	53.278	$\pm 2.02 \cdot 10^{-1}$	95,053	39.369	$\pm 3.82 \cdot 10^{-1}$	269,991	26.718	$\pm 6.36 \cdot 10^{-3}$	4,003
32	42.089	$\pm 1.61 \cdot 10^{-1}$	73,756	27.709	$\pm 3.91 \cdot 10^{-1}$	236,158	26.718	$\pm 6.36 \cdot 10^{-3}$	4,003
48	34.027	$\pm 1.12 \cdot 10^{-1}$	60,453	21.859	$\pm 4.09 \cdot 10^{-1}$	217,563	26.718	$\pm 6.36 \cdot 10^{-3}$	4,003
64	30.993	$\pm 8.11 \cdot 10^{-2}$	55,896	18.921	$\pm 3.83 \cdot 10^{-1}$	211,036	26.718	$\pm 6.36 \cdot 10^{-3}$	4,003
80	29.764	$\pm 6.77 \cdot 10^{-2}$	54,550	18.044	$\pm 3.76 \cdot 10^{-1}$	208,196	26.718	$\pm 6.36 \cdot 10^{-3}$	4,003
96	28.610	$\pm 4.98 \cdot 10^{-2}$	51,743	18.483	$\pm 4.17 \cdot 10^{-1}$	205,377	26.718	$\pm 6.36 \cdot 10^{-3}$	4,003
112	28.333	$\pm 5.67 \cdot 10^{-2}$	54,985	17.352	$\pm 3.77 \cdot 10^{-1}$	211,328	26.718	$\pm 6.41 \cdot 10^{-3}$	4,035
128	27.803	$\pm 5.52 \cdot 10^{-2}$	52,010	18.797	$\pm 4.44 \cdot 10^{-1}$	208,168	26.718	$\pm 6.41 \cdot 10^{-3}$	4,035
160	27.479	$\pm 4.13 \cdot 10^{-2}$	52,478	16.999	$\pm 3.83 \cdot 10^{-1}$	207,050	26.718	$\pm 6.41 \cdot 10^{-3}$	4,035
192	27.174	$\pm 2.63 \cdot 10^{-2}$	52,052	16.638	$\pm 3.77 \cdot 10^{-1}$	207,089	26.718	$\pm 6.41 \cdot 10^{-3}$	4,035
254	26.746	$\pm 2.44 \cdot 10^{-2}$	54,843	16.056	$\pm 3.60 \cdot 10^{-1}$	210,772	26.718	$\pm 6.41 \cdot 10^{-3}$	4,035

Pipes	__fw_dispatch			__fw_relinquish			__fw_unblock		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
12	144.580	$\pm 3.06 \cdot 10^{-1}$	451,360	157.503	$\pm 3.64 \cdot 10^{-1}$	188,533	136.226	$\pm 2.98 \cdot 10^{-1}$	258,577
16	91.638	$\pm 4.31 \cdot 10^{-1}$	242,619	102.333	$\pm 4.79 \cdot 10^{-1}$	73,191	87.793	$\pm 2.58 \cdot 10^{-1}$	165,177
20	70.820	$\pm 5.74 \cdot 10^{-1}$	156,357	75.949	$\pm 5.63 \cdot 10^{-1}$	41,604	66.956	$\pm 2.38 \cdot 10^{-1}$	110,751
24	58.634	$\pm 6.68 \cdot 10^{-1}$	125,077	61.208	$\pm 6.07 \cdot 10^{-1}$	30,025	54.397	$\pm 2.07 \cdot 10^{-1}$	91,050
32	50.285	$\pm 8.43 \cdot 10^{-1}$	95,191	45.932	$\pm 6.07 \cdot 10^{-1}$	21,437	42.921	$\pm 1.68 \cdot 10^{-1}$	69,753
48	47.016	$\pm 1.03 \cdot 10^{0}$	77,742	36.530	$\pm 5.83 \cdot 10^{-1}$	17,289	34.496	$\pm 1.19 \cdot 10^{-1}$	56,450
64	43.403	$\pm 1.03 \cdot 10^{0}$	71,719	32.507	$\pm 4.69 \cdot 10^{-1}$	15,823	31.273	$\pm 8.67 \cdot 10^{-2}$	51,893
80	41.905	$\pm 1.02 \cdot 10^{0}$	70,017	31.097	$\pm 4.54 \cdot 10^{-1}$	15,468	29.955	$\pm 7.27 \cdot 10^{-2}$	50,547
96	45.668	$\pm 1.19 \cdot 10^{0}$	66,709	30.039	$\pm 4.58 \cdot 10^{-1}$	14,967	28.719	$\pm 5.38 \cdot 10^{-2}$	47,740
112	41.256	$\pm 1.05 \cdot 10^{0}$	70,081	29.612	$\pm 3.96 \cdot 10^{-1}$	15,096	28.411	$\pm 6.10 \cdot 10^{-2}$	50,950
128	47.612	$\pm 1.29 \cdot 10^{0}$	66,966	28.735	$\pm 3.74 \cdot 10^{-1}$	14,956	27.844	$\pm 5.98 \cdot 10^{-2}$	47,975
160	41.782	$\pm 1.09 \cdot 10^{0}$	67,297	28.457	$\pm 3.39 \cdot 10^{-1}$	14,819	27.492	$\pm 4.47 \cdot 10^{-2}$	48,443
192	41.110	$\pm 1.09 \cdot 10^{0}$	66,733	28.299	$\pm 3.41 \cdot 10^{-1}$	14,681	27.162	$\pm 2.85 \cdot 10^{-2}$	48,017
254	38.461	$\pm 1.02 \cdot 10^{0}$	69,627	27.390	$\pm 2.25 \cdot 10^{-1}$	14,784	26.698	$\pm 2.63 \cdot 10^{-2}$	50,808

B.2 Composition Overhead

Table B.20: Overhead experiment results. The mean runtimes and confidence intervals are given in microseconds (cf. Section 8.6.2).

Cycles	1 Stage			2 Stages			3 Stages		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
0	0.299	$\pm 5.54 \cdot 10^{-5}$	17,616	0.513	$\pm 1.21 \cdot 10^{-4}$	15,787	0.727	$\pm 1.82 \cdot 10^{-4}$	15,962
50	0.324	$\pm 5.85 \cdot 10^{-5}$	15,784	0.538	$\pm 1.20 \cdot 10^{-4}$	16,026	0.749	$\pm 1.82 \cdot 10^{-4}$	15,964
100	0.349	$\pm 5.52 \cdot 10^{-5}$	17,741	0.563	$\pm 1.19 \cdot 10^{-4}$	16,161	0.772	$\pm 1.79 \cdot 10^{-4}$	16,626
150	0.374	$\pm 5.51 \cdot 10^{-5}$	17,784	0.588	$\pm 1.21 \cdot 10^{-4}$	15,692	0.802	$\pm 1.82 \cdot 10^{-4}$	16,011
300	0.449	$\pm 5.59 \cdot 10^{-5}$	17,265	0.663	$\pm 1.15 \cdot 10^{-4}$	17,539	0.877	$\pm 1.72 \cdot 10^{-4}$	17,949
450	0.524	$\pm 5.51 \cdot 10^{-5}$	17,791	0.738	$\pm 1.14 \cdot 10^{-4}$	17,742	0.952	$\pm 1.78 \cdot 10^{-4}$	16,694
600	0.599	$\pm 5.47 \cdot 10^{-5}$	18,075	0.813	$\pm 1.16 \cdot 10^{-4}$	17,187	1.027	$\pm 1.81 \cdot 10^{-4}$	16,119
900	0.749	$\pm 5.54 \cdot 10^{-5}$	17,622	0.963	$\pm 1.15 \cdot 10^{-4}$	17,443	1.177	$\pm 1.72 \cdot 10^{-4}$	18,023
1,200	0.899	$\pm 5.65 \cdot 10^{-5}$	16,901	1.113	$\pm 1.15 \cdot 10^{-4}$	17,497	1.327	$\pm 1.77 \cdot 10^{-4}$	16,848
2,100	1.349	$\pm 5.54 \cdot 10^{-5}$	17,623	1.563	$\pm 1.14 \cdot 10^{-4}$	17,691	1.777	$\pm 1.74 \cdot 10^{-4}$	17,502
3,000	1.799	$\pm 5.49 \cdot 10^{-5}$	17,907	2.013	$\pm 1.18 \cdot 10^{-4}$	16,608	2.227	$\pm 1.72 \cdot 10^{-4}$	17,933
4,500	2.549	$\pm 5.58 \cdot 10^{-5}$	17,353	2.763	$\pm 1.15 \cdot 10^{-4}$	17,484	2.976	$\pm 8.66 \cdot 10^{-4}$	18,085
6,000	3.299	$\pm 5.65 \cdot 10^{-5}$	16,907	3.513	$\pm 1.13 \cdot 10^{-4}$	17,960	3.727	$\pm 1.74 \cdot 10^{-4}$	17,588
7,500	4.049	$\pm 5.35 \cdot 10^{-5}$	18,873	4.263	$\pm 1.16 \cdot 10^{-4}$	17,186	4.477	$\pm 1.77 \cdot 10^{-4}$	16,886
9,000	4.799	$\pm 5.44 \cdot 10^{-5}$	18,224	5.013	$\pm 1.13 \cdot 10^{-4}$	18,147	5.227	$\pm 1.70 \cdot 10^{-4}$	18,414
10,500	5.549	$\pm 5.26 \cdot 10^{-5}$	19,526	5.763	$\pm 1.10 \cdot 10^{-4}$	18,943	5.977	$\pm 1.69 \cdot 10^{-4}$	18,658
12,000	6.299	$\pm 5.31 \cdot 10^{-5}$	19,151	6.513	$\pm 1.11 \cdot 10^{-4}$	18,875	6.727	$\pm 1.72 \cdot 10^{-4}$	17,941
13,500	7.049	$\pm 5.48 \cdot 10^{-5}$	18,021	7.263	$\pm 1.13 \cdot 10^{-4}$	17,990	7.477	$\pm 1.79 \cdot 10^{-4}$	16,626
15,000	7.799	$\pm 5.55 \cdot 10^{-5}$	17,536	8.013	$\pm 1.17 \cdot 10^{-4}$	16,998	8.227	$\pm 1.77 \cdot 10^{-4}$	16,851
15,000	7.799	$\pm 5.55 \cdot 10^{-5}$	17,536	8.013	$\pm 1.17 \cdot 10^{-4}$	16,998	8.227	$\pm 1.77 \cdot 10^{-4}$	16,851
16,500	8.549	$\pm 5.54 \cdot 10^{-5}$	17,589	8.763	$\pm 1.13 \cdot 10^{-4}$	18,196	8.977	$\pm 1.74 \cdot 10^{-4}$	17,589
25,000	12.799	$\pm 5.40 \cdot 10^{-5}$	18,532	13.013	$\pm 1.10 \cdot 10^{-4}$	19,045	13.222	$\pm 1.71 \cdot 10^{-4}$	18,223

