A New System Architecture for Heterogeneous Compute Units

Dissertation zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

Vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

Eingereicht von
M.Sc. Nils Asmussen
geboren am 31.01.1984
in Schleswig

Betreuender Hochschullehrer:	Prof. Dr. Hermann Härtig Technische Universität Dresden
Gutachter:	Prof. Dr. Timothy Roscoe ETH Zürich
Fachreferent:	Prof. Dr. Christof Fetzer Technische Universität Dresden
Statusvortrag:	31.01.2017
Abgabe:	20.11.2018
Verteidigung:	10.05.2019

15. Juli 2019

Contents

List of Figures **7**

List of Tables **9**

1 Introduction **11**
1.1 Motivation . . . 11
1.1.1 Increasing Heterogeneity . . . 11
1.1.2 Future Hardware Platforms . . . 13
1.1.3 Problems of Current OS Designs . . . 14
1.2 Approach . . . 14
1.3 Contributions . . . 16

2 Related Work **19**
2.1 Operating Systems . . . 19
2.2 Hardware Components . . . 22

3 Isolation and Communication **25**
3.1 Motivation . . . 25
3.2 Threat Model . . . 26
3.3 Overview and Comparison . . . 26
3.3.1 Privilege Levels . . . 27
3.3.2 Isolation . . . 27
3.3.3 Communication . . . 28
3.3.4 Role of the Kernel . . . 29
3.3.5 Trusted Computing Base . . . 29
3.4 Data Transfer Unit . . . 30
3.4.1 Integration . . . 30
3.4.2 Endpoints . . . 30
3.4.3 Commands . . . 31
3.4.4 Receiving Messages . . . 32
3.4.5 Replying to Messages . . . 33
3.4.6 Credit System . . . 33
3.4.7 Command Abortion . . . 34
3.4.8 Discussion . . . 34
3.5 The Operating System M3 . . . 35
3.5.1 System Calls . . . 36
3.5.2 Capabilities . . . 36
3.5.3 Virtual PEs . . . 37
3.5.4 Gates . . . 38
3.5.5 Memory Management . . . 39

3.5.6 Endpoint Multiplexing . . . 39
3.5.7 Discussion . . . 40
3.6 Interplay . . . 41
3.7 Evaluation . . . 42
3.7.1 Prototype Platforms . . . 42
3.7.2 System Call and IPC Performance . . . 43
3.7.3 Power Consumption and Chip Area . . . 45
3.8 Summary . . . 47

4 Operating-System Services 49
4.1 Motivation . . . 49
4.2 Services . . . 50
4.2.1 Service and Session . . . 50
4.2.2 Service Protocol . . . 50
4.3 File Protocol . . . 51
4.3.1 Design Goals . . . 51
4.3.2 The Protocol . . . 52
4.3.3 File Multiplexing . . . 53
4.4 File System . . . 53
4.4.1 Overview . . . 53
4.4.2 Data Organization . . . 54
4.4.3 File Session . . . 54
4.4.4 Metadata Session . . . 55
4.4.5 Limitations . . . 56
4.5 Pipe . . . 56
4.5.1 Overview . . . 56
4.5.2 Data Access . . . 56
4.6 Virtual File System . . . 57
4.6.1 Files and File Systems . . . 57
4.6.2 Selective Inheritance . . . 58
4.7 Discussion . . . 59
4.7.1 File System Access Control . . . 59
4.7.2 M3FS for Storage Devices . . . 59
4.7.3 POSIX Compatibility . . . 60
4.8 Evaluation . . . 60
4.8.1 File System Read/Write/Copy . . . 60
4.8.2 File Fragmentation . . . 62
4.8.3 Pipe . . . 62
4.9 Summary . . . 63

5 Virtual Memory 65
5.1 Motivation . . . 65
5.2 Goals . . . 65
5.3 Overview . . . 67
5.4 Related Work . . . 68
5.5 Integration . . . 69
5.6 Uniform Addressing . . . 70
5.7 Data Transfers . . . 71
5.8 Address Translation . . . 72
5.8.1 PE-Type B . . . 73

5.8.2 PE-Type C . . . 75
5.9 Virtual-Memory Management . . . 77
5.9.1 Overview . . . 77
5.9.2 Mapping Capabilities . . . 78
5.9.3 Page Table Entries . . . 79
5.9.4 Pager . . . 79
5.9.5 Message Passing . . . 81
5.10 Interplay . . . 81
5.11 Revisiting the TCB . . . 83
5.12 Discussion . . . 84
5.12.1 The VMA in Existing OSes . . . 84
5.12.2 Caches in Type B PEs . . . 84
5.12.3 Page Faults in Type B PEs . . . 85
5.12.4 Cache Coherency . . . 86
5.13 Evaluation . . . 86
5.13.1 Measurement Setup . . . 86
5.13.2 Revisiting System Calls . . . 87
5.13.3 Revisiting File Systems and Pipes . . . 87
5.13.4 TLB Misses and Page Faults . . . 88
5.13.5 VPE::run and VPE::exec . . . 90
5.14 Summary . . . 91

6 Autonomous Accelerators **93**
6.1 Motivation and Related Work . . . 93
6.2 Accelerator Types . . . 94
6.2.1 Memory Access . . . 94
6.2.2 Implementation Paradigm . . . 95
6.3 Goals . . . 96
6.4 Overview . . . 97
6.4.1 Accelerator Usage . . . 97
6.4.2 VPEs for Accelerators . . . 98
6.5 Request-Processing Accelerators . . . 99
6.5.1 Integration . . . 99
6.5.2 Interruptibility . . . 99
6.5.3 Usage . . . 100
6.6 Stream-Processing Accelerators . . . 101
6.6.1 Integration . . . 101
6.6.2 Direct Data Exchange . . . 102
6.6.3 Shell Extension . . . 102
6.7 Evaluation . . . 103
6.7.1 Accelerator Logic . . . 103
6.7.2 Request-Processing Accelerators . . . 103
6.7.3 Stream-Processing Accelerators . . . 105
6.8 Summary . . . 107

7 Context Switching **109**
7.1 Motivation . . . 109
7.2 Related Work . . . 110
7.3 Overview . . . 110
7.4 Context-Enabled Communication . . . 111

7.4.1 VPE-aware Communication . . . 112
7.4.2 Message Forwarding . . . 112
7.4.3 VPE Migration . . . 113
7.4.4 Computing vs. Idling . . . 113
7.4.5 Gang Scheduling . . . 113
7.4.6 Revisiting Command Abortion . . . 114
7.5 RCTMux Implementation . . . 114
7.5.1 General-Purpose Cores . . . 115
7.5.2 Accelerators . . . 115
7.6 Revisiting the TCB . . . 116
7.6.1 How Powerful is RCTMux? . . . 116
7.6.2 Is the Privileged CPU Mode Required? . . . 116
7.7 Evaluation . . . 117
7.7.1 Communication with Suspended VPEs . . . 117
7.7.2 Non-communicating Applications . . . 119
7.7.3 Communicating Applications . . . 119
7.8 Summary . . . 122

8 Evaluation **123**
8.1 Experimental Setup . . . 123
8.1.1 Evaluation Platform . . . 123
8.1.2 Systrace Infrastructure . . . 124
8.2 Performance . . . 124
8.2.1 Standalone Applications . . . 124
8.2.2 Pipelines of Applications . . . 126
8.3 Scalability . . . 127
8.3.1 Standalone Applications . . . 128
8.3.2 Pipelines of Applications . . . 129
8.3.3 Web Server . . . 130
8.4 Efficiency . . . 130
8.4.1 Single Application Instances . . . 131
8.4.2 Multiple Application Instances . . . 132
8.4.3 System Efficiency . . . 133
8.5 Autonomous Image Processing . . . 134
8.6 Accelerator Sharing . . . 136
8.7 Software Complexity . . . 137

9 Conclusion and Future Work **139**
9.1 Conclusion . . . 139
9.2 Extensions and Future Work . . . 142

Acknowledgements **145**

Glossary **147**

Bibliography **151**

List of Figures

1.1 Potential future platform . . . 13
1.2 The key ideas of the new system architecture . . . 15

3.1 System overview . . . 26
3.2 PE internals . . . 30
3.3 Capability system . . . 36
3.4 VPE state diagram . . . 37
3.5 Modules of the CU-specific helper . . . 38
3.6 DTU-based interactions in the producer-consumer example . . . 41
3.7 Schematic depiction of the Tomahawk platform . . . 42
3.8 Syscall and IPC performance on Linux, NOVA, and M^3 . . . 44
3.9 Power consumption for message transfers and RPC scenarios . . . 46

4.1 Client and server interaction . . . 50
4.2 The file protocol . . . 52
4.3 M^{3}FS' data structures . . . 54
4.4 Overview of the VFS in form of a UML class diagram . . . 57
4.5 M^{3}FS' performance in read, write, and copy . . . 61
4.6 Pipe performance . . . 63

5.1 Abstract integration concept of CUs with caches . . . 67
5.2 Detailed depiction of the integration of different PE types . . . 69
5.3 Overview of the DTU's XferUnit . . . 72
5.4 The internals of the MMU in the DTU of PE-type B . . . 73
5.5 Address translation in PE-type C . . . 75
5.6 Modules of the CU-specific helper . . . 76
5.7 Overview of the virtual-memory management . . . 77
5.8 The data structures of the pager . . . 80
5.9 The sequence of events for a RDMA read from virtual memory . . . 82
5.10 Different PE-type B configurations . . . 85
5.11 System call performance on the different PE types . . . 87
5.12 File system and pipe performance on different PE types . . . 88
5.13 TLB miss and page fault handling performance . . . 89
5.14 Performance comparison of `fork` and `VPE::run` . . . 90
5.15 Performance comparison of `vfork+exec` and `VPE::exec` . . . 91

6.1 Stream processing versus request processing . . . 96
6.2 Overview of the accelerator integration . . . 97
6.3 The integration of request-processing accelerators . . . 99
6.4 The integration of stream-processing accelerators . . . 101

6.5 Performance and autonomy of request-processing accelerators 104
6.6 Performance and autonomy of stream-processing accelerators 106

7.1 Overview of components and interfaces for context switching 111
7.2 Modules of the CU-specific helper 115
7.3 RCTMux in accelerators 115
7.4 Overhead of communication with running and suspended VPEs 118
7.5 Context switching overhead for non-communicating applications . . . 119
7.6 Context switching overhead for communicating applications 120
7.7 Context switching overhead for communicating apps on a single PE . . 121
7.8 Kernel load with context-switched communicating applications 122

8.1 Performance comparison to Linux using standalone applications 125
8.2 Performance comparison to Linux using a pipeline between two apps . 127
8.3 Parallel efficiency of standalone applications 129
8.4 Parallel efficiency of application pipelines and scalability of *nginx* . . . 130
8.5 Performance of standalone apps with a varying number of user PEs . . 131
8.6 Performance of app pipelines with a varying number of user PEs 131
8.7 Parallel efficiency of standalone apps and app pipelines 132
8.8 System efficiency with optimal number and placement of OS servers . 134
8.9 Runtime and required CPU time for accelerator chains 135
8.10 Context switching overhead for accelerators 136

List of Tables

2.1 Comparison of first-class support of different CU types 20

3.1 Comparison with the traditional system architecture 27
3.2 Chip area of a PE with DTU 46

8.1 Complexity of software components 137

Chapter 1

Introduction

The ongoing trend to more and more heterogeneous systems forces us to rethink the design of systems [33, 35, 132]. GPUs have entered almost all devices today, starting from mobile phones over desktop machines to high performance computers, FPGAs are deployed because of their performance, energy efficiency, and flexibility [45, 108], and custom accelerators are added to increase the performance and energy efficiency even further [69, 91, 101]. However, current operating systems (OSes) represent non-CPU components such as accelerators as devices, arguably *second-class citizens*. For example, devices have no access to OS services such as file systems or network stacks. This limitation and the increasing importance of accelerators spawned several research projects to improve the integration of accelerators by, for example, enabling access to OS services. However, all existing solutions known to me such as GPUfs [133], GPUnet [75], and PTask [124] for GPUs or BORPH [135] and ReconOS [26] for FPGAs are specific to a single type of accelerator. In this work, I am investigating the consequences of a new system architecture that is based on a uniform hardware interface for all *compute units*, ranging from complex general-purpose cores to fixed-function accelerators, with the goal to represent all compute units as *first-class citizens*.

1.1 Motivation

This section motivates my work in detail by starting with the development towards increasingly heterogeneous systems, followed by the platforms we can expect in the future based on this development. Finally, I explain the shortcomings of current OS designs to properly support these platforms.

1.1.1 Increasing Heterogeneity

From the invention of the first microprocessor at the beginning of the 1970's to about 2006, system designers had a "free lunch" [139]. In every generation, the size of transistors decreased, which allowed to increase the clock frequency without increasing the overall power consumption. The ability to keep the power density constant is known as Dennard scaling [46] and enabled substantial performance improvements without requiring any change to the architecture or software. Dennard scaling broke down at about 2006 due to thermal and power constraints and caused processor designers to transition to multi-core architectures [51, 141]. In contrast to the increasing clock frequency, the increasing core count requires software changes to actually turn the increased performance potential into performance gains for applications. However,

since not all software can be parallelized and also due to diminishing returns, academia and industry already entered the next era: the era of specialization.

Specialization is motivated by the ever increasing performance demands of applications and the efficiency requirements on mobile devices or data centers. Specializing a core or using an accelerator for a specific workload allows to both improve the performance and reduce the energy requirements [28, 68, 69, 91, 118, 153, 154]. Furthermore, it is expected that the increasing *dark silicon* problem will be an additional driver for specialization. Dark silicon is the portion of a circuit that cannot be powered due to thermal or power constraints [51]. The amount of dark silicon increases with every processor generation due to the end of Dennard scaling and current studies expect that 50 % of the chip area for 8 nm technology nodes will be dark [51, 64]. One approach to make better use of dark silicon is to specialize and integrate accelerators for various application domains [141]. For example, it is imaginable to build a chip that consists of multiple *islands* with various different compute units in each island and only a few compute units being active at a time, depending on the application. In summary, specialization is motivated by performance gains, efficiency improvements, and dark silicon and will lead to increasingly heterogeneous systems. The following lists the different types of heterogeneity that are already used in practice or explored in research.

Heterogeneous general-purpose cores One form of heterogeneity is the integration of cores with the same instruction set architecture (ISA), but different performance or energy characteristics into one system. An example is the ARM big.LITTLE platform that consists of two clusters that contain multiple powerful or energy efficient cores, respectively. Future processors might also employ heterogeneous ISAs to further improve the performance and energy efficiency. As has been shown by Venkat et al. [146], mixed-ISA systems, consisting of ARM Thumb, x86-64, and Alpha cores in this case, outperform single-ISA systems by 21 % and improve the energy efficiency by 23 %. Upcoming and more deviating ISAs such as Microsoft's E2 architecture [117] or the Mill architecture [13] might increase the motivation to build mixed-ISA systems even further. For example, the Mill architecture promises to run general-purpose code at comparable performance as out-of-order cores, but with the energy efficiency of digital signal processors (DSPs).

GPUs and FPGAs Most systems nowadays, ranging from mobile devices to high performance computers, possess a graphics processing unit (GPU). GPUs are increasingly used not only to accelerate graphics rendering, but also for general-purpose workloads, because many workloads benefit from the massive parallelism that GPUs offer. For example, GPUs have been used to accelerate machine learning [145], database engines [71], and computational fluid dynamics [92]. Field-programmable gate arrays (FPGAs) are increasingly deployed, because they allow to achieve even higher performance than GPUs [45, 108], while being more flexible than custom accelerators.

Custom accelerators Since custom accelerators are built specifically for a particular application domain or workload, they are less flexible than FPGAs, but offer typically higher performance at lower energy requirements. Due to these advantages and the increasing motivation to specialize, many custom accelerators have been designed by academia [86, 91, 144, 153, 154] and industry [69, 101]. For example, Wu et al. presented a data streaming framework that outperforms software by 7.8× with only 4.3 % of the power [153]. Google showed that its tensor processing units (TPUs) for the

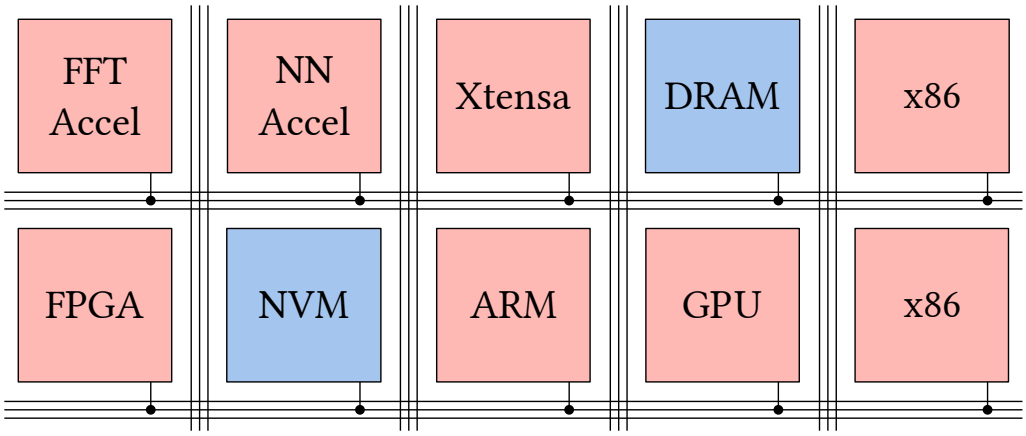


Figure 1.1: Potential future platform

inference phase of neural networks increases the performance per socket 30-fold and the performance per watt 80-fold over a contemporary CPU [69].

Programmable peripheral devices Peripheral devices such as solid state disks (SSDs) and network interface cards (NICs) are typically programmable today. In other words, these devices contain general-purpose cores that are used to provide complex features like single-root input/output virtualization (SR-IOV) or the flash translation layer on SSDs. Due to the programmability these devices become more similar to CPUs, which spawned research projects that explored how they can be used to perform near-data computing [48, 57, 73, 127].

Other forms of heterogeneity Besides the described increasing heterogeneity of compute units, research is being done on heterogeneous memories [87, 102], because different types of memories provide different advantages and disadvantages. For example, SRAM is faster and smaller than DRAM, while non-volatile memory is slower, but persistent. Furthermore, research projects like the Center for Advancing Electronics Dresden (cfAED) explore different materials to build processors such as silicon nanowires [44], carbon nanotubes [100], or organic electronics [94]. Therefore, future processors might not only use different ISAs and accelerators, but also different materials to build these components.

1.1.2 Future Hardware Platforms

Due to the strong motivation outlined in the previous section, I expect very heterogeneous platforms in the future. Figure 1.1 illustrates an exemplary future platform based on the current trend and based on the platforms that are already proposed and built in both academia [40, 41, 147] and industry [80, 123]. Various heterogeneous *compute units* (CUs) are integrated into one system and connected by an interconnect (e.g., a network-on-chip). The CUs may, for example, be fixed-function accelerators for fast Fourier transformation (FFT) or neural networks (NN), reconfigurable logic such as FPGAs, GPUs, specialized Xtensa cores, and complex general-purpose ARM or x86 cores. Furthermore, future platforms might contain heterogeneous memories such as DRAM and non-volatile memory (NVM) that are integrated close to the CUs (e.g., by using 3D-stacked memory) to enable low-latency access to the data. Note that Figure 1.1 is

a schematic representation and does not imply that all CUs and memories are on the same chip. For example, the FPGA might be off-chip, but reachable via interconnect.

1.1.3 Problems of Current OS Designs

Considering future platforms like the one depicted in Figure 1.1 and the design of current OSes reveals a large gap. Current mainstream OSes such as Linux, BSD, and Windows are built for homogeneous and CPU-centric systems. As a consequence, these OSes run a shared kernel on all CPU cores and require that these cores provide different CPU modes (user and kernel mode), a memory management unit (MMU), and other architectural features. In consequence, the OS kernel can only run on homogeneous general-purpose cores, potentially with different performance characteristics. Several research projects such as Barrelfish [35], Popcorn Linux [33], and K2 [88] explore the support of mixed-ISA systems to remove this limitation based on the multikernel concept.

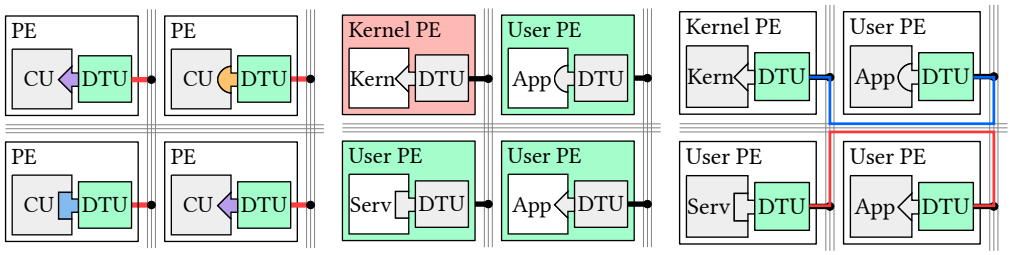
Another property of current OSes, inherent to their CPU-centric design, is that accelerators are treated as devices. IOMMUs can be used to prevent untrusted accelerators (or devices) from causing harm to other system components. However, the mechanisms that allow threads on the CPU to access OS services such as file systems, network stacks, and pipes are not available for accelerators. Additionally, accelerators cannot directly communicate with each other using the OSes inter-process communication mechanisms. In short, devices and therefore accelerators are second-class citizens, because the concepts and mechanisms that are available for threads on the CPUs cannot be used by accelerators. At the same time, accelerators become increasingly similar to CPUs and therefore demand the same functionality. The second-class handling of accelerators spawned several research projects that explore how the integration of one specific accelerator can be improved. For example, GPUfs [133], GPUnet [75], and PTask [124] improve the integration of GPUs, whereas BORPH [135] and ReconOS [26] focus on FPGAs. However, the degree of heterogeneity in future platforms suggests that island solutions are not sufficient.

In summary, current OS designs have difficulties to exploit the full potential of the expected future platforms that consist not only of multiple heterogeneous general-purpose cores, but employ also various accelerators. Accelerators do typically not provide the required architectural features to run an OS kernel, because their advantages over general-purpose cores stem also from their architectural differences. Current OSes can be extended by new concepts to improve the integration of a specific accelerator, as has been shown by approaches such as GPUfs [133] or BORPH [135]. However, the trend to more heterogeneous systems raises the question whether there should be a strong separation between CPUs and accelerators, because both are used to perform computations based on input data and produce output data. Therefore, the question that I am addressing in this thesis is:

> Can we design a system that handles all types of compute units as first-class citizens and in a uniform way?

1.2 Approach

In this work, I am investigating the consequences of a new system architecture based on a uniform hardware interface for all compute units (CUs) to integrate all types of CUs as first-class citizens. The first-class support enables all types of CUs to access OS services,



(a) Uniform interface (b) Remote control (c) Direct communication

Figure 1.2: The key ideas of the new system architecture

allows all CUs to directly communicate with each other, adds context switching support to all types of CUs, and allows accelerators to operate autonomously.

The system architecture I propose and analyze in this work is based on a hardware/OS co-design and builds upon the following key ideas, illustrated in Figure 1.2:

1. adding a uniform interface to all CUs,
2. controlling the applications on these CUs remotely, and
3. using direct communication between applications.

The uniform interface is provided by a new hardware communication component that is integrated next to each CU. The hardware component is called *data transfer unit* (DTU) and the combination of CU and DTU is called *processing element* (PE), as depicted in Figure 1.2. The DTU should on the one hand have minimal requirements on the CU to maximize the freedom for CU designers when specializing their CU. On the other hand, it needs to offer the functionality that 1) keeps the management of very heterogeneous CUs simple and 2) allows to treat all CUs as first-class citizens. Thus, the *internal interface* between CU and DTU (the different shapes in Figure 1.2a) depends on the CU, whereas the *external interface* (red lines) is uniform. As shown in this work, the required functionality for 1) and 2) is message passing and RDMA-like memory access (remote direct memory access).

Controlling applications remotely is also motivated by the CU's heterogeneity. Since not all CUs can be expected to provide the architectural features (e.g., different CPU modes and an MMU) to run an OS kernel, I decided to explore an OS design that executes the kernel on a dedicated *kernel PE* (red in Figure 1.2b) without any applications on this PE. Instead, applications and servers run on the remaining *user PEs* (green), remotely controlled by the kernel. Combined with the DTU as a communication device, this approach allows to integrate arbitrary CUs as first-class citizens.

Since applications now run on bare-metal without an OS kernel on the same PE, I use a different approach for communication between applications and also for communication with the kernel. Both communications are performed via the DTU's message passing and/or RDMA feature. Each application has a communication channel to the kernel (blue line in Figure 1.2c), which is set up by the kernel at application start. This channel can be used by the application to request the creation of other communication channels (red line), which can only be done by the kernel PE. After a channel has been established, applications can communicate directly without involving the kernel PE, which is beneficial for both performance and scalability.

Note that a unified interface at the hardware level, in contrast to a software-based abstraction layer, enables an easy interaction between *all* CUs. For example, even

non-programmable CUs can use the DTU to communicate with all other CUs in the system. In contrast, using different ways to communicate at the hardware level with a software-based abstraction layer on top requires accelerators that cannot use the abstraction layer to implement *all* these communication types in hardware.

1.3 Contributions

The main contribution of my work is the description, prototypical implementation, and evaluation of the new system architecture, which is based on a uniform hardware interface for all CUs. The following lists the contributions in more detail.

Uniform interface for all CUs The uniform interface in my system architecture is the data transfer unit (DTU) next to each compute unit (CU), which has two main goals. First, it acts as an abstraction layer for the CU's heterogeneity to unify the management of the hardware by the OS. Second, the uniform interface allows to integrate all types of CUs as first-class citizens by providing message passing and RDMA-like memory access. I show in Chapter 3 how the DTU can be designed to meet these goals and at the same time support arbitrary types of CUs. Supporting arbitrary CU types mandates to minimize the architectural requirements. For example, the message-passing feature of the DTU should not require virtual-memory support to keep a communication private between two communication partners or require exception support to handle corner cases in software. Additionally, I reuse existing hardware components such as general-purpose cores, accelerators, caches, memories, interconnects, etc. without mandating modifications to ensure that my approach is practical. In other words, I only add a new hardware component and change the way existing components are connected, but do not require, for example, to add instructions to existing cores.

Spatial isolation between CUs In a time in which almost every computing device gets connected to the Internet, security becomes increasingly important [62, 67]. At the same time, untrusted code and untrusted intellectual property blocks (e.g., accelerators) need to be supported, which requires systems to prevent unauthorized data access and manipulation of data. Even with only trusted CUs, it cannot be expected that all CUs in future platforms provide the architectural features for spatial isolation. For these reasons, I describe in Chapter 3 how the DTU in my system architecture adds an additional layer of protection that spatially isolates different CUs from each other.

Access to OS services for all CU types Accelerators work on data, just like threads on the CPU. The data will typically be stored in a file or is received from the network or is produced by a program. Thus, access to OS services enables the accelerator to load data from these sources and store data to these sinks. I describe in Chapter 4 how OS services can be provided based on the DTU and its communication capabilities to enable the access to OS services by arbitrary CUs.

Virtual-memory support Virtual memory is important to spatially isolate applications and use the system's memory efficiently. Some types of CUs such as general-purpose cores typically employ a memory management unit (MMU) to support virtual memory, whereas other CUs such as accelerators lack this support. I show in Chapter 5 how a DTU-based system architecture can reuse the MMU in existing general-purpose cores and also how virtual-memory support can be provided for accelerators.

Autonomous accelerators As mentioned above, accelerators offer substantial energy savings over general-purpose cores. Unfortunately, accelerators need to be assisted today due to the second-class handling of accelerators [132]. For example, the TPUs described in Google's paper burden their controlling CPU with 11 % to 76 % load just to operate the TPU [69]. For that reason, the system cannot fully benefit from these savings, if accelerators need to be continuously assisted by the typically power-hungry CPU. Additionally, if the CPU does not need to assist the accelerator, the CPU can perform other work in the meantime and does not become the bottleneck with an increasing number of accelerators. I show in Chapter 6 how accelerators can access OS services and thereby work autonomously and demonstrate how fine-grained interruptibility of accelerators can be combined with the autonomous operation. Additionally, I demonstrate how accelerators can communicate with each other without assistance by the CPU, based on the DTU's communication capabilities.

Context switching With my system architecture, communication between applications is primarily based on the DTU and bypasses the kernel. This leads to performance improvements as will be shown in the evaluation and has been shown by other approaches with kernel-bypassing communication [96] as well. However, context switching support is difficult if the kernel is not involved in communications, but required to use CUs efficiently. I explain in Chapter 7 how kernel-bypassing communication can be combined with context switching and also how context switching can be supported on all types of CUs, including accelerators.

Prototype implementation and evaluation To determine the feasibility of my system architecture, I implemented a prototype of both the DTU and the operating system that manages the hardware based on the DTU. I built the DTU in software for gem5 [38], a simulation platform for computer-architecture research. Additionally, the DTU has been implemented in silicon based on a cooperation with Benedikt Nöthen from the Vodafone Chair Mobile Communication Systems. The silicon implementation is based on the Tomahawk platform [28, 58, 106], which is a heterogeneous multiprocessor system-on-chip (MPSoC) designed at TU Dresden and primarily intended for mobile communication applications. The OS prototype is called M^3 for **m**icrokernel-based syste**m** for heterogeneous **m**anycores. In Chapter 8, I evaluate the performance of the new system architecture, its scalability with the number of PEs, the efficiency when sharing PEs, and the benefits if accelerators are handled as first-class citizens. The DTU model for gem5[1], the OS prototype[2], and the benchmark infrastructure[3] are available as open source to enable follow-up works and reproductions of the results.

Due to the overall complexity of the system architecture, consisting of several components in both hardware and software, I describe the system architecture by starting with a simplified platform and by extending it in multiple steps. After comparing my approach to related works in Chapter 2, Chapter 3 and Chapter 4 introduce the basic architecture and the OS service design based on Tomahawk-like platforms that consist of multiple processing elements (PEs), each containing a simple general-purpose core and

[1] `https://github.com/TUD-OS/gem5-dtu`
[2] `https://github.com/TUD-OS/M3`
[3] `https://github.com/TUD-OS/M3-bench`

a dedicated scratchpad memory for code and data. Additionally, applications are pinned on dedicated PEs and run on these PEs until their completion. These simplifications allow me to focus first on the most important properties of my system architecture without being distracted by details and complexity. Chapter 5 extends the simplified platform by caches and virtual-memory support to enable arbitrarily complex applications. Chapter 6 further extends the platform by describing the integration of accelerators with focus on their autonomous operation. Finally, Chapter 7 introduces context switching that enables applications to share PEs.

To strengthen the understanding of the reader, all chapters evaluate the basic properties of the concepts introduced in the chapter using micro-benchmarks and by focusing on a single aspect of the system. Chapter 8 evaluates the described system architecture as a whole and in more realistic settings. The basic system architecture and OS-service design described in Chapter 3 and Chapter 4, respectively, are based on the publication at ASPLOS'16 [31]. The autonomous execution of accelerators (Chapter 6) in combination with context switching (Chapter 7) has been published at USENIX ATC'19 [30].

Chapter 2

Related Work

This chapter discusses the differences and relations to other works that are similar to my system architecture regarding the operating system, called M^3, and the DTU. The related work on more specific aspects of my system architecture will be discussed in later chapters.

2.1 Operating Systems

The primary goal of this thesis is to integrate all types of compute units (CUs) as first-class citizens. The first-class integration of all CUs consists of the ability to access OS services, direct communication between CUs, context switching support, and spatial isolation for security and robustness reasons. When exploring related OS work, we find different approaches to shrink the gap between CPUs and accelerators, resulting in different degrees of first-class support for accelerators. I summarize the first-class support for various different CU types in Table 2.1 and compare my approach to others at the end of this section. Note that "Core w/o OS" denotes a general-purpose core in Table 2.1, potentially with instruction extensions to accelerate a specific workload, but without the architectural features to run an OS kernel (e.g., without different CPU modes). "HW accel" stands for a fixed-function hardware accelerator.

Traditional OSes Traditional OSes such as Linux, Windows, or L4 [85] have a CPU-centric OS design and therefore handle all accelerators as devices, that is, second-class citizens. Additionally, these OSes are built for homogeneous general-purpose cores and therefore run a shared kernel on all CPU cores and require the CPU's architectural features to isolate the kernel from applications. For example, these OSes require multiple CPU modes (user and kernel mode) and a memory management unit (MMU) to prevent malicious applications from harming other applications or the OS kernel. Hence, current OSes have difficulties to support heterogeneous instruction set architectures (ISAs), because each ISA requires a different kernel binary, implements the aforementioned architectural features in different ways, and has potentially a different word size, a different endianess, and so on. Additionally, supporting accelerators as first-class citizens is difficult, because accelerators typically do not have the architectural features to run an OS kernel.

Multikernel approaches Baumann et al. [35] introduced the multikernel model with the idea to divert from the shared-kernel OS design and treat the underlying hardware as a distributed system. The current implementation, called Barrelfish, runs an independent

	Hom. GPC	Het. ISA	Core w/o OS	GPU	FPGA	HW accel
Linux, Windows, L4	✓	x	x	x	x	x
Barrelfish	✓	✓	x	x	x	x
Popcorn Linux, K2	✓	(✓)	x	x	x	x
Helios	✓	✓	(✓)	x	x	x
Gdev, GPUfs, GPUnet	✓	x	x	✓	x	x
ReconOS, BORPH, OS4RS	✓	x	x	x	✓	(✓)
M^3	✓	(✓)	✓	(✓)	(✓)	✓

Table 2.1: First-class support of different CU types, ranging from homogeneous general-purpose cores to fixed-function hardware accelerators for different OS designs and OS extensions. Check marks denote that first-class support has been demonstrated, whereas check marks in parenthesis indicate that the current design should be able to support the CU type, but the support has not been demonstrated so far. Crosses denote that the current design does not support the CU type and therefore a different design or extension is required.

kernel on every core and replicates the global OS state across all cores as needed. The kernel is responsible to enforce protection by isolating applications via address spaces and to protect capabilities from applications. On top of the kernel, Barrelfish uses a so called *monitor* that is responsible for inter-core coordination such as the synchronization of the replicated OS state. This structure allows Barrelfish to support non-coherent and mixed-ISA systems comparatively easily, because all cores are handled by independent kernels. For example, Cosh [36] has shown how I/O abstractions such as files and pipes can be provided using Barrelfish as the foundation, if the hardware platform consists of multiple coherence domains. Barrelfish/DC [155] showed how the distributed nature of this approach can be taken one step further by decoupling kernels and cores. This allows, for example, to hot plug CPUs and live-update kernels. Barrelfish's support of mixed-ISA systems has been demonstrated for heterogeneous ARM [55] and x86 platforms [24, 99]. How Barrelfish can support CUs as first-class citizens that do not have the architectural features to run an OS kernel such as GPUs, FPGAs, or hardware accelerators has not been covered in publications.

Similar to Barrelfish, Popcorn Linux [33] explores how non-coherent or mixed-ISA systems can be supported by building upon the multikernel model. In contrast to Barrelfish's small-kernel-per-core approach to enforce protection, Popcorn Linux extends the Linux kernel to support multiple kernel instances. These instances synchronize their state via message passing and therefore do not rely on global cache coherency or shared memory. For example, each kernel instance can run in a separate coherence domain. To hide the complexity of the underlying platform from applications, Popcorn Linux provides distributed shared memory transparently for applications. As for Barrelfish, the support of mixed-ISA systems has not yet been shown, to the best of my knowledge. K2 [88] also uses multiple Linux instances to support multiple coherence domains in mobile platforms. The authors show that their approach enables substantial energy savings by combining high-performance cores with low-power cores and retains the programmability of existing OSes. In contrast to Popcorn Linux, K2 uses distributed shared memory for both the kernels and applications. Since both Popcorn Linux and K2 extend Linux, accelerators are handled as devices, just as in Linux.

Reducing the architectural requirements NIX [32], based on Plan 9 [115], does not target heterogeneous systems, but relaxes the requirement on executing a kernel on every core by introducing *application cores*. In contrast to still existing *time sharing cores*, application cores do not execute a kernel to prevent OS noise. OS noise is the interference with the application caused by the OS (e.g., by interrupt handling). Application cores can still access OS services by communicating via message passing with a time sharing core, similar to FlexSC [136]. Although NIX can support cores that are not able to execute a kernel, the communication in NIX is based on shared memory and isolation requires MMUs on all cores. Helios [105], a derivative from Singularity [52], reduces the requirements one step further by using software-based isolation instead of address space protection. This approach requires neither an MMU nor different CPU modes. According to the authors, it only requires a timer device, an interrupt controller, exceptions, and at least 32 MiB of memory. Based on *satellite kernels*, Helios has shown that heterogeneous ISAs can be supported. However, Helios restricts applications to managed programming languages such as C#.

A single accelerator type as first-class citizen Several research projects explore the extension of existing OSes for a single accelerator type. PTask [124] proposed a dataflow-based programming model for GPUs that simplifies the offloading of compute-intensive tasks to GPUs. Since the tasks in the dataflow graphs are known to the OS, the OS can provide system-wide guarantees such as fairness or performance isolation. Gdev [72] makes GPUs first-class citizens by providing applications access to the GPU in terms of *virtual GPUs* that are scheduled by the OS and that can establish shared memory via a POSIX-like API. Additionally, Gdev allows the OS itself to use the GPU as well. For example, the OS can use the GPU to accelerate the encryption and decryption operations of an encrypted file system. GPUfs [133] showed how a POSIX-like file system API can be designed for GPUs that enables GPU programs to access the host file system. GPUfs uses a buffer cache in GPU memory to handle reads and writes locally and uses remote procedure calls to the CPU on buffer-cache misses. Due to the parallel nature of GPUs, GPU programs perform file system calls at warp-granularity[1]. Similarly, GPUnet [75] presented a socket API that allows GPU programs to establish connections and exchange packets over the network. To avoid the complexity of packet processing on the GPU, GPUnet uses RDMA-capable NICs that are responsible for all low-level packet processing and deliver the payload directly to the GPU.

ReconOS [26] provides a unified multithreaded programming model for both software threads and hardware threads on FPGAs. ReconOS uses a *delegate thread* in software to execute OS calls on behalf of hardware threads. BORPH [135] goes one step further and represents an FPGA design as a Linux process by extending Linux accordingly. These *hardware processes* have access to the OS based on message passing between the CPU and the FPGA. Similarly to ReconOS, BORPH uses a dedicated software thread to support file I/O by hardware processes. OS4RS [107] represents FPGA designs as Linux processes as well and uses a hardware abstraction layer (HAL) to allow communication between software and hardware processes in a transparent way. The HAL bridges the gap between the different ways to communicate among software processes and hardware processes. In contrast to ReconOS, BORPH, and OS4RS, M^3 does not employ a dedicated software thread or a HAL to bridge the gap between FPGA designs and existing software interfaces, but introduces new interfaces that are suitable for both software and hardware and enables accelerators to operate autonomously.

[1] A warp is a group of threads on Nvidia GPUs that executes the same code in lockstep.

OS support for peripheral devices Many modern peripherals such as NICs and solid state disks (SSDs) are programmable, which spawned several research projects on *near-data computing* [48, 57, 73, 127]. For example, Willow [127] introduced *SSD Apps* that run on the SSD to perform near-data computing directly on the SSD or provide a block device interface for a conventional file system. An application on the CPU can communicate with an SSD App via remote procedure calls. FlexNIC [73] allows to offload packet processing tasks such as multiplexing and filtering to the NIC based on a special programming language. Omnix [132] introduced the idea to take this approach one step further by designing an OS for *omni-programmable systems* containing NICs for near-network computing, SSDs for near-data computing, and GPUs. Omnix extends OS abstractions such as tasks and I/O to GPUs and near-X accelerators by taking advantage of hardware virtualization support (SR-IOV) in near-X accelerators and using IOMMUs for protection. LegoOS [128] introduced the *splitkernel* that splits the OS into multiple *monitors*, similarly to a multikernel. Each monitor is responsible for a specific resource and runs on the corresponding hardware component. For example, the memory monitor could run on the memory controller and manage the physical memory, whereas the storage monitor manages and runs on the storage device.

Comparison The key difference between my work and the other approaches is that my work does not try to get the best out of *existing* hardware, but uses a hardware/OS co-design to investigate how the hardware/software interface should be designed for upcoming heterogeneous systems. My system architecture uses a uniform hardware interface for all types of CUs, the data transfer unit (DTU), and equips the DTU with communication endpoints that can be controlled externally. This approach allows the OS to establish communication channels to OS servers, which provides all CUs access to file systems or network stacks. Additionally, CUs are spatially isolated by controlling their communication channels to other CUs. In contrast, the other discussed approaches, except Helios, use the CU's architectural features (different CPU modes and the MMU) to isolate processes on general-purpose cores and typically rely on IOMMUs to control the access to physical memory for devices and accelerators. With my approach, the CUs do not need specific architectural features, which enables the support of *arbitrary* CUs as first-class citizens, as shown in Table 2.1. As a proof of concept, I show how the CU types on both ends of the spectrum can be supported: general-purpose cores on the one end with all architectural features to run an OS kernel and fixed-function accelerators on the other end that have none of these features and do not even execute software.

2.2 Hardware Components

My system architecture is based on the DTU as a uniform interface for all CUs. The DTU is used for cross-CU communication and spatial isolation between CUs. There are several hardware components that share similarities with the DTU, which can be grouped into three categories: components for memory translation and protection, components for message passing, and technologies for RDMA-like memory access.

Memory translation and protection At first, memory management units (MMUs) and memory protection units (MPUs) are related to the DTU. MMUs and MPUs are typically tightly integrated with the core architecture and used for memory translation and/or protection, controlled by the OS kernel on the same core. More recently, IOMMUs have been introduced to add translation and protection for I/O devices. Several

approaches also use IOMMUs to control the memory accesses of accelerators such as Omnix [132], the heterogeneous system architecture (HSA) [7, 123], and the coherent accelerator processor interface (CAPI) [15, 138].

Due to the increasing popularity of network-on-chips (NoCs) and the use of MMU-less CUs in such a NoC, various proposals [42, 53, 116] have been made on how to control the CUs' access to the NoC. For example, Fiorin et al. [53] introduced the data protection unit (DPU), which is added next to each CU, similar to the DTU. The DPU controls memory accesses that are sent to the NoC based on a permission table in the DPU, which allows to define the access to physical memory in a per-CU and page-granular fashion. Porquet et al. [116] proposed the NoC-MPU, which extends on this idea by implementing the permission table as a hierarchical, two-level table stored in memory and by caching recent translations in the NoC-MPU.

The main difference between these hardware components and the DTU is that MMUs, MPUs, IOMMUs, DPUs, and NoC-MPUs are built for memory translation and/or protection. The purpose of the DTU is to add a uniform interface to all CUs and provide the required features to handle all CUs as first-class citizens. In particular, this includes the ability to perform secure and efficient message passing between CUs.

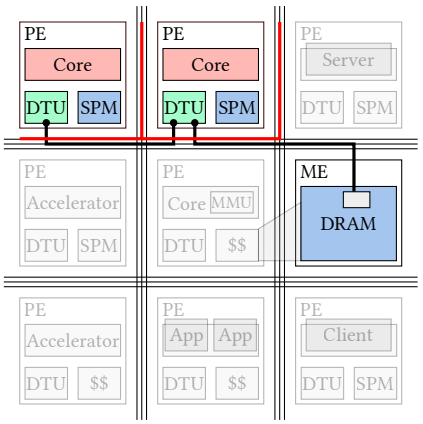
Message passing Besides approaches for memory accesses, various work [27, 63, 82, 95, 113, 151] has been done on (user-level) message passing. For example, Alpert et al. [27] describe protected message passing in userland on the Shrimp multi-computer. Protection is thereby based on the processor's virtual-memory support. Similarly, the MAGIC component in the FLASH multiprocessor [63, 82], is designed as a processor that executes software to allow the implementation of a variety of protocols like cache coherency. MAGIC requires an OS kernel on every core for secure user-level message passing. The Tilera manycore architectures use processor extensions to provide tile-to-tile communication based on register accesses [151]. For example, DLibOS [96] takes advantage of this feature by placing the network stack and applications on separate tiles and using the low-latency and kernel-bypassing communication to significantly improve network latency and throughput. In summary, all approaches rely on processor-specific features like virtual memory and/or an OS kernel on the sending/receiving CU for special cases. Since my goal is to integrate arbitrary CUs into the system, the DTU has to work independent of the CU. In other words, the DTU needs to execute all operations without involving the CU or an OS kernel.

RDMA and P2P DMA The DTU supports the access of other CUs' memories or memory modules such as DRAM in a way that is akin to remote direct memory access (RDMA) and peer-to-peer DMA (P2P DMA). RDMA is a feature of network interface controllers (NICs) such as InfiniBand controllers [8] or Ethernet controllers [16], which allows to access the memory of a remote machine without involving the OS on either side. Memory access via the DTU is similar, but performed within one machine – for example between an accelerator and a CPU core. P2P DMA enables peripheral devices to communicate directly via PCI express, bypassing main memory and the CPU. For example, recent GPUs support P2P DMA via GPUDirectRDMA [3] from Nvidia or DirectGMA [19] from AMD, which allows to transfer network packets directly from the NIC to the GPU's memory [75] or to transfer data directly from the SSD to the GPU [37]. Similarly, Si et al. [131] have shown that Xeon Phi co-processors can directly communicate with co-processors on other machines via InfiniBand without involving the host CPU. For that reason, the DTU's memory access feature is conceptually similar

to P2P DMA, but not only used for peripheral devices. Instead, the feature is used for all types of CUs and memories in my system architecture.

Chapter 3

Isolation and Communication



This chapter starts the description of the proposed system architecture by focusing on spatial isolation and communication between compute units (CUs). As explained in the introduction, the first chapters are based on a simple platform that integrates memory elements (MEs) containing DRAM and multiple processing elements (PEs) into a network-on-chip (NoC). Each PE consists of a simple general-purpose core as the CU, a scratchpad memory (SPM) for code and data, and a data transfer unit (DTU). The operating system (OS) runs each program on a dedicated PE without interruption from its start until its termination. Despite these simplifications, this chapter already considers the requirements of, for example, accelerators for the design of the DTU and the OS. The following chapters will gradually extend the system, as indicated by the picture on the top right at the beginning of each chapter.

3.1 Motivation

Integrating CUs that do not have the architectural features to run an OS kernel (e.g., a core without different CPU modes) as first-class citizens requires means to communicate with the CU that runs an OS kernel or an OS server. Furthermore, all CUs should be able to directly communicate with each other without involving the OS on the critical path. To keep the system manageable despite the growing heterogeneity, as outlined in Chapter 1, I decided to introduce a uniform communication interface for all CUs, called data transfer unit (DTU). The important point to support *all* types of CUs is to design the DTU in a CU-independent way, as I will describe in this chapter.

In a time in which almost every computing device gets connected to the Internet, security becomes increasingly important [62, 67]. Therefore, the different components of a system should be spatially isolated[1] from each other to prevent that components can access or manipulate foreign code or data. For example, without spatial isolation a malicious program could read confidential information from another program (e.g., cryptographic keys) or manipulate the data of another program to cause crashes or wrong results. Some CUs such as many general-purpose cores employ a memory management unit (MMU) to achieve spatial isolation between different software components on this CU by executing them in different address spaces. However, other CUs such as simple accelerators do not contain an MMU [116]. Furthermore, integrating untrusted third-party accelerators into the system demands an enforcement of spatial isolation by

[1] Other forms of isolation such as temporal or performance isolation are not considered in this work.

trusted components [109]. For these reasons, I use the DTU as an additional layer of protection to enforce spatial isolation between CUs.

3.2 Threat Model

For the isolation concept described in this chapter, I assume that an attacker wants to compromise the confidentiality, integrity, or availability of the system or parts of the system. I assume that the attacker controls the CU, including its memory such as SPM, and uses software or hardware to perform these attacks. However, I assume that the DTU, the network-on-chip, and the kernel PE are trustworthy. Attacks based on physical access to the system are not considered in this work. More specifically:

Confidentiality The goal of confidentiality is that every CU (hardware and software) can only access its own data and data to which access was granted. For example, it should not be possible to read confidential information such as cryptographic keys from other CUs.

Integrity Similarly, integrity has the goal that every CU can only modify its own data and data to which write access was granted. For example, it should not be possible to manipulate other applications' data to cause crashes or wrong results.

Availability Finally, availability has the goal to prevent denial-of-service attacks on other CUs. For example, it should not be possible for a client to prevent other clients from getting serviced by flooding the server with messages.

Achieving these goals requires not only to control the access to memory, but also to control the exchange of messages. Otherwise, an illegitimate message to a server could allow the sender to access confidential information or manipulate data.

3.3 Overview and Comparison

Before diving into the details of the DTU and the operating system M^3, this section provides an overview of the system architecture and compares it to the traditional system architecture. The key idea of my system architecture, depicted in Figure 3.1, is to introduce the DTU as a uniform interface for all CUs. The DTU can be used to communicate between different processing elements (PEs), each consisting of a CU, a DTU, and a local memory (SPM in this chapter). The *kernel PE* runs the M^3 kernel without any application on this PE. In contrast, M^3 runs the applications and servers on the remaining *user PEs*. Each application/server has a communication channel to the kernel (blue lines in Figure 3.1), which is set up by the kernel at application start. This channel is used for system calls to, among others, create new communication channels to other applications or memories (red lines), which can only be done by the kernel PE. After a communication channel has been established, the communication is performed directly without involving the kernel PE.

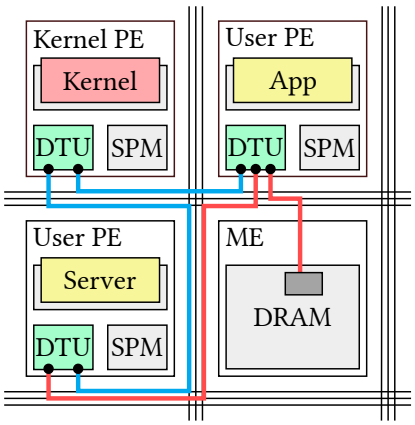


Figure 3.1: System overview

	Traditional	**M³**
Privilege levels	User/kernel mode	User/kernel PEs
Isolation	Memory management unit	DTU
Communication	System call	DTU
Role of the kernel	Resource access, communication	Channel creation
TCB	Kernel and all CUs	Kernel PE and DTUs

Table 3.1: Comparison with the traditional system architecture

Due to the large differences between this system architecture and traditional system architectures, the following compares their most important properties. A short overview is given in Table 3.1. Note that the purpose of the comparison is *not* the criticism of traditional system architectures. Instead, the comparison is intended to improve the reader's understanding of the proposed system architecture.

3.3.1 Privilege Levels

Most general-purpose cores today have at least two CPU modes. For example, the x86 architecture supports four CPU modes, called *protection rings*. Ring 0 is intended for the OS kernel, ring 1 and 2 for OS services, and ring 3 for user applications. The different rings allow the OS kernel to isolate itself from OS services and applications. For example, parts of the virtual address space can be configured as only accessible from ring 0 and the security-critical instructions are only allowed in ring 0. In other words, unprivileged software is distinguished from privileged software by the CPU mode.

M³'s system architecture also distinguishes different privilege levels. However, as the kernel runs on a dedicated PE, privilege is determined on a PE-basis, leading to kernel PEs and user PEs. Whether a PE is privileged or not is defined by its DTU. A privileged DTU can freely establish communication channels to other DTUs and between other DTUs. Like in traditional systems, which start in the privileged CPU mode, all DTUs are initially privileged. The kernel will downgrade the privilege level of the user PEs' DTUs during boot.

3.3.2 Isolation

The goal of spatial isolation is to keep errors and exploits of vulnerabilities local to one component. Traditional systems achieve spatial isolation by multiple CPU modes and virtual address spaces provided by memory management units (MMUs). In other words, isolation is based on the CU's architectural features, hence called *CU-based isolation* in the following.

In the targeted platforms with very heterogeneous CUs, including simple accelerators, it cannot be assumed that all CUs provide these features. For that reason, I introduce another level of isolation, called *NoC-level isolation*, which restricts the interactions between PEs. Since all interaction is based on the DTU, the DTU enforces isolation and thereby achieves confidentiality and integrity between PEs. For example, NoC-level isolation prevents that an unauthorized CU can send a message to a server and thereby gain access to confidential information or manipulate data. As mentioned in the previous section, only the kernel PE can establish communication channels between DTUs. Thus, user PEs are isolated by default and the kernel can selectively allow communication. In other words, access to PE-external resources is restricted, whereas PE-internal resources

can freely be used by the software running on that PE (e.g., privileged instructions). For that reason, NoC-level isolation can also be combined with CU-based isolation. For example, if a CU provides the required features, it can run a full-fledged operating system such as L4 or Linux.

3.3.3 Communication

The communication between applications in traditional systems, called inter-process communication (IPC), comes in various flavors. For the comparison, I focus on message passing as for example provided by UNIX-like OSes or used in microkernel-based OSes such as L4. In both cases, all communication involves the OS kernel. That is, for each message, the application performs a system call and the kernel is responsible for checking the application's permissions and delivering the message to the recipient.

In contrast to traditional systems, applications on M^3 communicate directly with each other via the DTU, bypassing the kernel. In other words, after the communication channel has been set up by the kernel, the kernel is no longer involved in the communication. The DTU provides so-called *endpoints* (EPs) for communication channels. Each EP can be configured to a *send EP*, *receive EP*, or *memory EP*. Message passing is connection oriented and point-to-point between a send EP and a receive EP. The kernel creates a message-passing channel by configuring an EP at the sender's DTU as a send EP that is connected to a receive EP at the recipient's DTU. Each send EP is connected to exactly one receive EP, whereas each receive EP can receive messages from multiple send EPs. Such a communication channel allows the sender to send messages to the recipient, whereas the recipient can only reply to those messages (once). This communication model is inspired by L4, where remote procedure calls are the dominant form of communication.

Messages consist of a header and a payload. The DTU loads the payload as specified by the sending CU from memory and prepends a header containing meta information like the length of the payload. Afterwards, the DTU sends the message over the NoC to the receiving DTU, which will store it into the *receive buffer* of the receive EP. Receiver buffers are allocated by the kernel before the configuration of receive EPs. Traditional kernels typically use a queue to handle multiple outstanding messages for a single recipient. M^3 uses no queue at sender side, but queues multiple messages in the receiver buffer at recipient side. To use a receive buffer securely for multiple senders without involving the OS kernel in the communication, M^3 uses a credit system. The credit system informs the sender how many messages can still be sent via a send EP and also prevents denial-of-service attacks by enforcing that limit. The credit system ensures availability of the kernel and also servers by enabling them to control the frequency and quantity of messages their clients can send.

The DTU's send operation is synchronous in the sense that the send is "in progress" until the message has been successfully stored in the receive buffer or the sending failed with an error (e.g., caused by insufficient credits). In other words, after the send operation finished successfully, the sender knows that the message has been delivered to the recipient. However, the recipient might not have processed it yet. On L4, the call operation (a combination of send and reply) is synchronous as well, because the kernel blocks the sender until the recipient has processed the message and sent the reply. The call operation is not synchronous on M^3, because the M^3 kernel is not involved in the communication. For that reason, the sender is free to perform other work between the send and the reply operation. However, for simplicity most APIs in M^3 perform calls synchronously by waiting for the reply after the send operation.

In contrast to a send EP, which links to a receive EP, a memory EP has no EP as its counterpart, but refers to a contiguous and byte-granular piece of memory in a specific PE or memory element (ME). If it refers to a PE, it is comparable to RDMA, because the CU of that PE is not involved in the access.

3.3.4 Role of the Kernel

The role of the kernel depends on the OS design. Monolithic kernels are responsible for (almost) all resources and services of the system, such as process management, memory management, file systems, network stacks, and drivers. In microkernel-based OSes, the kernel has only the required abstractions and mechanisms to let userland components provide the actual functionality of the OS. I decided to design M³ as a microkernel-based OS, because of the advantages in security and reliability [61, 62, 77]. For that reason, M³ shares more similarities with microkernels such as L4 or MINIX [65]. Most importantly, all kernels are responsible for the security-critical actions in the system. In microkernel-based system, security-critical decisions are typically done in userland, and the kernel is responsible for enforcing these decisions. The enforcement is sometimes delegated to hardware. For example in traditional systems, page tables are manipulated by the kernel, but enforced by hardware. In my system architecture, DTU endpoints are configured by the kernel, but enforced by the DTU. In both L4 and M³, capabilities are used to manage the permissions in the system, which allows to delegate the decision making to user components. Due to these similarities, I also use the term "kernel" for the entity that runs on the kernel PE.

Besides these similarities, there are significant differences. As mentioned in the previous section, the L4 kernel is involved in every communication. In monolithic systems, almost all resources are accessed via system calls. In other words, whereas traditional kernels are typically involved in the resource accesses and the communication, the M³ kernel is only required for the creation of communication channels, providing access to resources. Additionally, the M³ kernel is not entered via interrupt or exception, but is continuously running[2] and waiting for system calls in form of messages.

3.3.5 Trusted Computing Base

A common concept in security research is the *trusted computing base* (TCB) [84], which consists of all components (hardware, firmware, and software) the security of the system depends on. Since the CU-based isolation is achieved with the features of the CU, the TCB always contains the CU and often the entire hardware platform. With NoC-level isolation, not all CUs are part of the TCB, because isolation is based on the DTU.

Let us consider an example to better understand the differences. Suppose we have three CUs (CU_1 to CU_3) and two applications, running on CU_1 and CU_2. To meet our security goals, we need to enforce that these applications can communicate with each other, but *not* with other applications in the system. The traditional system architecture would run a kernel on each CU (e.g., the multikernel Barrelfish [35]), responsible for the enforcement of our restriction. This implies that all CUs and the kernels are in the TCB. With NoC-level isolation, the kernel runs on CU_3 and establishes a communication channel between CU_1 and CU_2, usable via the DTU. Thus, the TCB contains the kernel and CU_3, all DTUs and the NoC. CU_1 and CU_2 are not in the TCB[3], because they can neither change the communication channel nor create new channels. The NoC is in

[2] The CU is put into a low power mode while waiting for a message.

[3] CUs still need to be electrically sound (e.g., not contain short circuits).

the TCB as well, because it transfers security-critical information between PEs and is thus able to compromise security. Note also that, NoC-level isolation still requires the application running on CU_1 to trust CU_1 to deliver its service. However, an application on a different CU does not need to trust CU_1.

3.4 Data Transfer Unit

In this section, I will describe the DTU in detail, focusing on isolation and communication. Later chapters will extend on that by adding support for virtual memory and context switching. One of the main points for the design decisions explained in the following is that the DTU needs to work independently of the CU. For example, the DTU cannot raise an exception in the CU to handle corner cases, because the CU might be a simple accelerator that does not even execute software.

3.4.1 Integration

The DTU is integrated next to each CU with a CU-dependent internal interface. In particular, the internal interface depends on the way memory is accessed. This chapter starts with a simple general-purpose core that executes a single stream of instructions and has a dedicated scratchpad memory (SPM) that is accessed without address translation and without caches. The following chapters will extend this to support fixed-function accelerators as CUs on the one end of the spectrum and CUs with complex cache hierarchies and virtual memory on the other end. I will also discuss later how CUs with multiple cores or hardware multithreading can be supported.

Figure 3.2 depicts the type of PE we consider in this chapter in more detail. The core, SPM, and DTU are connected via a crossbar (thick line). The DTU is accessed via memory mapped I/O (MMIO) and handles the communication with the NoC. The crossbar receives all memory requests from the core and sends it to the DTU or SPM, depending on the destination address. Similarly, the DTU can access the SPM, for example, to load a message it should send or to store a received message. The core, SPM, and NoC are reused without any modification.

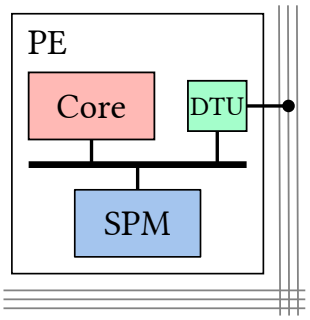


Figure 3.2: PE internals

3.4.2 Endpoints

The DTU has a number of *endpoints* (EPs) to establish communication channels, which can be configured to three different EP types: *send EPs* and *receive EPs* are used for message passing, whereas *memory EPs* are used for RDMA-like memory access. Each EP is represented by a DTU register and can be configured (at runtime) to one of these EP types. Depending on the type, the EP register holds information such as the receiver of sent messages, the receive buffer address and size, the remote memory address and access permissions, etc. In other words, the EPs define the access to PE-external resources. EPs are therefore similar to hardware capabilities, as for example used in CHERI [152]. In the current implementation, each EP register is 192 bit large, starting with 3 bit for the EP type, followed by 189 bits whose meaning depends on the EP type.

Memory Endpoint

A memory EP grants access to a byte-granular and contiguous region in PE-external memory. A memory EP stores the id of the element (processing element or memory element) in which the region resides, the base address of the region, the region size in bytes, and the access permissions (read or write). Due to the similarity to RDMA, read and write requests via memory EP are also called *RDMA reads* and *RDMA writes*.

Send Endpoint

A send EP allows to send messages to exactly one receive EP and holds the PE number of the recipient, the receive EP number, and the maximum message size supported by the receive EP. Additionally, it contains a so-called *label*. The idea was originally introduced by KeyKOS [60] with the name *numeric tag*, which is a value chosen by the recipient when the communication channel is created and unforgeable by the sender to securely identify the sender. Typically, the recipient sets it to the address of the object that corresponds to the sender, so that no additional lookup in a hash table or a similar data structure is necessary to find the object needed for the requested operation. Finally, the send EP contains the number of *credits*, which is explained in more detail in Section 3.4.6.

Receive Endpoint

A receive EP can receive messages from multiple send EPs and send replies to the received messages, as explained in more detail in Section 3.4.5. The messages are stored in the SPM in a so-called *receive buffer*. A receive buffer is organized in fixed-size slots, whose size is chosen during the creation of the receive EP. The receive buffer is allocated by the M^3 kernel to ensure that it resides in pinned memory. In the platform considered in this chapter, receive buffers reside in untranslated SPM and are therefore pinned by design. The motivation for pinned receive buffers is explained in Section 5.9.5 after the platform has been extended by memory support. The receive EP holds the address of the receive buffer, the number of slots and the size of each slot. Additionally, it stores a read and write position and a bitmap for occupied slots and unread slots to receive message and fetch messages, which will be explained in Section 3.4.4.

3.4.3 Commands

The DTU supports two types of commands: *internal commands* and *external commands*. Internal commands allow the CU to use the EPs, that is, the established communication channels. External commands cannot by used by the CU. Instead, RDMA write requests to the external command registers are required. As RDMA writes demand a corresponding memory EP, the kernel decides who is allowed to use external commands at which PEs. Currently, only the kernel itself uses them.

Internal Commands

The CU can use internal commands to trigger an action in the DTU. To keep the DTU simple, the current implementation supports only one command at a time. The DTU offers the following *command registers*: `COMMAND`, `EPID`, `DATA_ADDR`, `DATA_SIZE`, `OFFSET`, `REPLY_EPID`, and `REPLY_LABEL`.

The CU writes the command id to COMMAND to trigger the command with given id. Afterwards, it polls the same register to wait for its completion. This is acceptable in this case, because all commands require only a couple of cycles. The other registers are used to supply the DTU with additional information, depending on the command. The most important internal commands are:

SEND: Send the message stored at DATA_ADDR with DATA_SIZE bytes via the send EP EPID and let the recipient reply to REPLY_EPID with label REPLY_LABEL.

REPLY: Send the message stored at DATA_ADDR with DATA_SIZE bytes as a reply to the message at OFFSET with receive EP EPID.

READ: Read DATA_SIZE bytes from offset OFFSET of the region given by memory EP EPID into DATA_ADDR.

WRITE: Write the DATA_SIZE bytes at DATA_ADDR to offset OFFSET of the region given by memory EP EPID.

FETCH: Store the address of the next unread message in receive EP EPID into register OFFSET and mark it as read. If there is no unread message, store 0 into OFFSET.

The DTU ensures that the CU adheres to the restrictions given by the EP. For example, the offset and size of the data to read needs to be contained in the memory region that is defined by the memory EP. Similarly, the size of the message to send cannot exceed the maximum message size set in the send EP. If one of these restriction is violated, the DTU refuses to execute the command and stores a corresponding error code to the register COMMAND. Note that in contrast to the address of the receive buffer that is stored in the receive EP, the buffer for the commands SEND, REPLY, READ, and WRITE is specified via the DATA_ADDR register when executing the command. The reason is that messages can be received at any point in time, whereas the CU decides when to execute a command. Additionally, in contrast to receive buffers, no restrictions have to be enforced on the buffer for aforementioned commands. Since the buffer's address belongs to the address space of the current activity on the CU, any address can be specified. For example, messages are typically placed on the stack.

External Commands

The purpose of external commands is to enable the kernel to control PEs remotely. For example, the kernel needs to be able to invalidate EPs, as will be explained in more detail in Section 3.5.6. External commands are triggered by writing the command to execute to the EXT_CMD register of the remote DTU. As all external commands require more background to understand them, I will introduce them together with the concept that requires an external command.

3.4.4 Receiving Messages

Whenever the DTU receives a message from the NoC, it needs a place in the SPM to store the message. Since the CU might be a simple fixed-function accelerator, this has to be done without involving the CU, that is, the DTU cannot delegate the placement decision to the CU. For that reason, the receive EP contains all information that is required to make the decision autonomously. Furthermore, it is important that replies to received messages can be sent in any order and other messages can be handled in the meantime. For example, a request to a file system server might require to contact the storage driver to load data from the storage device. As this might take some time, the file system server

should be able to handle other requests in the meantime and reply to the client's request afterwards.

To fulfill these requirements, receive EPs maintain a bitmap for occupied slots and unread slots, a read position and a write position. The write position indicates the last slot where a message has been placed, while the read position indicates the last slot from which a message has been fetched. The occupied bitmap stores which slots are in use, whereas the unread bitmap stores which slots have not been fetched yet. To find a free slot in the receive buffer, the DTU walks in round-robin fashion over the occupied bitmap, starting at the slot behind the write position, and using the first slot that is not occupied. Similarly, the `FETCH` command starts at the slot behind the read position, returns the first slot that is unread to the CU and marks it read. Replying to a message or using the `ACK_MSG` command clears the bit in the occupied bitmap.

This concept allows the DTU to make the placement decision autonomously, while passing the messages in the order they arrived to the CU (to prevent starvation) and allowing the CU to defer the reply to a message. The disadvantage is that the number of messages per receive buffer is limited to the size of the bitmaps (32 at the moment). I think this limitation is acceptable, because multiple receive EPs can be used to increase the limit and, ultimately, many clients will require multiple PEs anyway to keep up with the potentially high request rate. Increasing the size of the bitmaps leads to larger endpoints and thus an increased DTU size. Alternatively, the larger bitmaps can be stored in the SPM to keep the DTU small at the cost of slightly slower message handling.

3.4.5 Replying to Messages

Replying to received messages is an essential feature, because many applications use the request-response communication pattern, such as client-server scenarios. This can be done with an additional communication channel in the other direction: DTU_A has a send EP to a receive EP on DTU_B and DTU_B has a send EP to a receive EP on DTU_A. However, EPs are a scarce resource and servers often have many clients, whereas clients are often only connected to a few servers. Thus, reducing the number of EPs required on the server side is important. In many client-server scenarios, the server does not need a communication channel to the client that allows it to initiate a communication. Instead, being able to reply to received messages is sufficient.

Since multiple send EPs can be connected to the same receive EP, replying to one of the received messages requires to know the destination of that reply, that is, the sender of the message. This information is part of the message header, which is added by the sending DTU. To keep the trusted computing base small, I decided to store this security-critical information not in the SPM with the message (or later, in the cache hierarchy), but in DTU-internal memory, called *header table*. The header table is per DTU instead of per EP and each receive EP stores its starting offset in the header table to store the headers of received messages. To reply to a message, the CU specifies the address of the message to reply to and the DTU sends the reply to the PE and receive EP as specified in the corresponding entry of the header table.

3.4.6 Credit System

Messages are exchanged between applications without involving the kernel. Additionally, the CU at the recipient is not involved to store the message. This raises two questions:

1. how does an application know whether a recipient has sufficient space?

2. how can we stop applications from flooding others with messages?

To solve these problems, the DTU uses a credit system, similar to Intel QuickPath [10]. The idea is to let the recipient hand out credits to its senders, decrease the credits on sent messages and increase them again on received replies. In other words, the credit system can be used for flow control between sender and receiver.

Since the receive buffers have fixed-size slots for messages, the credits are given in messages (e.g., 2 credits allow to send 2 messages). The credits are stored in the send EP. If the send EP has no credits left, the DTU denies to send the message. Otherwise, the credits are decreased by one and the message is sent. The reply to this message increases the number of credits by one again. Typically, the recipient will not hand out more credits than the number of message slots in the receive buffer to guarantee message delivery if a client has credits. However, as the DTU will simply drop messages if the receive buffer is full, it is also possible to hand out more credits if the scenario can tolerate message drops.

3.4.7 Command Abortion

As discussed in Section 3.3.5, a malicious CU (including its caches or SPM) can neither change existing channels nor create new channels. However, it can prevent a successful communication. For example, if PE_1 sends a message to PE_2, the DTU in PE_2 needs to store the message into PE_2's SPM. Thus, the DTU sends a write request to the SPM. If the SPM is malicious and does never send a response to the DTU, the `SEND` command issued by PE_1 will never complete, which renders PE_1's DTU unusable. For that reason, the DTU needs to offer a way to abort a running command in case the communication partner does not cooperate.

The DTU uses the special command register `ABORT` to allow the abortion of the current command. The DTU supports two abort modes: *hard* and *soft*. The former aborts the current command immediately. The latter expects a timeout in register `ABORT` and delays the abort until the timeout expired if and only if the current command is `SEND` or `REPLY`. The rational is that each message has to be received exactly once, whereas memory accesses can be repeated. Both abort modes are only performed in the local DTU, that is, the communication partner is not notified of the abort. Furthermore, both modes write the abort error code to the register `COMMAND` in case a command has been aborted. If so, the communication channel should not be used anymore (the message might have been delivered successfully or not). Hard aborts are intended in case the communication partner does not cooperate. Soft aborts are used if the communication channel should stay valid (e.g., on context switches as described in Chapter 7). Note also that aborted `SEND` commands still reduce the credits of the send endpoint and aborted `REPLY` commands deny further replies to the same message. This prevents that the abort can be abused to circumvent the credit system or the reply-once guarantee.

3.4.8 Discussion

Although the design of the DTU is constrained by the goal to be independent of the CU, there are design alternatives and extensions of which I discuss the most important ones.

DTU Sharing

As pointed out earlier, for the sake of simplicity I currently assume that the CU executes a single stream of instructions. Modern processors typically employ multiple cores and

multiple hardware threads per core to make the best use of the resources. Although it is possible to add a DTU for each hardware thread, it might be a better trade-off to share a DTU among multiple hardware threads. The DTU can be extended to employ a register file (EP registers, command registers etc.) per hardware thread in the DTU, but share the logic and buffers. This approach decreases the performance slightly, but reduces the required chip area and is transparent for the CU.

Message Headers in Memory

An alternative to the header table in the DTU is to load the header from the message in memory (the receive buffers store messages including their headers). This requires additional protection to prevent that the CU can change the header to, for example, send a message to a different PE: 1) message headers need to be read-only and 2) receive buffers cannot overlap. The latter is required, because otherwise a malicious application could let another PE send carefully crafted messages into receive buffer A, that overlaps with receive buffer B in such a way that message payloads from receive buffer A are interpreted as message headers for receive buffer B.

Relying on the headers in memory leads to a smaller DTU, because it removes the DTU-internal memory for the header table (2 KiB for 128 headers in the current implementation). The downside is that it moves the memory (SPM, caches, DRAM, …) into the trusted computing base (TCB) of the entire system. In case of complex general-purpose cores, where separating the caches from the core is difficult, it effectively moves the entire CU into the TCB. Furthermore, storing headers in memory requires a concept to make the message headers read-only without relying on the CU's memory protection features. In the collaboration with Benedikt Nöthen on Tomahawk 4 [58], we stored the headers in memory and introduced a DTU register that defines an address as the barrier between writable and read-only memory. The kernel will set it remotely, place all receive buffers above that address and make sure that they do not overlap. Since we placed the DTU between CU and SPM, the DTU received all memory requests from the CU and simply ignored write requests to addresses above the barrier. For simplicity, we put the entire receive buffers in read-only memory instead of just the message headers.

Due to the low memory requirement of the header table when considering todays CUs, the reduced TCB size, and the simplicity (no additional protection needed), the DTU described in this thesis stores the header table in the DTU.

Endpoints in Memory

Similarly to the message headers in memory, as just discussed, the EPs can also be stored in memory instead of in the DTU. This has comparable consequences: On the one hand, it makes the number of EPs configurable at runtime, removing the need to multiplex them (see Section 3.5.6), and reduces the size of the DTU. On the other hand, additional protection is required and the memory becomes a part of the TCB.

3.5 The Operating System M^3

The last section described the DTU as the foundation for isolation and communication. I will now move from hardware to software and demonstrate how the OS uses this foundation to achieve isolation and communication. The name of the OS is M^3 for **m**icrokernel-based syste**m** for heterogeneous **m**anycores. Alternatively, the abbreviation M^3 can be seen as L4 ±1, because some ideas are taken from the L4 microkernel

family [77, 83, 137]. I decided to design M³ as a microkernel-based OS, because of the advantages in security and reliability [61, 62, 77]. For that reason, the M³ kernel only provides basic abstractions and mechanisms and implements the actual functionality of the OS via servers running on user PEs.

3.5.1 System Calls

System calls in traditional systems are performed by executing a special instruction (for example, `SYSCALL` on x86-64) that switches from the unprivileged CPU mode into the privileged CPU mode. The kernel will typically save the CPU registers first, handle the system call, and restore the registers again before returning to user space. The system call number and its arguments are typically passed in registers or in memory.

On M³, system calls are performed by sending a message from the user PE to the kernel PE. This approach is comparable to FlexSC [136], which handles system calls asynchronously on a separate core. The message contains both the system call number and its arguments. As PEs are isolated by default, system calls require an established communication channel. This channel is set up by the kernel before starting the application. The kernel configures EP_0 as a send endpoint with a receive endpoint at the kernel's DTU as the destination. Furthermore, EP_1 is configured as a receive endpoint to receive the replies from the kernel. The credits of EP_0 are set to allow one system call at a time. To perform a system call, the application uses the DTU's `SEND` command with EP_0 and sets `REPLY_EPID` to 1 for receiving the reply[4]. Note also that M³ uses the same mechanism for system calls as for IPC, which allows to easily interpose the system call channel.