Cycles	4 Stage			5 Stages			6 Stages		
	Runtime	95% CI	Samples	Runtime	95% CI	Samples	Runtime	95% CI	Samples
0	0.941	$\pm 2.32 \cdot 10^{-4}$	17,643	1.155	$\pm 2.93 \cdot 10^{-4}$	17,442	1.369	$\pm 3.48 \cdot 10^{-4}$	17,862
50	0.961	$\pm 2.48 \cdot 10^{-4}$	15,518	1.180	$\pm 2.97 \cdot 10^{-4}$	16,942	1.384	$\pm 3.67 \cdot 10^{-4}$	16,048
100	0.991	$\pm 2.42 \cdot 10^{-4}$	16,242	1.204	$\pm 4.99 \cdot 10^{-4}$	16,290	1.414	$\pm 3.52 \cdot 10^{-4}$	17,466
150	1.011	$\pm 2.40 \cdot 10^{-4}$	16,514	1.230	$\pm 2.92 \cdot 10^{-4}$	17,605	1.444	$\pm 3.52 \cdot 10^{-4}$	17,511
300	1.091	$\pm 2.32 \cdot 10^{-4}$	17,721	1.305	$\pm 2.90 \cdot 10^{-4}$	17,868	1.519	$\pm 3.63 \cdot 10^{-4}$	16,488
450	1.161	$\pm 2.32 \cdot 10^{-4}$	17,730	1.380	$\pm 3.06 \cdot 10^{-4}$	16,038	1.594	$\pm 3.48 \cdot 10^{-4}$	17,885
600	1.241	$\pm 2.40 \cdot 10^{-4}$	16,561	1.455	$\pm 2.92 \cdot 10^{-4}$	17,580	1.669	$\pm 3.48 \cdot 10^{-4}$	17,848
900	1.391	$\pm 2.41 \cdot 10^{-4}$	16,470	1.605	$\pm 2.99 \cdot 10^{-4}$	16,805	1.819	$\pm 3.52 \cdot 10^{-4}$	17,502
1,200	1.541	$\pm 2.44 \cdot 10^{-4}$	16,002	1.755	$\pm 2.98 \cdot 10^{-4}$	16,853	1.969	$\pm 3.47 \cdot 10^{-4}$	18,041
2,100	1.991	$\pm 2.27 \cdot 10^{-4}$	18,452	2.205	$\pm 2.93 \cdot 10^{-4}$	17,503	2.419	$\pm 3.50 \cdot 10^{-4}$	17,663
3,000	2.441	$\pm 2.34 \cdot 10^{-4}$	17,398	2.655	$\pm 2.92 \cdot 10^{-4}$	17,613	2.869	$\pm 3.52 \cdot 10^{-4}$	17,537
4,500	3.191	$\pm 2.34 \cdot 10^{-4}$	17,333	3.405	$\pm 2.95 \cdot 10^{-4}$	17,169	3.619	$\pm 3.44 \cdot 10^{-4}$	18,288
6,000	3.941	$\pm 2.43 \cdot 10^{-4}$	16,120	4.155	$\pm 2.86 \cdot 10^{-4}$	18,366	4.369	$\pm 3.56 \cdot 10^{-4}$	17,062
7,500	4.691	$\pm 2.39 \cdot 10^{-4}$	16,742	4.905	$\pm 2.89 \cdot 10^{-4}$	17,968	5.119	$\pm 3.45 \cdot 10^{-4}$	18,234
9,000	5.441	$\pm 2.32 \cdot 10^{-4}$	17,712	5.655	$\pm 2.84 \cdot 10^{-4}$	18,527	5.869	$\pm 3.33 \cdot 10^{-4}$	19,497
10,500	6.191	$\pm 2.24 \cdot 10^{-4}$	19,070	6.405	$\pm 2.80 \cdot 10^{-4}$	19,058	6.619	$\pm 3.47 \cdot 10^{-4}$	17,990
12,000	6.941	$\pm 2.30 \cdot 10^{-4}$	17,949	7.155	$\pm 2.95 \cdot 10^{-4}$	17,190	7.369	$\pm 3.50 \cdot 10^{-4}$	17,734
13,500	7.691	$\pm 2.33 \cdot 10^{-4}$	17,549	7.905	$\pm 2.92 \cdot 10^{-4}$	17,542	8.119	$\pm 3.54 \cdot 10^{-4}$	17,332
15,000	8.441	$\pm 2.33 \cdot 10^{-4}$	17,627	8.655	$\pm 2.99 \cdot 10^{-4}$	16,730	8.869	$\pm 3.55 \cdot 10^{-4}$	17,174
15,000	8.441	$\pm 2.33 \cdot 10^{-4}$	17,627	8.655	$\pm 2.99 \cdot 10^{-4}$	16,730	8.869	$\pm 3.55 \cdot 10^{-4}$	17,174
16,500	9.191	$\pm 2.33 \cdot 10^{-4}$	17,490	9.405	$\pm 2.93 \cdot 10^{-4}$	17,429	9.619	$\pm 3.42 \cdot 10^{-4}$	18,526
25,000	13.441	$\pm 2.29 \cdot 10^{-4}$	18,242	13.655	$\pm 2.85 \cdot 10^{-4}$	18,415	13.864	$\pm 3.34 \cdot 10^{-4}$	19,483

B.3 Scheduler Performance

Table B.21: Performance evaluation results for the NAS benchmark suite. The mean runtimes and confidence intervals are given in seconds. Each benchmark was repeated 50 times (cf. Section 8.7.2).

NAS Bechmark	CoBaS		CFS		Ratio
	Runtime	95 % CI	Runtime	95 % CI	
BT A	14.84	$\pm 2.09 \cdot 10^{-2}$	14.51	$\pm 1.71 \cdot 10^{-2}$	0.97753
BT B	65.05	$\pm 6.68 \cdot 10^{-2}$	64.53	$\pm 1.66 \cdot 10^{-1}$	0.99208
BT C	265.02	$\pm 9.59 \cdot 10^{-2}$	264.12	$\pm 1.96 \cdot 10^{-1}$	0.99664
CG A	0.51	$\pm 1.47 \cdot 10^{-2}$	0.47	$\pm 5.63 \cdot 10^{-4}$	0.93149
CG B	21.15	$\pm 3.37 \cdot 10^{-2}$	21.00	$\pm 2.33 \cdot 10^{-2}$	0.99304
CG C	60.52	$\pm 1.79 \cdot 10^{-2}$	60.09	$\pm 1.47 \cdot 10^{-2}$	0.99295
EP A	2.14	$\pm 1.74 \cdot 10^{-3}$	2.13	$\pm 2.21 \cdot 10^{-3}$	0.99450
EP B	8.55	$\pm 5.58 \cdot 10^{-3}$	8.52	$\pm 8.68 \cdot 10^{-3}$	0.99661
EP C	34.19	$\pm 2.44 \cdot 10^{-2}$	34.07	$\pm 2.39 \cdot 10^{-2}$	0.99645
FT A	0.92	$\pm 2.10 \cdot 10^{-3}$	0.90	$\pm 1.34 \cdot 10^{-3}$	0.98517
FT B	11.72	$\pm 2.29 \cdot 10^{-2}$	11.62	$\pm 1.21 \cdot 10^{-2}$	0.99164
IS A	0.16	$\pm 6.82 \cdot 10^{-4}$	0.16	$\pm 0.00 \cdot 10^{0}$	0.99626
IS B	0.72	$\pm 1.60 \cdot 10^{-3}$	0.71	$\pm 1.36 \cdot 10^{-3}$	0.99442
IS C	3.05	$\pm 1.84 \cdot 10^{-3}$	3.04	$\pm 2.02 \cdot 10^{-3}$	0.99816
LU A	9.07	$\pm 1.60 \cdot 10^{-2}$	9.01	$\pm 2.64 \cdot 10^{-2}$	0.99323
LU B	58.95	$\pm 3.29 \cdot 10^{-1}$	59.28	$\pm 3.23 \cdot 10^{-1}$	1.00556
LU C	384.93	$\pm 5.99 \cdot 10^{-2}$	383.93	$\pm 1.12 \cdot 10^{-1}$	0.99740
MG A	0.77	$\pm 7.26 \cdot 10^{-3}$	0.76	$\pm 8.04 \cdot 10^{-4}$	0.98395
MG B	3.60	$\pm 1.32 \cdot 10^{-2}$	3.57	$\pm 3.68 \cdot 10^{-3}$	0.99050
SP A	15.16	$\pm 2.26 \cdot 10^{-2}$	15.12	$\pm 1.98 \cdot 10^{-2}$	0.99782
SP B	100.61	$\pm 1.37 \cdot 10^{-1}$	100.55	$\pm 1.14 \cdot 10^{-1}$	0.99946
SP C	476.18	$\pm 5.45 \cdot 10^{-2}$	475.31	$\pm 1.07 \cdot 10^{-1}$	0.99818
UA A	12.73	$\pm 3.25 \cdot 10^{-2}$	12.31	$\pm 1.77 \cdot 10^{-2}$	0.96706
UA B	59.06	$\pm 8.45 \cdot 10^{-2}$	58.35	$\pm 8.41 \cdot 10^{-2}$	0.98789

Table B.22: Performance evaluation results for the hackbench benchmark. The mean runtimes and confidence intervals are given in seconds. Each benchmark was repeated 50 times (cf. Section 8.7.2).