3.5.2 Capabilities

Inspired by L4, the M³ kernel uses capabilities to manage the access of applications to resources. A capability is thereby a pair, consisting of a pointer to the resource and permissions for this resource. Like L4, M³ supports different kinds of capabilities. However, at this point, I will only talk about *object capabilities*. The introduction of virtual memory in Chapter 5 will add *mapping capabilities*, which are comparable to L4's memory capabilities. Each object capability refers to a *kernel object*. Examples for kernel objects are send and receive gates for message passing (see Section 3.5.4), memory gates for memory access (also explained in Section 3.5.4), endpoint objects to *activate* gates, and virtual processing elements (VPEs). VPEs are explained in more detail in the next section, but can be roughly compared to processes. Each VPE has its own capability space.

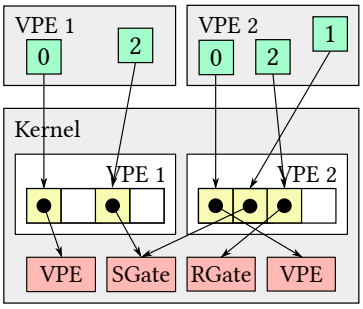


Figure 3.3: Capability system

As shown in Figure 3.3, capabilities are stored in the address space of the kernel and are thus inaccessible from applications. When performing system calls, applications use *capability selectors* (green in Figure 3.3) to reference capabilities (yellow) in their own capability space. Thus, capability selectors are comparable to file descriptors in UNIX-like OSes. Since multiple capabilities can point to the same kernel object (red),

[4] Specifying a different endpoint is fine, too. If it is no receive endpoint or has not enough space, the DTU will simply drop the reply, only hurting the application itself.

all kernel objects are reference counted. Capabilities can be created, exchanged, and revoked via system call:

Creation The kernel offers a system call for each type of kernel object, which creates both a new kernel object and a capability that points to it. The capability will be stored in the capability space of the requesting VPE at the specified selector.

Exchange Capabilities can be exchanged between VPEs in two forms: *delegate* and *obtain*. The former copies a range of capabilities from the requesting VPE to a specified VPE. The latter works in the other way around.

Revoke At any point in time, capabilities can be revoked, that is, deleted. Revocation is done recursively: if VPE_1 has delegated a capability to VPE_2 and VPE_2 to VPE_3 and VPE_1 revokes its capability, the capabilities are revoked from all three VPEs.

The ability to revoke capabilities recursively requires the kernel to store the chains of exchanges. Similar to other capability-based OSes, M^3 uses a so-called *mapping database*, which stores capabilities in a tree, according to the chains of exchanges. Revoking a capability will revoke the entire subtree.

3.5.3 Virtual PEs

The M^3 kernel uses the abstraction *virtual processing element* (VPE) to manage the access to user PEs. A VPE is a resource container and an execution context and hence a combination of a process and a thread in traditional OSes. This decision has been made for simplicity and because M^3 does currently not support multiple execution contexts within one resource container. If this support is added, execution contexts will be separated from VPEs and represented by an own kernel object. A VPE is created for a specific type of user PE. Each VPE has its own address space (consisting of the SPM and DTU) and capability space, isolating VPEs by default. For now, we assume that each VPE has its own PE and VPEs are neither preempted, nor migrated. That is, VPEs are assigned to a specific PE at creation and run on this PE until their completion. Chapter 7 will introduce context switching to share PEs between multiple VPEs. In contrast to traditional OSes, the kernel does not maintain any CU-specific data for the VPE such as the register state. Instead, the kernel is only concerned about the VPE's DTU and controls it remotely via RDMA requests. The name *virtual PE* has been chosen, because a VPE provides the illusion that an entire PE, consisting of CU, memory, and DTU, belongs to the owner of the VPE.

Lifecycle

Each VPE has a state, which is manipulated by the kernel according to the state diagram depicted in Figure 3.4. Upon creation, the VPE is in the state `INIT` and has an empty capability space (except for a capability for the own VPE, a capability for the own memory, and one endpoint capability for each endpoint, present at fixed positions in each capability space). This is in contrast to the `fork-exec` model used in UNIX-like OSes, where a new process starts with the same resources as its parent. As in L4, this is motivated by the *principle of least authority* (POLA), which is easier to achieve by selectively granting permissions instead of selectively removing permissions.

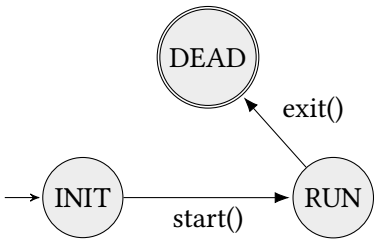


Figure 3.4: VPE state diagram

Creating a VPE provides the creator (the parent) with a VPE capability, a memory gate capability for the entire address space of the VPE, and one endpoint capability for each endpoint (see Section 3.5.4). This moves the potentially complex task of application loading into user space (except for the first VPE, comparable to "init" on UNIX, which is simple to load). During the `INIT` state, the parent typically delegates the capabilities to the child that are required for its operation. Afterwards, the parent loads the desired application into the child VPE via the memory capability and uses the start system call, which causes a transition to the `RUN` state. Finally, the exit system call of the child VPE leads to a transition to the `DEAD` state. The parent can use a system call to wait for this event. Note also that the application loading step is optional to use the same API for non-programmable PEs, which will be discussed in more detail in Chapter 6.

CU-specific Helper

Since the kernel is running on a different PE and controls the user PEs remotely, the kernel requires a small helper on each user PE to run software on this PE. This small piece of software is called *CU-specific helper*. As each VPE has its own instance of the CU-specific helper, the kernel loads it onto the target PE at the creation of a VPE. At this point, the CU-specific helper is only responsible to initialize the CU (e.g., the CPU registers) and to wait for the kernel's signal to transfer control to a just loaded application. As shown in Figure 3.5, the CU-specific helper contains two modules, which will be described in more detail in later chapters. The remotely controlled time multiplexer (RCTMux) is responsible for context switching, whereas the virtual-memory assistant (VMA) is used to support virtual memory.

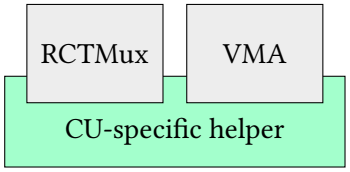


Figure 3.5: Modules of the CU-specific helper

Application Loading

Based on the memory gate capability received upon VPE creation, M^3 currently offers two ways to load an application into a VPE: `run` and `exec`. `run` takes a function as an argument and executes it asynchronously in the child VPE. Thus, the code and data of the parent is cloned, similar to `fork`, but without the capabilities. The parent VPE is free to perform other tasks while the child VPE executes. Due to the cloning of the state, `run` is primarily intended for PEs with compatible ISAs. However, compiler techniques for cross-ISA migration [47] can relax this requirement. Since we are currently considering PEs with SPM, all code and data is copied eagerly. On PEs with virtual-memory support, introduced in Chapter 5, demand loading is used.

As on UNIX-like OSes, `exec` expects the path of the program to execute and command line arguments. `exec` loads code and data of the specified program from the file system into the VPE. Analogously to `run`, this is done eagerly for PEs with SPM and via demand loading in case virtual memory is supported. In contrast to `exec` on UNIX, M^3's `exec` is not used to replace the program in the own VPE, but in a different VPE. Thus, like `run`, it is asynchronous and allows the caller of `VPE::exec` to perform other tasks during the runtime of the executed program.

3.5.4 Gates

As described in the last section, VPEs are isolated by default. The kernel provides mechanisms to establish communication channels between VPEs, if desired. There are

two types of communication channels: message passing channels and memory access channels. Both channels are used via the DTU and are represented as kernel objects, called *gates*. Gates come in three forms:

1. *receive gates* to receive messages,
2. *send gates* to send messages to a receive gate, and
3. *memory gates* to access PE-external memory.

To use a gate, the VPE has to ask the kernel to configure an endpoint of the DTU. This procedure, called *activation*, is done via an RDMA write from the kernel's DTU to the endpoint registers of the target DTU. In other words, the kernel writes to the MMIO region of the corresponding endpoint in the target DTU. Afterwards, the VPE can use the communication channel by executing the corresponding DTU command (`SEND` for send gates, `READ` or `WRITE` for memory gates, and `FETCH` or `REPLY` for receive gates). Section 3.6 provides an example of how gates are used for communication.

As mentioned in the previous section, activating a gate requires an endpoint capability, which denotes the DTU endpoint that should be configured. Every VPE has an endpoint capability for each of its DTU endpoints, which allows the VPE to activate gates on its own endpoints. Additionally, parents receives endpoint capabilities for their child VPEs and endpoint capabilities can be exchanged with other VPEs. Hence, VPEs can activate gates for other VPEs. This is important for non-programmable PEs, which require established communication channels at start, and will for example be used by OS services, as explained in Chapter 4.

3.5.5 Memory Management

In the current implementation, the M^3 kernel is responsible for the physical memory management (DRAM). As the management can become more complex in the future to, for example, support non-uniform memory-access (NUMA) architectures, this responsibility should later be moved to userland, similarly as in L4. The M^3 kernel offers the `create_mgate` system call that allocates physical memory and creates a memory gate for the allocated memory. Additionally, the `derive_mem` system call is offered to derive a new memory gate capability from an existing one and restrict the new memory gate to a subset of the memory and the permissions.

3.5.6 Endpoint Multiplexing

Endpoints (EPs) are a scarce resource in the current implementation as they are hardware registers (see Section 3.4.8). To support more communication channels than the available EPs, M^3 supports multiplexing of the EPs among gates. For security reasons, EPs can only be configured by the kernel, so this needs to be done via system call. However, the kernel cannot simply reconfigure an EP due to the credit system. Let us assume that the VPE wants to replace one send gate with another one and the send EP used for the current send gate has outstanding credits, that is, has lost credits by sending a message and is still waiting to get them back. This leads to two problems:

1. If the credits arrive after the send EP has been reused for a new gate, the new gate would receive the credits, because only the EP id is used as identification.
2. Reading the current number of credits and reconfiguring the EP is not atomic. If the credits are received in between, the kernel stores the wrong number of credits and the already received credits are lost.

To solve these problems, I decided to let the kernel use an external command (see Section 3.4.3) to invalidate an EP. This command atomically checks whether no credits are missing and if so, invalidates the EP. Otherwise, an error is reported to the kernel. The first problem is solved, because outstanding credits are impossible and the second problem is solved, because invalid EPs cannot receive credits anymore. The downside is that send EPs cannot be reconfigured if credits are outstanding.

Memory EPs can be multiplexed without any further measures. Multiplexing receive EPs is possible by invalidating its send EPs first, but not yet supported in the current implementation.

3.5.7 Discussion

The design of M^3, running the kernel on a dedicated PE and remotely controlling the user PEs, is different from traditional OSes. This has advantages and disadvantages:

- As the kernel does not run on all CUs supporting heterogeneous CUs becomes easier. In particular, the DTU as the common interface of all CUs allows the kernel to (mostly) ignore the differences of the CUs and only take care of the DTUs and their communication channels.
- Applications do not share resources like CPU registers, caches and TLBs with the kernel. First, not sharing these resources can lead to performance improvements, because CPU registers do not need to be saved and restored on system calls and the kernel does not evict applications' cache lines and TLB entries. Second, resource sharing can lead to covert channels, which threaten the system security.
- Recently discovered security flaws in most modern general-purpose cores that use speculative execution (e.g., Meltdown [89], Spectre [78], and Foreshadow-NG [149]), raise the question whether we should still base our system's security on the proper enforcement of the different CPU modes by such complex cores. Instead, outsourcing this responsibility to a simple hardware component like the DTU looks promising to increase the security of our systems.
- Furthermore, as no application is running on the kernel PE, the kernel does not need different CPU modes, support context switching, or use paging to isolate itself. Instead, the only hardware feature that a kernel needs to support is the DTU to communicate with the applications and control them from the outside. This simplifies the kernel and creates opportunities such as implementing the kernel on an FPGA instead of running it on a general-purpose core.
- The kernel does not necessarily benefit from the same hardware features (instruction extensions, number of functional units, memory architecture etc.) as the application. Giving the kernel and applications different PEs allows to accelerate both in the best possible way.
- One downside of this approach is less system utilization, because one PE is dedicated to the kernel and thus not usable by applications. However, it is expected that the power consumption and heat generation will be the limiting factors in the future [59, 142]. That is, even if an OS could fully utilize all PEs at all times, physical limits will prevent the OS from doing so.
- The direct communication between applications is beneficial for performance and scalability. However, it has the downside that the kernel does not know what

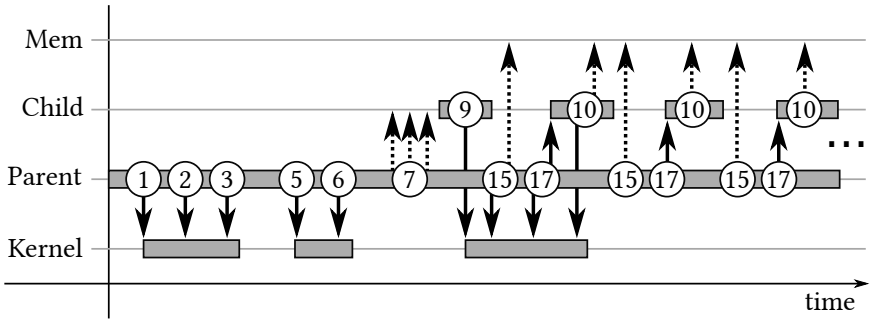


Figure 3.6: DTU-based interactions in the producer-consumer example

the applications are doing (e.g., computing or idling). We will experience this problem in the design of context switching in Chapter 7.

3.6 Interplay

After the description of the DTU and M^3, this section gives an example to provide a better understanding of the interplay between hardware and software on the one hand, and applications and the kernel on the other hand. Let us look at the following producer-consumer example (simplified):

```
1  RecvGate rgate = RecvGate::create();
2  SendGate sgate = SendGate::create(rgate);
3  MemGate mgate = MemGate::create(512 * 1024);
4
5  VPE consumer = VPE::create();
6  consumer.delegate(rgate, mgate);
7  consumer.run([&rgate, &mgate] {
8      while(true) {
9          msg = rgate.receive();
10         mgate.read(...);
11     }
12 });
13
14 while(true) {
15     mgate.write(...);
16     msg = ...
17     sgate.send(msg);
18 }
```

Figure 3.6 illustrates the interactions between the participating components when running the producer-consumer example. The gray horizontal bars represent activity of the corresponding component. The numbers in circles refer to the line numbers in the code and all arrows represent DTU-based communications. The dotted arrows are memory accesses, whereas the solid arrows represent message exchanges. As can be seen, the first three lines issue system calls to create the three gates (①, ②, and ③). The send gate is created for the receive gate, allowing to send messages to the receive gate. The memory gate will be used to exchange the data between producer and consumer. Afterwards, we create a child VPE that acts as the consumer (⑤) and delegate the receive and memory gate to that VPE (⑥). The method `VPE::run` executes the given function (a C++ lambda in the example) on the VPE asynchronously. `VPE::run` first copies the program state into the child VPE, leading to the memory transfers in

line (7). In line (9), the receive gate is implicitly activated via system call to configure an endpoint for message receptions[5]. Afterwards, the parent VPE writes to the memory gate in line (15) and sends a message to the child VPE in line (17). Since both gates were not activated yet, activate system calls are performed on their first usage. As soon as the child VPE has received the message, it reads from the memory gate leading to another activate system call ((10)). Afterwards, all gates are activated, so that the kernel is no longer involved in the communication. In other words, after the setup phase, both VPEs interact autonomously and the kernel is idling.

3.7 Evaluation

This section evaluates the basic attributes of the described architecture. Application-level benchmarks and other benchmarks that evaluate the system as a whole will follow in Chapter 8. Before starting with the evaluation, the following section shortly introduces the used prototype platforms and provides the details on the evaluation platform.

3.7.1 Prototype Platforms

M^3 runs on different prototype platforms, which all have their individual advantages and disadvantages as described in the following.

Tomahawk The first prototype platform is Tomahawk [28], which is a heterogeneous multiprocessor system-on-chip (MPSoC) designed at TU Dresden and primarily intended for mobile communication applications. As illustrated in Figure 3.7, Tomahawk consists of multiple PEs integrated into a NoC and a memory controller that provides access to an external DRAM. Each PE contains an Xtensa LX4 general-purpose core, a scratchpad memory (SPM) for instructions and data, and a DTU. The Tomahawk platform is therefore comparable to the platform considered in this chapter.

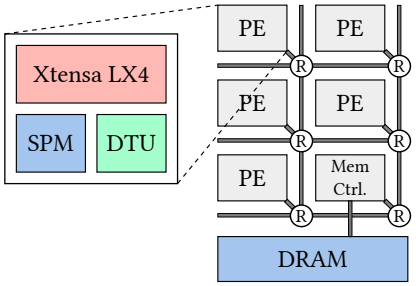


Figure 3.7: Schematic depiction of the Tomahawk platform

This platform was interesting for my work, because the Xtensa cores lack the required architectural features (e.g., multiple CPU modes and a memory management unit) to run a traditional OS kernel, similar to accelerators. Tomahawk exists as custom silicon in, among others, version 2 [106] and version 4 [58]. The PEs in Tomahawk 2 contain no full DTU, but a simple DMA unit that allows transfers between PEs and between PEs and DRAM. Tomahawk 4 contains a DTU[6] similar to the one described in this thesis, designed by Benedikt Nöthen and myself.

gem5 Due to the large effort that comes with a silicon implementation, I used simulation to study the more advanced aspects of this system architecture. I chose gem5 [38], which is a widely spread platform for computer-architecture research. gem5 is an ideal candidate for my work, because of its modularity and flexibility that allowed me to explore new ways of combining existing hardware components like general-purpose

[5]Since both VPEs execute independently, the parent might also execute line (15) first. In any case, activating the send gate in line (17) blocks until the associated receive gate has been activated.

[6]The DTU is called iDMA (intelligent direct memory access) controller in the Tomahawk 4 publication.

cores, caches, and memories. Additionally, gem5 supports multiple CPU models, memory back-ends, and execution modes that allow for a trade-off between simulation speed and accuracy. In particular, gem5 supports a cycle-accurate out-of-order CPU model and detailed cache models that enable accurate performance measurements. Although gem5 supports multiple instruction set architectures (ISAs) such as x86, ARM, MIPS, and ALPHA, gem5 does currently not support mixed-ISA systems. To evaluate my system architecture on gem5, I implemented a DTU model based on an early prototype by Christian Menard, and integrated fixed-function accelerators.

Linux Due to the slow simulation speed of gem5 and because some parts of M^3 are independent of the underlying platform, I use Linux as another prototype platform. On Linux, each PE is represented as a Linux process and the DTU is emulated in software based on UNIX domain sockets. Linux is valuable for early prototyping and debugging (e.g., by running a program on one PE in Valgrind [104]), but not suitable for performance measurements due to the slow communication primitive. The simulation of potential future platforms natively on top of Linux has been published on the third workshop on Systems for Future Multicore Architectures (SFMA'13) [29].

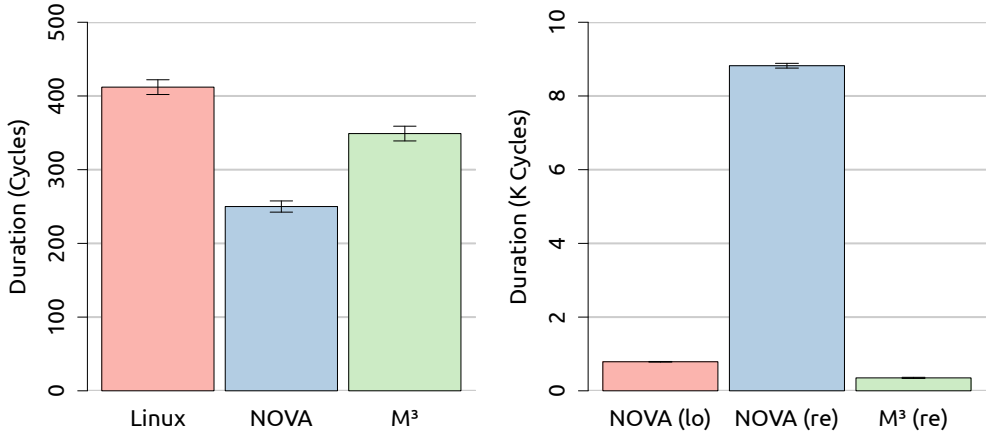
Evaluation Platform For the evaluation in this work, except for the power and chip area measurements in the following, I chose gem5 as the evaluation platform instead of Tomahawk. The reason is that the simple DMA unit in Tomahawk 2 requires to emulate a significant portion of the DTU's functionality in software. The emulation leads to unrealistic performance results for DTU-based operations. Tomahawk 4 does not require software emulation, but is unfortunately not usable for non-trivial scenarios due to a hardware bug in the network-on-chip or DTU. In contrast, the gem5-based prototype platform allows accurate performance simulations and supports all features of M^3 such as virtual memory and accelerators. Furthermore, the flexibility of gem5 enables the evaluation of many different system configurations.

On gem5, M^3 supports both x86-64 and ARMv7. However, the x86-64 support is more complete and gem5 does not support mixed-ISA systems. Since M^3 achieves comparable results on both ISAs and to prevent a distraction from the important points in this work by a comparison of ISAs, I decided to only show the results for x86-64.

3.7.2 System Call and IPC Performance

I start by comparing the performance of two basic operations on M^3 to existing systems: system calls and communication between processes (VPEs on M^3). The performance is compared to Linux and NOVA [137]. NOVA is a microkernel/-hypervisor of the L4 family. I compare the performance of system calls on all three OSes. However, as Linux is a monolithic kernel, inter-process communication (IPC) is less important. In contrast to that, it is one of most important and optimized primitives of a microkernel such as NOVA, so that I compare IPC performance only to NOVA. The benchmarks were done with Linux 4.10, modified versions of NOVA, and one of its userlands called NOVA runtime environment (NRE). NOVA and NRE were modified to run on gem5[7].

[7] The modified NOVA version is available at `https://github.com/TUD-OS/NOVA/tree/gem5` and NRE at `https://github.com/TUD-OS/NRE/tree/gem5`.



(a) System call performance

(b) IPC performance

Figure 3.8: Syscall and IPC performance on Linux, NOVA, and M³

Measurement Setup

All three OSes are run on gem5 in full-system mode with x86-64 cores[8], using the out-of-order CPU model and the classical memory system. Linux and NOVA use a typical cache configuration consisting of 32 KiB L1 instruction cache, 32 KiB L1 data cache, and 256 KiB L2 cache. Since this chapter is considering PEs with SPM, M³ runs on such a PE with a sufficiently large SPM for the used programs. I will later revisit these benchmarks with other PE configurations. All cores are simulated with a 3 GHz clock frequency. The `DDR3_1600_8x8` model of gem5 is used as the physical memory, clocked at 1 GHz. The PEs are connected via a crossbar instead of a full NoC, because gem5's NoC model is strongly coupled with the cache coherency protocol. A crossbar is sufficient in this case, because the evaluation does not require many PEs.

System Calls

On all three OSes, 100 system calls were performed with warm caches. On Linux and NOVA, system calls are done by executing the system call instruction of x86-64. On M³, a message is sent via the DTU from the application to the kernel, which handles the system call and replies via the DTU to the application. As shown in Figure 3.8a, the performance is similar on all three OSes. On Linux and NOVA, a large portion of the time is spent with switching the CPU mode and saving and restoring registers. On M³, 86 cycles are spent by the DTU to send the two messages (request and reply), 111 cycles are spent by the kernel and 154 cycles are required for pre and post-processing on the application-side. Note that neither NOVA nor the used Linux version contain mitigations for Meltdown [89] and Spectre [78]. For example, the introduction of kernel page table isolation (KPTI) in Linux as a mitigation for the Meltdown attack, decreased Linux's system call performance significantly. In contrast, the M³ kernel already runs on a separate PE, preventing the Meltdown attack and Spectre attacks against the kernel by design.

[8] Since virtual memory is mandatory on x86-64 and the currently considered PEs do not support it, an identity mapping is set up at application start to "ignore" virtual memory during runtime.

Inter-process Communication

To compare the performance of IPC with a different microkernel, I executed a "IPC-pingpong" benchmark on NOVA and M^3, again with 100 repetitions and warm caches. On NOVA, two processes are created (called *protection domains* in NOVA) and an empty message is sent from one process to the other and back (round-trip). Figure 3.8b shows the time for *local* IPC on one core ("NOVA (lo)") and *remote* IPC between two cores ("NOVA (re)"). This is compared to IPC on M^3, always performed between two cores. Chapter 7 will show how context switching can be used to perform IPC on one core. As can be seen in Figure 3.8b, local IPC on NOVA is quite fast (788 cycles), whereas remote IPC is rather slow (8821 cycles). The main reason is that remote IPC requires inter-processor interrupts (IPIs), which are expensive. On M^3, an IPC round-trip takes 349 cycles, which is even faster than the local IPC on NOVA. This is because the DTU accelerates message passing in hardware and M^3 dedicates an entire PE to an application, leading to a shorter code path between message reception and its handling. Note that the time for IPC is the same as for system calls on M^3, because there is no technical difference between a system call and IPC.

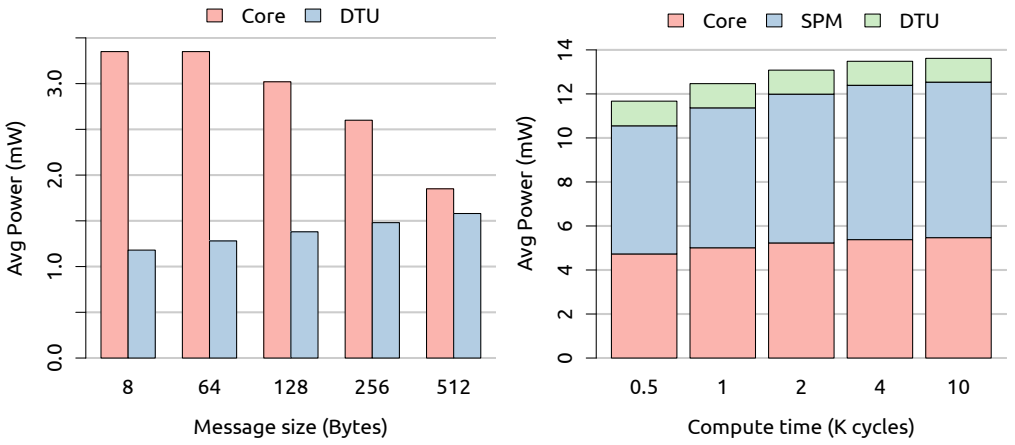
3.7.3 Power Consumption and Chip Area

To quantify the costs of adding a DTU next to every CU, I performed experiments that measure the power overhead of the DTU and its required chip area. This has been done in a collaboration with the Vodafone Chair Mobile Communication Systems at TU Dresden. We based our work on an early version of the Tomahawk 4 Platform [58] and extended it with a DTU. For the measurement, the modules have been synthesized with the Synopsys Design Compiler for a 65 nm low-power TSMC CMOS process at 100 MHz using typical case conditions (e.g., temperature: 25 °C, supply voltage: 1.25 V). To measure the power consumption, we fed Synopsys Prime-Time with a dump file of all switching activities, which was obtained by simulating the execution of the programs on the synthesized gate-level netlists with back-annotated timing information using Mentor Questa.

Power Consumption

The goal of the first experiment is to measure the power consumption of message transfers, depending on the size of the message. We developed a small program that uses a previously configured send endpoint to send messages of varying sizes to a receive endpoint at a different PE. The measurement starts right before a message is sent and ends after this message has been received by the recipient. The left part of Figure 3.9 shows the average power consumption for this period of time. As can be seen, the power consumption of the core decreases with increasing message sizes, while the DTU's power consumption increases linearly, but slowly. The reason is that, at the beginning, the DTU is idling, while the core is active. After the core has instructed the DTU to send the message, the core is put in a low-power mode, while the DTU becomes active. Note that the core is a simple Xtensa LX5 core, optimized for a low power consumption (comparable to the ARM Cortex-M0). That is, the relative power consumption of the DTU compared to more complex CUs will be smaller.

To measure the power overhead of the DTU in more realistic settings, we created a benchmark that resembles an RPC scenario. The benchmark consists of a client and a server, whereas the client sends a request to the server. We let the server compute for different amounts of time to study the overhead of the DTU depending on the usage



(a) Message transfers

(b) RPC scenarios

Figure 3.9: Power consumption for message transfers and RPC scenarios

frequency. We configured the Xtensa core to use scratchpad memory (SPM) as the memory for instructions and data of 32 KiB in total. The average power consumption of the recipient is shown in the right part of Figure 3.9. As can be seen, if the DTU is only used occasionally, which I consider realistic since the DTU is only required to access PE-external resources, the power consumption of the DTU is rather small compared to the core and memory. The average power consumption of core and memory increases with longer compute times, because the instruction mix of the computation is slightly more energy intensive than the mix for message passing and setup time and also produces more memory accesses.

Chip Area

The final question we strived to answer is the chip area required for the DTU. We used the same 65 nm TSMC LP process to determine the chip area of the involved modules. The result is shown in Table 3.2. In this configuration, the DTU makes up 20 % of the chip area. The majority of the area is spent for the endpoint registers. However, the evaluated DTU allows to use all endpoints at the same time. This requires command registers per endpoint and also increases the area for the logic to handle commands in parallel. On the other hand, the evaluated DTU stores message headers in the SPM, not in DTU-internal memory as described in Section 3.4.5. Note also that, as for the power consumption, an Xtensa core is significantly smaller than modern ARM or x86 cores. Additionally, the latter typically employ at least hundreds of KiB of cache per core. Thus, the relative area overhead of the DTU would be significantly smaller. In the hardware implementation of Tomahawk 4 [58], which exists as custom silicon by now, each PE consists of two simple cores (Xtensa and ARM), allowing to use either of them at a time, and employs 128 KiB of SPM. For that reason, the DTU makes up only 6 % of a PE's chip area.

Module	**Area** (mm^2)
PE	0.476 864
├ SPM (32 KiB)	├ 0.211 805
├ Xtensa LX5	├ 0.185 259
└ DTU	└ 0.0798
├ Memory (8 EPs)	├ 0.060 991
└ Logic	└ 0.018 809

Table 3.2: Chip area of a PE with DTU

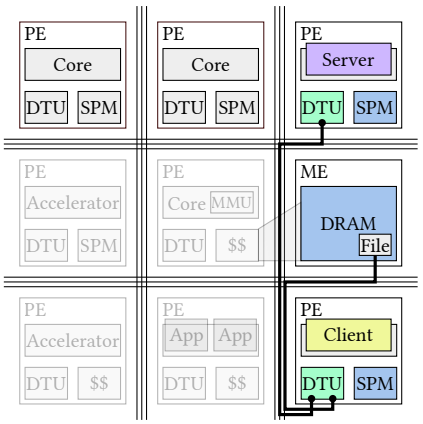
3.8 Summary

In this chapter, I described the foundation of the proposed system architecture. I introduced the data transfer unit (DTU), which provides a uniform interface for all compute units (CUs) and enables a secure communication between all CUs. The DTU supports two communication types: message passing and RDMA-like memory access. Established communication channels for both communication types are represented by the DTU's endpoints.

I introduced M^3 as an operating system that runs the kernel on a dedicated processing element (PE) and controls the applications, represented as virtual processing elements (VPEs), remotely on potentially very heterogeneous user PEs. The primary task of the M^3 kernel is to establish communication channels between VPEs by connecting their DTUs' endpoints. To control the access to resources, the M^3 kernel uses capabilities and offers means to exchange capabilities between VPEs. After a communication channel has been established between two VPEs, these VPEs communicate directly, without involving the kernel again. To this end, the DTU offers internal commands (e.g., `SEND` or `WRITE`), which allow untrusted VPEs to *use* established communication channels, but do not allow to change existing channels or create new ones.

Chapter 4

Operating-System Services



In the previous chapter, I described the basic structure of the system. I introduced the DTU as a new hardware component that provides isolation at the NoC-level and offers communication endpoints. Based on the DTU, the M³ kernel provides the virtual processing element (VPE) abstraction to create isolated components and means to establish communication channels between them. Communication channels are represented in software as gates. To use a gate, a DTU endpoint needs to be configured. Since the number of endpoints is limited, endpoints are multiplexed among potentially many gates. This chapter describes the part of the OS that is hosted on user PEs, which uses the abstractions and mechanisms of the kernel to provide the actual functionality of the OS to applications.

4.1 Motivation

As described in Chapter 2 on related work, there are several existing extensions to OSes such as GPUfs [133], GPUnet [75], OS4RS [107], or BORPH [135] that show how one specific accelerator can get access to OS services. These approaches either add a new protocol to access these services, specifically for the accelerator, or let the accelerator communicate with a software thread that performs the actual access.

In this chapter, I show how the abstractions and mechanisms of the M³ kernel and the DTU can be used to design a new and unified protocol that allows all types of compute units (CUs) to access OS services and, as will be shown in Chapter 6, enables accelerators to run autonomously. More specifically, I introduce a new protocol for all file-like objects, called *file protocol*, that can be used by all kinds of CUs. In other words, the file protocol allows all CUs, ranging from simple fixed-function accelerators to complex general-purpose cores, to access OS services such as file systems, pipes, or network stacks in a uniform way.

This chapter starts with the description of the service infrastructure, which is used later in the chapter to provide access to files and pipes. Next, I introduce the file protocol, followed by a file system server and a pipe server as two examples, which implement the file protocol and use the service infrastructure. Finally, I explain how the virtual file system in M³'s library, called *libm3*, makes file systems and pipes easy to use for applications. Note that this chapter still focuses on simple general-purpose cores, as introduced in the previous chapter. The integration of accelerators and their implementation of the file protocol will be the topic of Chapter 6.

4.2 Services

A *service* on M^3 offers functionality for applications (*clients*) and is hosted on one or more user PEs by a *server*. The service interface is service specific. For example, remote procedure calls (RPC) can be used to request a service's functionality. Another example is that the server sends events to its clients, requiring a communication channel in the opposite direction. To establish the required communication channels between client and server, they need to exchange capabilities. On L4-style microkernels, capability exchange is coupled with IPC, allowing to exchange capabilities during IPC. This approach cannot be used on M^3, because IPC is performed directly between applications via the DTU without involving the kernel. However, the kernel is responsible for the capability management and thus needs to perform the actual exchange. Hence, as illustrated in Figure 4.1, M^3 uses an indirect communication channel (red arrows), involving the kernel, to exchange the capabilities for the direct communication channel (green line). Afterwards, the direct communication channel can be used without involving the kernel again.

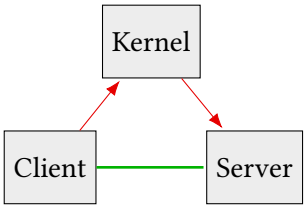


Figure 4.1: Client and server interaction

4.2.1 Service and Session

The M^3 kernel uses two types of kernel objects to handle the access to services: a *service* object, representing the service itself, and a *session* object. To use a service, a client needs to first create a session for the service. The server is aware of all sessions for its service and maintains client-specific state for each session. Furthermore, as a session represents an established connection between client and server, a session allows to exchange capabilities between client and server, indirectly via the kernel.

Servers on M^3 register their service to the M^3 kernel via a system call, which creates a service kernel object. Clients create sessions via system call as well, resulting in a session object. In contrast to other microkernels, the M^3 kernel is currently responsible for maintaining a list of services and deciding whether a client can use a service. This approach has been chosen for simplicity. As the complexity of decision making can grow significantly (e.g., by using a scripting language to configure permissions), it should later be moved to userland. However, since the same approach as for example in L4 can be used, I leave this for future work.

4.2.2 Service Protocol

To notify servers about the creation and destruction of sessions and to negotiate the exchange of capabilities, the kernel creates a communication channel between itself and the server upon service registration. After the service registration, the server listens to requests over this channel. The channel is used on behalf of clients. For example, if a client requests the creation of a new session via system call, the kernel sends a corresponding message to the server via this channel. The protocol that is used on this channel is called *service protocol* and consists of the following messages:

`OPEN` to create a new session,

`DELEGATE` to copy capabilities from the client to the server,

`OBTAIN` to copy capabilities from the server to the client, and

CLOSE to close a session.

The OPEN message is sent by the kernel upon a session creation request and asks the server for permission. If the server is willing to accept the new client, the server creates a new session capability via a system call and sends its selector (the local identifier of the capability) back to the kernel. The kernel will create a copy of the session capability as a child capability and store it in the client's capability table. This way, the server is free to revoke its session capability from its own capability space at any time, which will revoke the capability from the client's capability space as well, because the client's session capability is a child of the server's session capability. Each session has an identifier specified by the server when creating the session capability, which is sent as label (see Section 3.4.2) for all further messages that are exchanged between kernel and server for this session. In the current implementation, the server uses the pointer to the client-specific state as the identifier.

The messages DELEGATE and OBTAIN are used upon capability exchange requests from the client based on an existing session. For DELEGATE, the client specifies the capability range that should be copied to the server. The server is expected to reply to the kernel whether it agrees to the exchange and if so, to specify where the capabilities should be stored in its capability space. For OBTAIN, the client specifies where capabilities should be stored in its capability space and the server is expected to reply to the kernel which capabilities should be copied to the client. Both DELEGATE and OBTAIN provide space for arguments, which are simply forwarded from the client to the server. Note that OBTAIN is similar to *take* in the take-grant model [90]. However, like other microkernel-based systems [83, 137] and in contrast to the take-grant model, M^3's capability tree does not define which capabilities a specific VPE is allowed to obtain. Instead, OBTAIN asks the server for a specific number of capabilities. Thus, the client is not guaranteed to receive specific capabilities from the server.

Finally, CLOSE is sent if the client's session object is revoked. The server will revoke its session capability upon this message. Typically, the client uses the revoke system call to destroy the session as soon as it is done using the service. The revoke can also be caused by a crash of the client, which destroys the VPE including all its capabilities.

4.3 File Protocol

One of the primary goals of this work is to treat *all* kinds of CUs as first-class citizens. To reach this goal, all CUs should be able to access OS services. For example, both general-purpose cores and simple fixed-function accelerators should be able to read from files or write to network sockets. To this end, I introduce a file protocol that can be used by all kinds of CUs to access all file-like objects.

4.3.1 Design Goals

To suit all kinds of CUs, the protocol needs to fulfill certain requirements:

Simplicity: The protocol needs to be sufficiently simple to enable an implementation for fixed-function accelerators or FPGAs, as described in Chapter 6.

Flexibility: At the same time, it needs to be flexible to suite general-purpose cores. For example, applications should be able to open an arbitrary number of file-like objects and use them simultaneously. Additionally, applications should have the option to access file-like objects via a POSIX-compatible API.

Performance and scalability: The protocol has to enable a good read and write performance and should scale with the number of clients, which mandates performing as much work on the client-side as possible.

Considering these requirements, I decided to perform the potentially complex setup and tear-down phases always in software and offer a simple protocol for the data access for *all* kinds of CUs. In other words, software is responsible to establish the communication channels (for example, to the file system) and to tear them down again after the operation, while all CUs, including simple accelerators can use these communication channels to access the data of file-like objects. Since the required steps in the setup and tear-down phases depend on the service, I will describe them in Section 4.4 for M^3's file system service and in Section 4.5 for M^3's pipe service.

4.3.2 The Protocol

The file protocol is depicted in Figure 4.2 and uses a message-passing channel between a send endpoint (EP) at the client (S in the figure) and a receive EP at the server (R). The server is expected to make the data available in memory and provides the client access to the data via the memory EP (M). Note that the protocol is described using EPs instead of gates, because it also intended for simple accelerators that do not use gates (a software abstraction), but directly work with EPs.

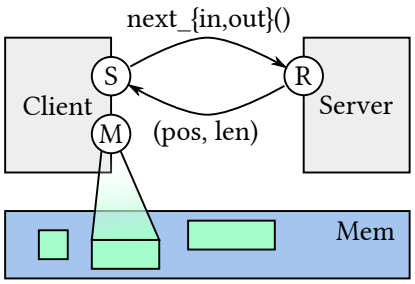


Figure 4.2: The file protocol

The protocol consists of two main requests: `next_in()` and `next_out()`. The former requests access to the next piece of data to read, whereas the latter requests access to the memory to which the next piece of data should be written to. Additionally, the protocol supports the request `commit(nbytes)` to commit the first `nbytes` of the previous input or output request, which is discussed in more detail below. To enable the client to access the data, the server configures the memory EP at the client correspondingly, using the EP capability that the client has delegated to the server during the setup phase. As for example depicted in Figure 4.2, a file system server will provide the client access to a fragmented file (green blocks in the figure) piece by piece. After the EP has been configured, the server replies the position of the current piece within the green block and the size of the piece to the client. Although the position will typically be zero and the size will be the size of the green block, this is not necessarily the case. For example, the pipe server uses a single green block and tells its clients where to read or write next within this block, as will be described in more detail in Section 4.5. Upon receiving this reply, the client can use the memory EP to access the data in RDMA fashion via the DTU. As soon as the data has been completely read or written, the client issues another `next_in()` or `next_out()` request to the server. Receiving a length of 0 from the server denotes end-of-file.

As the client accesses the data on its own via the DTU, the server does not know how many bytes the client has actually read or written. To this end, input and output requests need to be *committed*. Each `next_in()` and `next_out()` request implicitly commits the complete previous piece of input or output data, respectively. Additionally, the `commit(nbytes)` request can be used to explicitly commit the first `nbytes` of the previous input or output request. In contrast to the input and output requests, the commit request does not request access to new data. The commit request is used, for

example, if a client wants to stop writing to a file, in which case it might have written less than it got access to. Another use case is file multiplexing, as described in the following section.

4.3.3 File Multiplexing

As mentioned in the design goals, applications should be able to work with an arbitrary number of files simultaneously. However, the protocol demands a dedicated memory EP per file and EPs are a limited hardware resource. To solve that problem, I decided to multiplex a set of endpoints at the client-side among the open files. Note that file multiplexing is a feature of libm3 and therefore only available for software.

The file multiplexer is contacted whenever the application tries to access the data of a file, say *A*. If file *A* already has a dedicated EP, the data access is performed directly, potentially preceded by an input or output request to the server. Otherwise, the file multiplexer determines a victim file, say *B*, via round robin in the current implementation. Before the EP can be used for file *A*, file *B* calls `commit(nbytes)` to commit the bytes read or written so far. Afterwards, the EP capability is revoked from the server to prevent this server from reconfiguring the EP, while it is used for a different purpose. File *A* now delegates the EP capability to its server and can afterwards perform the read or write operation. Note that this always requires a new input or output request to the server, because the file has either committed the last input or output request during the multiplexing or not yet sent a request.

4.4 File System

After the generic description of the file protocol, this section presents an in-memory file system called *M³FS*, which implements the file protocol and uses the service infrastructure. In-memory file systems are becoming increasingly important due to growing memory sizes and upcoming non-volatile RAM. In the current implementation, the file system image is loaded into DRAM at boot (containing executables for the boot process, for example) and is maintained at runtime by M³FS.

4.4.1 Overview

M³FS is implemented as an M³ service and uses two types of sessions for its clients. The creation of a new session (`OPEN` in the service protocol) creates a *metadata session* at M³FS. Besides performing meta operations (see Section 4.4.4), clients can use the metadata session to open a file via the `OBTAIN` operation, which creates a *file session* (see Section 4.4.3). Both sessions offer an RPC interface for clients based on one send gate (corresponding to the send endpoint of the file protocol) per session. The RPC interface of the file session uses an extended version of the file protocol, additionally allowing, for example, to seek within files. Furthermore, M³FS offers the option to clone a session, resulting in a new pair of session and send gate capability. Cloning a session allows, for example, to provide another VPE access to a specific file without granting the VPE access to the entire file system. Before explaining the file session and metadata session in more detail, the next section described how the data of M³FS is organized.

4.4.2 Data Organization

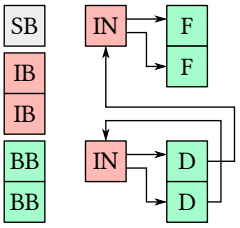


Figure 4.3: M³FS' data structures

The data of M³FS is organized similarly to typical UNIX file systems such as ext4 [97], as shown in Figure 4.3. The file system is split into fixed-size blocks (4 KiB in the current implementation). The superblock (SB in the figure) stores meta information about the entire file system like the total number of blocks. Inode blocks (IN) contain multiple inodes, each representing a file. An inode stores the file's meta data such as the size, the creation date, and pointers to the data. File blocks (F) contain arbitrary data, whereas directory blocks (D) store a list of directory entries in a format defined by M³FS. To prevent clients from directly manipulating directory blocks, M³FS grants only read access to directory blocks and offers meta operations to add and remove directory entries. A directory entry consists of a name and an inode number. To keep track of the available inodes and blocks, M³FS uses an inode bitmap (IB) and block bitmap (BB), respectively.

4.4.3 File Session

As mentioned in the overview, file sessions are created based on a metadata session via the `OBTAIN` operation. In other words, the client requests to obtain capabilities from M³FS. The path and desired access mode are transferred via the arguments of the `OBTAIN` operation (see Section 4.2.2). If the file exists and can be opened for the desired mode (e.g., for reading and writing), M³FS creates a new file session and passes the session capability and a send gate capability for the RPC interface to the client. Afterwards, the client can use the send gate to access the file. The RPC interface implements the `next_in`, `next_out`, and `commit` requests of the file protocol and additionally a `seek` request and an `fstat` request.

To handle the input and output requests, consolidated here as *data requests*, M³FS needs to configure the memory endpoint of the client to provide the client access to the data. The M³ kernel offers the `activate` system call to configure a DTU endpoint, specified by using an endpoint capability, for a given gate capability. Since a memory gate can only refer to a contiguous region of memory, M³FS needs to determine which areas of the file are contiguous in memory. For that reason, inodes organize the files' data in *extents*, similar to other modern file systems [97, 122]. An extent is a contiguous region defined by a starting block number and a number of blocks. An inode contains three extents directly pointing to data blocks, one singly-indirect extent pointing to a block with extents, which point to data blocks, and one doubly-indirect extent. Pointing to data via extents directly indicates which parts of the files are contiguous, eliminating the need to search for them on data requests.

Since data requests always ask for the next piece of data, the file session stores the current file position as a pair of extent number and offset within this extent. The offset is only non-zero if the client seeks or performs a commit request, explained below in more detail. On every data request, M³FS configures the memory endpoint of the client to point to the next extent of the file. To this end, M³FS first creates a new memory gate capability for the next extent. The M³ kernel offers the `derive_mem` system call to create a new memory gate capability based on an existing memory gate capability (here, the file system image in memory), reduced to a subset of its memory region and access permissions. M³FS uses the derived memory gate capability to configure the endpoint of the client. At the end of each data request, after M³FS has sent the reply to the client,

the memory gate capability for the previous extent is revoked.

Output requests beyond the end of the file append to the file. In this case, M³FS adds a new extent to the file with a preferably large number of blocks (currently 128 blocks, i.e., 512 KiB). Using a large number of blocks is important for two reasons:

1. minimizing the number of output requests for appending and
2. minimizing the file fragmentation.

The former reason leads to a better performance of appends. Interestingly, although M³FS is an in-memory file system, file fragmentation is relevant. The reason is that a low file fragmentation minimizes the number of data requests when reading or writing a file, which improves the performance. That is, in contrast to file systems for spinning disks, the distance between two fragments is not important, but the number of fragments (extents) is. Additionally, keeping the number of data requests to the server low is important for its scalability.

To prevent other clients from accessing file parts that have not been written yet (these parts are initialized to zero, though), appends are not visible to other clients until the appending client has committed the append. Appends are committed either implicitly with the next output request or explicitly via a commit request. Besides moving the file position forward, these requests truncate the file accordingly, in case the previous output request reserved more space than the client committed. If another client is already appending to the same file, M³FS denies further appends until the running append has been committed.

Besides data requests, M³FS supports seeking within files. Since the client specifies the file position and not the extent number, M³FS first needs to determine the extent number and offset within the extent. In the worst case, M³FS needs to walk sequentially through the extents to determine the extent for a given file position. For that reason, seek requests are handled on the client side, if the requested position still refers to the data the client already has access to, which is frequently the case. Therefore, I consider the time complexity of $O(n)$ for server-side seeks acceptable. This can be further improved by, for example, using binary search.

Finally, the file session does also support the `OBTAIN` operation to obtain memory capabilities for a specified region of the file. This operation is used by the pager to map files into the address space of a VPE, as will be described in more detail in Chapter 5 on virtual memory.

4.4.4 Metadata Session

The metadata session allows to access and modify the data structures of the file system. Like the file session, M³FS offers this functionality via an RPC interface to its clients. The current implementation supports `open`, `stat`, `mkdir`, `rmdir`, `link`, and `unlink`. M³FS accesses the meta-data blocks (SB, IN, IB, BB, and D in Figure 4.3) via RDMA in DRAM and uses a small *block cache*, similar to the buffer cache in UNIX-like OSes, to minimize the number of RDMA accesses to DRAM. However, since data accesses are performed on the client side via RDMA, M³FS's block cache is rather small. The current implementation has a cache for 16 blocks and employs the least recently used (LRU) strategy for eviction.

4.4.5 Limitations

A difference between M³FS and other file systems is that M³FS does not guarantee atomic writes, because the clients are writing the data on their own. This access type is comparable to `mmap`'ing a file for writing on UNIX-like OSes. However, in case atomic writes are required, M³FS can be extended to first provide clients access to unused memory and move/copy the data to the desired place on commit (for example, enabled by a flag on `open`), similar to copy-on-write file systems [39, 122]. In this way, each client writes to its own memory and the actual write to the file is done by M³FS atomically.

The current implementation of M³FS is not crash consistent. Hence, using M³FS as is with non-volatile memory can lead to inconsistencies if the system crashes. Since metadata is completely controlled by M³FS, crash consistency for metadata can be guaranteed with the same strategies as for traditional file systems (e.g., journaling). In case the data should be kept consistent as well, M³FS can be modified to only use atomic writes, as discussed above. In this way, the file's data is never updated in place. How M³FS can be extended by a block cache for data to efficiently support storage devices as well is discussed in Section 4.7.

4.5 Pipe

This section describes how pipes are provided based on the service infrastructure and the file protocol. The goal is to offer the same semantics as an anonymous pipe on UNIX, that is, a unidirectional, first-in-first-out (FIFO) communication channel. A UNIX-like pipe requires in particular to support multiple readers and writers, which allows, for example, the reader to spawn worker VPEs that concurrently read from the pipe.

4.5.1 Overview

To support multiple readers and writers, M³ implements pipes with a server that manages the pipe at a central place. Data is exchanged via shared memory and the readers and writers ask the server for the next read or write position. Similar to M³FS, the pipe server uses two kinds of sessions. The creation of a new session creates a *pipe session*, representing the pipe. To use the pipe, the client needs to delegate a memory gate capability via the pipe session to the server, which is used to exchange data over the pipe. Furthermore, the pipe session allows to create *channel sessions*, either for reading or for writing, via the `OBTAIN` operation. In this way, the client obtains a capability for the channel session and a send gate capability for the pipe's RPC interface, implementing the file protocol. Each channel session can either be used for reading or for writing and can be cloned to add another reader or writer, respectively, to the pipe.

4.5.2 Data Access

Data in the pipe is accessed according to the file protocol. To this end, the pipe server manages the shared memory, referenced by the memory gate that was delegated via the pipe session at the beginning, as a ring buffer with variable element sizes. For readers, the pipe server derives a read-only memory gate from the shared memory, whereas for writers it derives a write-only memory gate. In contrast to M³FS, the pipe server configures the memory endpoint of each client only once upon the first data access. The reason is that the shared memory area has a fixed size and is always contiguous.

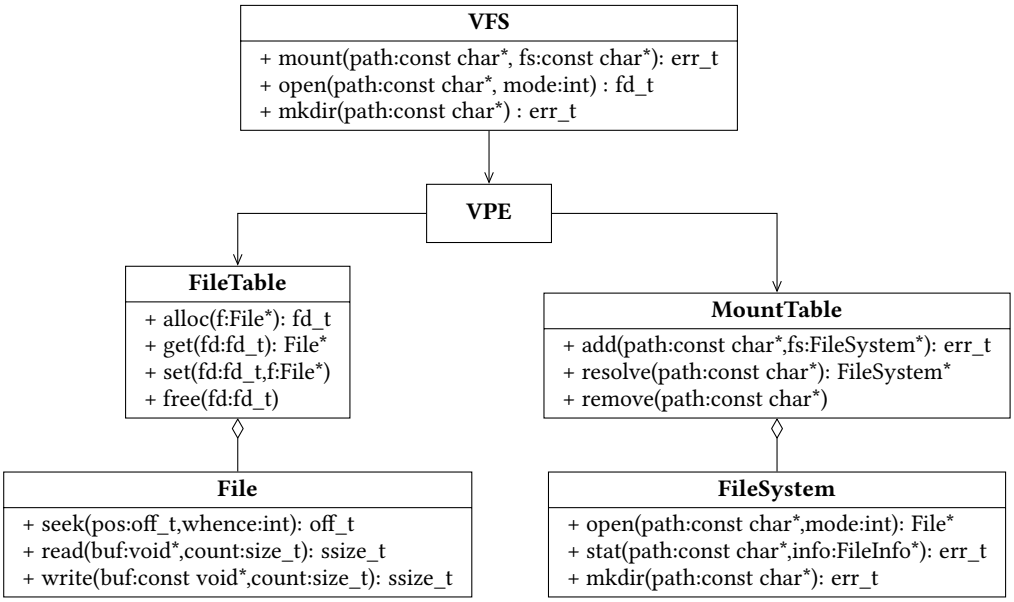


Figure 4.4: Overview of the VFS in form of a UML class diagram

Upon input and output requests, the pipe server checks if the ring buffer has data to read or space to write to. If so, the pipe server replies with the position within the shared memory and the length. Otherwise, the pipe server queues the request and revisits it later. Each input and output request implicitly commits the previous request to the pipe. Additionally, commit requests commit the specified number of bytes to the pipe without requesting new access. To support multiple readers and writers, only one client at a time can read or write, respectively. Further requests are queued as well. After committing an input or output request, pending requests are reconsidered in FIFO order. As soon as all write channel sessions have been closed and all data has been read, readers receive the end-of-file reply from the pipe server. Similarly, if all read channel sessions are closed, writers receive end-of-file since all readers lost interest in the data.

4.6 Virtual File System

The virtual file system (VFS) is a feature of libm3 that offers the ability to work with multiple file systems and access file-like objects. This section takes a brief look at its implementation to provide a better understanding of how files are used in applications. Since M^3 is written in C++, the VFS is implemented in an object-oriented fashion.

4.6.1 Files and File Systems

Figure 4.4 provides an overview of the involved classes with their relations and most important methods. libm3 uses an object of the `VPE` class to represent the application's own VPE and one object for each created child VPE. Each `VPE` object has a `FileTable` object and a `MountTable` object. The `FileTable` maintains a table of open files. The table is indexed by the file descriptor, as in UNIX-like OSes, and each open file is represented by a `File` object. The `File` class implements the file protocol, introduced in Section 4.3. Analogous to the `FileTable`, the `MountTable` maintains a table of file systems, mounted at specific paths and represented by `FileSystem` objects. In contrast to UNIX-like OSes, the VFS is a library feature and files and file systems are not passed

to child VPEs by default. Instead, as described in the next section, the VFS provides mechanisms to *selectively* pass files and file systems to child VPEs, allowing a shared access to, for example, the same file system object.

The `VFS` class offers an abstraction layer on top of the two tables, always referring to the own VPE. Besides forwarding mount and unmount requests to the mount table of the own VPE, it is responsible for determining the `FileSystem` object for a given path, and forwarding the call to this object. For example, `VFS::open` will first use `MountTable::resolve` to determine the `FileSystem` object responsible for the path. Afterwards, it calls the `open` method of the found `FileSystem` object and passes the remaining path within this file system as an argument. The `open` call results in a `File` object, which `VFS::open` associates with a free file descriptor by calling `FileTable::alloc`.

The `File` class implements the file protocol and is therefore used for all file-like objects. The `read` and `write` methods of `File` have a POSIX-like interface and abstract the details of the file protocol. For example, the write method is implemented as follows (simplified):

```
ssize_t File::write(const void *buffer, size_t count) {
    delegate_ep_cap_if_not_done();

    if(pos == len)
        (pos, len) = send_request(NEXT_OUT);

    size_t amount = min(count, len - pos);
    DTU::write(ep, buffer, amount, pos);
    pos += amount;
    return amount;
}
```

At first, the EP capability is delegated to the server in case this has not been done yet or needs to be done again after file multiplexing (see Section 4.3.3). Afterwards, an output request is sent to the server, if no space is left for writing. Finally, the data is written via the DTU from the application's buffer to the position chosen by the server. Note that the commit request is only used when closing the file, as each output request implicitly commits the previous request.

4.6.2 Selective Inheritance

The VFS supports a selective inheritance of file and file system objects to share them with child VPEs. For example, this allows to create a pipe between two VPEs and perform I/O redirection. The key idea, leading to the class design depicted in Figure 4.4, is to separate between the creation of `File` and `FileSystem` objects and their reference. Although each VPE has a file table and a mount table (referencing `File` and `FileSystem` objects), the `VFS` class always creates these objects for the own VPE (`VPE::self()`). If for example a file should be shared with a child VPE, the application copies the pointer to the file object into the file table of the child VPE. This allows to build a collection of files and file systems for an inheritance using the same data structures, without creating new objects, but merely referencing them. After filling the file and mount table of the child VPE, the file sessions and file system sessions selected for the inheritance are cloned for the child VPE, as explained in Section 4.4 and Section 4.5. As most other OSes, M^3 defines the file descriptors 0, 1, and 2 for standard input, standard output, and standard error, respectively.

The following shows an example for a pipe between the parent and child VPE:

```
1  Pipe pipe = Pipe::create(64 * 1024);
2  VPE rd = VPE::create();
3
4  rd.fds()->set(STDIN_FD,
5                VPE::self().fds().get(pipe.reader_fd()));
6  rd.obtain_fds();
7  rd.run([] {
8      File *in = VPE::self().fds().get(STDIN_FD);
9      in->read(buffer, sizeof(buffer));
10 });
11 pipe.close_reader();
12
13 File *out = VPE::self().fds().get(pipe.writer_fd());
14 out->write(buffer, sizeof(buffer));
15 pipe.close_writer();
16
17 rd.wait();
```

The first line constructs a new pipe with a 64 KiB shared memory area, which creates two new `File` objects in our file table. Afterwards, we create a new VPE (starting with an empty file and mount table) and set its standard input to the reading end of the pipe. The call to `obtain_fds` in line 6 lets the child VPE obtain the required capabilities defined by its file table. Finally, `run` executes the given function in the child VPE, which reads from the `File` object associated with standard input. After closing the read end of the pipe in line 11, the parent writes to the pipe and closes the write end. Finally, the parent waits in line 17 until the child VPE has completed the function call.

4.7 Discussion

As in the previous chapter, this section discusses extensions to the presented concepts.

4.7.1 File System Access Control

A metadata session at M^{3}FS currently provides full access to all its files and directories. If a more fine-grained access control is desired, M^3 can for example be extended to restrict metadata sessions to a specific subtree of the file system and to a specific access mode (e.g., read-only or read-write). The `OBTAIN` operation to clone an existing metadata session can be extended to only allow downgrades of these permissions to enable hierarchical restrictions. For example, a session with read-write access to the entire file system can create a new session with read-only access to the sub-directory "/shared" and delegate this session to a different VPE. This VPE can create new sessions as well (e.g., for "/shared/data"), but is restricted to this sub-directory and read-only access.

4.7.2 M^{3}FS for Storage Devices

M^{3}FS is currently an in-memory file system. As the data organization is already similar to popular disk file systems, it can be used for storage devices as well. While the current implementation already uses a block cache for meta data blocks, it grants applications direct access to data, which is already in memory. However, using the existing block cache for data blocks as well and granting applications access to cached data blocks is not desirable in this case, because it eliminates one of M^{3}FS's advantages: granting

applications access to a large contiguous region at once. To support that with storage devices, the cache should store extents instead of blocks.

4.7.3 POSIX Compatibility

As described in Section 4.6, libm3 currently uses an API for the access to pipes and files, which is not POSIX compatible. However, the API has been designed with POSIX compatibility in mind, making it easy to add a POSIX compatibility layer on top. Such a layer does already exist in a slightly different form for the system-call tracing infrastructure, called *systrace*, used for application-level benchmarks in Chapter 8. Systrace translates (a subset of) Linux' system calls to libm3's API, which can easily be provided as a library to run legacy applications.

4.8 Evaluation

This section evaluates the presented OS services in comparison to Linux 4.10 using micro-benchmarks. The benchmarks are performed with the same gem5 configurations as in the previous chapter (see Section 3.7): Linux is using caches, whereas M^3 is using scratchpad memory (SPM). However, this difference is not important for the following benchmarks. Running M^3 on PEs with caches, as is supported with the additions presented in the next chapter, leads to comparable results.

4.8.1 File System Read/Write/Copy

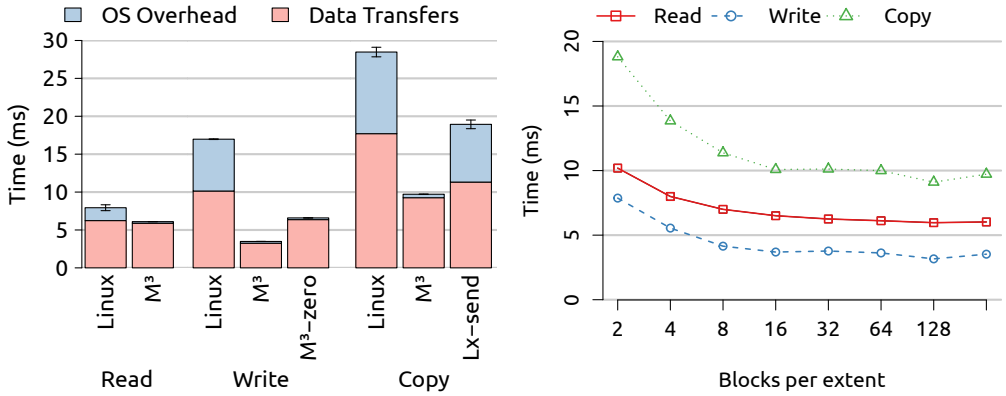
The first benchmark evaluates the performance when reading, writing, and copying files. Since M^{3}FS is an in-memory file system, I compare its performance to tmpfs on Linux. Linux is running on a single core, whereas M^3 uses three PEs: one for the kernel, one for M^{3}FS and one for the benchmark. However, M^3 does not take advantage of multiple PEs, that is, at no point in time are multiple PEs doing useful work in parallel. Instead, the execution merely transitions from PE to PE when the benchmark performs calls to the kernel or M^{3}FS. To ensure that, M^{3}FS revokes the memory gate capability for the previous extent *before* replying to the client in these benchmarks instead of afterwards as described in Section 4.4.3.

The benchmarks perform the following tasks:

1. *Read*: read a file, discarding the data,
2. *Write*: write precomputed data into a new file, and
3. *Copy*: copy a file by reading an existing file and writing to a new file.

All files are 32 MiB large and the benchmarks use an 8 KiB buffer to read/write the file step by step. 8 KiB has been chosen as the buffer size, because it is the sweet spot on Linux for these benchmarks. M^{3}FS and tmpfs use a block size of 4 KiB. M^{3}FS has been configured to use extents of at most 512 KiB for both existing and created files. I analyze the influence of the extent size on the performance in the next section. On Linux, the file operations are performed via the system calls `read` and `write`. On M^3, the methods `File::read` and `File::write` are used.

Figure 4.5a shows the average time over four runs, preceded by one warm-up run. The times are split into data transfers (DTU transfers on M^3 and `memcpy` on Linux) and the OS overhead. As can be seen, both the transfer times and the OS overhead show



(a) Comparison to Linux

(b) Performance with varying extent lengths

Figure 4.5: Read, write, and copy performance, using 32 MiB files

large differences. In explaining this behavior, I will start with the transfer times and go into the OS overhead afterwards.

M³ and Linux use different ways to perform the data transfers. While M³ uses the DTU's RDMA feature to load data from DRAM into SPM or store data from SPM to DRAM, Linux copies the data within its CPU cache from one location to the other. As shown in Figure 4.5a, this leads to comparable transfer times when reading a file. However, on Linux, writing is slower than reading, whereas on M³, writing is faster than reading. The former stems from the fact that gem5's cache model uses the write-alloc policy for write misses (as is common in today's hardware). Thus, write requests that cause a cache miss first load the cache line from memory. This effectively doubles the transfer volume, leading to 64 MiB in total. When writing the file on M³, the DTU is repeatedly instructed to read the buffer from SPM and write it to DRAM. Thus, the file's data is only written to DRAM, not read from DRAM. Interestingly, writing to DRAM is even faster than reading, because read requests cannot be answered until the data is available. Write requests are queued in the DRAM controller and written to DRAM later. This has the consequence, that the DTU can already send the next write request, while the controller is writing the queued requests to DRAM. Hence, writing to DRAM achieves a higher data rate than reading from DRAM. On both OSes, the transfer time for copying a file is the sum of the transfer times for reading and writing.

Besides the transfer time differences, M³ and Linux show different OS overheads. The main reason for the different OS overheads is that M³FS grants its clients direct access to potentially large contiguous regions of the file's data, whereas Linux accesses the file's data page by page. Thus, on M³, after the client has access to the data, it only issues DTU transfers, leading to less overhead. When reading, the overhead is 0.30 ms on M³ and 1.70 ms on Linux. When writing, the overhead is 0.33 ms on M³ and 6.83 ms on Linux. The write overhead on Linux is primarily caused by the block allocation (this has been determined by comparing the overhead when appending new data to a file with the overhead when overwriting existing data). In the default configuration (the columns labeled "M³" in Figure 4.5a), M³FS does not overwrite blocks with zeros on allocation, because it assumes that enough zeroed blocks are available. Overwriting blocks with zeros can be done in idle times of M³FS. In contrast to Linux in this benchmark, M³FS is required to zero the blocks, because it grants its clients direct access (similar to `mmap` on Linux). Otherwise, clients could get access to deleted confidential data. If no zeroed blocks are available anymore, M³FS needs to perform this task on allocation. The results

for this case are shown in the column "M³-zero".

On both OSes, the OS overhead for copying is again roughly the sum of the OS overheads for reading and writing. However, Linux does offer the system call `sendfile` to transfer data between two file descriptors. When using `sendfile` to copy the file instead of `read` and `write`, the performance improves, as shown by the column "Lx-send". The reason is, that `sendfile` can copy directly within the buffer cache, whereas `read` and `write` uses the application's buffer as an intermediate step.

In summary, M³ outperforms Linux in these benchmarks by taking advantage of the DTU's RDMA feature to grant clients direct access to large contiguous file regions. The access can be granted in constant time, independent of the size of the region. CPU caches are inherently at a disadvantage to copy data, because they operate at cache line granularity and have no knowledge about the software's intention. This can be mitigated to some degree by prefetching (enabled in the benchmark), but not completely eliminated, as this benchmark shows. In contrast, the DTU can be instructed to read or write a large amount of data at once, leading to better results (e.g., writing does not require reading). Another positive side-effect of DTU transfers is that the CU can be put into a low power mode during the transfer. Alternatively, the DTU can already prefetch the next data block, while the CU is working on the current block.

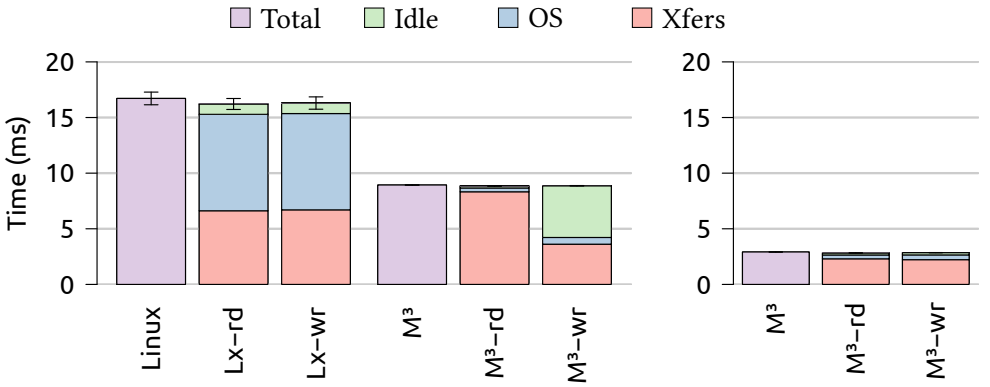
4.8.2 File Fragmentation

As mentioned previously, the performance on M³ for reading or writing a file depends on the fragmentation of the file, that is, the number of extents. To quantify the performance impact of fragmentation, I repeated the three benchmarks of the previous section with varying extent sizes for the 32 MiB file. For reading, the file is created with a varying number of blocks per extent, leading to a fragmentation of the file. Similarly, for writing, M³FS is restricted to find contiguous regions with a limited number of blocks when appending to the file. The results are shown in Figure 4.5b, which are again the averages of four runs, preceded by one warm-up run. As can be seen, the performance increases with increasing extent sizes, but levels off at 16 blocks per extent. For that reason and to reduce the number of requests to M³FS, files are extended in steps of 128 blocks as a trade-off between good performance/scalability and low memory wastage.

4.8.3 Pipe

The second OS service that has been introduced in this chapter is the pipe. Similar to the file system benchmarks in the previous section, I evaluate the pipe performance in this section in comparison to Linux. This time, Linux is running on a two-core system with per-core L1 instruction and data caches with 32 KiB each and a shared 512 KiB L2 cache. M³ uses five PEs: kernel, pipe server, creator, reader, and writer. The creator starts reader and writer and is thus only used at setup time. Reader and writer are working in parallel, as on Linux, whereas the pipe server and the kernel are only active on behalf of the reader/writer. During that time, the reader/writer is inactive.

The creator starts a reader and a writer and lets them exchange 32 MiB of data via the pipe using a 8 KiB buffer. Analogous to the file system benchmarks, the writer writes precomputed data into the pipe and the reader reads and then discards the read data. On Linux, the `fork` and `execve` system calls are used to create reader and writer, whereas on M³, two VPEs are created and `VPE::exec` is used. M³ uses a shared memory (SHM) area of 128 KiB in DRAM to exchange the data. The pipe accesses are performed via `read` and `write` on Linux and via `File::read` and `File::write` on M³.



(a) Comparison to Linux using a SHM area in DRAM (b) Using a SHM area in SPM

Figure 4.6: Pipe performance, exchanging 32 MiB of data

Figure 4.6a shows the average time over eight runs. The bars "Linux" and "M³" show the total time of the benchmark, that is, the time from starting reader and writer until both have exited. The two bars next to each of them show the time for the reader and writer, respectively, split into transfer time, OS overhead, and idle time. On M³, the transfer times behave comparably to the file system benchmarks: writing is faster than reading, leading to a lot of idle time at the writer. In this case, the transfers on Linux are faster than in the file system benchmarks, because the in-kernel buffer for the pipe is much smaller than the file size in the previous section. Thus, the write requests do not leave the L2 cache in most cases.

The comparison also shows that Linux has a much higher OS overhead than M³. The primary reason is that Linux uses a single lock for the entire code to read from or write to the pipe, including the data transfers. Thus, since the user applications in this benchmark use precomputed data and discard the read data, Linux spends almost the entire time in these critical sections. On M³, the data transfers are performed by the clients, not by the pipe server. Thus, the pipe server does not become a bottleneck with one reader and one writer. Furthermore, similar to M³FS, the pipe server lets its clients access a large portion of the pipe's shared memory area at once (by default 25 % of the shared memory size), which reduces the number of requests to the pipe server and therefore the OS overhead.

The performance on M³ can be further improved, because so far, M³ used a shared memory area in DRAM to exchange the data. Since any kind of memory can be used for that purpose, the data can for example be stored in the SPM of an unused neighboring PE. As illustrated in Figure 4.6b, this reduces the transfer times significantly, because SPM is much faster than DRAM. Additionally, the SPM's read and write performance is symmetric, removing the idle times at the writer.

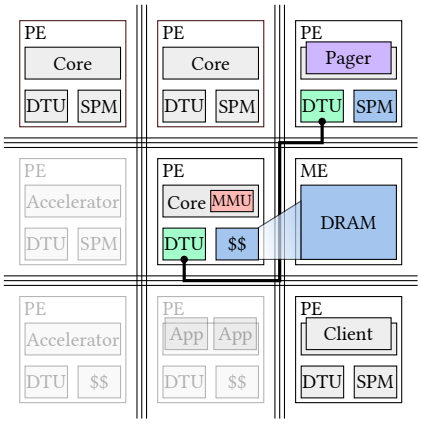
4.9 Summary

In this chapter, I demonstrated how OS services are provided based on the DTU's features and the mechanisms offered by the M³ kernel. I introduced a file protocol, which can be used for all file-like objects such as files, pipes, network sockets, and so on. Most importantly, this protocol is sufficiently simple and yet flexible enough to be suitable for all kinds of compute units (CUs), ranging from simple fixed-function

accelerators to complex general-purpose cores. As a proof of concept, I designed and implemented an in-memory file system, called M^{3}FS, and pipes. The evaluation shows that their performance is not only competitive with, but even outperforms Linux by taking advantage of the DTU's RDMA feature and granting clients direct access to large amounts of data.

Chapter 5

Virtual Memory



The previous chapters used simple scratchpad memory-based processing elements as in the Tomahawk platform. This chapter describes how virtual memory and a cache-based memory access strategy is supported in the proposed system architecture.

5.1 Motivation

The scratchpad-based processing elements (PEs) considered in the previous chapters are well suited for, for example, small applications that fit into the scratchpad memory (SPM) or fixed-function accelerators that benefit from many parallel SPM accesses or predictable access times. However, the SPM is typically small (in the order of hundreds of KiB) due to its expensive SRAM. While larger applications and accelerators can move data manually between the SPM and larger memories, this is rather inconvenient and complicated. Instead, using a cache-based access strategy to a large memory is more attractive, because the cache performs these data movements transparently to the compute unit (CU), while being reasonably efficient.

Cache-based access to a large memory leads to the desire to share this large memory securely among multiple applications to make efficient use of the memory. Thus, applications should be enabled to access *virtual memory*. Besides secure sharing of the physical memory, virtual memory enables other well-known techniques such as copy-on-write, demand loading, swapping, memory mapped files, and page deduplication. The preferred way to implement virtual memory today is paging[1], because it is both performant and simple to use for applications.