Number of Groups	CoBaS		CFS		Ratio
	Runtime	95 % CI	Runtime	95 % CI	
1	0.82	$\pm 3.78 \cdot 10^{-3}$	0.69	$\pm 3.86 \cdot 10^{-2}$	0.84677
2	1.23	$\pm 3.95 \cdot 10^{-3}$	1.24	$\pm 5.14 \cdot 10^{-2}$	1.00973
4	1.57	$\pm 1.38 \cdot 10^{-2}$	2.57	$\pm 3.99 \cdot 10^{-2}$	1.64459
8	2.73	$\pm 8.45 \cdot 10^{-3}$	4.85	$\pm 7.60 \cdot 10^{-2}$	1.77811
16	5.34	$\pm 1.23 \cdot 10^{-2}$	8.17	$\pm 1.76 \cdot 10^{-1}$	1.52895
32	10.46	$\pm 1.79 \cdot 10^{-2}$	13.74	$\pm 2.32 \cdot 10^{-1}$	1.31405
64	20.56	$\pm 2.98 \cdot 10^{-2}$	25.48	$\pm 3.72 \cdot 10^{-1}$	1.23945
128	40.72	$\pm 3.84 \cdot 10^{-2}$	45.74	$\pm 3.81 \cdot 10^{-1}$	1.12309
256	80.83	$\pm 4.53 \cdot 10^{-2}$	83.73	$\pm 2.45 \cdot 10^{-1}$	1.03586

B.4 Scheduler Adaptation

Table B.23: Adaptation experiment results. The mean runtimes and confidence intervals are given in seconds. Each experiment was repeated 100 times (cf. Section 8.8.2).

(a) Results for the round-robin scheduling policy.

Threads	Short		Middle		Long	
	Runtime	95% CI	Runtime	95% CI	Runtime	95% CI
1	$4.34 \cdot 10^{-3}$	$\pm 1.70 \cdot 10^{-4}$	$4.32 \cdot 10^{-3}$	$\pm 1.59 \cdot 10^{-4}$	$6.40 \cdot 10^{-2}$	$\pm 0.00 \cdot 10^{0}$
2	$5.24 \cdot 10^{-3}$	$\pm 1.47 \cdot 10^{-4}$	$5.08 \cdot 10^{-3}$	$\pm 1.31 \cdot 10^{-4}$	$1.81 \cdot 10^{-1}$	$\pm 1.21 \cdot 10^{-4}$
4	$6.41 \cdot 10^{-3}$	$\pm 2.17 \cdot 10^{-4}$	$1.15 \cdot 10^{-2}$	$\pm 1.77 \cdot 10^{-4}$	$5.94 \cdot 10^{-1}$	$\pm 4.97 \cdot 10^{-4}$
8	$2.57 \cdot 10^{-2}$	$\pm 6.73 \cdot 10^{-4}$	$3.68 \cdot 10^{-2}$	$\pm 7.42 \cdot 10^{-4}$	$2.13 \cdot 10^{0}$	$\pm 7.96 \cdot 10^{-4}$
16	$1.11 \cdot 10^{-1}$	$\pm 1.23 \cdot 10^{-3}$	$1.63 \cdot 10^{-1}$	$\pm 2.14 \cdot 10^{-3}$	$8.08 \cdot 10^{0}$	$\pm 2.04 \cdot 10^{-3}$
32	$4.72 \cdot 10^{-1}$	$\pm 2.06 \cdot 10^{-3}$	$6.99 \cdot 10^{-1}$	$\pm 4.51 \cdot 10^{-3}$	$3.14 \cdot 10^{1}$	$\pm 5.54 \cdot 10^{-3}$
64	$2.45 \cdot 10^{0}$	$\pm 8.67 \cdot 10^{-2}$	$2.94 \cdot 10^{0}$	$\pm 1.64 \cdot 10^{-2}$	$1.24 \cdot 10^{2}$	$\pm 1.63 \cdot 10^{-2}$
128	$1.16 \cdot 10^{1}$	$\pm 9.13 \cdot 10^{-2}$	$1.16 \cdot 10^{1}$	$\pm 7.09 \cdot 10^{-2}$	$4.92 \cdot 10^{2}$	$\pm 4.60 \cdot 10^{-2}$

(b) Results for the CFS policy.

Threads	Short		Middle		Long	
	Runtime	95% CI	Runtime	95% CI	Runtime	95% CI
1	$3.00 \cdot 10^{-5}$	$\pm 4.42 \cdot 10^{-5}$	$1.01 \cdot 10^{-3}$	$\pm 1.98 \cdot 10^{-5}$	$6.01 \cdot 10^{-2}$	$\pm 9.22 \cdot 10^{-5}$
2	$1.31 \cdot 10^{-2}$	$\pm 1.74 \cdot 10^{-3}$	$1.43 \cdot 10^{-2}$	$\pm 1.55 \cdot 10^{-3}$	$1.78 \cdot 10^{-1}$	$\pm 6.25 \cdot 10^{-4}$
4	$3.67 \cdot 10^{-2}$	$\pm 4.91 \cdot 10^{-3}$	$3.57 \cdot 10^{-2}$	$\pm 4.71 \cdot 10^{-3}$	$6.04 \cdot 10^{-1}$	$\pm 2.44 \cdot 10^{-3}$
8	$1.01 \cdot 10^{-1}$	$\pm 1.16 \cdot 10^{-2}$	$1.16 \cdot 10^{-1}$	$\pm 1.13 \cdot 10^{-2}$	$2.18 \cdot 10^{0}$	$\pm 5.60 \cdot 10^{-3}$
16	$3.55 \cdot 10^{-1}$	$\pm 3.06 \cdot 10^{-2}$	$3.30 \cdot 10^{-1}$	$\pm 3.50 \cdot 10^{-2}$	$8.01 \cdot 10^{0}$	$\pm 1.28 \cdot 10^{-2}$
32	$1.24 \cdot 10^{0}$	$\pm 1.04 \cdot 10^{-1}$	$1.09 \cdot 10^{0}$	$\pm 9.45 \cdot 10^{-2}$	$3.07 \cdot 10^{1}$	$\pm 2.77 \cdot 10^{-2}$
64	$4.62 \cdot 10^{0}$	$\pm 3.49 \cdot 10^{-1}$	$4.40 \cdot 10^{0}$	$\pm 3.39 \cdot 10^{-1}$	$1.20 \cdot 10^{2}$	$\pm 7.80 \cdot 10^{-2}$
128	$1.79 \cdot 10^{1}$	$\pm 1.32 \cdot 10^{0}$	$1.79 \cdot 10^{1}$	$\pm 1.31 \cdot 10^{0}$	$4.73 \cdot 10^{2}$	$\pm 1.96 \cdot 10^{-1}$

(c) Results for the optimized scheduling policy.

Threads	Short		Middle		Long	
	Runtime	95% CI	Runtime	95% CI	Runtime	95% CI
1	$8.23 \cdot 10^{-3}$	$\pm 4.90 \cdot 10^{-4}$	$7.98 \cdot 10^{-3}$	$\pm 4.60 \cdot 10^{-4}$	$6.71 \cdot 10^{-2}$	$\pm 5.56 \cdot 10^{-4}$
2	$7.50 \cdot 10^{-3}$	$\pm 3.48 \cdot 10^{-4}$	$7.70 \cdot 10^{-3}$	$\pm 2.88 \cdot 10^{-4}$	$1.27 \cdot 10^{-1}$	$\pm 3.45 \cdot 10^{-4}$
4	$7.92 \cdot 10^{-3}$	$\pm 2.83 \cdot 10^{-4}$	$1.02 \cdot 10^{-2}$	$\pm 2.34 \cdot 10^{-4}$	$2.45 \cdot 10^{-1}$	$\pm 4.09 \cdot 10^{-4}$
8	$7.52 \cdot 10^{-3}$	$\pm 3.06 \cdot 10^{-4}$	$1.35 \cdot 10^{-2}$	$\pm 3.23 \cdot 10^{-4}$	$4.84 \cdot 10^{-1}$	$\pm 5.59 \cdot 10^{-4}$
16	$8.49 \cdot 10^{-3}$	$\pm 4.77 \cdot 10^{-4}$	$2.22 \cdot 10^{-2}$	$\pm 4.16 \cdot 10^{-4}$	$9.60 \cdot 10^{-1}$	$\pm 8.49 \cdot 10^{-4}$
32	$7.65 \cdot 10^{-3}$	$\pm 4.56 \cdot 10^{-4}$	$3.63 \cdot 10^{-2}$	$\pm 4.58 \cdot 10^{-4}$	$1.91 \cdot 10^{0}$	$\pm 1.28 \cdot 10^{-3}$
64	$1.49 \cdot 10^{-2}$	$\pm 1.16 \cdot 10^{-3}$	$7.09 \cdot 10^{-2}$	$\pm 9.82 \cdot 10^{-4}$	$3.83 \cdot 10^{0}$	$\pm 2.25 \cdot 10^{-3}$
128	$3.16 \cdot 10^{-2}$	$\pm 2.82 \cdot 10^{-3}$	$1.45 \cdot 10^{-1}$	$\pm 2.70 \cdot 10^{-3}$	$7.69 \cdot 10^{0}$	$\pm 3.98 \cdot 10^{-3}$

List of Publications

A. Busse, R. Karnapke, and H.-U. Heiss. "CoBaS: Introducing a Component Based Scheduling Framework". In: *2015 International Symposium on Computer Architecture and High Performance Computing Workshop*. SBAC-PADW. Oct. 2015, pp. 79–84. DOI: 10.1109/SBAC-PADW.2015.23.

A. Busse, J. H. Schönherr, M. Diener, P. O. A. Navaux, and H.-U. Heiß. "Partial Coscheduling of Virtual Machines Based on Memory Access Patterns". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15. Salamanca, Spain: ACM, Apr. 2015, pp. 2033–2038. DOI: 10.1145/2695664.2695736.

A. Busse, J. H. Schönherr, M. Diener, G. Mühl, and J. Richling. "Analyzing Resource Interdependencies in Multi-core Architectures to Improve Scheduling Decisions". In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC '13. Coimbra, Portugal: ACM, Mar. 2013, pp. 1595–1602. DOI: 10.1145/2480362.2480661.

A. Busse. "SSD-basiertes Caching von Blockgeräten". In: *Informatiktage 2011*. Ed. by Gesellschaft für Informatik. Vol. S-10. Mar. 2011, pp. 139–142. ISBN: 978-3-88579-444-8.