5.2 Goals

For the virtual-memory support, I strive for the following properties:

Uniform External Interface The M^3 kernel should be able to manage all PEs in a uniform way and all PEs should be able to collaborate in an easy fashion. Thus, although the DTU's internal interface may vary, all DTUs should have the same external interface to hide the differences between the PEs.

[1] As there is some disagreement concerning these terms, I want to shortly define how they are used in this thesis: *virtual memory* introduces virtual address spaces that are mapped via some mechanism to physical memory. One of these mechanisms is *paging*, which splits the virtual and physical address spaces into same-size blocks called *pages* in the virtual address space and *frames* in the physical address space.

Virtual Interface to DTU As will be introduced in this chapter, a VPE has a virtual address space if it has been assigned to a PE with virtual-memory support. This raises the question whether such VPEs should use virtual or physical addresses when interacting with the DTU (e.g., to specify the address of a message that should be sent or the destination buffer for an RDMA read request). Requiring physical addresses has the downside, that applications cannot specify them on their own, because this would compromise the security of the system. To prevent the security problem, the M^3 kernel could store the physical address in the affected DTU endpoint and let the application work with the corresponding virtual address. For message passing, this approach is acceptable, as has been shown by L4, which uses a per-thread buffer at a fixed location called user thread control block (UTCB). However, it has significant downsides for RDMA. Let us consider a file system with a buffer cache as an example. Being forced to use physical addresses leaves us with three options:

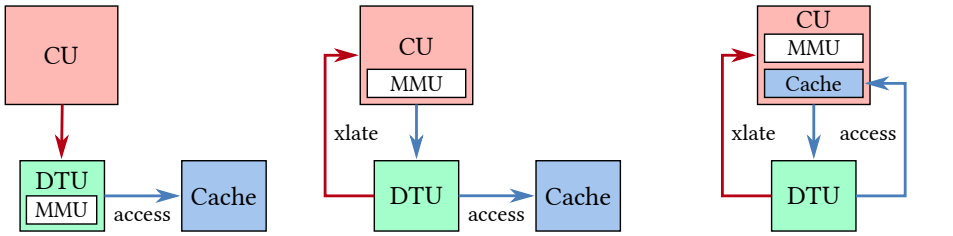
1. We register the entire buffer cache at the M^3 kernel to enable RDMA for it. This requires to allocate the entire buffer cache upfront in physically contiguous memory and pin it in virtual memory. Thus, we lose most of the advantages of virtual memory.

2. We register a single block in the buffer cache at the M^3 kernel on-demand. For example, before loading a block from persistent storage into the buffer cache, we register the destination area in the buffer cache for RDMA. After performing the RDMA read operation, we unregister the area again. Since both registering and unregistering requires a system call, this adds significant overhead to the operation. Depending on the size of the block, the overhead could even be larger than the transfer time for the block.

3. We register a block outside the buffer cache at the M^3 kernel, which we use for all RDMA transfers. This doubles the transfer time for all RDMA operations. For example, a load into the buffer cache requires a load into our RDMA block first, followed by a copy to the actual destination.

As outlined, all three options are undesirable. For that reason, I decided that the interaction with the DTU should use virtual addresses. This enables the applications to freely choose the address without compromising security, because the access permissions are defined by page tables. Thus, in the example, the file system can load data from persistent storage directly into the desired place without any overhead. At the same time, all memory-management techniques such as demand loading, swapping, and so on can be used for the buffer cache. Furthermore, applications on PEs with memory support interact with the DTU in the same way as applications on SPM-based PEs: both use the use the addresses of their own address space (physical or virtual).

Uniform Addressing In the previous chapters, I have shown that RDMA can be used to access all memory in the system, including SPM in PEs and external memory such as DRAM or NVM. That is, VPEs were able to access each other's physical address spaces. In this chapter, I will extend this by allowing RDMA to virtual address spaces as well. Furthermore, it should be possible to map all types of physical memory into virtual memory. This requires *uniform addressing*, which I will introduce in the following sections. Remember that we are not yet considering context switching in this chapter. For that reason, each PE has only a single virtual address space at a time: the virtual address space of the running VPE. Thus, RDMA requests to such a PE access the virtual



(a) CUs w/o MMU (b) CUs w/ MMU (idealized) (c) CUs w/ MMU (reality)

Figure 5.1: Abstract integration concept of CUs with caches

memory of the running VPE. Upon each VPE's termination, the kernel invalidates the communication endpoints at the VPE's communication partners. This ensures that no other VPE can still access the virtual address space of the just terminated VPE.

Optional Cache Coherency Cache-coherent shared-memory systems are the predominant form of systems today. However, it is still unclear whether future systems will be (globally) cache coherent:

1. The costs of cache coherency in terms of energy, chip area, complexity, and performance are expected to increase with an increasing number and variety of cores [74, 98].

2. Heterogeneous systems as targeted by this work increase the challenge to provide global cache coherency: all caches need to use the same cache coherency protocol, which requires changes to at least a subset of the CUs.

3. Whereas implicit communication via cache-coherent shared memory simplifies the programming of homogeneous systems, it is less attractive for heterogeneous systems: CUs might use different endiannesses, different word sizes, different atomic operations, etc.. Thus, communication between different CUs is always explicit to perform the necessary conversions. In other words, the value of cache coherency in heterogeneous systems is arguably smaller than in homogeneous systems.

For these reasons, I keep cache coherency optional. In other words, M^3 does not take advantage of cache coherency. To keep it simple, this chapter uses non-coherent caches and discusses the consequences of coherent caches at the end.

5.3 Overview

To achieve the stated goals, the DTU needs to access the virtual memory of the current VPE. For example, sending a message requires to load the message from the specified virtual address and receiving a message requires to store the message into the receive buffer in virtual memory. Additionally, incoming RDMA requests require access to the virtual memory of the current VPE.

Existing CUs can be split into two categories: CUs that already support virtual memory and CUs that do not support virtual memory. Most general-purpose cores fall into the first category, because they contain a memory management unit (MMU). In contrast to that, accelerators such as DSPs, FPGAs, or fixed-function accelerators do not

necessarily contain an MMU and therefore fall into the second category. The integration of these two categories of CUs is depicted in Figure 5.1.

Starting with the simpler case, if a CU does not contain an MMU as in Figure 5.1a, the address translation should be done externally and transparently to the CU. For that reason, the DTU receives all memory accesses of the CU and interprets the addresses as virtual (red arrows in the figure). The DTU-internal MMU translates the virtual address to a physical address and sends a request with the physical address (blue arrows) to the cache. This leads to physical caches, which are typically preferred over virtual caches (see Section 5.12.2 for a discussion of virtual caches). With this structure, the DTU can easily access the virtual address space of the current VPE, because the DTU can translate the virtual address on its own before accessing the physical cache.

If a CU contains an MMU as in Figure 5.1b, virtual addresses are translated by the CU-internal MMU and the CU's external interface uses physical addresses. Thus, the DTU is not able to translate virtual addresses and requires assistance from the CU (red arrow named "xlate") to access the physical cache based on virtual addresses. Unfortunately, this structure requires an integration of the DTU *between* the CU and the first-level cache. While this might be achievable in some cases, I consider it unrealistic for today's complex general-purpose cores (e.g., x86 cores), because the caches are typically difficult to separate from the core. Solving the problem requires to reuse of existing cores *including* their caches without any modification as depicted in Figure 5.1c. In this case, the DTU needs an interface to access the CU-internal caches, which is typically provided by the cache-coherency interconnect. Note that it is also imaginable to integrate entire CPUs, potentially including DRAM and other components, as a single PE into the system. For example, an Intel Xeon CPU can be integrated in this way. However, this thesis focuses on the creation of system-on-chips by reusing existing cores instead of entire CPUs.

The remainder of this chapter is organized as follows. In Section 5.5, I explain the integration of the two just discussed categories of CUs in more detail, followed by the introduction of *uniform addressing* in Section 5.6. Before continuing with the address translation procedure, Section 5.7 provides the required background on DTU-based data transfers. Section 5.8 describes the address translation in detail for CUs without MMU (Section 5.8.1) and for CUs with MMU (Section 5.8.2), assuming that all pages are already mapped. This is relaxed in Section 5.9 on virtual-memory management, which describes how page faults are resolved. Afterwards, Section 5.10 walks through an example to show the interplay of the different components that provide virtual-memory support. Finally, Section 5.11 examines the implications on the trusted computing base, followed by a discussion of miscellaneous aspects in Section 5.12 and the evaluation in Section 5.13.

5.4 Related Work

There are other works that add address translation or protection to accelerators. At first, the DTU-internal MMU is comparable to an IOMMU in conventional architectures in the sense that the DTU and the IOMMU are managed externally and add virtual memory transparently to a CU. Additionally, this approach is similar to the heterogeneous system architecture (HSA) [7, 123] and the coherent accelerator processor interface (CAPI) [15, 138], which allow the integration of accelerators into a cache-coherent virtual-memory system. IOMMUs, CAPI, and the DTU-internal MMU add virtual-memory support transparently to an accelerator. On the one hand, this approach

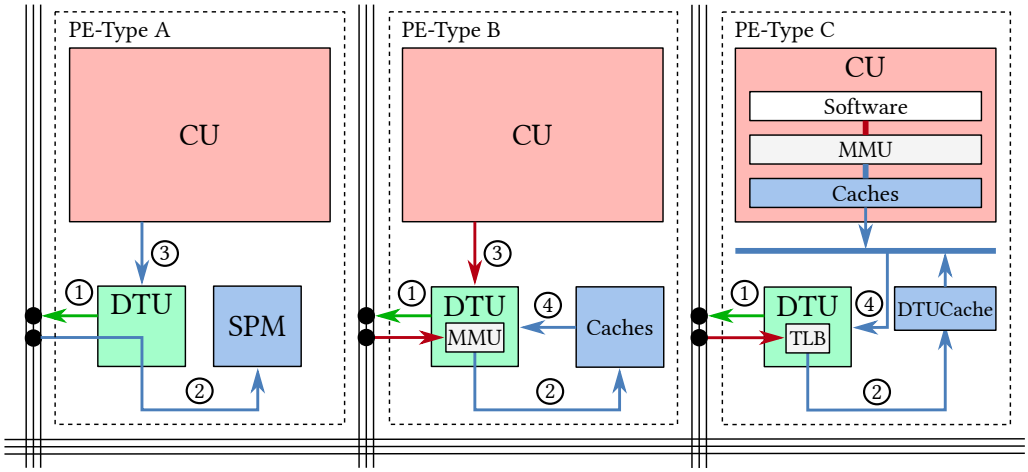


Figure 5.2: Detailed depiction of the integration of different PE types: A for accelerators, B for CUs without MMU, and C for complex general-purpose cores

simplifies the development of accelerators and also leads to secure systems, because memory protection is enforced by a trusted component from the point of view of the system designer (the accelerator can be developed by a third-party vendor). On the other hand, the accelerator vendor cannot tightly integrate the memory translation with the accelerator and optimize it for a specific purpose. For that reason, Border Control [109] separates address translation from protection by leaving the address translation up to the accelerator and performing protection checks outside of the accelerator. In contrast to my approach, HSA, CAPI, and Border Control focus on cache-coherent systems, while my approach keeps cache coherency optional.

5.5 Integration

After the rather abstract description in the overview, this section explains in detail how these two categories of CUs can be integrated. This leads to three *PE types* as depicted in Figure 5.2. All PE types have the same external interface, but different internal interfaces, as described in the following.

In general, the DTU is connected to the NoC (arrow ①) and can access memory (arrow ②), for example, to load a message that should be send or to store a received message. The CU uses memory mapped I/O (MMIO) requests to interact with the DTU, which are handled in different ways, depending on the PE type. Blue arrows indicate the use of physical addresses and red arrows the use of virtual addresses. Green arrows indicate the use of *NoC addresses*, which are introduced in the following section.

PE-type A was the foundation of the previous chapters and is primarily intended for simple accelerators. For example, some accelerators prefer an untranslated access to scratchpad memory (SPM), because SPM easily supports many parallel accesses and has a predictable access latency [43, 93, 129, 130, 144]. Since the SPM is not shared with other PEs[2], an address translation on the path to the SPM is not required. To simplify the sharing of the DTU's implementation for different PE types, all memory requests of the CU are sent to the DTU (arrow ③), which either forwards the request to the SPM or

[2]Only the CU in the same PE sends memory requests to the SPM. Other PEs can access the SPM via RDMA requests (arrow ②), but not without explicit permission by the VPE on the target PE.

handles it, if the request refers to the DTU's MMIO region. In other words, the crossbar shown in Figure 3.2 is part of the DTU.

PE-type B is intended for CUs, for which virtual memory is desired, but unsupported by the CU itself. In this case, the DTU is integrated between the CU and the caches. As in PE-type A, all memory requests from the CU are sent to the DTU (arrow ③) and handled by the DTU if a request refers to the DTU's MMIO region. Otherwise, the DTU performs an address translation before forwarding the request to the first-level cache, similar to an IOMMU. Last-level cache misses are sent to the DTU (arrow ④) and forwarded over the NoC to the physical memory.

PE-type C is intended for complex general-purpose cores that have an MMU and caches. I connect the DTU via a *DTUCache* to the coherency interconnect (thick horizontal line) after the last-level cache, which allows the DTU to access the caches of the CU. Therefore, existing cores, including their MMUs and caches, do not need to be modified. The DTUCache does not need to be large to achieve good performance (8 KiB is used in the evaluation). To perform MMIO requests, the DTU's MMIO region needs to be configured as "uncachable", leading to last-level cache misses, which are sent to the DTU (arrow ④). If a last-level cache miss does not refer to the MMIO region, it is sent over the NoC to the physical memory.

5.6 Uniform Addressing

The key idea of uniform addressing is to split the system into processing elements and memory elements (MEs) and make these elements "look the same" from the outside. In contrast to PEs, MEs do not contain a CU, but only memory such as DRAM, NVM or SPM. An RDMA request to an element *always* refers to its address space. In MEs, the RDMA request refers to their internal physical memory. In type-A PEs, it refers to the SPM; that is, the physical address space of the running VPE. In type B and C PEs, it refers to the virtual address space of the running VPE. This is directly supported in type-B PEs, because the DTU is responsible for the address translation. However, since the purpose of type-C PEs is to reuse existing cores, which communicate with other hardware components via physical addresses, directly accessing the virtual address space is not possible for the DTU. I solve this problem by running a small assistant on the core, called virtual-memory assistant (VMA), which will be described in Section 5.8.

To support the element-centric addressing, I introduce *NoC addresses*, which are split into an element ID and an offset for the element's address space. Currently, I use 64-bit NoC addresses with an 8-bit element ID and 56-bit offset. The element ID is used for routing on the NoC to determine the destination element. The destination element only receives the offset of the NoC address. To support mapping all physical memory into virtual memory, virtual addresses are translated to physical addresses, which are in turn converted to NoC addresses. The conversion depends on the PE type: in PE-type B, no conversion is required, because physical addresses and NoC addresses have the same format. In PE-type C, the required conversion depends on the physical address format of the CU (for example, physical addresses on x86 are 48-bit wide), which is explained in more detail in Section 5.8.2. Without any additional measures, this allows to map virtual memory into virtual memory, possibly leading to cycles. For example, a VPE running on PE 1 could map a virtual address to a NoC address with element ID 1, referring to the VPE's own address space. The M^3 kernel prevents this by only allowing to map physical memory (i.e., NoC addresses referring to elements with physical memory) into virtual memory.

The arrow colors in Figure 5.2 indicate where the different address types are used in the system. All PE types use physical addresses (blue) to access the PE-internal SPM or cache, potentially after a translation from virtual (red) to physical. In type-B PEs, the MMU in the DTU translates the virtual address to a physical address. In type-C PEs, the MMU in the CU performs the translation and the DTU uses a small, software-managed translation lookaside buffer (TLB) to cache recent translations. Incoming requests from the NoC provide the DTU with the offset from the NoC address. In PE-type A, the offset is a physical address, which can be passed directly to the SPM. In type B and C PEs, the offset is a virtual address, which needs to be translated first. In general, NoC addresses are used for outgoing requests such as RDMA and last-level cache misses. Hence, the DTU converts physical addresses to NoC addresses upon last-level cache misses.

This addressing scheme allows to map all physical memory, possibly residing in different elements and using different technologies such as SPM, DRAM, and NVM, into virtual memory. Furthermore, the scheme provides the same semantics for all RDMA requests, independent of the destination element, because all RDMA requests refer to the element's address space: the physical memory in memory elements or the virtual/physical address space of the running VPE in processing elements. Hence, a memory endpoint (enabling RDMA requests) simply holds a NoC address and a size to refer to a piece of memory. In other words, these concepts abstract the differences of the PE types and allow us to (mostly) ignore these differences in the following chapters, because all PEs interact with the DTU in the same way and all PEs can interact with each other in the same way.

5.7 Data Transfers

Before we can understand how address translation works, we first need to take a closer look at how the DTU performs data transfers. These data transfers take place between two DTUs in the system: the sending DTU loads the data from its internal memory and sends it via the NoC to the receiving DTU, which will store the data into its internal memory. Due to this symmetry, I will concentrate on one DTU, that is, the transfers between internal memory and NoC.

XferUnit

The component in the DTU that is responsible for the data transfers between the internal memory and the NoC is called *XferUnit*. As explained in Section 5.5, the DTU accesses the internal memory in the same way in all PE types (arrow ② in Figure 5.2). The XferUnit is used in two different situations:

Commands Executing a command for the CU often involves data transfers. For example, sending a message requires loading the message from memory and sending it via the NoC to the recipient. Similarly, reading from external memory requires to transfer the data into the internal memory.

NoC Requests If PE_A has a communication channel to PE_B, PE_B might receive requests from the NoC. For example, if PE_A has a send endpoint referring to a receive endpoint at PE_B, PE_B needs to store incoming messages into its memory.

As depicted in Figure 5.3, the XferUnit supports multiple transfers at the same time by providing multiple *transfer slots*. Each transfer slot maintains the information about the transfer and contains a buffer, which is used as intermediate storage when transferring data between internal memory and NoC packages. In memory elements and type-A PEs, the `addr` field holds a physical address referring to the internal memory. That is, no address translation is performed. In type B and type-C PEs, the `addr` field holds a virtual address, which refers to the virtual address space of the running VPE. Thus, the XferUnit uses the DTU's MMU/TLB to perform the address translation, whose details will be explained in the next section. Note that supporting multiple simultaneous transfers also leads to potentially multiple simultaneous address translations.

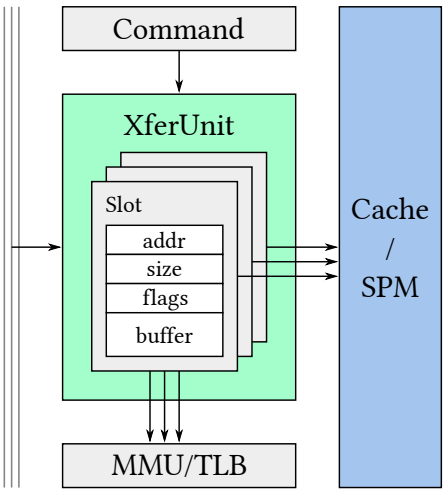


Figure 5.3: The XferUnit

Another responsibility of the XferUnit is to respect cache line boundaries in case the internal memory is a cache. For that reason, accesses are performed in cache-line granularity and the XferUnit takes care of alignment. Supporting arbitrary alignments and sizes in the DTU relieves the CU of aligning data structures accordingly, which simplifies the usage of DTU commands significantly. However, aligning data structures is beneficial for performance.

Page Faults

As explained in more detail in the following sections, transfers in type B and C PEs can cause page faults. Resolving these page faults can lead to new transfers if, for example, page table entries are manipulated (see Section 5.9.3). Without further measures, a deadlock is caused in case all transfer slots are already in use and a new transfer slot is required to free a used transfer slot. To solve this problem, I decided to dedicate the first transfer slot to transfers that cannot page fault. In other words, if a page fault is caused by a transfer in the first slot, the transfer is aborted with an error. The dedicated slot is sufficient, because transfers can only generate new transfers on page faults and resolving page faults cannot cause new page faults. Thus, all transfers that are performed to resolve a page fault specify the "no page fault" flag to indicate that the first transfer slot should be used. The dedicated slot also implies that each DTU needs at least two transfer slots.

5.8 Address Translation

The translation of virtual addresses to physical addresses is handled differently in PE-type B and C. In PE-type B, the DTU is performing the translation autonomously, that is, without involving the CU. In PE-type C, the translation is performed by the MMU of the CU and the DTU contains a software-managed TLB to cache recent translations in the DTU. Thus, this section explains their address translations separately. The virtual-memory management is the same for both PE types, apart from minor details, and will be explained in the following section.

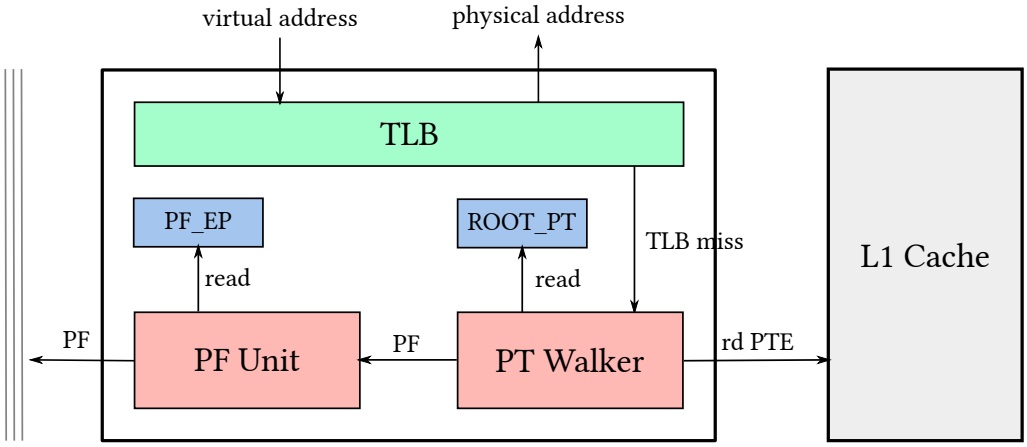


Figure 5.4: The internals of the MMU in the DTU of PE-type B

5.8.1 PE-Type B

As mentioned earlier, for PE-type B, the DTU is extended by address translation functionality, inspired by IOMMUs. IOMMUs are similar in that they are used by devices, but managed remotely by the CPU. The DTU in PE-type B is also used by the CU in this PE, but managed by other PEs. Thus, a key point for both is to not require support by the CU/device. To this end, the DTU performs the address translation with an internal MMU (see Figure 5.2) transparently to the CU. An address translation is performed during transfers with the XferUnit, as explained in the previous section. Additionally, in PE-type B, all memory accesses by the CU (e.g., via load/store instructions) are routed through the DTU to the cache and require an address translation. This section zooms into the MMU and explains its internals.

Overview

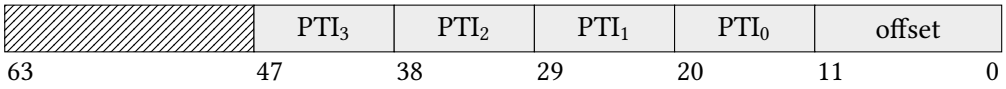
I starts with a high-level perspective on the translation of a virtual address. Figure 5.4 provides an overview of the DTU's MMU. It consists of:

1. the `ROOT_PT` register, holding the physical address of the root page table,
2. the `PF_EP` register, holding the endpoint number for page fault messages,
3. the translation lookaside buffer (TLB) to cache recent translations,
4. the page table walker to resolve TLB misses, and
5. the page fault unit to send page fault messages.

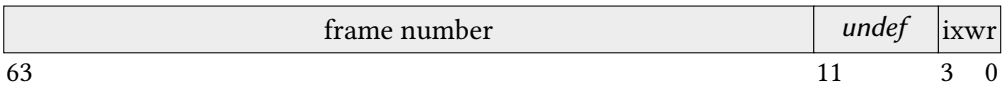
Whenever the DTU receives a virtual address, the MMU is used to translate it to a physical address. To do so, the MMU first consults the TLB in case it translated this page before and still hold the result (*TLB hit*). If not (*TLB miss*), the TLB asks the page table walker to translate the address. The page table walker loads the corresponding page table entries (PTEs) from the cache and inserts the resulting entry into the TLB, if all PTEs exist and the access is allowed. Afterwards, the resulting physical address is returned to the DTU. In case a PTE is missing or has insufficient permissions, the page fault unit sends a page fault as a message over the NoC via the send endpoint defined by `PF_EP` to resolve the page fault. The following explains this process in more detail.

Page Table Walker

To perform the page table walk, the DTU needs to define the paging-related data structures. Note however, that many of the decisions are purely implementation choices and not inherent to the system architecture. Like other paging mechanisms, I decided to use 4 KiB pages, because this size offers a good trade-off between performance and memory usage. Since NoC addresses are 64 bits, I decided to use 64-bit wide physical addresses as well, which results in 64-bit (8 B) wide PTEs and 512 PTEs (4096/8 = 512) in each page table. Thus, virtual addresses have a 12 bit offset (2^{12} = 4096) and 9 bit per page table index (2^9 = 512). I decided to use four levels of page tables, leading to 48-bit virtual addresses:



PTEs contain the frame number (physical address divided by the page size), which is either the next page table or the frame the page is mapped to. Some of the bits in the offset are used to store the permissions:



The last three bits (xwr) grant execute, write, and read permission, respectively. If any of the bits is set, the PTE is considered valid. If bit 3 (i) is set, the memory can be accessed internally, that is, by the CU. Otherwise, only RDMA request are allowed to access the memory. This is comparable with x86's user/supervisor bit and is primarily intended to let the kernel map PTEs into virtual memory, as will be explained in Section 5.9.3, but hide them from the CU. The bits 4 to 11 are not defined by the DTU and thus freely usable by software.

The page table walker starts the translation with the root page table from the `ROOT_PT` register and loads the PTE at index PTI_3 from the cache (caches are indexed and tagged with physical addresses). If the permissions are sufficient, the page table walker continues with the page table at the physical address set in the just loaded PTE and so on until the last level is reached. If the translation was successful, the page number and the PTE of the last level are inserted into the TLB.

Page Fault Unit

As mentioned, if a PTE has insufficient permissions, the page table walker turns to the page fault unit to resolve the page fault. Since I strive to keep the address translation transparent to the CU, the page fault has to be resolved without involving the CU. For that reason, the page fault unit sends page faults as messages to a predefined *pager* that resolves the fault. The pager will be described in more detail in Section 5.9.4. The message contains the virtual address and the required permissions and is sent via the endpoint given by the register `PF_EP`. If the fault has been resolved (e.g., by creating PTEs), the pager sends a reply, on which behalf the page table walker retries the translation. Note that the pager knows the VPE for which the page fault should be resolved from the label of the message (see Section 3.4.2), which has been defined by the pager at the VPE's creation.

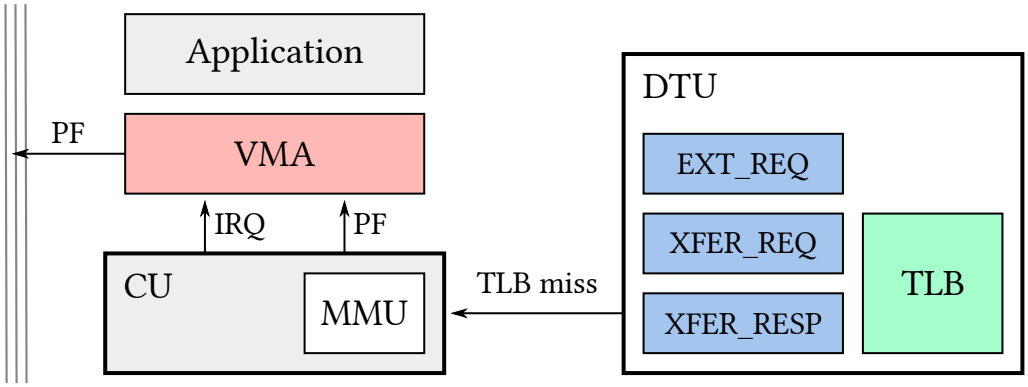


Figure 5.5: Address translation in PE-type C

Translation Lookaside Buffer

The translation lookaside buffer (TLB) is responsible for caching translation results, which is essential for performance since every instruction requires at least one memory access (to load the instruction itself). The current implementation consists of a configurable number of entries, whereas each entry stores the page number, frame number, and permission bits. Additionally, it stores the LRU sequence number as the TLB uses the least-recently-used (LRU) replacement strategy. To keep the cached translations in sync with the page tables, the DTU offers external commands to flush the TLB entry for a specific page number and to flush the complete TLB (to load a new VPE).

5.8.2 PE-Type C

In contrast to type-B PEs, the CU in type-C PEs is expected to have a MMU to translate virtual addresses to physical addresses. The goal is to use such a CU without any modification and let the DTU access the virtual address space defined by the MMU. As illustrated in Figure 5.5, the basic idea to achieve this goal is to integrate a small TLB into the DTU, which is managed by a software running on the CU, called *virtual-memory assistant* (VMA), via a register interface provided by the DTU (blue in the figure).

Physical vs. NoC Addresses

If an existing core is used without modification, its MMU will translate virtual address to physical addresses. However, as outlined in Section 5.6, the physical address needs to be converted to a NoC address on last-level cache misses. For example, current x86-64 cores use 48-bit physical addresses. To use such a core without modification, the 48-bit physical addresses are converted to NoC addresses by the DTU. Currently, a simple translation scheme is used, which dedicates 8-bit of the physical address to the element ID and 40-bit to the offset and widens the offset to 56-bit when translating a physical address to a NoC address. As a consequence, such a core can only map the first 1 TiB (2^{40}) of memory in each memory element into virtual memory. To access the memory beyond 1 TiB, the core can use the DTU's RDMA feature, though.

Virtual-Memory Access

To serve an incoming RDMA request to the virtual memory of the running VPE, the DTU uses the XferUnit, as outlined in Section 5.7. Since the RDMA request refers to a virtual

address, the XferUnit needs to translate this address to a physical address before being able to send a request to the DTUCache. The XferUnit first consults the TLB in case the translation for this page has been done before. On a TLB hit, the translation is finished and the DTUCache access can be performed. In contrast to PE-type B, TLB misses are handled in software by the VMA. On a TLB miss, the XferUnit writes the virtual address, the requested permissions, and the transfer slot id to the `XFER_REQ` register and injects an interrupt request (IRQ) into the CU. After the VMA has performed the address translation, the VMA needs to write the physical address, permissions, and transfer slot id to the `XFER_RESP` register. Upon this register write, the XferUnit will insert a corresponding entry into its TLB and continue with the transfer.

Since the XferUnit has multiple transfer slots, multiple transfers can request an address translation simultaneously. Thus, on a second translation, the XferUnit has to wait until the VMA has read the `XFER_REQ` register for the first translation. For that reason, the VMA is expected to write zero to the register as soon as it has read its content, which notifies the XferUnit to request the second translation. If the address translation fails, the VMA is expected to write the transfer slot id without any permissions to `XFER_RESP`. In this case, the XferUnit will abort the transfer.

Virtual-Memory Assistant

As described in Section 3.5.3, the kernel loads a CU-specific helper onto the target PE at the start of each VPE. The CU-specific helper is responsible of the initialization of the CU and contains two modules, as depicted in Figure 5.6. One of these modules is the virtual-memory assistant (VMA). To protect page table entries from applications, the CU-specific helper is running in the privileged CPU mode on type-C PEs. The VMA module is responsible for the CU-specific part of virtual-memory management. The current prototype contains an implementation for x86-64.

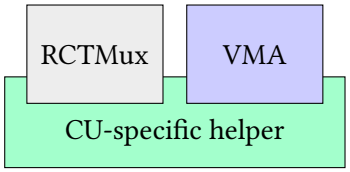


Figure 5.6: Modules of the CU-specific helper

The DTU reports TLB misses to the VMA via an interrupt vector dedicated to the DTU. The VMA first reads the `XFER_REQ` register. If it is non-zero, the VMA sets it to zero and translates the virtual address by performing a page table walk in software. Afterwards the VMA writes the result to `XFER_RESP` as described earlier.

Besides handling the TLB misses of the DTU, the VMA is also responsible for handling page faults caused by the MMU. If the MMU causes a page fault exception, the VMA sends a corresponding page fault message to the pager, just as the page fault unit in PE-type B does in hardware. Afterwards, the VMA waits for the reply of the pager, upon which the execution of the application is resumed. The VMA follows the same protocol in case the page table walk on behalf of a TLB miss causes a page fault.

As pointed out earlier, the DTU might request another translation once the VMA has set `XFER_REQ` to zero. For that reason, the VMA disables interrupts by default to delay the handling of the next translation until the completion of the current translation. However, when sending a page fault message and waiting for its reply, interrupts need to be enabled. The reason is, that the handling of the page fault might again cause address translation requests. For example, the page fault message itself might cause a TLB miss if the DTU tries to load it from the VPE's virtual address space. Thus, the VMA handles address translations during the page fault handling by enabling interrupts for this period of time. If more page faults are caused during this time, the VMA queues them and services them as soon as the current page fault handling is finished.

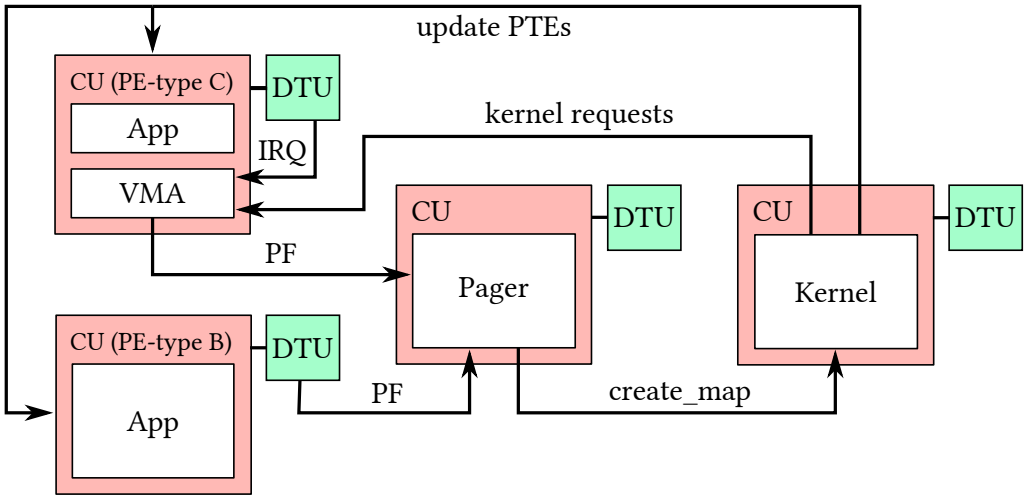


Figure 5.7: Overview of the virtual-memory management, with type B and C PEs sharing the pager and kernel

Another problem the VMA has to solve is that page fault messages are sent via a DTU command (with the send endpoint given by the `PF_EP` register), but commands can cause page faults. First, this implies that the page fault message and its reply cannot cause another page fault, requiring to store them in pinned memory. Second, for example a `READ` command of the application might page fault if the DTU tries to store the read data at the specified virtual address. Since the DTU supports only one command at the time, the VMA cannot send the page fault message to the pager, because the `READ` command is still in progress. To solve that, the VMA soft aborts the current command before sending the page fault message and retries the command afterwards.

5.9 Virtual-Memory Management

The previous section described the ways address translation is performed in PE-type B and PE-type C: in PE-type B, TLB miss resolution and page fault messaging is done completely in hardware by the DTU and transparently to the CU. In PE-type C, both is done in software by the virtual-memory assistant (VMA) and the DTU only contains a software-managed TLB to cache recent translations. In both cases, page faults are reported via messages to the pager, which is responsible for resolving the faults. This section describes how the pager manages the virtual memory of applications.

5.9.1 Overview

As illustrated in Figure 5.7, page faults are resolved by two components: the pager and the M^3 kernel. The kernel is responsible for the security-critical operations such as page table manipulation, because it would allow applications to access arbitrary physical memory. The potentially complex part of virtual-memory management is done by the pager in userland. In other words, the kernel offers a mechanism for page table manipulation and the userland implements policies based on it, as in L4 [83, 137].

Every VPE has its own address space. VPEs on type-A PEs have a physical address space (the SPM in the PE), whereas VPEs on type B and C PEs have a virtual address space. In case of the latter, the VPE optionally has a pager. The pager allows the VPE to map anonymous memory or files and benefit from virtual-memory techniques such as

demand loading and copy on write. Page faults are sent to the pager, which will typically allocate new physical memory and instruct the kernel to create page table entries with the `create_mapping` system call (see Section 5.9.2 for details). Afterwards, the pager sends a reply, which finishes the page fault handling and resumes the execution of the VPE. If the VPE runs on a type-C PE, the kernel sends *kernel requests* to the VMA for operations that cannot be handled remotely (e.g., flush TLB entries).

Note that VPEs can also benefit from virtual memory without using a pager by allocating physical memory and mapping it into their virtual address space via the `create_mapping` system call. However, the system call maps the physical memory eagerly and does not support page faults to implement virtual-memory techniques such as copy on write, demand loading, swapping, and so on. For example, the pager itself can use virtual memory without a pager.

5.9.2 Mapping Capabilities

Memory gates were introduced in Section 3.5.4 as kernel objects to access PE-external memory via the DTU in an RDMA-like fashion. A *memory gate capability* or *memory capability* for short is a capability that references a memory gate. A memory gate is byte granular and can either refer to physical memory or the virtual memory of a VPE. For that reason, the memory gate stores the NoC address, the VPE id, the number of bytes, and the access permissions. To support virtual memory, I introduced another type of capability called *mapping capability*. A mapping capability references a kernel object called *mapping*, consisting of the NoC address, the number of pages and the access permissions. Note that mapping capabilities and memory capabilities seem similar, because they share some of their fields, but are fundamentally different: mapping capabilities represent page table entries (see below) and the memory is accessed via load and store instructions. In contrast, memory capabilities do not represent page table entries and can only be used via the DTU and thus require to configure a DTU endpoint before the memory can be accessed.

Mapping capabilities are created via the `create_mapping` system call, which receives a VPE capability (destination address space), a virtual address, a memory capability, the offset and size of the region in the memory capability to use and the permissions. That is, a mapping capability is always based on a memory capability. To prevent mapping loops (e.g., virtual memory referring to itself), the memory capability has to refer to physical memory. The creation of a mapping capability creates the corresponding PTEs for the destination VPE. Consequently, revoking a mapping capability invalidates these PTEs again. Furthermore, revoking a memory capability does also revoke all mapping capabilities that are based on it. However, revoking mapping capabilities does not revoke memory capabilities that refer to this part of the virtual address space. In other words, if VPE_1 creates a memory capability M for a part of its own virtual address space and passes M to VPE_2, VPE_1 should revoke M if the part of the virtual address space is no longer available. Without the revoke, VPE_2 can still try to access M via its DTU, which will either fail if the virtual addresses have not been reused yet or access the new memory.

Note that mapping capabilities cannot be exchanged between VPEs, because all mapping capabilities are based on memory capabilities and only memory capabilities are supported on all PE types. In contrast, mapping capabilities are only supported on PE-type B and C. Hence, it is more attractive to exchange the underlying memory capability to, for example, establish shared memory. If only a subset of the memory capability should be shared, the `derive_mem` system call can be used to create a memory

capability for a subset of the memory and a subset of the access permissions. Using memory capabilities also allows to establish shared memory in a smaller granularity than pages, because memory capabilities are byte granular.

5.9.3 Page Table Entries

As mentioned in the overview, the kernel is responsible for the manipulation of page table entries (PTEs). Since the PEs are potentially heterogeneous, the kernel needs to support multiple page table formats. In the current implementation, the kernel supports the DTU's page table format (PE-type B) and x86-64's format (PE-type C). Alternatively, the kernel can delegate the task of page table manipulation to the VMA (to only support the DTU's format), but for simplicity and because I expect the number of different page table formats in one system to be small, I decided against this approach.

At the creation of mapping capabilities, the kernel creates the corresponding PTEs to establish the mapping. In traditional OSes, PTEs are created by accessing the page table via load and store instructions. However, as described in Section 5.8, both the page table walker of the DTU and the MMU load PTEs from physical memory via the cache. Since the M^3 kernel is running on a different PE, it has a different cache (if at all; maybe it runs on a type-A PE) and I do not postulate cache coherency. The M^3 kernel could write the PTEs to physical memory and invalidate the corresponding cache lines in the user PE to force a reload from physical memory. However, there is a more efficient way to remotely manipulate PTEs: use the RDMA feature of the DTU to directly push them into the cache of the user PE. Unfortunately, page tables are stored in physical memory and RDMA requests refer to the virtual address space on PE-type B and C. Therefore, the kernel makes all page tables accessible via virtual memory. For PE-type B, the kernel clears the internal bit (see Section 5.8.1) in the PTEs to make the page tables only accessible via RDMA. For PE-type C, the kernel uses the user/supervisor bit to make the page tables only accessible for the VMA. In summary, the kernel writes a PTE by calculating the PTE's virtual address and sending an RDMA write request to the user PE that writes to this virtual address.

Manipulating PTEs also requires to set the root page table and to flush TLB entries in case permissions are downgraded. On PE-type B, the kernel performs an RDMA write request to the `ROOT_PT` register of the DTU to set the root page table and uses an external command to invalidate TLB entries. On PE-type C, the kernel uses the external command as well to invalidate the TLB entries in the DTU. However, the TLB entries in the MMU of the CU also need to be invalidated, which cannot be done remotely (for x86-64). Similarly, the root page table (`CR3`) cannot be set remotely. For that reason, the kernel instructs the VMA to perform these operations via the already mentioned kernel requests. In the current implementation, kernel requests are done by injecting an IRQ in the user PE and polling until the VMA acknowledges the completion of the request. Since all kernel requests are completed in a few hundred cycles, I consider the solution acceptable.

5.9.4 Pager

The pager is an M^3 service, which manages the virtual address space of its clients (VPEs). On VPE creation, the creator decides whether a pager is used and if yes, which one. If a pager is used, a session is created at the pager and the pager obtains the VPE capability and the memory capability of the virtual address space to manage. The pager offers an RPC interface to create and remove mappings. The general design of the pager is

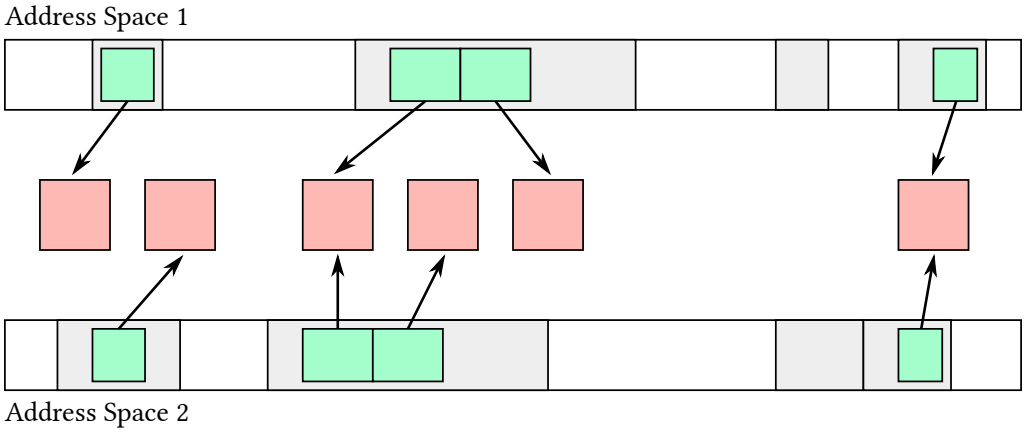


Figure 5.8: The data structures of the pager: each address space consists of dataspaces (grey), which in turn consist of areas (green) pointing to physical memory (red).

similar as on L4 [83, 137], with the difference that M^3's current pager implementation does not support multiple pagers per address space. This is supported on L4 via the region manager indirection, which can be added to M^3 in the future.

Figure 5.8 illustrates the data structures that are used to manage the virtual memory of the pager's clients. Each address space contains a list of so-called *dataspaces*. A dataspace is an abstraction for anything that contains data (anonymous memory, files, I/O memory, ...). For example, the typical M^3 application has one dataspace for its code, one for static data, one for the heap, and one for the stack. The dataspaces for code and static data are file dataspaces, whereas the other two are anonymous dataspaces. The pager manages each dataspace with so-called *areas* to allocate and copy physical memory in smaller granularity than the dataspace size, but in larger granularity than pages. Areas have a variable size and point to a physical memory object, which may be shared by multiple areas.

Application Loading

Section 3.5.3 described how applications are loaded on type-A PEs: code and data is eagerly loaded from the executable and copied into the SPM before the application is started. Application loading is done by libm3, which offers the method `VPE::exec` to execute a given executable in a VPE. Since type B and C PEs support virtual memory, demand loading is used to reduce the loading times of applications. In other words, `VPE::exec` uses the pager's RPC interface to create the dataspaces for code, data, and so on, and starts the VPE. Creating a dataspace only creates the data structure, which stores the virtual address, the size, and the access permissions. As soon as the VPE is started, it will trigger a page fault. If the page fault occurred in an anonymous dataspace, the pager allocates physical memory. If it occurred in a file dataspace, the pager obtains memory capabilities from M^{3}FS for the corresponding part of the file. In both cases, the pager creates a new area in the dataspace and uses the `create_mapping` system call to create PTEs for the physical memory. To improve the performance, the pager tries to map multiple pages at once.

Application Cloning

The other operation supported by libm3 to load applications is `VPE::run`, which copies the current state of the application and executes a given function in a child VPE. Analogously to application loading, if the child VPE runs on a type-A PE, libm3 eagerly copies the entire state into the SPM of that PE. If the child VPE runs on a type B or C PE, the state is copied on demand. Since I do not postulate coherent caches, I decided to implement *copy-on-access* instead of copy-on-write. The reason is that providing the child read access to a page is already similarly expensive as copying the page, because the corresponding cache lines need to be flushed first. Otherwise, the child gets access to stale data if the parent has dirty cache lines. For that reason, the pager copies the data on the first read *or* write access instead of the first write access. However, without coherent caches the pager cannot simply copy the memory via `memcpy`. Instead, the physical memory object stores which VPE has a writable copy of that memory. If there is such a VPE, the pager copies the memory via RDMA from the VPE's virtual address space and therefore, from the PE's caches. Otherwise, the pager copies the memory via RDMA from the physical memory in the memory element.

5.9.5 Message Passing

If a message is sent between DTUs, the receiving DTU stores the message into the receive buffer that is associated with the message's destination receive endpoint. As mentioned in Chapter 3, receive buffers are allocated by the kernel to ensure that they reside in pinned memory. Consider the following example to understand the reasoning behind the decision. Let us assume that an application sends a system call message to the kernel and instructs the DTU to use a receive endpoint, whose receive buffer is not mapped, to receive the kernel's reply. Consequently, trying to store the kernel's reply into the receive buffer at application side causes a page fault, leading to a message to the pager to resolve the page fault. As explained in the previous sections, the pager might need the kernel to create page table entries. However, the kernel is still busy with the reply to the application, because the DTU's `SEND` and `REPLY` commands are synchronous, leading to a deadlock. To prevent such deadlocks, I decided to forbid page faults in receive buffers. Note that this model is similar to L4's model [83, 137], which pins message buffers to prevent "nested messages": a page fault during message passing would cause another message to the pager to resolve the page fault.

5.10 Interplay

After the description of data transfers, address translation, and virtual-memory management, this section explains their interplay. Let us consider a DTU `READ` command as an example: VPE1 performs a `READ` command with a memory endpoint that refers to the virtual address space of VPE2. The existence of such a memory endpoint means that VPE2 has delegated a corresponding memory capability to VPE1. Let us assume that VPE2 runs on a type-C PE and uses a pager. Figure 5.9 illustrates a possible sequence of events for the DTU `READ` command. To focus on the relevant components for this example, the figure shows only the DTUs of the participating VPEs (DTU1 and DTU2), the virtual-memory assistant (VMA) of VPE2, the pager, and the kernel.

The sequence starts at DTU1, which sends the read request (1) to DTU2. The activity of DTU1 ends at this point, because the DTU handles these operations asynchronously. DTU2 accepts the read request, allocates a transfer slot in the XferUnit and starts the

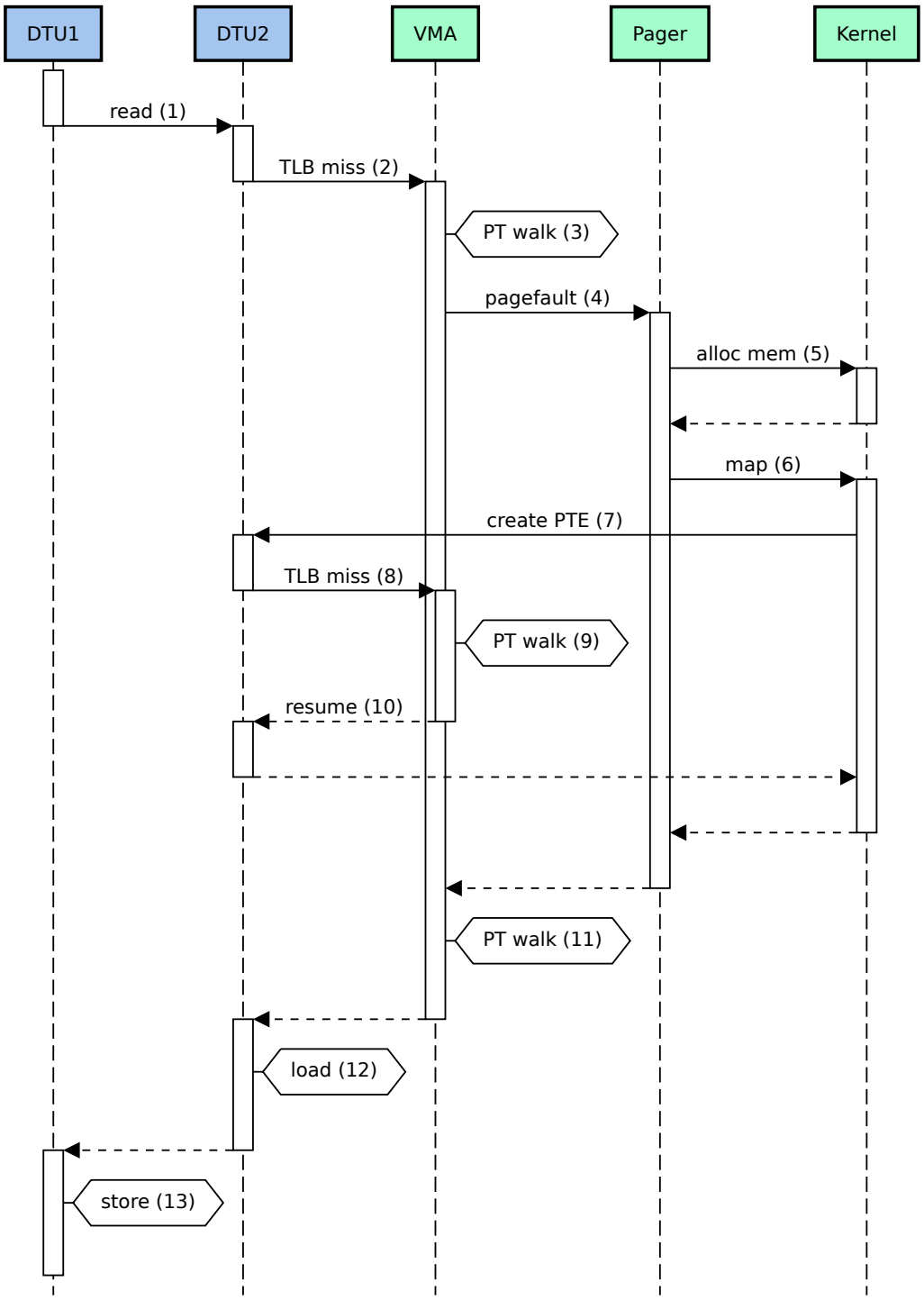


Figure 5.9: The sequence of events for a RDMA read from virtual memory. The numbers in parentheses are used to refer to the individual steps in the text.

transfer. In this example, the targeted region in virtual memory has not been touched yet. Hence, the translation causes a TLB miss in DTU2 and activates the VMA by injecting a TLB-miss IRQ (2). The VMA notices the invalid page table entry (PTE) during the page table walk (3) and sends a page fault message (4) to the pager. To resolve the page fault, the pager first allocates new physical memory (5) from the kernel (we assume here that the DTU read refers to an anonymous dataspace) and uses the `create_mapping` system call (6) afterwards to create the PTE. The kernel creates the PTE via an RDMA write request (7) to the corresponding virtual address in VPE2's address space (see Section 5.9.3). This requires a new transfer slot in DTU2, besides the already occupied slot for the read request. Since the kernel knows that this transfer will not and cannot page fault, it specifies the "no page fault" flag for this DTU write request, instructing DTU2 to use the first transfer slot (see Section 5.7).

In this example, the DTU transfer that creates the PTE also leads to a TLB miss in DTU2. Thus, another TLB-miss IRQ (8) is injected, causing a nested interrupt in the VMA. In this case, the page table walk (9) raises no page fault (if so, the VMA does not send a page fault message, but reports an error because the first transfer slot may not cause page faults), so that the VMA reports the resulting physical address to the DTU, which resumes the transfer (10). After the PTE has been written, the kernel replies to the pager, which in turn replies to the VMA. The VMA repeats the page table walk (11) and reports the resulting physical address to the DTU. Afterwards, the DTU resumes the transfer, which now loads the requested data (12) from the cache using the received physical address and sends the response to DTU1. Finally, DTU1 stores the loaded data (13) at the desired place in VPE1's address space.

5.11 Revisiting the TCB

As this chapter has increased the complexity of the system significantly, this section revisits the discussion of the trusted computing base (TCB).

In Section 3.3.5, I argued that the CUs including their memories are not part of the system's TCB. Instead, only the DTUs, the NoC, and the kernel's CU are part of the system's TCB. For PE-type B (and A) this still holds, because the address translation is performed by the DTU without involving the CU. However, PE-type C uses the VMA to perform page table walks and handle page faults for the DTU. Without further measures, the VMA (and the CU) is as powerful as the M^3 kernel, because the VMA can set the root page table and has write access to all page tables[3]. This allows the VMA to access all physical memory in the system. However, the DTU handles all last-level cache misses and already converts physical addresses to NoC addresses. This can be extended to restrict the physical memory access of the CU. For example, the DTU can use a dedicated endpoint to define the range of accessible physical memory for the CU and deny accesses outside this range. This limits the flexibility regarding the sharing of physical memory, but reduces the power of the VMA and the CU to the level of all other CUs and user-level components. Alternatively and similarly to Border Control [109], the DTU can use a protection table to validate the physical memory access on a per-frame basis without performing another translation.

Revisiting the interplay example of Section 5.10 does also show that the kernel requires assistance from the VMA to create page table entries: the VMA may need to handle TLB misses for the DTU. A malicious VMA could refuse to handle the TLB misses,

[3]Current general-purpose cores do not allow to prevent privileged software from accessing parts of the memory such as page table entries.

preventing a completion of the DTU WRITE command issued by the kernel. Thus, the kernel must be prepared to abort these commands after a timeout. Fortunately, a short timeout can be used, because these commands take only a couple of hundred cycles.

As in L4-based systems, an application that uses virtual memory needs to trust the components that are responsible for its management. Let us call them virtual-memory management components (VMMCs) for a moment. Specifically on M^3, the VMMCs are the kernel, the pager, and the VMA (on type C PEs). All VMMCs control the virtual memory of the VPEs they are responsible for and possess therefore complete power over these VPEs. If memory mapped files are used for critical parts of the application (e.g., code or data), M^{3}FS is part of the VMMCs as well.

In contrast to L4, M^3 also supports the access to the virtual memory of other VPEs via the DTU's RDMA feature. Accessing other virtual address spaces requires to also trust their VMMCs to some degree. For example, consider the interplay example of Section 5.10 again: in contrast to VPE1's VMMCs (if it has a virtual address space), the VMMCs of VPE2 cannot access or manipulate VPE1's address space. However, using a malicious pager for VPE2 that never replies to page fault messages, prevents a successful completion of VPE1's DTU READ command. If VPE1 does not trust VPE2 (or its VMMCs), VPE1 can abort the DTU READ command at any time.

5.12 Discussion

As in the previous chapters, this section discusses possible alternatives and extensions to the presented design.

5.12.1 The VMA in Existing OSes

In Section 5.8.2, I presented the VMA as a small piece of software running in the privileged CPU mode as the virtual-memory assistant for the VPE on this PE. It is also possible to integrate the VMA into an existing OS such as Linux or L4. For example, a type-C PE can run a full-fledged L4-based system, represented as a single VPE in M^3 and extended by the VMA to handle TLB misses of the DTU. In this case, the VMA would not send page faults as messages to the M^3 pager, but use L4's memory management to resolve them. This allows L4 applications to use the DTU to communicate with other PEs. To share the DTU among multiple L4 applications, the L4 kernel can mediate the access to the DTU via system calls.

5.12.2 Caches in Type B PEs

In Section 5.5, type-B PEs were introduced with the DTU placed between CU and caches, translating virtual addresses to physical addresses on the way. This section shortly discusses other possible configurations as shown in Figure 5.10. The arrows show where virtual addresses (red) and physical addresses (blue) are used. To keep it simple, only two levels of caches are considered. The leftmost configuration places the DTU between CU and caches, leading to physically tagged caches. The downside of this configuration is that the DTU is placed at a performance-critical position. Without further measures such as a virtually indexed and physically tagged cache, the L1 hit latency increases due to the address translation in the DTU. This problem can be mitigated by increasing the distance of the DTU to the CU as in the two configurations on the right in Figure 5.10. However, an address translation after the first-level cache makes some of the caches virtual. In consequence, maintaining cache coherence requires a reverse translation

from physical addresses to virtual addresses. Furthermore, it leads to aliasing problems, because the same physical address might be mapped at two different virtual addresses. To avoid these problems, I am using the leftmost configuration in the evaluation.

5.12.3 Page Faults in Type B PEs

If a system designer wants to integrate a specific CU into a system as considered in this work, it needs to pick the PE type that fits best for the CU. If the CU is a general-purpose core that has an MMU, PE-type C is typically preferable. In contrast, type-B PEs are primarily intended for CUs that do not have an MMU. However, it is also possible to integrate an x86-64 core into PE-type B, which is for example done in the evaluation in the following section to make the PE types comparable. Doing so leads to interesting effects and problems regarding page faults, that I want to explain shortly in this section.

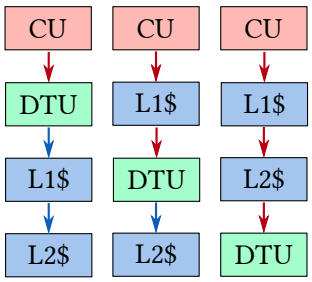


Figure 5.10: Different PE-type B configurations

As already described in the previous section, the DTU in type-B PEs is placed between the CU and the first level cache and performs an address translation on the way. The address translation can lead to a page fault, which the pager will try to resolve, transparently to the CU. The transparent page-fault resolution raises the question what should happen if the page fault cannot be resolved. Currently, the pager terminates the VPE, but it is also imaginable to allow the VPE to handle unresolved page faults, similarly to the `SIGSEGV` signal on UNIX-like OSes. However, existing general-purpose cores typically only support to raise interrupts externally, but not exceptions. Unfortunately, interrupts are not guaranteed to be issued immediately, which is required if the handler of unresolved page faults should be able to continue the execution at the failed load/store instruction. Thus, to support such a handler, the CU could raise an exception upon failed cache accesses.

The opportunity to raise such an exception would also improve the situation for CUs that perform speculative execution. Speculative execution generates loads and stores to more or less arbitrary addresses, which can lead to page faults. In traditional architectures, speculative execution is performed by the same entity as the address translation (and protection check): the core. Thus, if the protection check fails for the speculatively executed instruction, the core will flag the instruction correspondingly. If it turns out that the instruction should indeed be executed, the core raises an exception. Otherwise, the instruction is discarded. If the protection check is moved into the DTU, as is done when integrating a corresponding CU into a type-B PE, the only way to flag the instruction correspondingly is to let the DTU pass this information to the CU. Hence, the DTU should be able to report page faults upon cache accesses to the CU. However, existing general-purpose cores do not necessarily support such an interface. For example, the x86 family supports a machine-check exception, which is intended for errors such as system bus errors, ECC errors, and cache errors. But as the machine-check exception is an abort-class exception, it does not allow to resume the execution afterwards [9, p. 3181].

Without the possibility to inform the CU about page faults, applications on type-B PEs generate many page faults due to speculative execution, which leads to a performance degradation. To reduce the number of page faults, I decided to store non-existing mappings in the TLB as well. To this end, I extended the DTU to support a special error code in replies from the pager. If the pager sends this error code as the page fault reply

to the DTU, the DTU inserts an entry into the TLB with a special flag set. If an address causes a TLB hit with the flag set, the DTU will not send a page fault message to the pager, but only pretend the memory access: stores are not performed and loads load zeros. In the current implementation, the pager uses this error code only for accesses to the first page, because it turned out that many misspeculations access the first page and the first page is never mapped. If necessary, this scheme can be extended to all unmapped parts of the address space.

5.12.4 Cache Coherency

In this work, I am using non-coherent caches to prepare for future systems that might not be (globally) cache coherent, as argued in Section 5.2. This section shortly discusses implications of coherent caches.

Maintaining cache coherency requires that all caches use the same cache coherency protocol. Assuming that this is the case and that all caches are physical (see Section 5.12.2), no changes are necessary. However, it is worthwhile to reconsider DTU-based access to memories, which are also accessed via caches (e.g., DRAM). Without cache coherence, DTU-based accesses lead to inconsistencies as well, if one party accesses the memory via the cache and another party via DTU transfers directly from the memory. However, this can be considered acceptable, because the system is non-coherent, requiring manual coordination in any case. If caches are coherent, this behavior is arguably less desirable. To solve the problem, a cache can be placed in front of the memory, which participates in the cache coherency protocol, and DTU transfers can access the memory indirectly through this cache. This solution is transparent to applications and avoids inconsistencies if both cache-based and DTU-based accesses are used.

Note also that M^3 can take advantage of cache coherency. For example, the pager can implement copy-on-write instead of copy-on-access and the M^3 kernel can update page table entries with load and store instructions instead of RDMA requests.

5.13 Evaluation

This evaluation analyzes the implications of virtual-memory support and its basic performance. After describing the measurement setup, I will start by revisiting the system call, file system, and pipe benchmarks of the previous chapters to show their behavior on the three different PE types that are now supported. Afterwards, I will show the performance of page fault handling and application loading in comparison to Linux.

5.13.1 Measurement Setup

This evaluation will primarily focus on the differences between type A, B, and C PEs. All PE types use a single out-of-order x86-64 core clocked at 3 GHz as the CU. Type-A PEs contain scratchpad memory as the internal memory without any caches. Both Type-B and Type-C PEs use 32 KiB L1 instruction cache, 32 KiB L1 data cache, and 256 KiB L2 cache. The same cache configuration is used for Linux. Type-C PEs additionally use an 8 KiB DTUCache. The DTU's TLB has 128 entries in type-B PEs and 32 entries in type-C PEs. The reason is that the DTU in type-B PEs handles *all* memory accesses of the CU, leading to many address translations. In type-C PEs, the DTU's TLB is only used for DTU transfers. In type-B PEs, the L1 hit latency is 5 cycles instead of 4 cycles

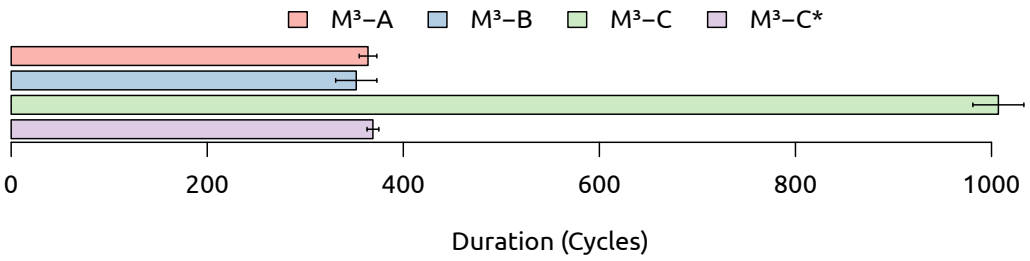


Figure 5.11: System call performance on the different PE types

as in type-C PEs due to the address translation in the DTU. As in the previous chapters, the `DDR3_1600_8x8` model of gem5 is used as the physical memory, clocked at 1 GHz. To ease the interpretation of the results, each system contains only PEs of the same type (e.g., only type-A PEs).

5.13.2 Revisiting System Calls

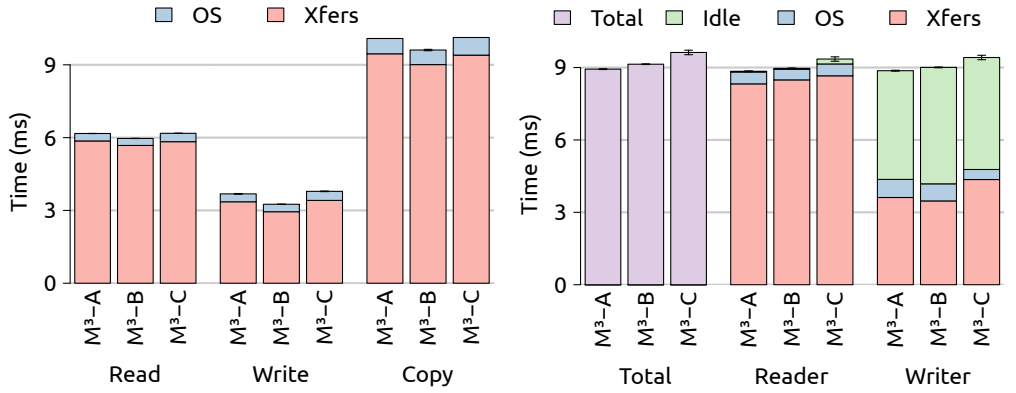
To compare the performance of M^3 system calls on different PE types, I repeated the system call benchmark used in Section 3.7.2. Each system call sends a message to the M^3 kernel, which sends a reply back to the application. The benchmark executes 100 no-op system calls with warm caches. Figure 5.11 shows the results obtained with only type-A PEs ("M^3-A"), only type-B PEs ("M^3-B"), and only type-C PEs ("M^3-C"). The figure shows that the performance is comparable on type-A PEs and type-B PEs, but decreases significantly on type-C PEs.

To analyze the cause of the slowdown, I ran the benchmarks on type-C PEs again, but routed the memory-mapped I/O (MMIO) requests to the DTU's registers from the CU directly to the DTU, bypassing the caches (called "M^3-C*" in the figure). As can be seen in Figure 5.11, this leads to roughly the same performance as on the other PE types. The reason is that in M^3-C, accesses to DTU registers have a latency of about 40 cycles to 50 cycles, because they need to first travel through the caches until they reach the DTU. Since a system call requires quite a few DTU register accesses to send messages, fetch messages, and so on, the duration of a system call increases significantly.

5.13.3 Revisiting File Systems and Pipes

After the system calls benchmark, this section revisits the file system and pipe benchmark of the previous chapter. The file system benchmark reads, writes, and copies a 32 MiB file using a 8 KiB buffer. The pipe benchmark transfers 32 MiB of data from the writer to the reader also using a 8 KiB buffer (at reader and writer) and a 128 KiB shared memory area in DRAM. Figure 5.12a shows the average times of four runs after one warm-up run. As can be seen, all file system benchmarks require roughly the same runtime on all PE types. The reason is that the slower MMIO accesses on PE-type C are less important for data transfers than for message passing, because data transfers require much less accesses to DTU registers. In particular, the relative overhead is smaller, because the data size for data transfers is typically larger than for message transfers, so that DTU `READ` and `WRITE` commands executes longer.

Figure 5.12b shows the average of eight runs of the pipe benchmark. As for the file system benchmarks, the total runtime is roughly the same on all PE types. However, interestingly, the performance is slightly worse on type-B PEs, in contrast to the file system benchmark. The reason is, that the pipe benchmark has two PEs that access



(a) File system performance

(b) Pipe performance

Figure 5.12: File system and pipe performance on different PE types

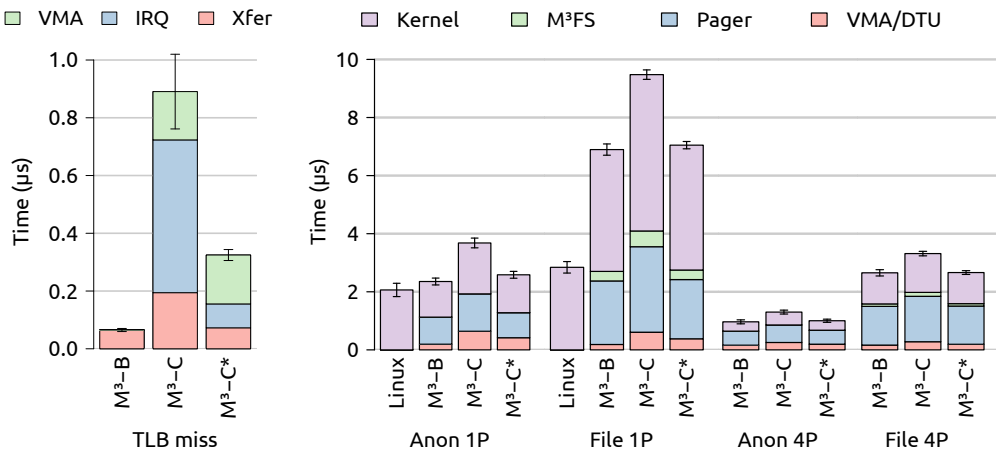
the DRAM with RDMA requests. Hence, the transfer performance depends on whether and how these requests overlap. For that reason, small timing differences (for example, faster MMIO accesses on type B than on type-C PEs) can change the overlap and thus the transfer performance. In this particular benchmark, the overlap on type-B PEs is more beneficial for the transfer performance as on type-C PEs. Another effect that can be seen in Figure 5.12b is that the reader idles more on type-C PEs than on the other PE types. This stems from the fact that MMIO accesses are slower, which increases the time for calls to the pipe server, leading to idle times at the client.

5.13.4 TLB Misses and Page Faults

This section evaluates the time to handle TLB misses and resolve page faults. To measure the time of TLB misses, the benchmark eagerly maps 16 pages of anonymous memory into its address space and reads 8 byte from every page via the DTU. Since the memory has not been touched yet, every `READ` command leads to a TLB miss in the DTU. Figure 5.13a shows the time per TLB miss for eight runs after one warm-up run. The figure compares the performance of the different PE types, again including M^3-C*, where DTU register accesses bypass the caches. To understand the differences, the figure splits the time into multiple parts: *VMA* denotes time spent in the VMA, *IRQ* denotes the time from the interrupt injection until the begin of its handling, and *Xfer* denotes the time for the data transfer itself.

On type-B PEs, the DTU performs the page table walk to handle the TLB miss completely in hardware. For that reason, a TLB miss requires only 63 ns. On type-C PEs, the DTU injects an IRQ to let the VMA perform the page table walk. The VMA itself needs about 150 ns, regardless of whether DTU register accesses bypass the caches or not. Interestingly, PE-type C spends the majority of the time waiting until the x86-64 core starts to handle the injected IRQ. To start handling an IRQ, the core needs to drain its pipeline first. Since the benchmark executes a loop at this point, which repeatedly reads a DTU register to wait for the completion of the DTU command, the pipeline typically already contains many load instructions for the DTU register. Thus, if these loads need to travel through the caches first, draining the pipeline costs significantly more time. The pipeline drain is also the main cause for the variation for M^3-C.

The next benchmark compares the handling of page faults between Linux and M^3 on different PE types. Note that a comparison to L4 is not done, because L4Re [83] does not run on gem5 and NOVA's [137] userland NRE is not mature enough. On Linux, the



(a) TLB miss handling

(b) Page fault handling time per page

Figure 5.13: TLB miss and page fault handling performance on different PE types

MMU raises a page fault exception, which is completely handled by the Linux kernel. On M^3, page faults are handled in user space by the pager (involving M^{3}FS for memory mapped files) and the kernel is only responsible for the manipulation of page table entries. The page fault message is either sent by the DTU (type-B PEs) or by the VMA (type-C PEs). Similarly to the TLB-miss benchmark, the page-fault benchmark maps 64 pages into its address space and causes page faults by writing to every page. Figure 5.13b shows the average time per page, using eight runs with warm caches, and shows the contribution of the participating components to the time. I measured both the time for anonymous memory and file-backed memory.

As shown in Figure 5.13b, M^3-B and M^3-C* achieve similar performance than Linux for anonymous memory if one page is mapped at once ("Anon 1P"). M^3-C is significantly slower due to the slower DTU register accesses. The majority of the time is spent in the kernel and the pager. The kernel needs to allocate new physical memory and is responsible for the page table manipulation. The pager spends most of the time with overwriting the allocated memory with zeros. Page faults for memory mapped files are significantly slower on M^3 in all configurations. The main reason is that M^3 is a microkernel-based system, requiring some indirections and multiple components to resolve each page fault. However, in contrast to Linux, page fault handling is based on memory capabilities (e.g., the pager obtains a memory capability from M^{3}FS). Thus, mapping larger pieces of memory at once is both natural and more efficient. As shown in Figure 5.13b, even mapping only four pages at once ("Anon 4P" and "File 4P"), leads to comparable or better performance than on Linux. Comparing the runtime distribution of a single page with multiple pages also shows that the pager needs roughly the same time per page, whereas the kernel needs significantly less time when mapping multiple pages at once. The reason is that the pager needs to copy the memory from the file to anonymous memory first (as it is mapped with write permissions), which scales linearly with the number of pages. The time for page table manipulation or physical memory allocation depends much less on the number of pages.

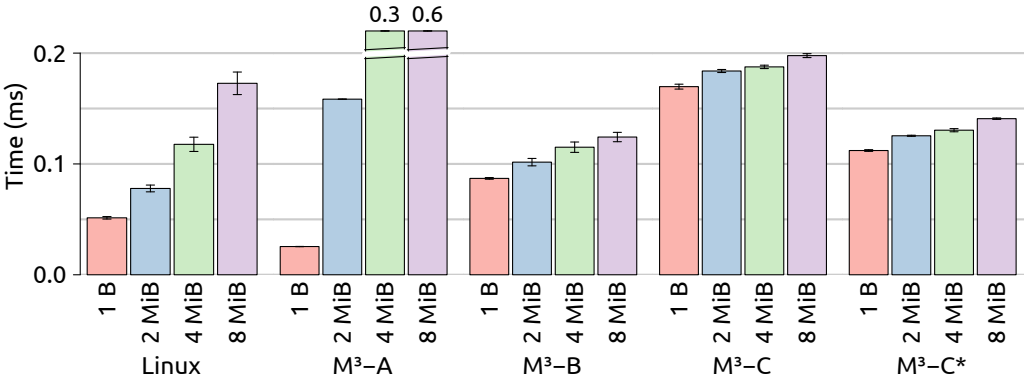


Figure 5.14: Performance comparison of `fork` and `VPE::run` on different PE types and with varying application sizes.