S. Sydow, M. Nabelsee, **A. Busse**, S. Koch, and H. Parzyjegla. "Performance-Aware Device Driver Architecture for Signal Processing". In: *2016 International Symposium on Computer Architecture and High Performance Computing*. SBAC-PAD. Accepted for Publication. Oct. 2016.

M. Nabelsee, **A. Busse**, H. Parzyjegla, and G. Mühl. "Load-aware Scheduling for Heterogeneous Multi-core Systems". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC '16. Pisa, Italy: ACM, Apr. 2016, pp. 1844–1851. DOI: 10.1145/2851613.2851738.

F. Rabuske, C. A. da Costa, R. da Rosa Righi, G. Rostirolla, A. Alberti, **A. Busse**, and H.-U. Heiss. "GetLB++: Improving Transaction Load Balancing on the Electronic Funds Transfer Landscape". In: *Proceedings of the 21st International Conference on Parallel and Distributed Processing Techniques and Applications*. PDPTA 2015. Las Vegas, Nevada, USA: CSREA Press, July 2015, pp. 13–19. ISBN: 1-60132-424-3.

M. Diener, E. H. Cruz, P. O. Navaux, **A. Busse**, and H.-U. Heiß. "Communication-aware Process and Thread Mapping Using Online Communication Detection". In: *Journal Parallel Computing* 43.C (Mar. 2015), pp. 43–63. DOI: 10.1016/j.parco.2015.01.005.

N. Jeremic, G. Mühl, **A. Busse**, and J. Richling. "Dataset Management-Aware Software Architecture for Storage Systems Based on SSDs". In: *7th IEEE International Conference on Networking, Architecture, and Storage*. NAS 2012. June 2012, pp. 288–292. DOI: 10.1109/NAS.2012.42.

N. Jeremic, G. Mühl, **A. Busse**, and J. Richling. "Operating System Support for Dynamic Over-provisioning of Solid State Drives". In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC '12. Trento, Italy: ACM, 2012, pp. 1753–1758. DOI: 10.1145/2245276.2232061.

H. Parzyjegla, D. Graff, A. Schröter, A. Schepeljanski, **A. Busse**, J. Richling, M. Werner, and G. Mühl. "Rebeca - eine autonome Publish/Subscribe Middleware". In: *Praxis der Informationsverarbeitung und Kommunikation (PIK)* 34.3 (Aug. 2011), pp. 135–137. DOI: 10.1515/piko.2011.025.

H. Parzyjegla, A. Schröter, D. Graff, **A. Busse**, A. Schepeljanski, J. Richling, M. Werner, and G. Mühl. "Autonomy Features and Feature Composition in REBECA". In: *8th IEEE/ACM International Conference on Autonomic Computing*. ICAC'11. Karlsruhe, Germany: ACM, June 2011.

N. Jeremic, G. Mühl, **A. Busse**, and J. Richling. "The Pitfalls of Deploying Solid-state Drive RAIDs". In: *Proceedings of the 4th Annual International Conference on Systems and Storage*. SYSTOR '11. Haifa, Israel: ACM, 2011, 14:1–14:13. DOI: 10.1145/1987816.1987835.

Bibliography

Literature

[1] L. Abeni and G. Buttazzo. "Integrating Multimedia Applications in Hard Real-Time Systems". In: *Proceedings of the IEEE Real-Time Systems Symposium*. RTSS '98. Washington, DC, USA: IEEE Computer Society, Dec. 1998, pp. 4–13. DOI: 10.1109/REAL.1998.739726.

[2] L. Abeni and J. Regehr. "How to Rapidly Prototype a Real-Time Scheduler". Presented as a work in progress at the 23rd IEEE Real-Time Systems Symposium. Dec. 2002. URL: http://www.cs.utah.edu/flux/papers/hls-rtss02/hls-wip-rtss02.pdf (Retrieved 06/25/2016).

[3] J. Agron, D. Andrews, M. Finley, and W. Peck. "FPGA Implementation of a Priority Scheduler Module". In: *Proceedings of the 25th IEEE International Real-Time Systems Symposium, Works in Progress Session*. 2004.

[4] A. H. Alhusaini, V. K. Prasanna, and C. S. Raghavendra. "A Unified Resource Scheduling Framework for Heterogeneous Computing Environments". In: *Proceedings of the 8th Heterogeneous Computing Workshop*. HCW '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 156–165. DOI: 10.1109/HCW.1999.765123.

[5] G. M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560.

[6] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism". In: *ACM Transactions on Computer Systems* 10.1 (Feb. 1992), pp. 53–79. DOI: 10.1145/146941.146944.

[7] C. Angelini. *AMD Bulldozer Review: FX-8150 Gets Tested*. Tom's Hardware. Oct. 2011. URL: http://www.tomshardware.com/reviews/fx-8150-zambezi-bulldozer-990fx,3043.html (Retrieved 01/25/2015).

[8] ARM Limited. "ARM Unveils its Most Energy Efficient Application Processor Ever". Press Release. Oct. 2011. URL: http://www.arm.com/about/newsroom/arm-unveils-its-most-energy-efficient-application-processor-ever-with-biglittle-processing.php (Retrieved 07/04/2016).

[9] ARM Limited. *big.LITTLE Technology: The Future of Mobile. Making very high performance available in a mobile envelope without sacrificing energy efficiency*. White Paper. 2013. URL: https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf (Retrieved 07/16/2016).

[10] J. Arnold and M. F. Kaashoek. "Ksplice: Automatic Rebootless Kernel Updates". In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. Nuremberg, Germany: ACM, 2009, pp. 187–198. DOI: 10.1145/1519065.1519085.

[11] M. Åsberg, T. Nolte, S. Kato, and R. Rajkumar. "ExSched: An External CPU Scheduler Framework for Real-Time Systems". In: *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Aug. 2012, pp. 240–249. DOI: 10.1109/RTCSA.2012.9.

[12] T. Aswathanarayana, D. Niehaus, V. Subramonian, and C. Gill. "Design and Performance of Configurable Endsystem Scheduling Mechanisms". In: *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*. RTAS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 32–43. DOI: 10.1109/RTAS.2005.17.

[13] M. J. Bach. *The Design of the UNIX Operating System*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1986. ISBN: 0-13-201757-1.

[14] D. H. Bailey et al. "The NAS Parallel Benchmarks – Summary and Preliminary Results". In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing '91. Albuquerque, NM, USA: ACM, 1991, pp. 158–165. DOI: 10.1145/125826.125925.

[15] D. Bailey et al. *The NAS Parallel Benchmarks*. Tech. rep. RNR-94-007. Moffett Field, CA, USA: NASA Advanced Supercomputing Devision, Mar. 1994.

[16] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff. "OpenPiton: An Open Source Manycore Research Framework". In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, GA, USA: ACM, 2016, pp. 217–232. DOI: 10.1145/2872362.2872414.

[17] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran. "Popcorn: bridging the programmability gap in heterogeneous-ISA platforms". In: *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. Apr. 2015, 29:1–29:16. DOI: 10.1145/2741948.2741962.

[18] T. O. Barnett and L. L. Constantine, eds. *Modular Programming: Proceedings of a National Symposium*. Information & Systems Institute. Cambridge, MA, USA, July 1968.

[19] L. Barreto and G. Muller. "Bossa: a Language-based Approach for the Design of Real Time Schedulers". In: *Proceedings of the 10th International Conference on Real-Time Systems*. Paris, France, 2002, pp. 19–31.

[20] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. "The Multikernel: A New OS Architecture for Scalable Multicore Systems". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, MT, USA: ACM, 2009, pp. 29–44. DOI: 10.1145/1629575.1629579.

[21] M. Bautin, A. Dwarakinath, and T. Chiueh. "Graphics Engine Resource Management". In: *Proceedings of the 15th Annual Multimedia Computing and Networking Conference*. Jan. 2008.

[22] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. "Extensibility Safety and Performance in the SPIN Operating System". In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 267–283. DOI: 10.1145/224056.224077.

[23] B. Betkaoui, D. B. Thomas, and W. Luk. "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing". In: *Proceedings of the 2010 International Conference on Field-Programmable Technology*. Beijing, China, Dec. 2010, pp. 94–101. DOI: 10.1109/FPT.2010.5681761.

[24] N. Binkert et al. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. DOI: 10.1145/2024716.2024718.

[25] C. Binnie. *Practical Linux Topics*. Berkeley, CA: Apress, 2016. DOI: 10.1007/978-1-4842-1772-6.

[26] *Bitcoin Hash ASIC*. BM1384. Datasheet. Bitmain Technologies Limited. Dec. 2014. URL: https://www.bitmaintech.com/files/download/BM1384_Datasheet_v2.0.pdf (Retrieved 06/11/2015).

[27] *Bitcoin Hashing Chip*. BE-200. Datasheet. ASICMiner. 2014. URL: http://ozr.name/Cryptocurrency/Mining/ASIC/ASICMiner/BE200/documents/AM%20BE200%20Datasheet.pdf (Retrieved 06/11/2015).

[28] D. P. Bovet and M. Cesati. *Understanding the Linux kernel*. 3rd ed. Beijing, Cambridge, Farnham, Köln, Sebastopol, Tokyo: O'Reilly, Nov. 2005. ISBN: 978-0-596-00565-8.

[29] G. Brebner. "A virtual hardware operating system for the Xilinx XC6200". In: *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*. Ed. by R. W. Hartenstein and M. Glesner. Vol. 1142. Lecture Notes in Computer Science. Berlin and Heidelberg: Springer, 1996, pp. 327–336. DOI: 10.1007/3-540-61730-2_35.

[30] D. Bryant. *Disrupting the Data Center to Create the Digital Services Economy*. The Data Stack. Intel Corporation. June 2014. URL: https://communities.intel.com/community/itpeernetwork/datastack/blog/2014/06/18/disrupting-the-data-center-to-create-the-digital-services-economy (Retrieved 02/01/2015).

[31] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J. A. Stankovic, G. Wallace, and C. Weems. "The Spring Scheduling Coprocessor: A Scheduling Accelerator". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7.1 (Mar. 1999), pp. 38–47. DOI: 10.1109/92.748199.