5.13.5 VPE::run and VPE::exec

M³ supports `VPE::run` and `VPE::exec` to clone and load applications. Due to the similarities to `fork` and `exec` on Linux, this section compares their performance. As described in Section 5.9.4, `VPE::run` clones the current state of the application and executes a given function in another VPE. If virtual memory is supported, copy-on-access is used to clone the state, similar to copy-on-write for `fork` on Linux. `VPE::exec` loads a new application from the file system into another VPE. If virtual memory is supported, demand loading is used on M³, as for `exec` on Linux. In contrast to `exec` on Linux, `VPE::exec` can currently only be used on other VPEs, not on the own VPE. It should also be mentioned, that `fork` and `exec` on Linux offer more features than their equivalent on M³. For these reasons, the results of the comparison should be interpreted with a grain of salt. Nevertheless, comparing them puts the performance of M³'s operations into perspective.

First, I compare the performance of `VPE::run` and `fork`. On M³, the benchmark creates a new VPE and calls `VPE::run`. On Linux, the benchmark calls `fork`. In both cases, the measurement is started before the VPE creation or `fork` and is stopped as soon as the child starts executing. The benchmark has an array of varying size (1 B to 8 MiB) in its static data segment, which is initialized before the measurement, to evaluate the influence of the application's size on the performance. Figure 5.14 shows the average of four runs after one warm-up run. M³ shows comparable performance to Linux on type-B PEs, but is significantly slower on type-C PEs due to the slower DTU register accesses. As can be seen, the performance depends on the application's size in all configurations. On all configurations except PE-type A, the reason is that copy-on-access/copy-on-write requires to set all writable pages to read-only. On M³, this is only done for the parent and all pages for the child are created on demand. On Linux, all page table entries of the parent are set to read-only *and* are copied to the child, which is the main reason why Linux's performance scales worse with the application size in this benchmark. On PE-type A, all data needs to be copied eagerly due to the missing virtual-memory support, leading to bad performance for large applications. However, this is acceptable, because the scratchpad memory in type-A PEs is typically in the order of 100 KiB, limiting the application size anyways.

The comparison of `VPE::exec` and `exec` has been done similarly. On M³, the benchmark creates a new VPE and calls `VPE::exec` to execute an application. On Linux, the benchmark calls `vfork` and also executes an application in the child process. On

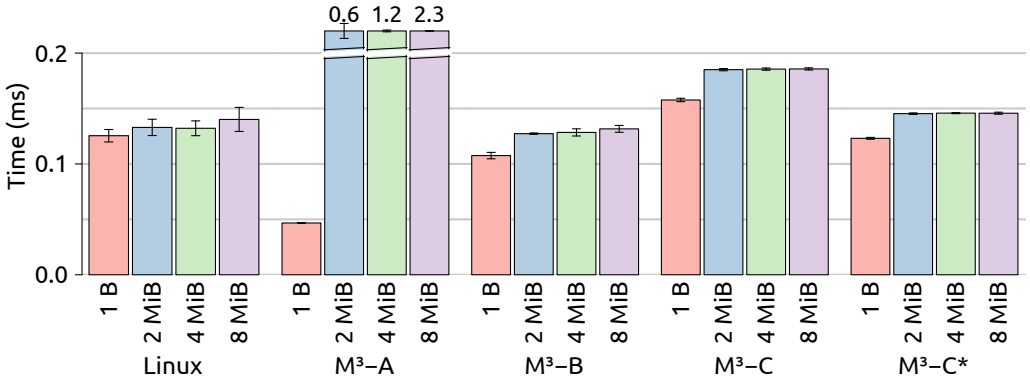


Figure 5.15: Performance comparison of `vfork+exec` and `VPE::exec` on different PE types and with varying application sizes.

both OSes, the time is measured from the VPE creation or `vfork` until the child starts executing. In this case, `vfork` instead of `fork` is used to let the child process borrow the parent's address space until the call of `exec`, improving the performance on Linux. Like for the previous benchmark, the application that is executed in the child VPE/process contains an array of varying size (1 B to 8 MiB) in its static data segment. Figure 5.15 shows again the average of four runs after one warm-up run. Since the array of varying size does not need to be cloned in this case and is also not touched, the performance is mostly independent of the application's size (except for M³-A). Similarly to the previous benchmark, M³'s performance is roughly on the same level as Linux's performance. On PE-type A, the entire application needs to be loaded, which costs significantly more time than using demand loading. Note also that on type-A PEs, `VPE::exec` takes much longer than `VPE::run`, because `VPE::exec` requires to load the data first from the file into the parent's address space and to copy it afterwards into the child's address space. In contrast to that, `VPE::run` simply copies the data from the parent's address space into the child's address space.

5.14 Summary

This chapter extended the system by caches and virtual-memory support. To this end, I introduced two new PE types (called PE-type B and PE-type C) to the existing type (called PE-type A). All three PE types have the same external interface to collaborate seamlessly with each other, but differ in the way the CU is attached to the DTU and the internal memory (scratchpad memory or caches). PE-type A is intended for accelerators that prefer untranslated access to scratchpad memory (SPM). PE-type B is intended for accelerators that desire cached-based access to large amounts of data. Since these accelerators typically lack virtual-memory support, the DTU adds the support externally and transparently to the accelerator. Finally, PE-type C is intended for general-purpose cores that already have a memory management unit (MMU). For that reason, the MMU is reused for virtual-memory support and the DTU offloads the translation of virtual addresses to a small component running on the core called virtual-memory assistant (VMA).

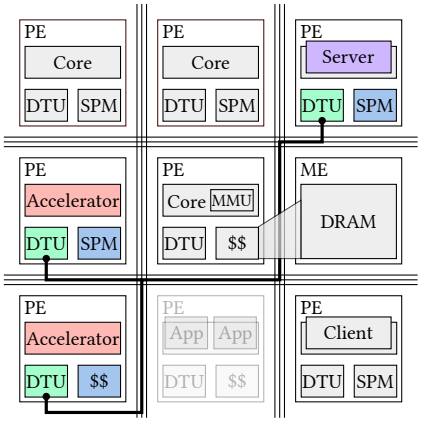
Fortunately, in the remaining chapters of the thesis and when working with the system, the differences between the PE types can mostly be ignored. The reason is that all PE types have the same external interface, hiding these differences. In particular, all PE types accept RDMA requests, which always refer to the address space of the

running VPE: the SPM on PE-type A and the virtual address space in PE-type B and C. Furthermore, page faults are handled by M³'s pager in the same way, independent of whether the paged application is running on PE-type B or PE-type C. The pager receives page faults as messages from the application's PE (either from the VMA or from the DTU) and handles the page faults by using the kernel's mechanism to update page table entries.

The evaluation has shown that the performance of DTU commands is worse on PE-type C, because accessing the DTU's registers is slower. However, this slowdown is mostly negligible in more realistic settings such as file and pipe accesses, which show comparable performance. Comparing the performance of page fault handling and application loading to Linux has also shown that M³ is competitive in this regard.

Chapter 6

Autonomous Accelerators



In the previous chapters, I introduced the architecture that enables the integration of very different kind of compute units (CUs) as first-class citizens. I also introduced different processing element (PE) types that suit different kind of CUs. After focusing on general-purpose cores in the previous chapters, this chapter integrates accelerators into type A and type-B PEs and shows how M^3's concepts enable accelerators to run autonomously.

6.1 Motivation and Related Work

Running accelerators autonomously has multiple benefits. First, accelerators typically offer substantial energy savings over general-purpose cores [69, 86, 91]. However, if accelerators need to be assisted by a typically power-hungry general-purpose core during their operation, the system cannot benefit from the energy savings. For example, Google's TPUs burden their controlling CPU with 11 % to 76 % load just to operate the TPU [69]. Second, if the CPU does not need to assist the accelerator, the CPU can perform other work in the meantime. Third, with an increasing number of accelerators that the CPU needs to assist simultaneously, the CPU becomes the bottleneck.

In this chapter, I first explain how accelerators can access OS services such as file systems or network stacks based on the file protocol introduced in Chapter 4. I also show how this protocol can be used for direct accelerator-to-accelerator communication. There are already specialized solutions that allow access to OS services by a specific type of accelerator, like GPUfs [133] and GPUnet [75] for GPUs or BORPH [135] and ReconOS [26] for FPGAs. However, there is no general solution that would grant any accelerator first-party access to OS services and also allow direct communication between multiple accelerators without assistance by the CPU. CAPI [15] uses a similar approach to integrate accelerators into a system, but focuses on cache coherency and address translation, whereas I am focusing on protocols to access OS services.

Second, I show how fine-grained interruptibility can be combined with autonomous operation. If a system wants to support multiple activities with different priorities on a single accelerator, a low-latency context switch to the prioritized activity is needed. However, accelerators are typically invoked by software and are not interruptible until the computation is complete. One way to lower the latency is to reduce the amount of data per invocation. Consequently, the compute time per invocation is reduced, but software needs to continuously invoke the accelerator, which causes more

CPU utilization and power consumption. Alternatively, the fine-grained invocation can be done in hardware by adding a simple state machine with preemption points next to the accelerator logic. This approach allows to get the best of both worlds: operate accelerators autonomously and interrupt them with low latency. As the context-switching mechanism will be subject of the next chapter, this chapter only explains how accelerators can be interrupted with low latency.

6.2 Accelerator Types

Before I focus on specific accelerators, this section provides an overview of the most important types of accelerators and how they are or can be supported in my system architecture.

6.2.1 Memory Access

At first, accelerators can be categorized by their memory access type, because some accelerators prefer direct memory access (DMA)-based bulk transfers into local scratchpad memory (SPM) and others prefer cache-based memory access [130].

DMA-based Memory Access

Some accelerators perform their computation on local SPM, which requires to first load the data into the SPM via a DMA transfer and to copy the result after the computation from the SPM to main memory. This type of memory access is called *DMA-based memory access* and is often preferable if the accelerated algorithm performs its computation block-wise. For example, a stream-processing accelerator (e.g., for AES encryption) receives a block of data from its source, performs its computation on the block, and sends the resulting block to its sink. Additionally, many accelerators prefer to compute on SPM, because it has a predictable access latency and can easily support many parallel accesses by partitioning the SPM [43, 93, 129, 130, 144]. In my system architecture, PE-type A is the most suitable PE type for DMA-based memory access. DMA transfers can be performed with the DTU's RDMA feature. Since the M^3 kernel configures the memory endpoint to restrict the accessible memory, an untrusted accelerator can issue the RDMA transfers on its own without compromising the system's security. On conventional systems, this is typically done via an IOMMU.

Cache-based Memory Access

Other types of accelerators (e.g., accelerators for matrix multiplication) perform fine-grained and irregular memory accesses to large amounts of data (too large for SPM). In this case, a *cached-based memory access* is preferable, because a cache allows fine-grained memory accesses without the setup overhead of DMA transfers and caches the data locally in case it is accessed again. A cache can also be preferable for computations on smaller amounts of data, if a small cache can capture enough locality to achieve comparable performance to a larger SPM [130]. If the main memory should be securely shared among multiple CUs or the accelerator's access to the memory should be restricted, virtual-memory support is desirable. Like with DMA-based memory accesses, this is typically done via an IOMMU on conventional systems. In my system architecture, PE-type B is intended for this kind of accelerators, in which the DTU supports virtual memory, similar to an IOMMU.

6.2.2 Implementation Paradigm

Apart from the memory access type, accelerators can be distinguished by the used implementation paradigm. The following discusses the most important paradigms.

ASIP

The first way to build an accelerator is to take an existing general-purpose core and add instructions that speed up specific computations. This paradigm is called application-specific instruction-set processor (ASIP). For example, Cadence [22] offers several digital signal processors (DSPs) based on an extensible general-purpose instruction set architecture (ISA). Adding instructions to a general-purpose core requires a flexible ISA such as Xtensa [22] or RISC-V [17], which typically leads to mixed-ISA systems if cores with a high single-thread performance for general-purpose workloads such as x86 or ARM cores are desired as well. Fortunately, the design of M^3 can easily support systems with multiple ISAs. However, this cannot be shown in this work, because gem5 does currently not support the simulation of mixed-ISA systems.

ASIC

The application-specific integrated circuit (ASIC) provides an alternative to the ASIP, which can further improve the accelerators performance and energy efficiency, at the price of less flexibility [28]. In contrast to ASIPs, ASICs do not execute any software, but implement a finite-state machine in hardware, typically designed in a hardware description language such as Verilog [20] or VHDL [21]. For that reason, ASIC-based accelerators are also called *fixed-function accelerators* [129]. Since fixed-function accelerators are the most difficult to integrate as first-class citizens and can be added to gem5 with reasonable effort, this chapter focuses on this implementation paradigm.

FPGA

Field-programmable gate array (FPGA) designs are similar to ASICs in the sense that both are implemented in a hardware description language. In contrast to ASICs, FPGAs are more flexible, because they can be reconfigured at runtime. Due to the similarities, I believe that the concepts that are introduced in this chapter for fixed-function accelerators can also be applied to FPGAs. However, due to the complexity of FPGAs and the missing support in gem5, the integration of FPGAs is left for future work.

GPGPU

Finally, general-purpose computing on graphics processing units (GPGPU) is an attractive and common way to accelerate a specific workload. A GPU consists basically of many cores, where each of them contains a large number of single instruction, multiple data (SIMD) functional units. Typically, each core is split into two groups of functional units to execute two instruction streams at a time. For example, the GeForce GTX 1070 from Nvidia's Pascal generation consists of 15 cores called *streaming multiprocessors* with 128 functional units, called *CUDA cores*, each. These CUDA cores are split into two groups to execute two instruction streams, called *warps*, at a time [14]. I believe that GPUs can be integrated by adding one DTU per group, leading to two DTUs per streaming multiprocessor. However, due to the complexity of GPUs, their integration is left for future work.

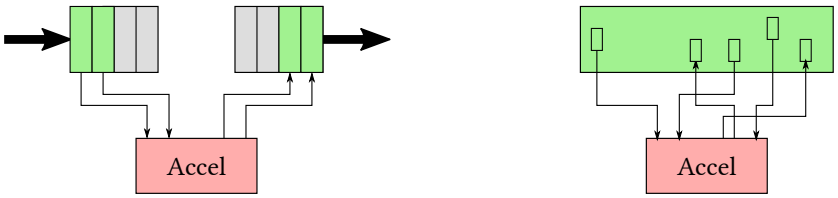


Figure 6.1: Stream processing (left) versus request processing (right)

6.3 Goals

In this work, I focus on fixed-function accelerators for two major reasons. First, for OSes, fixed-function accelerators represent one of the extreme points in the design space of CUs: on the one end of the spectrum, complex general-purpose cores provide all architectural features that are required for an OS kernel. Fixed-function accelerators are on the other end of the spectrum, because they have none of these features and do not even execute software. Hence, in this sense a fixed-function accelerator is the most difficult type of CU to support as a first-class citizen. Second, fixed-function accelerators can be easily integrated into gem5.

Since both memory access types are important, I will show the integration of fixed-function accelerators with DMA-based memory access and fixed-function accelerators that prefer cache-based memory access. For each memory-access type I chose one example use case. For DMA-based memory access, I chose stream processing, because the continuous and block-based arrival of data fits well to DMA-based memory access. The characterizing property of stream processing, as shown on the left in Figure 6.1, is that each block of data in the stream (green in the figure) is seen and processed by the accelerator exactly once. As shown in Figure 6.1, a buffer is used for incoming and outgoing data and the accelerator loads the data block-wise from the input buffer and stores the result of the computation block-wise into output buffer[1].

For cache-based memory access, I chose a use case I call *request processing*, shown on the right of Figure 6.1. In contrast to stream processing, request processing provides the entire data for the computation to the accelerator with a single request. In other words, the accelerator can access the complete data (green in the figure) during the whole computation. To support large requests and fine-grained data accesses, a cache-based memory access is preferable.

Note that the memory-access type for both use cases is not mandatory, but a design choice. Stream processing can be performed with cache-based memory access and request processing can be performed with DMA-based memory access. However, as stream processing typically benefits from DMA-based memory access and request processing typically benefits from cache-based memory access, I do not evaluate the alternatives.

For the integration of both stream-processing accelerators and request-processing accelerators, I strive for the following goals:

Uniformity: The usage of accelerators should be represented as VPEs just like the usage of general-purpose cores is represented as VPEs. This simplifies the usage of accelerators and minimizes the differences between cores and accelerators.

Direct access to OS services: The accelerators should be able to autonomously access OS services such as pipes, files, or network sockets.

[1]The data can also "flow through" the accelerator without the buffers, as will be described later.

Interruptibility with low latency: Despite running autonomously, the accelerators should be interruptible with a low latency in case a more important job for the accelerator arrives.

No changes to existing accelerators: The approach should allow to reuse an existing accelerator logic without modifications.

The following sections explain how these goals are reached, starting in Section 6.4 with an overview on the integration and usage of both types of accelerators. Afterwards, Section 6.5 describes request-processing accelerators in more detail, followed by the details on stream-processing accelerators in Section 6.6.

6.4 Overview

Both considered types of accelerator are integrated into the system as depicted in Figure 6.2. The CU consists of the accelerator logic, performing the computation, and the accelerator support module (ASM). The only requirements on the accelerator logic are, 1) one or more ports to access memory, which will be connected to a cache or scratchpad memory, 2) a way to invoke the logic, and 3) a notification upon its completion. Since these interfaces are always required to make use of an accelerator logic, I believe that an existing accelerator logic can be reused without any modification. The tasks of the ASM are to interact with the DTU, invoke the accelerator logic, and receive the completion notifications of the accelerator logic. In the current implementation, the ASM is simulated as a piece of hardware, because dedicated hardware is typically faster, more energy-efficient, and requires less chip area than a core-based solution. However, it is also imaginable to use a small core to implement the functionality of the ASM in software.

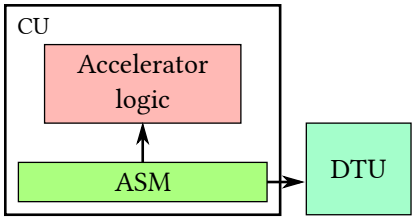


Figure 6.2: Overview of the accelerator integration

6.4.1 Accelerator Usage

Due to the major differences between stream-processing and request-processing accelerators, M^3 provides different interfaces to them for applications. Stream-processing accelerators are well suited to form a chain by connecting multiple accelerators together. As this scheme is similar to UNIX pipelines, M^3 allows to build such chains, containing mixtures of programs and accelerators, in M^3's shell. This allows to execute each step of the pipeline on the CU that suits it best. The details will be described in Section 6.6.3.

Request-processing accelerators are not designed to be connected, because they work on a single request until its completion and notify the application afterwards. The request is split into the data, which is provided in memory, and the meta-data that describes the data and is sent as a message to the accelerator. To this end, M^3 provides a library for applications to use request-processing accelerators.

The differences in the accelerator usage lead to different implementations of the ASM. The ASM for stream-processing accelerators, called SASM, uses the DTU to load data from a source (e.g., a file) into the SPM, invokes the accelerator logic, and sends the result to a sink (e.g., a pipe) via the DTU. In contrast, the ASM for request-processing accelerators, called RASM, waits for a message from the application, invokes the accelerators logic multiple times, and sends the response to the application upon

completion. Additionally, the RASM is interruptible between invocations. Due to the small block sizes of the considered stream-processing accelerators, the invocation of their logic does not need to be interruptible.

6.4.2 VPEs for Accelerators

Although applications use these two types of accelerators in different ways, the M^3 kernel represents the usage of *all* PEs, ranging from general-purpose PEs to accelerator PEs, by virtual processing elements (VPEs). The VPE abstraction simplifies the integration and usage of heterogeneous PEs, because the same concepts and mechanisms are used for all kinds of PEs. For example:

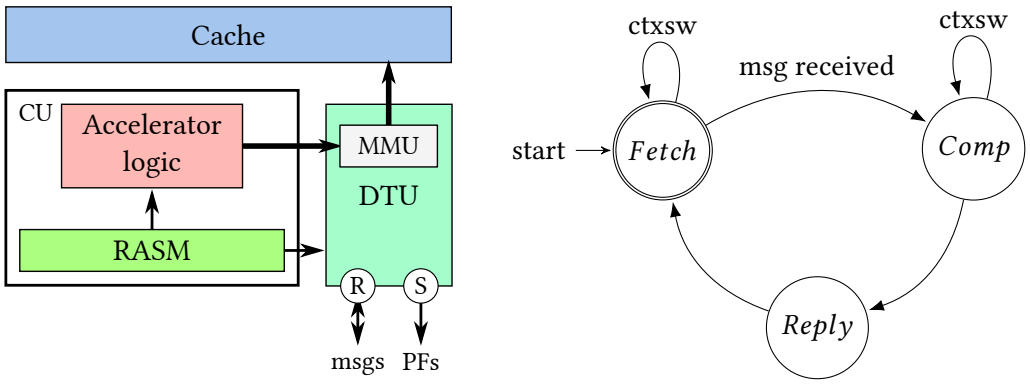
- system calls can be performed by all PEs,
- communication channels between VPEs are established by the kernel in the same way, independent of the PEs to which the VPEs are assigned, and
- context switching is handled in the same way for all PEs, as will be discussed in the next chapter.

Since the VPE abstraction is used for all PEs, the same rules apply to the use of accelerators as to the use of general-purpose cores. For example, spatial isolation is enforced between VPEs and only the kernel can establish communication channels. In particular, if an application creates an accelerator VPE[2], the application can only create communication channels for the accelerator VPE that it can also create for itself. Additionally, the memory of the accelerator is owned by the application while its VPE is active on the accelerator PE. Currently, the VPE is active from its start until its termination. The next chapter on context switching will extend this to allow sharing of accelerator PEs among multiple VPEs with only one VPE being active at a time.

Representing the usage of all PEs as VPEs and providing applications direct access to accelerator's memory has the benefit that the kernel stays small and simple, because the kernel does not need to know the specifics to setup or use accelerators. Instead, the kernel only knows the available PEs in the system, their PE type (A, B, or C), and their CU type (x86, ARM, FFT accelerator, ...). This information is used to find a suitable PE on VPE creation and to know whether and how virtual memory is supported. Applications use the direct access to the accelerator's memory that the VPE provides to setup and use the accelerator. As explained in the previous section, the setup and usage is supported by a library for request-processing accelerators and performed by the shell for stream-processing accelerators.

The direct access to accelerators by untrusted applications is similar to the approach proposed by Gelado et al. [56], which maps FPGA-based accelerators into the virtual address space of applications and uses the virtual memory indirection to dynamically multiplex the FPGA. The direct access by untrusted applications requires to ensure that no cause physical damage can be caused (e.g., by overclocking). If a direct access can lead to physical damage, a trusted component needs to mediate the access to the accelerator, as in other OSes.

[2] Although the current implementation allows every application to create a VPE for every PE, *PE capabilities* can be introduced to control the access to PEs.



(a) Components

(b) Finite state machine of the RASM

Figure 6.3: The integration of request-processing accelerators

6.5 Request-Processing Accelerators

In this and the next section, I will show how two different kinds of accelerators are integrated in a way that achieves the mentioned goals. This section starts with request-processing accelerators.

6.5.1 Integration

The request-processing accelerators considered in this work perform fine-grained accesses to large amounts of data and therefore use cache-based memory access. Figure 6.3a shows the involved components and their interaction to integrate such an accelerator into the system. As described in Section 6.4, the accelerator logic is connected to the request-processing accelerator support module (RASM). The RASM is implemented as a finite state machine that interacts with the DTU and invokes the accelerator logic. The RASM uses one receive endpoint (R in the figure) to receive messages and one send endpoint (S) to send page faults to the pager. The accelerator logic accesses the cache indirectly via the DTU, whose MMU translates the virtual addresses to physical addresses as described in Chapter 5. Page faults that occur during the translation are sent by the DTU via the send endpoint (S) to the pager and are resolved transparently to the accelerator.

Figure 6.3b shows the finite state machine implemented by the RASM. Initially, the RASM is in the *Fetch* state, which waits for incoming messages. As soon as a message arrives at the receive endpoint, the RASM decodes the message, which contains the addresses and lengths of the input and output areas in memory that should be used for the computation. Afterwards, the RASM transitions to the state *Comp*, which invokes the accelerator logic. After the computation, the RASM sends a reply to notify the caller of the completion and transitions back to the *Fetch* state.

6.5.2 Interruptibility

As discussed in Section 6.1, accelerators should be interruptible with low latency, requiring fine-grained invocations. At the same time, accelerators should run autonomously, asking for coarse-grained interactions with software. I achieve both by using the RASM as an indirection. Software performs the coarse-grained invocation of the RASM via message, which in turn invokes the accelerator logic in a fine-grained fashion in state

Comp and is interruptible between these invocations. As will be described in more detail in the next chapter, the kernel sends a signal to the accelerator if a context switch is desired. The RASM checks for the signal in the *Fetch* state and between the invocations in the *Comp* state, as indicated by the loops labeled "ctxsw" in Figure 6.3b. Upon receiving the signal, the RASM acknowledges to the kernel that the accelerator is ready for a context switch.

Since request-processing accelerators use virtual memory, their memory accesses can cause page faults. If the application that uses the accelerator can freely chose the pager to resolve these page faults, a malicious application could chose a pager that refuses to resolve page faults (e.g., sends no reply). In other words, applications could block accelerator PEs forever by causing their logic to wait for a page fault resolution. This problem has two solutions. First, applications can be forced to use a trusted pager, like in other OSes. Second, the RASM can reset the accelerator logic and use the DTU's soft-abort command on context switch requests. Performing the reset immediately requires to repeat the last step of the computation. Alternatively, the worst-case execution time of page fault resolutions can be determined to only reset the logic after a corresponding timeout.

Note that low-latency context switches can require changes to the accelerator logic in some cases. For example, if the amount of data per invocation is hardcoded in the accelerator logic, the time for an invocation cannot be influenced by the component that performs the invocation. In such cases, resetting the logic on a context switch request is an option to achieve low-latency context switches. However, if the accelerator is not idempotent[3], a checkpoint of the input data is required before each accelerator invocation. For example, all accelerators that do not change their input data are idempotent. Note that these problems are not specific to my system architecture, but exist in general.

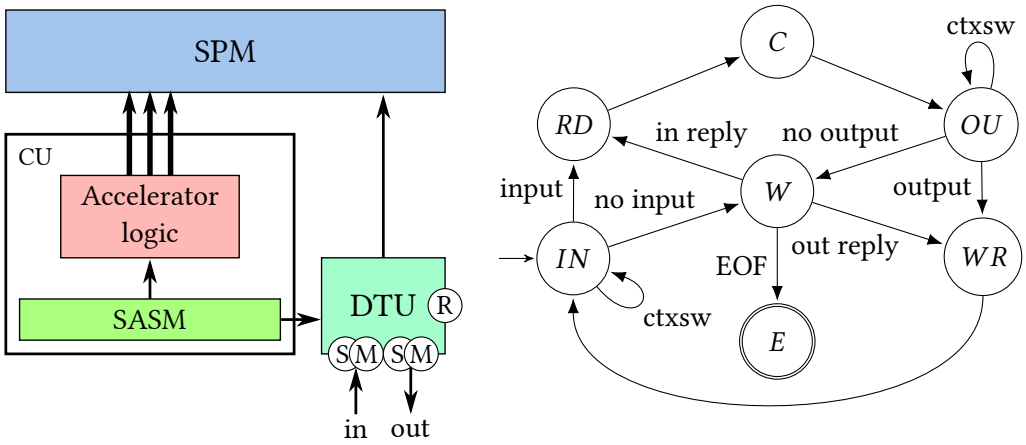
6.5.3 Usage

As explained in Section 6.4.2, applications get direct access to accelerator PEs. Hence, to use an accelerator, the application first creates a VPE for the accelerator PE by specifying its properties (PE type and CU type). If such a PE exists and the M^3 kernel grants the application access, the application creates a pager for the VPE, because request-processing accelerators are integrated in PE-type B and therefore use virtual memory. In the next step, the application maps the input data and the memory for the output by communicating with the pager. For example, the input data can be stored in a file, in which case this file is mapped into the address space of the accelerator VPE.

In the next step, the application sets up the communication channels with the accelerator. To this end, the application let's the M^3 kernel configure the accelerator's receive endpoint (R) to receive messages from the application. Analogously, the accelerator's send endpoint (S) is configured to let the accelerator's DTU send page faults messages to the pager.

Finally, the accelerator setup is complete and the application can start to use the accelerator. To this end, it sends a message to the accelerator's receive endpoint, which describes the mapped memory areas and waits for a reply to this message. Afterwards, the application can repeat these steps or terminate the VPE. Note that the accelerator setup can be provided as a library, simplifying the accelerator usage for applications.

[3] An operation is idempotent if it can be applied multiple times without changing the result.



(a) Components

(b) Finite state machine of the SASM

Figure 6.4: The integration of stream-processing accelerators

6.6 Stream-Processing Accelerators

This section describes the integration and usage of accelerators that perform stream processing, which is used in various domains such as image processing, mobile communication, or audio processing. In these domains, filter chains of multiple CUs are constructed and data is streamed through these chains. As will be described in this section, M^3's concepts allow to construct such chains in the shell and run each element of the chain on the CU that serves it best.

6.6.1 Integration

Stream-processing accelerators are integrated into PE-type A as shown in Figure 6.4a. As for the request-processing accelerators, the accelerator logic is placed next to the stream-processing accelerator support module (SASM). The SASM is implemented as a finite state machine that interacts with the DTU and invokes the accelerator logic. The accelerator logic has direct access to the SPM, because no address translation is performed, and expects the input data to be available in the SPM before the computation. Hence, the SASM needs to load the input data into the SPM before invoking the accelerator logic and needs to copy the result of the computation to somewhere else afterwards. To allow the accelerator the participation in pipelines, the SASM follows the file protocol (see Section 4.3) for both the data source and the data sink. The former is comparable to `stdin` in UNIX-like OSes and the latter to `stdout`. As the file protocol demands, source and sink use two endpoints each: a send endpoint to request access to new data (`S` in Figure 6.4a) and a memory endpoint to access the data (`M`). The receive endpoint (`R`) is used to receive replies to the request messages.

The SASM implements the finite state machine shown in Figure 6.4b. The SASM starts in the state *IN*, which checks whether the input has data left to read. If there is data left to read, it directly transitions to state *RD* to read the next block of data via the memory endpoint into the SPM. Otherwise, the SASM sends an input request (`next_in()`) to the input server to request access to new input data and transitions to state *W*. State *W* waits until a reply arrives at the receive endpoint and transitions to *RD* as soon as the reply to the input request has been received. After the SASM has read the next data block into the SPM, the accelerator logic is invoked and the SASM

transitions to state C, which in turn transitions to OU as soon as the computation has been completed. Analogously to the input phase, OU first checks whether the output area, to which the memory endpoint provides access to, has space left to write the result of the computation. If so, the SASM directly transitions to WR and writes the data and otherwise it first requests new space for the result from the output server (`next_out()`). Afterwards, the SASM transitions back to the state IN, which repeats the procedure until the reply to an input request receives a length of zero, which indicates end-of-file. In this case, the SASM commits the written data by sending `commit(bytes_written)` to the output server and transitions to the state E. Finally, the SASM sends the exit system call to the kernel using the system call send endpoint that the M³ kernel configures for all VPEs and stops.

Similarly to the RASM, the SASM is ready for a context switch at specific points in the state machine, indicated by the loops labeled "ctxsw" at state IN and OUT in Figure 6.4b. Thus, the SASM checks for the context switching signal in these states.

6.6.2 Direct Data Exchange

The solution presented so far has the disadvantage that multiple accelerators that are connected to a chain exchange their data indirectly via pipe based on a shared memory area in PE-external memory. The longer the chains are, the more time and energy is wasted due to this indirection. For that reason, I extended the SASM presented in Section 6.6.1 to enable a direct exchange of the data between the SPMs of the accelerator PEs. The idea is to let the writer push its result directly into the SPM of the reader, removing the need for the reader to load the data into the SPM. This is achieved by transitioning into state W during the computation as well (instead of C) and handling incoming requests for input or output from the neighbors according to the file protocol. An output request of the predecessor is answered as soon as the own SPM has space and the input request of the successor is answered as soon as the result has been produced and has been written to the successor. For simplicity, the accelerator's SASM assumes that the endpoints have already been configured accordingly.

6.6.3 Shell Extension

Since stream processing with accelerators is similar to UNIX pipelines, I extended M³'s shell to allow the construction of such pipelines from the shell. To execute a command such as "`preproc input | accel1 | accel2 > output`", the shell needs to distinguish between parts that should be executed on programmable CUs and parts that should be executed on non-programmable CUs. To this end, I introduced an *accelerator description file*, similar to a shell script, that starts with "`@=`", followed by the name of the accelerator and potentially configuration options in the future. After determining the required CU types based on the executables or accelerator description files, the shell creates VPEs for the corresponding PE and CU types. In the next step, the shell opens files or creates pipes as their `stdin` and `stdout`. For programmable CUs, the shell delegates the capabilities for the files to the child VPE, whereas for non-programmable CUs the shell configures their endpoints. Two consecutive accelerators are connected directly via the optimization explained previously. Afterwards, all VPEs are started and the shell waits for their completion. Remember that the accelerators perform the exit system call just as the programs on programmable CUs, removing the need to treat accelerators special. Since accelerators are first-class citizens in M³, supporting accelerators in pipelines required adding less than 50 lines of code to the shell.

This concept enables the use of stream-processing accelerators in arbitrary pipelines. Depending on their position within the pipeline, accelerators read from a file, a pipe, or the previous accelerator and write to a file, a pipe, or the next accelerator. Since a pipeline can contain an arbitrary mixture of programs and accelerators, each pipeline stage can be executed on the CU that serves it best.

6.7 Evaluation

This section analyzes the basic properties of the two types of accelerators. I start by explaining how the logic of the two accelerator types is simulated. Afterwards, I compare the performance and system utilization of the traditional approach, that continuously assists the accelerators during their operation, with M^3's approach, that lets accelerators work autonomously.

6.7.1 Accelerator Logic

For both accelerator types, I use Aladdin [129, 130] to simulate the accelerator logic. Aladdin is a power-performance accelerator-modeling framework that can be used to make a design-space exploration for fixed-function accelerators. Aladdin estimates performance, power, and chip area within 0.9 %, 4.9 %, and 6.6 %, respectively, compared to hardware implementations. At the same time, Aladdin performs the estimation based on C Code and a configuration file that specifies the desired loop-unroll factors and memory sizes, which makes Aladdin easy to use. Aladdin exists in two flavors:

1. the original Aladdin [129], simply called "Aladdin" in the following, determines the behavior of an accelerator offline, assuming a constant memory access time. I use Aladdin for the stream-processing accelerators, because of their deterministic execution model and the SPM's constant access time.

2. gem5-Aladdin [130] is integrated with gem5 to accurately simulate the memory accesses using gem5's memory subsystem. I use gem5-Aladdin for the request-processing accelerators, because their memory access times vary due to page faults and cache misses. I slightly adapted gem5-Aladdin to invoke gem5-Aladdin from the RASM and notify the RASM upon completions of gem5-Aladdin.

6.7.2 Request-Processing Accelerators

As described in Section 6.5, software invokes the RASM via message, which in turn invokes the accelerator logic. In this section, I evaluate the impact of the invocation granularity on the performance and CPU wake-up frequency.

I use different accelerator workloads from the MachSuite [119]. MachSuite has been analyzed by gem5-Aladdin with the result that some accelerators benefit from DMA-based memory access and others benefit from cache-based memory access. For this evaluation, I picked the accelerators that benefit from cache-based memory access:

1. Stencil-3D: a three-dimensional stencil computation,
2. MD-KNN: a k-nearest-neighbor computation from molecular dynamics,
3. FFT-1D: a one-dimensional fast Fourier transform, and
4. SPMV: a sparse matrix-vector multiplication.

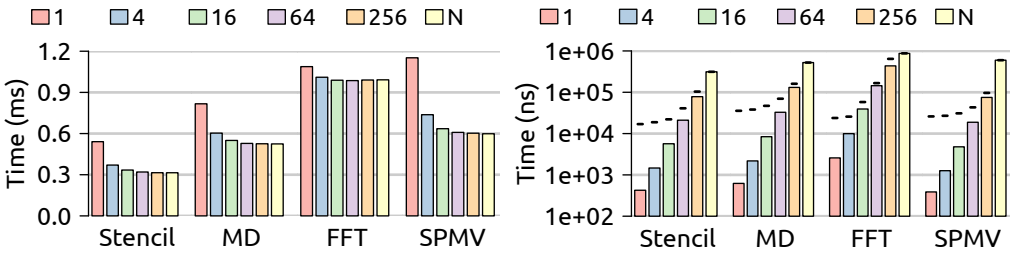


Figure 6.5: Total runtime (left) and the average (bars on the right) and maximum execution times (lines on the right) for different batch sizes

I adjusted each accelerator to perform a single indivisible step per invocation by the RASM. Multiple invocations are batched in a single invocation by the CPU. I analyze the spectrum between assisted and autonomous operation by varying the batch size.