[32] A. Busse, R. Karnapke, and H.-U. Heiss. "CoBaS: Introducing a Component Based Scheduling Framework". In: *2015 International Symposium on Computer Architecture and High Performance Computing Workshop*. SBAC-PADW. Oct. 2015, pp. 79–84. DOI: 10.1109/SBAC-PADW.2015.23.

[33] A. Busse, J. H. Schönherr, M. Diener, G. Mühl, and J. Richling. "Analyzing Resource Interdependencies in Multi-core Architectures to Improve Scheduling

Decisions". In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC '13. Coimbra, Portugal: ACM, 2013, pp. 1595–1602. DOI: 10.1145/2480362.2480661.

[34] A. Busse, J. H. Schönherr, M. Diener, P. O. A. Navaux, and H.-U. Heiß. "Partial Coscheduling of Virtual Machines Based on Memory Access Patterns". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15. Salamanca, Spain: ACM, Apr. 2015, pp. 2033–2038. DOI: 10.1145/2695664.2695736.

[35] G. M. Candea and M. B. Jones. "Vassal: Loadable Scheduler Support for Multi-policy Scheduling". In: *Proceedings of the 2nd Conference on USENIX Windows NT Symposium*. WINSYM'98. Seattle, Washington: USENIX Association, 1998.

[36] L. Cardelli. "Bad Engineering Properties of Object-orient Languages". In: *ACM Computing Surveys – Special Issue: Position Statements on Strategic Sirections in Computing Research* 28.4es (Dec. 1996). DOI: 10.1145/242224.242415.

[37] G. Chrysos. *Intel Xeon Phi Coprocessor – the Architecture. The first Intel Many Integrated Core (Intel MIC) architecture product*. Nov. 2012. URL: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner (Retrieved 10/09/2015).

[38] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. "Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs?" In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 225–236. DOI: 10.1109/MICRO.2010.36.

[39] E. G. Coffman, M. Elphick, and A. Shoshani. "System Deadlocks". In: *ACM Computing Surveys* 3.2 (June 1971), pp. 67–78. DOI: 10.1145/356586.356588.

[40] K. Compton and S. Hauck. "Reconfigurable Computing: A Survey of Systems and Software". In: *ACM Computing Surveys* 34.2 (June 2002), pp. 171–210. DOI: 10.1145/508352.508353.

[41] J. Corbet. *Extending extended BPF*. July 2014. URL: https://lwn.net/Articles/603983/ (Retrieved 09/23/2016).

[42] I. Corporation. *Intel Many Integrated Core Architecture*. 2011. URL: http://www.intel.com/technology/architecture-silicon/mic/index.htm (Retrieved 05/09/2015).

[43] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. "Design Of Ion-Implanted MOSFET's with Very Small Physical Dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. DOI: 10.1109/JPROC.1999.752522.

[44] M. Desnoyers and M. R. Dagenais. "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux". In: *Proceedings of the 2006 Linux Symposium*. Vol. 1. Ottawa, Canada, July 2006, pp. 261–268.

[45] L. P. Deutsch. "Design Reuse and Frameworks in the Smalltalk-80 System". In: *Software Reusability. Applications and Experience*. Vol. 2. Ed. by T. J. Biggerstaff and A. J. Perlis. New York, NY, USA: ACM, 1989, pp. 57–71. DOI: 10.1145/75722.75725.

[46] E. W. Dijkstra. “The Structure of the "THE"-multiprogramming System”. In: *Communications of the ACM* 11.5 (May 1968), pp. 341–346. DOI: 10.1145/363095.363143.

[47] B. Doud. *Accelerating the Data Plane With the TILE-Mx Manycore Processor*. EZchip. Feb. 2015. URL: http://www.tilera.com/files/drim__EZchip_LinleyDataCenterConference_Feb2015_7671.pdf (Retrieved 10/09/2015).

[48] F. Duhem, F. Muller, and P. Lorenzini. “FaRM: Fast Reconfiguration Manager for Reducing Reconfiguration Time Overhead on FPGA”. In: *Proceedings of the 7th International Conference on Reconfigurable Computing: Architectures, Tools and Applications*. ARC’11. Belfast, UK: Springer-Verlag, 2011, pp. 253–260. DOI: 10.1007/978-3-642-19475-7_26.

[49] A. Duran and M. Klemm. “The Intel Many Integrated Core Architecture”. In: *Proceedings of the 2012 International Conference on High Performance Computing & Simulation (HPCS 2012)*. Madrid, Spain, July 2012, pp. 365–366. DOI: 10.1109/HPCSim.2012.6266938.

[50] *E16G301 Epiphany 16-core microprocessor*. Datasheet. Adapteva. June 2013. URL: http://www.adapteva.com/docs/e16g301_datasheet.pdf (Retrieved 10/09/2015).

[51] M. Ebbers, J. Kettner, W. O’Brien, and B. Ogden. *Introduction to the New Mainframe: z/OS Basics*. 3rd ed. Redbooks SG24-6366-02. IBM. Mar. 2011. ISBN: 0-7384-3534-1.

[52] F. C. Eigler. “Problem Solving with Systemtap”. In: *Proceedings of the 2006 Linux Symposium*. Vol. 1. Ottawa, Canada, July 2006, pp. 261–268.

[53] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. “Dark Silicon and the End of Multicore Scaling”. In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA ’11. San Jose, CA, USA: ACM, 2011, pp. 365–376. DOI: 10.1145/2000064.2000108.

[54] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. “The Many Faces of Publish/Subscribe”. In: *ACM Computing Surveys (CSUR)* 35.2 (June 2003), pp. 114–131. DOI: 10.1145/857076.857078.

[55] M. Fayad and D. C. Schmidt. “Object-oriented Application Frameworks”. In: *Communications of the ACM* 40.10 (Oct. 1997), pp. 32–38. DOI: 10.1145/262793.262798.

[56] *Fibers*. Microsoft. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682661(v=vs.85).aspx (Retrieved 07/03/2016).

[57] M. J. Flynn. “Some Computer Organizations and their Effectiveness”. In: *IEEE Transactions on Computers* 21.9 (Sept. 1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[58] B. Ford, K. Van, M. Jay, L. Stephen, C. Bart, and R. J. Turner. “The Flux OS Toolkit: Reusable Components for OS Implementation”. In: *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*. HOTOS ’97. Washington, DC, USA: IEEE Computer Society, May 1997, pp. 14–19. ISBN: 0-8186-7834-8.

[59] R. Freund, T. Kidd, D. Hensgen, and L. Moore. “SmartNet: a scheduling framework for heterogeneous computing”. In: *Proceedings of the 2nd International*

Symposium on Parallel Architectures, Algorithms, and Networks. June 1996, pp. 514–521. DOI: 10.1109/ISPAN.1996.509034.

[60] P. H. Fröhlich. "Component-Oriented Programming Languages: Why, What, and How". PhD thesis. University of California, Irvine, 2003.

[61] Y. Fu, T. M. Nguyen, and D. Wentzlaff. "Coherence Domain Restriction on Large Scale Systems". In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: ACM, 2015, pp. 686–698. DOI: 10.1145/2830772.2830832.

[62] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. "A New Kernel Approach for Modular Real-Time systems Development". In: *Proceedings of the 13th Euromicro Conference on Real-Time Systems*. June 2001, pp. 199–206. DOI: 10.1109/EMRTS.2001.934032.

[63] V. G. Gaitan, N. C. Gaitan, and I. Ungurean. "CPU Architecture Based on a Hardware Scheduler and Independent Pipeline Registers". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23.9 (Sept. 2015), pp. 1661–1674. DOI: 10.1109/TVLSI.2014.2346542.

[64] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. With a forew. by G. Booch. Boston, MA, USA: Addison-Wesley, 1995. ISBN: 0-201-63361-2.

[65] A. Gomez, L. Schor, P. Kumar, and L. Thiele. "SF3P: A Framework to Explore and Prototype Hierarchical Compositions of Real-Time Schedulers". In: *Proceedings of the 25th IEEE International Symposium on Rapid System Prototyping (RSP)*. Oct. 2014, pp. 2–8. DOI: 10.1109/RSP.2014.6966685.

[66] A. Greenberg. *Crypto Currency*. Forbes. Apr. 2011. URL: http://www.forbes.com/forbes/2011/0509/technology-psilocybin-bitcoins-gavin-andresen-crypto-currency.html (Retrieved 05/09/2015).

[67] G. Grey. *big.LITTLE Software Update*. Linaro. July 2013. URL: http://www.linaro.org/blog/hardware-update/big-little-software-update/ (Retrieved 01/25/2015).

[68] T. Hamada, K. Benkrid, K. Nitadori, and M. Taiji. "A Comparative Study on ASIC, FPGAs, GPUs and General Purpose Processors in the $\mathcal{O}(N^2)$ Gravitational N-body Simulation". In: *Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware and Systems*. July 2009, pp. 447–452. DOI: 10.1109/AHS.2009.55.

[69] N. Hemsoth. *Intel Marrying FPGA, Beefy Broadwell for Open Compute Future*. Mar. 2016. URL: http://www.nextplatform.com/2016/03/14/intel-marrying-fpga-beefy-broadwell-open-compute-future/ (Retrieved 06/17/2016).

[70] J. L. Hennessy and D. A. Patterson. *Computer Architecture. A Quantitative Approach*. 5th ed. Waltham, MA, USA: Morgan Kaufmann, Sept. 2011. ISBN: 978-0-12-383872-8.

[71] J. Hildebrandt, F. Golatowski, and D. Timmermann. "Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems". In: *Proceedings of 11th Euromicro Conference on Real-Time Systems*. Euromicro RTS'99. 1999, pp. 208–215. DOI: 10.1109/EMRTS.1999.777467.

[72] J. Hildebrandt and D. Timmermann. "An FPGA Based Scheduling Coprocessor for Dynamic Priority Scheduling in Hard Real-Time Systems". In: *Field-Programmable Logic and Applications. The Roadmap to Reconfigurable Computing*. Ed. by R. W. Hartenstein and H. Grünbacher. Vol. 1896. Lecture Notes in Computer Science. Berlin and Heidelberg: Springer Berlin Heidelberg, 2000, pp. 777–780. DOI: 10.1007/3-540-44614-1_83.

[73] M. D. Hill and M. R. Marty. "Amdahl's Law in the Multicore Era". In: *Computer* 41.7 (July 2008), pp. 33–38. DOI: 10.1109/MC.2008.209.

[74] J. R. Holmevik. "Compiling SIMULA: A Historical Study of Technological Genesis". In: *IEEE Annals of the History of Computing* 16.4 (Dec. 1994), pp. 25–37. DOI: 10.1109/85.329756.

[75] S. Huang, S. Xiao, and W.-C. Feng. "On the Energy Efficiency of Graphics Processing Units for Scientific Computing". In: *5th IEEE Workshop on High-Performance, Power-Aware Computing (in conjunction with the 23rd International Parallel and Distributed Processing Symposium (IPDPS 2009))*. May 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5160980.

[76] J. W. Hunt and M. D. McIlroy. *An Algorithm for Differential File Comparison*. Tech. rep. 41. Bell Telephone Laboratories, June 1976.

[77] W. L. Hürsch and C. V. Lopes. *Separation of Concerns*. Tech. rep. College of Computer Science, Northeastern University, Feb. 1995.

[78] *Information technology - Programming languages - C*. ISO/IEC 9899 (International Standard). Dec. 2011.

[79] Intel Corporation. *From a Few Cores to Many. A Tera-scale Computing Research Overview*. White Paper. 2006. URL: http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-tera-scale-research-paper.pdf (Retrieved 07/04/2016).

[80] *Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*. Intel. Sept. 2012. URL: https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf (Retrieved 06/04/2016).

[81] *Intel Xeon Phi Coprocessor x100 Product Family*. Datasheet. Intel. Apr. 2015. URL: http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-phi-coprocessor-datasheet.pdf (Retrieved 10/09/2015).

[82] S. Jennings and J. Poimboeuf. *Dynamic Kernel Patching*. Youtube. Apr. 2014. URL: https://www.youtube.com/watch?v=juyQ5TsJRTA#t=2m6s (Retrieved 06/23/2016).

[83] H. Jin, M. Frumkin, and J. Yan. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance*. Tech. rep. NAS-99-011. Moffett Field, CA, USA: NASA Ames Research Center, Oct. 1999.

[84] R. E. Johnson and B. Foote. "Designing Reusable Classes". In: *Journal of Object-Oriented Programming* 1.2 (June 1988), pp. 22–35.

[85] M. T. Jones. *Inside the Linux 2.6 Completely Fair Scheduler*. IBM. Dec. 2009. URL: https://www.ibm.com/developerworks/library/l-completely-fair-scheduler/ (Retrieved 09/23/2016).

[86] N. Jouppi. *Google Supercharges Machine Learning Tasks with TPU Custom Chip*. May 2016. URL: https://cloudplatform.googleblog.com/2016/05/

Google-supercharges-machine-learning-tasks-with-custom-chip.html (Retrieved 07/04/2015).

[87] M. Jung. "Coupling gem5 with SystemC TLM 2.0 Virtual Platforms". Presented at the second gem5 user workshop. Portland, OR, USA, June 2015. URL: http://www.m5sim.org/wiki/images/4/4c/2015_ws_09_2015-06-14_Gem5_ISCA.pptx (Retrieved 10/24/2016).

[88] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. "TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments". In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'11. Portland, OR: USENIX Association, 2011.

[89] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. "Gdev: First-Class GPU Resource Management in the Operating System". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012.

[90] G. J. Kiczales, J. O. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. "Aspect-oriented programming". In: *ECOOP'97 — Object-Oriented Programming*. Ed. by M. Akşit and S. Matsuoka. Vol. 1241. Lecture Notes in Computer Science. Berlin and Heidelberg: Springer, 1997, pp. 220–242. DOI: 10.1007/BFb0053381.

[91] C. Kolivas. *The Brain Fuck Scheduler, BFS461*. Feb. 2015. URL: http://ck.kolivas.org/patches/bfs/3.0/3.19/3.19-sched-bfs-461.patch (Retrieved 03/15/2015).

[92] C. Kolivas. *The Staircase Deadline CPU Scheduler Policy*. May 2007. URL: http://mirror.linux.org.au/linux/kernel/people/ck/patches/staircase-deadline/2.6.21/2.6.21-sd-0.48.patch (Retrieved 03/15/2015).

[93] R. Kurzweil. "The Law of Accelerating Returns". In: *Alan Turing: Life and Legacy of a Great Thinker*. Ed. by C. Teuscher. Berlin and Heidelberg: Springer, 2004, pp. 381–416. DOI: 10.1007/978-3-662-05642-4_16.

[94] J. L. Lawall, H. Duchesne, G. Muller, and A.-F. Le Meur. "Bossa Nova: Introducing Modularity into the Bossa Domain-specific Language". In: *Generative Programming and Component Engineering. Proceedings of the 4th International Conference on Generative Programming and Component Engineering*. Ed. by R. Glück and M. Lowry. Vol. 3676. Lecture Notes in Computer Science. Berlin and Heidelberg: Springer, 2005, pp. 78–93. DOI: 10.1007/11561347_7.

[95] J. L. Lawall, G. Muller, and L. P. Barreto. "Capturing OS Expertise in an Event Type System: The Bossa Experience". In: *Proceedings of the 10th ACM SIGOPS European Workshop*. EW 10. Saint-Emilion, France: ACM, 2002, pp. 54–61. DOI: 10.1145/1133373.1133384.

[96] J. L. Lawall, G. Muller, and H. Duchesne. "Invited Application Paper: Language Design for Implementing Process Scheduling Hierarchies". In: *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '04. Verona, Italy: ACM, 2004, pp. 80–91. DOI: 10.1145/1014007.1014016.

[97] C. L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *Journal of the Association for Computing Machinery* 20.1 (Jan. 1973), pp. 46–61. DOI: 10.1145/321738.321743.

[98] J. Lozi, B. Lepers, J. R. Funston, F. Gaud, V. Quéma, and A. Fedorova. "The Linux Scheduler: A Decade of Wasted Coress". In: *Proceedings of the 11th European Conference on Computer Systems*. EuroSys 2016. London, United Kingdom: ACM, Apr. 2016, 1:1–1:16. DOI: 10.1145/2901318.2901326.

[99] E. Lübbers and M. Platzner. "ReconOS: Multithreaded Programming for Reconfigurable Computers". In: *ACM Transactions on Embedded Computing Systems* 9.1 (Oct. 2009), 8:1–8:33. DOI: 10.1145/1596532.1596540.

[100] M. M. K. Martin, M. D. Hill, and D. J. Sorin. "Why On-chip Cache Coherence is Here to Stay". In: *Communications of the ACM* 55.7 (July 2012), pp. 78–89. DOI: 10.1145/2209249.2209269.

[101] J. Mauro and R. McDougall. *Solaris Internals: Core Kernel Architecture*. Mountain View, CA, USA: Sun Microsystems, Inc., 2001. ISBN: 0-13-022496-0.

[102] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmer's Manual*. MIT Press, 1962. ISBN: 0-262-13011-4.

[103] J. McCarthy, R. Brayton, D. J. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, D. Park, and S. Russell. *LISP I Programmer's Manual*. Cambridge, Massachusetts, Mar. 1960.

[104] J. McGregor, J. Goodacre, and B. Carlson. "The Evolution of Mobile Technology Series. Part 3 – The Evolution of Mobile Processing Architectures". Webinar. June 2009. URL: http://www.ti.com/pdfs/wtbu/smp_webinar.pdf (Retrieved 07/04/2016).

[105] D. McIlroy. "'Mass produced' Software Components". In: *Software Engineering. Report of a conference sponsored by the NATO Science Committee*. Ed. by P. Naur and B. Randell. Garmisch, Germany: Scientific Affairs Division, NATO, Jan. 1969, pp. 138–156.

[106] S. McIntosh-Smith, T. Wilson, A. Á. Ibarra, J. Crisp, and R. B. Sessions. "Benchmarking Energy Efficiency, Power Costs and Carbon Emissions on Heterogeneous Systems". In: *The Computer Journal* 55.2 (Feb. 2012), pp. 192–205. DOI: 10.1093/comjnl/bxr091.

[107] P. Menage. *CGROUPS*. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt (Retrieved 02/08/2016).

[108] *Microft Windows Hotfix KB2592546: An update to optimize the performance of AMD Bulldozer CPUs*. Hofix withdrawn 12/16/2011. Microsoft Corporation, Dec. 2011.

[109] *Microft Windows Hotfix KB2645594: An update is available for computers that have an AMD FX, AMD Opteron 4200, AMD Opteron 6200, or AMD Bulldozer series processor installed and that are running Windows 7 or Windows Server 2008 R2*. Microsoft Corporation. Jan. 2012. URL: http://support.microsoft.com/kb/2645594 (Retrieved 01/21/2015).

[110] *Microft Windows Hotfix KB2646060: An update that selectively disables the Core Parking feature in Windows 7 or in Windows Server 2008 R2 is available.*

Microsoft Corporation. Jan. 2012. URL: http://support.microsoft.com/kb/2646060 (Retrieved 01/21/2015).

[111] D. Moloney, B. Barry, R. Richmond, F. Connor, C. Brick, and D. Donohoe. "Myriad 2: Eye of the Computational Vision Storm". Presentation at 26th Hot Chips Symposium on High Performance Chips. Aug. 2014. URL: http://www.hotchips.org/wp-content/uploads/hc_archives/hc26/HC26-12-day2-epub/HC26.12-6-HP-ASICs-epub/HC26.12.620-Myriad2-Eye-Moloney-Movidius-provided.pdf (Retrieved 07/04/2015).

[112] G. E. Moore. "Cramming More Components onto Integrated Circuits". In: *Electronics* 38.8 (Apr. 1965), pp. 114–117. DOI: 10.1109/jproc.1998.658762.

[113] G. Muller, C. Consel, R. Marlet, L. P. Barreto, F. Mérillon, and L. Réveillère. "Towards Robust OSes for Appliances: A New Approach Based on Domain-specific Languages". In: *Proceedings of the 9th ACM SIGOPS European Workshop. Beyond the PC: New Challenges for the Operating System*. EW 9. Kolding, Denmark: ACM, 2000, pp. 19–24. DOI: 10.1145/566726.566732.

[114] G. Muller, J. L. Lawall, and H. Duchesne. "A Framework for Simplifying the Development of Kernel Schedulers: Design and Performance Evaluation". In: *Proceedings of the 9th IEEE International Symposium on High Assurance Systems Engineering*. Heidelberg, Germany, Oct. 2005, pp. 56–65. DOI: 10.1109/HASE.2005.1.

[115] M. Nabelsee, A. Busse, H. Parzyjegla, and G. Mühl. "Load-aware Scheduling for Heterogeneous Multi-core Systems". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC '16. Pisa, Italy: ACM, Apr. 2016, pp. 1844–1851. DOI: 10.1145/2851613.2851738.

[116] *NO_HZ: Reducing Scheduling-Clock Ticks*. The Linux Kernel Organization. May 2015. URL: https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt (Retrieved 07/09/2015).

[117] NVIDIA Corporation. *Variable SMP — A Multi-Core CPU Architecture for Low Power and High Performance*. White Paper. 2011. URL: http://www.nvidia.com/content/pdf/tegra_white_papers/tegra-whitepaper-0911b.pdf (Retrieved 07/16/2016).

[118] Oracle Corporation. *Oracle Solaris ZFS Administration Guide*. E37384–02. June 2013. URL: https://docs.oracle.com/cd/E26505_01/pdf/E37384.pdf (Retrieved 02/08/2016).

[119] *Parallella-1.x Reference Manual*. Parallella. 2014. URL: http://www.parallella.org/docs/parallella_manual.pdf (Retrieved 07/16/2016).

[120] D. L. Parnas. "A Technique for Software Module Specification with Examples". In: *Communications of the ACM* 15.5 (May 1972), pp. 330–336. DOI: 10.1145/355602.361309.

[121] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Communications of the ACM* 15.12 (Dec. 1972), pp. 1053–1058. DOI: 10.1145/361598.361623.

[122] A. Patil and N. Audsley. "Implementing Application Specific RTOS Policies using Reflection". In: *Proceedings of the 11th IEEE Real Time and Embedded*

Technology and Applications Symposium. Mar. 2005, pp. 438–447. DOI: 10.1109/RTAS.2005.27.

[123] T. Pattison. *Programming Distributed Applications with COM+ and Microsoft Visual Basic*. 2nd ed. Redmond, WA, USA: Microsoft Press, 2000. ISBN: 0-7356-1010-X.

[124] V. Pavlík. "kGraft. Live patching of the Linux kernel". online. 2014. URL: http://events.linuxfoundation.org/sites/events/files/slides/kGraft.pdf (Retrieved 06/23/2016).

[125] S. Peter. "Resource Management in a Multicore Operating System". PhD thesis. Zurich, Switzerland: ETH Zurich, Oct. 2012.

[126] R. N. Pittman, N. L. Lynch, and A. Forin. *eMIPS, A Dynamically Extensible Processor*. Tech. rep. MSR-TR-2006-143. Redmond, WA, USA: Microsoft Research, Oct. 2006. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=70354 (Retrieved 11/29/2015).

[127] G. J. Popek and R. P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. DOI: 10.1145/361011.361073.

[128] *Programming languages – C*. ISO/IEC 9899:2011 (Standard). Dec. 2011.

[129] K. Ramamritham and J. A. Stankovic. "Scheduling Algorithms and Operating Systems Support for Real-Time Systems". In: *Proceedings of the IEEE* 82.1 (Jan. 1994), pp. 55–67. DOI: 10.1109/5.259426.

[130] E. S. Raymond. *The Art of UNIX Programming*. Addison-Wesley, Oct. 2003. ISBN: 0-13-142901-9.

[131] J. D. Regehr. "Using Hierarchical Scheduling to Support Soft Real-Time Applications in General-Purpose Operating Systems". PhD thesis. University of Virginia, May 2001.

[132] J. Regehr and J. A. Stankovic. "HLS: A Framework for Composing Soft Real-Time Schedulers". In: *Proceedings of the 22nd IEEE Real-Time Systems Symposium*. Dec. 2001, pp. 3–14. DOI: 10.1109/REAL.2001.990591.

[133] J. Richling, J. H. Schönherr, G. Mühl, and M. Werner. "Towards Energy-Aware Multi-Core Scheduling". In: *Praxis der Informationsverarbeitung und Kommunikation (PIK)* 32.2 (Apr. 2009), pp. 88–96. DOI: 10.1515/piko.2009.0017.

[134] J. Roberson. "ULE: A Modern Scheduler for FreeBSD". In: *Proceedings of the BSD Conference 2003 on BSD Conference*. BSDC'03. San Mateo, California: USENIX Association, 2003, pp. 17–28.

[135] M. P. Robillard and G. C. Murphy. "Representing Concerns in Source Code". In: *ACM Transactions on Software Engineering and Methodology* 16.1 (Feb. 2007). DOI: 10.1145/1189748.1189751.

[136] S. Rostedt and D. V. Hart. "Internals of the RT Patch". In: *Proceedings of the 2007 Linux Symposium*. Vol. 2. Ottawa, Canada, June 2007, pp. 161–172.

[137] R. Russell. *Unreliable Guide To Locking*. 2003. URL: https://www.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/index.html (Retrieved 02/08/2016).

[138] J. C. Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto-Matias. "ACFS: A Completely Fair Scheduler for Asymmetric Single-ISA Multicore Systems". In:

Proceedings of the 30th Annual ACM Symposium on Applied Computing. SAC '15. Salamanca, Spain: ACM, 2015, pp. 2027–2032. DOI: 10.1145/2695664.2695714.

[139] K. Sakamura and H. Takada, eds. *μITRON 4.0 Specification*. Version 4.00.00. TRON Association. Tokyo, Japan, June 1999.

[140] J. H. Schönherr, B. Lutz, and J. Richling. "Non-intrusive Coscheduling for General Purpose Operating Systems". In: *Multicore Software Engineering, Performance, and Tools. International Conference, MSEPT 2012 Prague, Czech Republic*. Ed. by V. Pankratius and M. Philippsen. Vol. 7303. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 66–77. DOI: 10.1007/978-3-642-31202-1_7.

[141] J. H. Schönherr, J. Richling, G. Mühl, and M. Werner. "A Scheduling Approach for Efficient Utilization of Hardware-Driven Frequency Scaling". In: *Proceedings of the 23rd International Conference on Architecture of Computing Systems*. ARCS 2010. Hannover, Germany: VDE Verlag, Feb. 2010, pp. 367–376. ISBN: 978-3-8007-3222-7.

[142] K. Sewell, R. G. Dreslinski, T. Manville, S. Satpathy, N. R. Pinckney, G. Blake, M. Cieslak, R. Das, T. F. Wenisch, D. Sylvester, D. Blaauw, and T. N. Mudge. "Swizzle-Switch Networks for Many-Core Systems". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 2.2 (June 2012), pp. 278–294. DOI: 10.1109/JETCAS.2012.2193936.

[143] D. Sheldon and A. Forin. *An Online Scheduler for Hardware Accelerators on General-Purpose Operating Systems*. Tech. rep. MSR-TR-2010-169. Redmond, WA, USA: Microsoft Research, Dec. 2010. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=80234 (Retrieved 11/29/2015).

[144] S. Singh. "Integrating FPGAs in High-performance Computing: Programming Models for Parallel Systems – the Programmer's Perspective". In: *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*. FPGA '07. Monterey, CA, USA: ACM, 2007, pp. 133–135. DOI: 10.1145/1216919.1216942.

[145] H. K.-H. So. "BORPH: An Operating System for FPGA-Based Reconfigurable Computers". PhD thesis. EECS Department, University of California, Berkeley, July 2007.

[146] Sony Computer Entertainment Inc. *Open Source Software used in PlayStation 4*. 2014. URL: http://doc.dl.playstation.net/doc/ps4-oss/ (Retrieved 02/08/2016).

[147] W. Stallings. *Operating Systems. Internals and Design Principles*. 7th ed. Pearson, 2011. ISBN: 978-0-273-75150-2.

[148] R. Stallman. *GNU General Public License*. June 2007. URL: http://www.gnu.org/licenses/gpl.html (Retrieved 02/08/2016).

[149] F. Steimann and P. Mayer. "Patterns of Interface-Based Programming". In: *Journal of Object Technology* 4.5 (July 2005), pp. 75–94. DOI: 10.5381/jot.2005.4.5.a1.

[150] I. Stoica and H. Abdel-Wahab. *Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation*. Tech. rep. TR-95-22. Norfolk, VA, USA: Old Dominion University, 1995.

[151] J. E. Stone, D. Gohara, and G. Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science Engineering* 12.3 (May 2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69.

[152] *SUSE Linux Enterprise Live Patching*. Datasheet. SUSE. Dec. 2014. URL: https://www.suse.com/docrep/documents/kmf54g86yg/sle_live_patching_data_sheet.pdf (Retrieved 06/20/2016).

[153] C. Szyperski. *Component Software. Beyond Object-Oriented Programming*. 2nd ed. Boston, MA, USA: Addison-Wesley, 2002. ISBN: 0-201-74572-0.

[154] C. Szyperski and D. G. Messerschmitt. "The Flexible Factory". In: *Software Development Magazine* (Sept. 2003). URL: http://www.drdobbs.com/the-flexible-factory/184415060 (Retrieved 06/20/2015).

[155] S. T. Taft, R. A. Duff, R. Brukardt, E. Plödereder, P. Leroy, and E. Schonberg. *Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/2012 (E)*. Vol. 8339. Lecture Notes in Computer Science. Springer, 2013. ISBN: 978-3-642-45418-9.

[156] P. Tahchiev, F. Leme, V. Massol, and G. Gregory. *JUnit in Action*. 2nd ed. Stamford, CT, USA: Manning Publications Co., 2010. ISBN: 978-1-935182-02-3.

[157] A. S. Tanenbaum. *Modern Operating Systems*. 3rd ed. Pearson Prentice Hall, 2007. ISBN: 978-0-13-813459-4.

[158] M. Taylor. "A Landscape of the New Dark Silicon Design Regime". In: *IEEE Micro* 33.5 (Sept. 2013), pp. 8–19. DOI: 10.1109/MM.2013.90.

[159] M. B. Taylor. "Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. San Francisco, California: ACM, 2012, pp. 1131–1136. DOI: 10.1145/2228360.2228567.

[160] The FreeBSD Project. *Simplified BSD License*. URL: https://www.freebsd.org/copyright/freebsd-license.html (Retrieved 02/08/2016).

[161] *The Open Group Base Specifications Issue 7*. IEEE Std 1003.1, 2013 Edition (Standard). 2013.

[162] The Rust Core Team. *Announcing Rust 1.0*. May 2015. URL: http://blog.rust-lang.org/2015/05/15/Rust-1.0.html (Retrieved 03/06/2016).

[163] The Rust Project Developers. *Rust Documentation*. Module std::option. 2016. URL: https://doc.rust-lang.org/std/option/index.html (Retrieved 08/25/2016).

[164] The Santa Cruz Operation and AT&T. *System V Application Binary Interface*. Version 4.1. Mar. 1997.

[165] *The SCC Platform Overview*. Intel Labs. May 2010. URL: http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-platform-overview-paper.pdf (Retrieved 10/10/2015).

[166] *The SPARC Architecture Manual. Version 8*. SPARC International. 1992. ISBN: 0-13-825001-4.

[167] D. B. Thomas, L. Howes, and W. Luk. "A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation". In: *Proceedings of the ACM/SIGDA 17th International Symposium on Field Programmable Gate Arrays*. FPGA '09. Monterey, CA, USA: ACM, 2009, pp. 63–72. DOI: 10.1145/1508128.1508139.

[168] C. J. Thompson, S. Hahn, and M. Oskin. "Using Modern Graphics Architectures for General-purpose Computing: A Framework and Analysis". In: *Proceedings of the 35th Annual International Symposium on Microarchitecture*. Istanbul, Turkey: IEEE Computer Society Press, Nov. 2002, pp. 306–317. ISBN: 0-7695-1859-1.

[169] *TIOBE Index*. TIOBE Software. June 2015. URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (Retrieved 06/01/2015).

[170] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. Version 1.2. May 1995.

[171] A. Vajda. *Programming Many-Core Chips*. New York: Springer US, 2011. ISBN: 978-1-4419-9738-8.

[172] G. Vanderburg. *Tricks of the Java Programming Gurus*. Carmel, IN, USA: SAMS, 1996. ISBN: 978-1575211022.

[173] S. Vitkar and M. Shah. *Performance report on hardware accelerator with EMC Retrospect 7.5*. San Jose, CA, USA: Indra Networks, May 2007.

[174] VMware Inc. *The CPU Scheduler in VMware vSphere 5.1*. Performance Study. Technical White Paper EN-001115-00. VMware Inc., Feb. 2013. URL: https://www.vmware.com/files/pdf/techpaper/VMware-vSphere-CPU-Sched-Perf.pdf (Retrieved 11/13/2015).

[175] N. Watkins, J. Straub, and D. Niehaus. "A Flexible Scheduling Framework Supporting Multiple Programming Models with Arbitrary Semantics in Linux". Presented at the 11th Real-Time Linux Workshop. Dresden, Germany, Dec. 2009. URL: https://users.soe.ucsc.edu/~jayhawk/watkins-rtlws09.pdf (Retrieved 10/24/2016).

[176] V. M. Weaver. *Linux perf_event Features and Overhead*. Appeared in the 2013 FastPath Workshop. Apr. 2013.

[177] V. M. Weaver. "Self-monitoring overhead of the Linux perf_event performance counter interface". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2015, pp. 102–111. DOI: 10.1109/ISPASS.2015.7095789.

[178] B. White, M. Almeida, E. Bakker, M. Ebbers, P. Hamid, N. Hinuma, R. Kleven, J. McDonough, F. Nogal, V. Ranieri Jr., K.-E. Stenfors, and J. Yang. *IBM zEnterprise System Technical Introduction*. 1st ed. Redbooks SG24-7832-00. IBM. Sept. 2010. ISBN: 0-7384-3476-0.

[179] W. A. Wulf and S. A. McKee. "Hitting the Memory Wall: Implications of the Obvious". In: *ACM SIGARCH Computer Architecture News* 23.1 (Mar. 1995), pp. 20–24. DOI: 10.1145/216585.216588.

[180] *Zynq-7000 All Programmable SoC First Generation Architecture*. Product Specification. Xilinx. Jan. 2016. URL: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf (Retrieved 06/04/2016).

Software References

[S1] *AngularJS*. Google. 2015. URL: https://angularjs.org (Retrieved 06/04/2015).

[S2] *The Barrelfish Operating System*. Nov. 2015. URL: http://www.barrelfish.org (Retrieved 11/29/2015).

[S3] D. P. Sydow. *Programming the Be Operating System. Writing Programs for the Be Operating System*. O'Reilly Media, June 1999.

[S4] *Bossa - A Framework for Scheduler Development*. Nov. 2004. URL: http://bossa.lip6.fr (Retrieved 06/19/2015).

[S5] *The BSD libc*. 2015. URL: https://svnweb.freebsd.org/base/head/lib/libc/ (Retrieved 06/11/2015).

[S6] *The C Run-Time Library Reference*. Microsoft. 2013. URL: https://msdn.microsoft.com/en-us/library/59ey50w6.aspx (Retrieved 06/11/2015).

[S7] *The Cocoa Layer*. Apple. 2015. URL: https://developer.apple.com/library/mac/navigation/#section=Frameworks&topic=Cocoa%20Layer (Retrieved 06/04/2015).

[S8] *The CRIU Project*. 2015. URL: https://criu.org (Retrieved 11/13/2015).

[S9] *The Compute Unified Device Architecture (CUDA)*. June 2007. URL: http://www.nvidia.com/object/cuda_home_new.html (Retrieved 05/05/2015).

[S10] *The Darwin kernel, 14.3.0*. Apple. Nov. 2014. URL: http://www.opensource.apple.com/source/xnu/xnu-2782.1.97/ (Retrieved 06/04/2015).

[S11] *The Django Project*. 2015. URL: https://www.djangoproject.com (Retrieved 06/04/2015).

[S12] *The eCos real-time operating system, v3.0*. Mar. 2009. URL: ftp://ecos.sourceware.org/pub/ecos/releases/ecos-3.0/ (Retrieved 03/15/2015).

[S13] *The ExSched Scheduling Framework*. URL: http://www.idt.mdh.se/~exsched/ (Retrieved 07/03/2016).

[S14] *The FreeBSD kernel, v10.2*. Mar. 2016. URL: https://svnweb.freebsd.org/base/stable/10/ (Retrieved 07/31/2015).

[S15] *ftrace – Function Tracer*. 2012. URL: https://www.kernel.org/doc/Documentation/trace/ftrace.txt (Retrieved 06/23/2016).

[S16] *The GNU C Library (glibc)*. 2015. URL: http://www.gnu.org/software/libc/ (Retrieved 06/11/2015).

[S17] R. S. Engelschall. *GNU Pth – The GNU Portable Threads*. June 2006. URL: https://www.gnu.org/software/pth/ (Retrieved 07/03/2016).

[S18] *The Go programming language*. 2015. URL: https://golang.org (Retrieved 06/04/2015).

[S19] *The Apache Hadoop Framework*. Apache Software Foundation. 2015. URL: http://hadoop.apache.org (Retrieved 06/04/2015).

[S20] *The Haiku operating system*. 2015. URL: https://git.haiku-os.org/haiku (Retrieved 06/04/2015).

[S21] *Hierarchical Loadable Schedulers*. Nov. 2002. URL: https://sourceforge.net/projects/linux-hls/ (Retrieved 06/25/2016).

[S22] *kpatch: dynamic kernel patching*. May 2016. URL: https://github.com/dynup/kpatch (Retrieved 06/23/2016).

[S23] *The Kernel-based Virtual Machine*. 2015. URL: http://www.linux-kvm.org (Retrieved 11/13/2015).

[S24] *The Linux kernel, v4.4*. Jan. 2016. URL: https://www.kernel.org/pub/linux/kernel/v4.x/ (Retrieved 02/15/2015).

[S25] *LTTng*. 2009. URL: http://lttng.org (Retrieved 09/23/2016).

[S26] *The Microsoft Foundation Class Library*. Microsoft. 2015. URL: https://msdn.microsoft.com/en-us/library/d06h2x6e(v=VS.120).aspx (Retrieved 06/04/2015).

[S27] *NAS Parallel Benchmarks*. Mar. 2016. URL: https://www.nas.nasa.gov/publications/npb.html (Retrieved 10/28/2016).

[S28] *The NetBSD Project*. Nov. 2015. URL: https://www.netbsd.org (Retrieved 11/29/2015).

[S29] *The Open Computing Language (OpenCL)*. Aug. 2009. URL: https://www.khronos.org/opencl (Retrieved 05/05/2015).

[S30] *perf*. Dec. 2009. URL: https://perf.wiki.kernel.org/index.php/Main_Page (Retrieved 09/23/2016).

[S31] *The Qt Project*. 2015. URL: http://www.qt.io (Retrieved 06/04/2015).

[S32] *The Redox Operating System*. 2016. URL: https://www.redox-os.org (Retrieved 08/25/2016).

[S33] *Ruby on Rails*. 2015. URL: http://rubyonrails.org (Retrieved 06/04/2015).

[S34] *The CONFIG_PREEMPT_RT Patch Set*. Oct. 2016. URL: https://www.kernel.org/pub/linux/kernel/projects/rt/ (Retrieved 10/23/2016).

[S35] *The Rust Programming Language*. 2016. URL: https://doc.rust-lang.org/book/ (Retrieved 02/08/2016).

[S36] *rust-bindgen – A Native Binding Generator for the Rust Language*. 2016. URL: https://github.com/Yamakaky/rust-bindgen (Retrieved 08/25/2016).

[S37] *rust.ko – A Minimal Linux Kernel Module Written in Rust*. 2016. URL: https://github.com/tsgates/rust.ko (Retrieved 08/25/2016).

[S38] *The S.Ha.R.K. Project*. Feb. 2008. URL: http://shark.sssup.it (Retrieved 06/30/2016).

[S39] *systemtap*. 2005. URL: https://sourceware.org/systemtap/ (Retrieved 09/23/2016).

[S40] *The Standard Performance Evaluation Corporation (SPEC)*. June 2016. URL: https://www.spec.org (Retrieved 07/04/2016).

[S41] *The Windows Research Kernel, v1.2*. Microsoft. June 2006. URL: http://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=7366&c1=en-us&c2=0 (Retrieved 09/07/2013).