As explained in Section 6.5.3, to use such an accelerator, an application creates a VPE for the desired accelerator, creates the memory mappings for the input and output data in the VPE's virtual address space, and establishes the communication channel to invoke the RASM. In this case, the input data is stored in files and the output data should be written to files as well. Therefore, these files are mapped into the virtual address space of the accelerator VPE.

The application that uses the accelerator and the remaining software components (kernel, M^{3}FS, pager) run on type-C PEs, which are clocked at 3 GHz and contain a single out-of-order x86-64 core, 32 KiB L1 instruction cache, 32 KiB L1 data cache and 256 KiB L2 cache. Each accelerator is integrated into a type-B PE, which is clocked at 1 GHz and contains 32 KiB L1 cache and a TLB with 128 entries. The `DDR3_1600_8x8` model of gem5 is used as the physical memory, clocked at 1 GHz.

Figure 6.5 shows the results from three runs after one warmup run with batch sizes of 1 to 256. Performing all invocations in a single batch is shown as 'N', because the total number of invocations depends on the workload. The standard deviation is less than 1 %. As can be seen in the left part of the figure, larger batch sizes lead to better performance. More importantly, the right part of the figure shows the average (the bars in the figure) and maximum (lines) accelerator execution times for each RASM invocation. For example, running the MD workload with a batch size of 16 shows acceptable performance, because the total runtime does no longer improve significantly with larger batch sizes as shown in the left part of Figure 6.5. However, as shown in the right part of the figure, a batch size of 16 leads to an average execution time of 8 µs and a maximum execution time of 48 µs. In other words, the context switching latency is up to 48 µs and the CPU is woken up every 8 µs on average. High wake-up frequencies are a problem on modern cores, which can only achieve significant power savings in deep sleep states. However, the deeper the sleep state, the longer the time to bring the core back into a functional state (e.g., on Intel's Haswell generation, dozens of microseconds to leave C6 and up to several milliseconds to leave C10 [81, 126]). Hence, deeper sleep states are only beneficial during longer idle periods. As mentioned earlier, resetting the accelerator logic allows to interrupt the accelerator immediately. Resetting the accelerator logic eliminates the context switching latency, but requires to repeat the last batch. In summary, invoking the accelerator logic in a fine-grained fashion from software results in a trade-off. Performing too fine-grained invocations degrades performance and prevents the CPU from entering deep sleep states. Performing too coarse-grained invocations leads to too high context switching latencies or requires to repeat too much work in case the logic is reset.

M³ achieves the best of both worlds by invoking the RASM just once from software and invoking the accelerator logic in a fine-grained and interruptible fashion in hardware. On the one hand, invoking the RASM just once from software (as with batch size 'N') leads to the best performance and longest deep sleep of the CPU. On the other hand, the fine-grained and interruptible invocation of the accelerator logic by the RASM allows to context-switch to a more important VPE with a low latency (as with batch size 1). Resetting the accelerator logic allows an immediate context switch as well. In this case, only a single indivisible step of the accelerator needs to be repeated.

6.7.3 Stream-Processing Accelerators

Stream-processing accelerators use DMA-based memory access to load a block of data from a source (e.g., a file or network socket), perform the computation on the block, and store the result to a sink. On traditional systems, the access to OS services can only be done in software and therefore, accelerators need to be assisted by a general-purpose core. On M³, accelerators can access OS services autonomously. In this section, I will analyze the differences between these approaches in general by using a chain of accelerators with different lengths and different accelerator speeds. Chapter 8 shows the advantages of the autonomous approach on a real-world scenario.

The benchmark uses an application that creates a chain of accelerators and lets the chain process 4 MiB of data. The data is read from a file and the result is stored into a file as well. I compare three different variants:

Assisted: The first variant tries to resemble the traditional approach. To this end, I developed another ASM that waits for a message, computes for a given number of cycles, and sends a reply. Hence, software is responsible to load the input data into the SPM, start the accelerator via a message, and move the result upon receiving the reply from the SPM to the next accelerator or to the output file.

Auto-Pipes: The second variant uses M³'s approach introduced in Section 6.6, but does not use the direct data exchange. In other words, the first and last accelerator are connected to the input and output file, respectively, and all accelerators are connected via pipes.

Autonomous: The final variant is the same as the previous variant, but uses the direct data exchange between the accelerators as introduced in Section 6.6.2.

The application and the remaining software components (kernel, M³FS, and pipe server) run again on type-C PEs, using the same configuration as in the benchmark with request-processing accelerators. The accelerators are integrated in type-A PEs, as described in Section 6.6.1. These PEs are clocked at 1 GHz and contain 4 KiB SPM.

The results are shown in Figure 6.6 using different chain lengths (1 to 8 accelerators) and different accelerator speeds (0.5 GB/s to 4 GB/s). The left figure shows the total runtime, whereas the right figure shows the CPU time spent in the rest of the system (application, kernel, M³FS, and pipe server), relative to the total runtime. As can be seen in the left figure, the performance of both the assisted and the auto-pipes variant degrades with increasing chain lengths and increasing accelerator speeds. The slowdowns have two different reasons. The performance of the assisted variant degrades primarily, because the load of the application increases with an increasing number of accelerators, as can be seen in the right figure. From a certain number of accelerators onwards, the application is not fast enough to fully utilize the accelerators. Furthermore, each

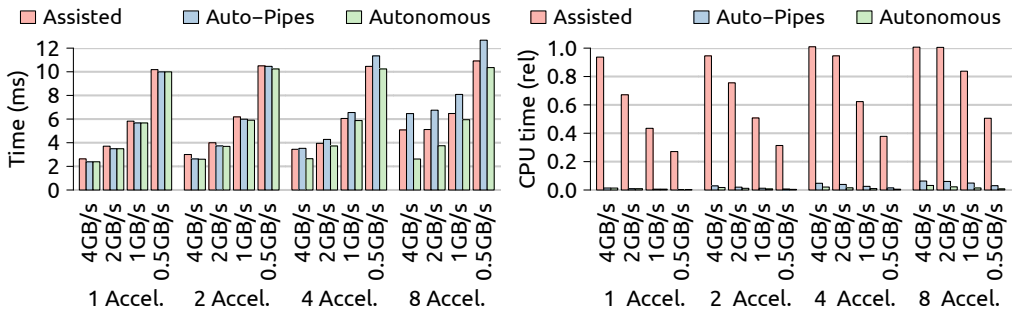


Figure 6.6: Total runtime (left) and the CPU time spent in the rest of the system, relative to the total runtime (right). Both are shown depending on the chain length (1 to 8) and the accelerator speed (0.5 GB/s to 4 GB/s).

invocation introduces overhead. The performance of the auto-pipes variant degrades, because each accelerator that reads from a pipe needs to wait at the beginning of the benchmark until its predecessor has written the data into the pipe. This is done in steps of 128 KiB. With an increasing chain length, these wait times sum up and lead to an increasing performance degradation. Furthermore, since the auto-pipes variant exchanges all data between the accelerators via shared memory areas in DRAM, an increasing chain length and accelerator speed puts more load onto the DRAM, leading to a slowdown. In contrast, the autonomous variant shows the same performance for all chain lengths and increasing accelerator speeds do not lead to performance problems.

The right figure shows the load that is caused in the rest of the system by the accelerator chain. As can be seen, the assisted variant leads to a lot of CPU load (between 0.5 to 1 with 8 accelerators), which is primarily spent in the application that drives the accelerator chain. The CPU time increases with increasing chain length and accelerator speed, because these two factors increase the frequency the accelerators need assistance by the application. Both other variants cause little CPU time in the rest of the system. The auto-pipe variant causes CPU time primarily in the pipe server (between 0.03 to 0.06 with 8 accelerators), which increases slightly with increasing chain length. The autonomous variant causes almost no load in the rest of the system (between 0.01 to 0.03 with 8 accelerators).

The performance and CPU time in the rest of the system can be improved for both the assisted and autonomous variant by increasing the size of the SPM to let the accelerators work autonomously for longer periods of time. However, SPM is expensive in terms of chip area and energy. The autonomous variant removes this trade off, because a small SPM size already achieves the best performance and allows to power-off the rest of the system for almost the entire runtime.

To put the accelerator speeds into perspective, I used Aladdin [129] to determine the computation time for several typical accelerator use cases. Aladdin uses a configuration file to analyze trade-offs between power, chip area, and performance by, for example, specifying loop unrolling factors. To get reasonable results, I generated all sensible configurations and picked the sweet spot between performance and the product of chip area and power consumption. Aladdin reports 6400 cycles to generate a SHA256 hash for 4 KiB of data (0.64 GB/s with 1 GHz clock frequency), 17 000 cycles for 512-point 1D fast Fourier transformation (4 KiB; 0.24 GB/s), 2000 cycles for a general matrix multiplication (GEMM) using a 32 × 32 matrix (8 KiB; 4.17 GB/s), and 1400 cycles for a 9-point 2D stencil on a 32 × 32 image (4 KiB; 2.94 GB/s). The results indicate that the chosen accelerator speeds are in line with typical accelerators.

6.8 Summary

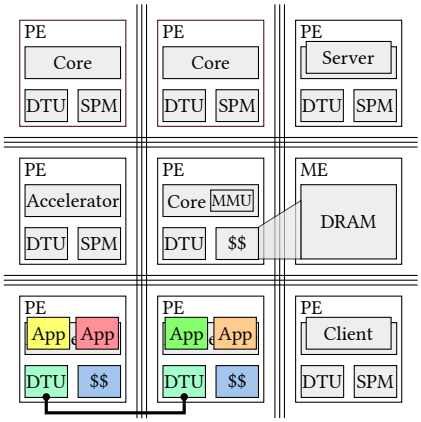
In this chapter, I introduced and discussed two types of accelerators: request-processing accelerators and stream-processing accelerators. Both types are seamlessly integrated into M^3, because the activity on accelerators is represented as VPEs, just as in the case of general-purpose cores. This simplifies the integration and usage of heterogeneous compute units (CUs), because the same abstractions and mechanisms are used for all types of CUs. As a demonstration, I extended M^3's shell to enable the construction of pipelines containing a arbitrary mixture of programs and accelerators.

For both accelerator types, an existing accelerator logic can be reused without modification by placing the logic next to an accelerator support module (ASM), which interacts with the DTU. In particular, the ASM for request-processing accelerators allows to combine fine-grained interruptibility with autonomous operation. The ASM for stream-processing accelerators enables the access to arbitrary file-like objects as sources or sinks. The ASMs can be provided to CU vendors as a library.

In the evaluation, I showed that using the ASM for a fine-grained and interruptible invocation of the accelerator logic achieves the best performance and also allows to interrupt the accelerator with low latency. Furthermore, I showed that running stream-processing accelerators autonomously leads to superior performance with longer accelerator chains and faster accelerators. Finally, I demonstrated that both types of accelerators allow to power off the rest of the system for almost the entire runtime of the accelerators.

Chapter 7

Context Switching



In the previous chapters, M^3 assigned a virtual processing element (VPE) to a specific processing element (PE) at its creation and ran the VPE on this PE without any interruption until its completion. In this chapter, I explain how multiple VPEs can time-share a PE by context switching between them. In particular, the subject of this chapter will be context switching in combination with DTU-based communication that bypasses the kernel and context-switching support on accelerators.

7.1 Motivation

Traditional communication via UNIX pipes, sockets, or L4-like IPC [83, 137] involves the kernel in every communication. For that reason, the kernel can buffer messages until the recipient is ready to receive them, can schedule recipients based on pending messages, and can easily switch to a different thread if the current thread needs to wait for I/O. In contrast, communication on M^3 via the DTU and on DLibOS [96] via Tilera's on-chip network bypasses the kernel with the consequence that the kernel cannot perform these actions. For that reason, DLibOS decided to omit context switching support altogether. I also believe the still increasing core counts and the dark silicon effect [51, 64] will reduce the context-switching frequency and lead to dedicated cores for applications by default. However, in the foreseeable future, there will always be situations where dedicated cores for all applications are not feasible or resources are underutilized. Thus, context switching is necessary to use the system's resources as efficiently as possible.

Combining kernel-bypassing communication with context switching is a hard problem, though. If the kernel is not involved in the communication, another means is required to determine whether the recipient is running. Since both M^3 and DLibOS [96] do not rely on shared memory for communication, the message cannot be stored in case the recipient is not running. However, the naive solution of waking up the recipient and retrying the kernel-bypassing communication is not sufficient. Since these two steps are not atomic, the recipient can be suspended in between, leading to no progress at the sender side. Furthermore, the kernel can no longer make scheduling or placement decisions if it cannot tell whether applications are currently waiting for a message or are doing useful work.

Context switching is not only beneficial for general-purpose cores, but also for accelerators. For example, in multi-tenant cloud environments context switching of accelerators is essential, because a single user will typically underutilize these accelerators. Furthermore, context switching is important to keep wait times and response times minimal. For example, a TPU-like deep-learning accelerator [69] service might

have multiple customers that want to run jobs of different lengths. Without context switching, a short job would have to wait a long time if a long job is already running. I also envision advantages for small embedded and edge devices. Due to their limited hardware resources, these devices benefit from the power efficiency of accelerators and require context switching to flexibly time-share these resources. However, accelerators typically lack the architectural features to run an OS kernel locally that switches between different contexts. Heavyweight hardware features such as single root I/O virtualization (SR-IOV) add a sufficient number of contexts to the hardware that can all be active simultaneously. I decided to keep the hardware simple by using a combination of hardware and software that only requires a single context in hardware.

7.2 Related Work

Sharing accelerators between multiple applications has been explored specifically for GPUs [34, 112, 140]. However, context switching is expensive on GPUs due to the typically large context of hundreds of kilobytes [25, 111]. For that reason, research has been done on alternative ways to support multitasking on GPUs. For example, *Draining* [140] takes advantage of the GPU execution model by preempting the execution at a thread block boundary, so that no context needs to be saved and restored. Chimera [112] uses a technique called *Flushing*, which detects and exploits idempotent execution to instantly preempt a streaming multiprocessor of a GPU.

Similarly, previous work has shown how context switching can be supported on FPGAs [56, 70, 134]. For example, Simmler et al. [134] presented fully preemptive multitasking on FPGAs. In contrast, my approach does not require a preemptible accelerator logic, but the accelerator support module supports interruptions at specific preemption points. Similarly to M^3, Gelado et al. [56] provide untrusted applications access to FPGA-based accelerators by mapping them into their virtual address space. The virtual-memory indirection allows Gelado et al. to dynamically multiplex the FPGA. The M^3 kernel uses VPEs as an indirection to multiplex PEs of arbitrary types.

An alternative to context switching is to add sufficiently many contexts to the hardware. An example is SR-IOV that is supported by recent peripheral devices. SR-IOV has been adopted by the OS community by, for example, Arrakis [114] and OmniX [132] to support peripherals and accelerators. Instead of requiring the architectural features to run an OS kernel, these works assume the hardware to manage multiple contexts. I decided to keep the hardware simple and therefore perform the potentially complex scheduling decisions in software and only simple save and restore actions in hardware.

7.3 Overview

Supporting context switches on accelerators and other types of CUs that do not provide the architectural features to perform these switches locally, requires to perform (at least) the complex part of context switches remotely. Additionally, the state that needs to be saved and restored depends on the CU and might also be inaccessible from a remote CU. Therefore, context-switching support is split in two parts: the potentially complex decisions and security-critical operations are performed by the M^3 kernel, whereas the CU state is saved and restored by the CU the context switch is performed on. Note that this separation is not required for general-purpose cores that typically have the architectural features to perform both parts locally. However, for simplicity I decided to perform context switches on general-purpose cores remotely as well.

Context switching involves four components, depicted in Figure 7.1: the CU of the user PE, the DTU, the *context switcher* (CtxSw) in the M³ kernel, and a small component on the user PEs, called *remotely controlled time multiplexer* (RCTMux). RCTMux's responsibility is to save and restore the CU state (e.g., CPU registers or local memory of accelerators) during a context switch. RCTMux is CU-specific and either a piece of software on programmable CUs or a piece of logic in case of non-programmable accelerators. The M³ kernel maintains one context switcher for each user PE and is responsible for scheduling and placement decisions. The context switcher initiates context switches and RCTMux reacts on its behalf.

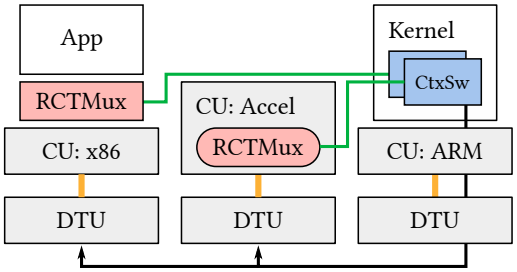


Figure 7.1: The involved components and their interfaces. The grey boxes at the bottom denote hardware, whereas the narrow boxes on top denote software.

These four components have two important interfaces: the DTU-CU interface (orange in Figure 7.1) and the CtxSw-RCTMux interface (green). The DTU-CU interface is used by the kernel to signal a remote CU (indirectly via the remote CU's DTU) about a planned context switch. Depending on the type of CU, the signal injects an interrupt request into a core or notifies the accelerator support module (containing RCTMux).

The CtxSw-RCTMux interface is used for the interaction between RCTMux and the context switcher. Both components use a shared variable for the interaction. The context switcher initiates each interaction by writing a request (e.g., to save the state) to the shared variable and signaling RCTMux via the DTU-CU interface. RCTMux acknowledges the completion of a request to the context switcher by writing to the shared variable as well. If the kernel decides to perform a context switch on a specific user PE, the context switcher first asks RCTMux to save the CU state. Afterwards, the context switcher saves the DTU state of the current VPE, restores the DTU state of the new VPE, and asks RCTMux restore the CU state. The context switcher executes each of these steps individually to allow the kernel to handle other requests (e.g., system calls) in the meantime. For the same reason, the context switcher polls the shared variable for RCTMux's response only a couple of times. If RCTMux has still not written to the shared variable, the context switcher checks the variable again later.

The remainder of this chapter is organized as follows. Section 7.4 explains the necessary steps to combine DTU-based communication that bypasses the kernel with context switching. Afterwards, Section 7.5 describes two implementations of RCTMux: for general-purpose cores and accelerators. Finally, Section 7.6 revisits the trusted computing base (TCB) and discusses whether RCTMux and the CU are part of the TCB.

7.4 Context-Enabled Communication

On M³ and other systems like DLibOS [96], applications can directly communicate with each other without involving the kernel. However, if applications can be *suspended*, for example due to a preemption, other means are required to deliver a message to a suspended application. This section describes the necessary steps to combine such communication with context switching.

7.4.1 VPE-aware Communication

Since communication on M³ is not based on shared memory, messages cannot be delivered if the receiving VPE is suspended. There are two basic solutions to this problem:

1. *Eagerly* invalidate all incoming communication channels to a VPE before suspending the VPE or
2. Keep the communication channels alive, but *lazily* detect communication attempts with suspended VPEs.

The eager approach does not require hardware support, but leads to more context switching overhead that grows linearly with the number of communication channels. In contrast, the lazy approach requires hardware support, but communication channels do not need to be invalidated on context switches. I decided to use the lazy approach, because M³ supports many incoming communications (at most $(n-1) * m$ per PE with n PEs and m endpoints per DTU). Furthermore, many communication channels are typically not used while a VPE is suspended. To this end, I extended the DTU by a register that holds the ID of the running VPE and added the ID of the destination VPE to send endpoints and memory endpoints. On message sends and RDMA-like memory accesses, the DTU adds the destination VPE ID to the request. This allows the DTU in the destination PE to determine whether the request targets the current VPE. Whenever the DTU receives a request to the wrong VPE, it responds with an error that can be handled at the sender PE.

7.4.2 Message Forwarding

Independent of eager invalidation or lazy detection, the DTU reports an error to the sender if the intended recipient is not running. Unfortunately, the naive solution of scheduling the recipient and retrying the communication introduces the following race condition. Since the kernel is not involved in this communication, it does not know when the communication has been completed successfully. If the kernel is suspending the recipient before the communication has been finished, the sender does not make progress. The reason for this problem is that context switching and communication are decoupled, because the kernel performs the context switching, but VPEs bypass the kernel when performing DTU-based communication. For example, if multiple senders try to communicate with multiple recipients scheduled on the same PE, the kernel could decide to schedule the next recipient before the communication with the current recipient has been finished.

I resolve the race condition by falling back to the traditional kernel-based communication model, if a communication failed due to a suspended VPE. The kernel performs both the context switching and the communication: if VPE A receives an error after trying to send a message to VPE B, VPE A asks the kernel to *forward* this message to VPE B. When receiving the forward request, the kernel will first schedule VPE B and afterwards send the message to VPE B. To guarantee progress, the kernel does not suspend VPE B until the message has been successfully delivered to VPE B. Note that the kernel does not perform a context switch on the target PE immediately in case the time slice of the current VPE on the target PE is not depleted yet.

Memory accesses to virtual address spaces via the DTU can fail for the same reason as messages. Thus, the kernel forwards them to the destination VPE on failures as well. However, the kernel can only execute the memory access, if no page faults occur (the

part of the virtual address space might be mapped on demand), because the kernel itself is required to handle the page fault, leading to a deadlock. To solve this problem, the kernel tries the access, but instructs the DTU to abort it if a page fault occurs. In this case, the kernel answers the forward request with an error and asks the VPE to retry the access. Thus, if memory accesses to other virtual address spaces are used and a progress guarantee is required, the memory should be pinned beforehand.

7.4.3 VPE Migration

To prefer direct communication that bypasses the kernel, the kernel should migrate the VPEs among compatible PEs accordingly. Currently, migration is supported if two PEs have the same instruction set architecture (ISA). With compiler support, even cross-ISA migration can be supported [47]. The M^3 kernel migrates VPEs in two situations:

- If two VPEs are scheduled on the same PE and attempt to communicate, the kernel tries to migrate the currently suspended communication partner to another PE. If migration is not possible (e.g., no other compatible PE is available), the kernel instead performs a context switch from the VPE that attempted the communication to the suspended VPE.
- If a VPE is idling (see next section), the kernel tries to work-steal a ready VPE from a compatible PE.

7.4.4 Computing vs. Idling

Another consequence of the DTU-based communication that bypasses the kernel is that the kernel does not know whether a VPE is currently computing or idling (e.g., because the VPE waits for a message). I solve this problem by sending an idle notification in form of a system call to the kernel. Alternatively, the kernel could poll all PEs periodically to check whether the current VPE is performing useful work, but I opted against this solution in favor of a less loaded and more scalable kernel.

As an optimization, I decided to delay the sending of idle notifications by a kernel-defined value to prevent too frequent context switches. For all application VPEs, the kernel sets the idle delay to 20 000 cycles[1]. For server VPEs, the kernel uses an idle delay of 1 cycle with the rational that server VPEs are typically only activated on demand. Hence, switching to an application VPE is more beneficial for the system's performance. Finally, the kernel asks a VPE to not send any idle notifications at all by setting the idle delay to 0 cycles, if there is no ready VPE that can run on its PE.

Note that the kernel cannot force VPEs to report idling. However, threads on traditional systems can also decide to poll instead of using blocking system calls. On both systems, CPU-hogging threads/VPEs can be penalized (e.g., priority degradation) and forcefully preempted.

7.4.5 Gang Scheduling

The described concepts so far allow to suspend VPEs, resume VPEs based on communication attempts, and use the system's resources efficiently by switching to a different VPE in case the current VPE idles. However, if a set of heavily-communicating VPEs contend with other VPEs for the same PEs, a systematic scheduling approach is required

[1] This idle delay turned out to be a good trade-off between context switching too often and overly long idle periods.

to maintain good performance. For example, consider a chain of accelerator VPEs performing stream processing and therefore exchanging messages and data at a high rate. If multiple such chains are contending for the same accelerator PEs, the kernel needs to context switch these VPEs. However, uncoordinated context switching among the VPEs of all chains leads to many failed communication attempts, because VPEs of different chains can run simultaneously.

I solve this problem by introducing a simple form of gang scheduling [110]. To this end, I added a new kernel object called *VPE group* and a system call to create a new VPE group. When creating a new VPE, applications can optionally specify the VPE group. The kernel pins all VPEs within a group on different PEs and schedules them at the same time to let them run simultaneously on different PEs. In this way, multiple sets of heavily-communicating VPEs can efficiently share the same PEs.

7.4.6 Revisiting Command Abortion

If the kernel decides to perform a context switch on a user PE, the DTU might currently be busy with a communication. As described in Section 3.4.7, the DTU supports two types of command abortions. The *soft abort* is used for context switching, while the *hard abort* is used for non-cooperative communication partners. Context switching uses soft aborts, because the communication channel should stay valid. Soft aborts use a timeout to cope with malicious communication partners. If the command is still in progress after the timeout expired, the command is hard aborted, rendering the communication channel unusable.

The DTU supports two different kinds of commands that need to be considered during command abortion: `SEND` and `REPLY` as well as `READ` and `WRITE`. The important difference is that `READ` and `WRITE` can be repeated, which is not possible for `SEND` and `REPLY`. The reason is that every message needs to be delivered to the recipient exactly once. Since receive buffers are pinned (see Section 5.9.5), page faults cannot occur, which leads to a fast message delivery in any case. Therefore, if the current command is `SEND` or `REPLY`, the DTU simply waits until the command is completed or the timeout expired. If the current command is `READ` or `WRITE`, the DTU aborts the command immediately, because it can be retried later. Aborting the command instead of waiting for its completion is important for `READ` and `WRITE`, because these commands can cause page faults, whose resolution can take arbitrary amounts of time.

Another important point of the soft abort is that it disables incoming communication from other unprivileged DTUs. In other words, as soon as RCTMux used the soft abort, all incoming communication requests fail with an error, as mentioned in Section 7.4.1. The M^3 kernel will enable the communication again (for the new VPE) after the context switch has been completed. Privileged DTUs can still communicate with VPEs whose communication has been disabled, to, for example, access the DTU state. Furthermore, outgoing communication is still allowed after the VPE used the soft abort, because the VPE might need to save its state elsewhere (e.g., the accelerator's local memory).

7.5 RCTMux Implementation

In this section, I show two implementations of RCTMux (remotely controlled time multiplexer) for different kind of CUs to show the generality of the context-switching approach. I start with the implementation for general-purpose cores, followed by the implementation for fixed-function accelerators, as introduced in the previous chapter.

7.5.1 General-Purpose Cores

This section describes the implementation of RCTMux for general-purpose cores. Currently, x86-64 and ARMv7 are supported. In general, RCTMux is a module of the CU-specific helper that has been mentioned in earlier chapters, as depicted in Figure 7.2. The CU-specific helper is responsible to initialize the CU to a usable state (e.g., by initializing the CPU registers and configuring interrupts). As described in Section 7.3, the RCTMux module acts on behalf of the M³ kernel to perform context switches. The other module of the CU-specific helper is the virtual-memory assistant (VMA), that was introduced in Chapter 5 on virtual memory to translate virtual addresses to physical addresses on type-C PEs. Both modules are entered upon a specific interrupt to either save or restore the state or perform an address translation. Note also that both modules are optional. RCTMux is only required, if context switching should be supported, whereas the VMA is only required on type-C PEs.

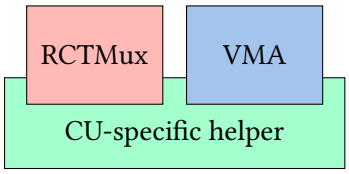


Figure 7.2: Modules of the CU-specific helper

If the VMA module detects a page fault during the address translation, it sends a message to the pager to resolve the page fault and waits for the response. Since the time until the reception of the pager's response is unbounded, this procedure is interruptible. In other words, if the kernel requests a context switching during this procedure, the VMA module will pass the control to RCTMux to save the state and switch to a different VPE. As soon as the VPE is resumed, RCTMux will pass the control back to the VMA to continue the page fault procedure.

On x86-64, the CU-specific helper runs in the privileged CPU mode (ring 0) and sets up interrupt and exception handling during its initialization. To this end, it initializes the Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT) and the Task State Segment (TSS). On ARMv7, the CU-specific helper runs in supervisor mode. In contrast to x86-64, the interrupt table is stored at a fixed address and contains a single instruction per interrupt vector (the instruction jumps to the interrupt handler) and therefore does not need initialization. However, the current implementation for ARMv7 does not implement the VMA module.

7.5.2 Accelerators

In contrast to general-purpose cores, accelerators are not forcefully preempted via an interrupt request. Instead, accelerators receive a signal by the DTU if the M³ kernel plans a context switch. As illustrated in Figure 7.3, RCTMux is part of the accelerator support module (ASM) that I provide to reuse the accelerator logic without modification. The ASM checks for this signal only at convenient points in time, as explained in the previous chapter. For example, request-processing accelerators check the signal after the invocation of the accelerator logic has been completed, because the accelerator logic is not assumed to be interruptible. After receiving the signal, the ASM hands over the control to RCTMux, which reads the shared variable that is used for the communication between the M³ kernel and RCTMux to determine the required action. If the CU state should

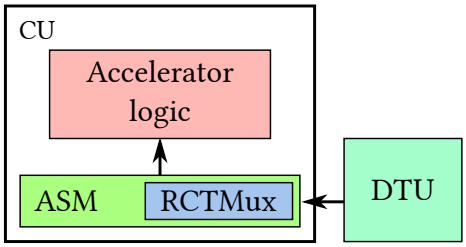


Figure 7.3: RCTMux in accelerators

be saved, RCTMux saves the state of the ASM, which holds, for example, the current position within the memory area the memory endpoint provides access to. For the stream-processing accelerators, it also saves the local scratchpad memory, because the scratchpad is shared among all VPEs on this PE. RCTMux saves the state based on a memory endpoint that has been configured beforehand. The restore action is performed analogously.

Besides the addition of RCTMux to the ASM, the ASM needed to be extended to take communication failures into account. For example, if the ASM sends a message to the file server to request the next piece of input data, the file server might currently be suspended. In such cases, the ASM performs the forward system call to let the kernel forward the message to the recipient.

7.6 Revisiting the TCB

After the description of the context-switching mechanism, this section revisits the discussion on the trusted computing base (TCB). I start with RCTMux's influence on the rest of the system, followed by a discussion of whether RCTMux needs to be protected from applications.

7.6.1 How Powerful is RCTMux?

RCTMux is running in the privileged CPU mode on general-purpose cores, as described in Section 7.5.1, which raises the question whether RCTMux is a privileged component in the overall system. As this chapter is concerned with context switching, I will discuss this question for RCTMux. Note however that all arguments apply for the CU-specific helper as well, because RCTMux is a module of the CU-specific helper. Since RCTMux runs on a user PE, it has an unprivileged DTU. Thus, RCTMux cannot change or create communication channels. In consequence, RCTMux has only influence on the user PE it is running on and is therefore in the TCB of the applications on this PE. However, RCTMux on PE_1 is not in the TCB of an application running on PE_2. Note that on PE-type C, RCTMux's influence is only limited to its own PE if the DTU restricts its access to a specific part of the physical memory, as discussed in Section 5.11. In summary, RCTMux is not privileged in the overall system, but has only power on its own PE.

On accelerators, RCTMux is implemented in hardware (as part of the ASM), just as the accelerator logic. However, for the same reason as for general-purpose cores, the ASM and therefore RCTMux are not privileged in the overall system: the DTU is unprivileged.

7.6.2 Is the Privileged CPU Mode Required?

The discussion in the previous section raises the question why RCTMux needs to run in the privileged CPU mode, if it has no power over the rest of the system. Without further means, RCTMux needs to be protected from the VPEs running on its PE to ensure that these VPEs cannot harm each other. For example, one VPE could otherwise block the PE forever by preventing context switches to other VPEs.

In contrast to traditional kernels, RCTMux and also the CU-specific helper, does not need access to multiple VPEs at the same time. The virtual-memory assistant only performs translations for the currently running VPE. Similarly, RCTMux only needs to save the state of the current VPE and restore the state later. If each VPE has its own instance of the CU-specific helper and the M^3 kernel has a mechanism to remotely switch

between VPEs, the CU-specific helper does not need to be protected. In other words, the CU does not need to enforce its different CPU modes properly (for example due to Meltdown [89] and Spectre [78]), because each VPE can only harm its own CU-specific helper. Thus, a malicious VPE *can* manipulate or destroy its own CU-specific helper, which prevents that the VPE's state is properly stored before a context switch. However, a VPE *cannot* prevent that other VPEs successfully resume their execution on the PE.

To proof the feasibility of this approach, I implemented a CU-specific helper that operates on exactly one VPE. Furthermore, I extended the DTU-CU interface by a *reset* operation that allows the M³ kernel to reset a PE into a defined state at any point in time. On x86-64, the reset sets the root page table (`CR3`) and the instruction pointer. Alternatively, the core could start executing code from a ROM that reads these values from the DTU and sets the core's registers correspondingly.

In summary, the M³ kernel performs a context switch on a user PE by first interrupting the current VPE and asking its CU-specific helper to save the state. Afterwards, the kernel resets the PE, which sets the root page table appropriately for the address space of the new VPE and the instruction pointer to the entry point of the CU-specific helper. Since the CU-specific helper has already initialized the CU, it skips this step and waits for an interrupt by the kernel. The kernel sends another interrupt request to ask the CU-specific helper of the new VPE to restore its state and resume the execution.

7.7 Evaluation

This section evaluates the context-switching concept described in this chapter using micro-benchmarks. The following chapter will look at larger scenarios and application-level benchmarks to analyze the behavior in more realistic settings.

7.7.1 Communication with Suspended VPEs

The first question to answer is: how expensive is a context switch? To evaluate that, I setup a benchmark that performs a remote procedure call (consisting of request and response) with running or suspended VPEs in different settings. The kernel PE and the PE that runs the benchmark always use PE-type C, containing a single out-of-order x86-64 core clocked at 3 GHz with 32 KiB L1 instruction cache, 32 KiB L1 data cache, and 256 KiB L2 cache. The PE of the communication partner varies to show the context switch on all PE types. PE-type A and B contain stream-processing and request-processing accelerator, respectively, and are clocked at 1 GHz. PE-type C is configured as the two other type C PEs. As in the previous chapters, the `DDR3_1600_8x8` model of gem5 is used as the physical memory, clocked at 1 GHz. As a reference point, I show the time for core-local and cross-core IPC on the microkernel/-hypervisor NOVA [137] as well, because NOVA has a well-optimized IPC mechanism. The results for NOVA have been obtained on gem5 with the same core configuration as used in type-C PEs (see Section 3.7.2).

Figure 7.4 shows the average time over 16 runs with warm caches. The times are split into multiple components to explain the behavior. "Wake" denotes the time from the moment the benchmark application fails to send the message to its suspended communication partner until the wakeup of the target PE to perform a context switch. "CtxSw" denotes the time for the actual context switch, "Fwd" the time to forward the message to the communication partner, and "Comm" denotes the remaining time for the communication itself. The first two rows in Figure 7.4 show the time for core-

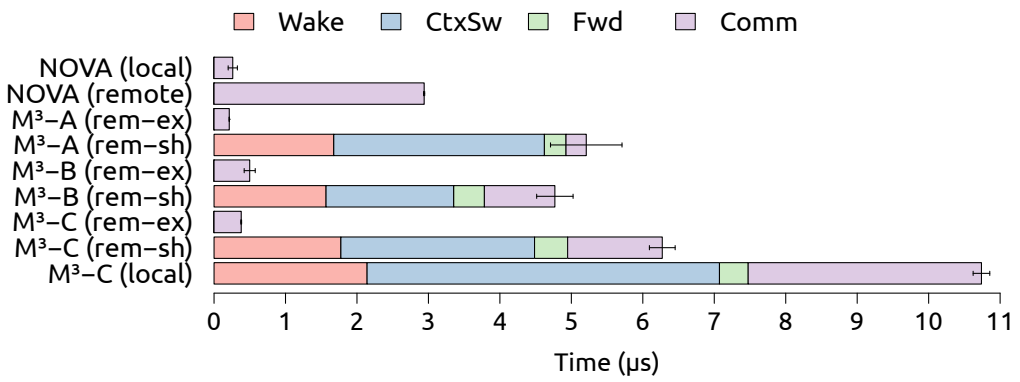


Figure 7.4: Overhead of communication with running and suspended VPEs

local ("local") and cross-core ("remote") communication on NOVA[2]. The next six rows show the times to communicate with a VPE on another PE of different PE types. For each PE type, the first row shows the case where the communication partner has its PE exclusively for itself ("rem-ex"), followed by the case where two VPEs share a PE ("rem-sh"). In the former case, no context switch is required. In the latter case, the benchmark communicates with the two VPEs in an alternating fashion, which requires a context switch for every communication. The final row shows the time for a core-local communication on M^3, requiring two context switches.

As can be seen in Figure 7.4 when comparing the rows labeled "M^3-* (rem-sh)" with the rows labeled "M^3-* (rem-ex)", communication on M^3 is significantly more expensive if the communication partner is suspended due to the context-switching overhead. The overhead is similar on all three PE types, but has different causes. At first, PE-type C is clocked at 3 GHz, while PE-type A and B are clocked at 1 GHz, with the consequence that PE-type C performs more work in a comparable period of time. This stems primarily from the fact that PE-type C executes software, whereas PE-type A and B use a finite state machine that performs the context switch in hardware. The context switch itself is more expensive on PE-type A than on PE-type B, because the content of the scratchpad memory in PE-type A needs to be saved and restored. On the other hand, the communication overhead ("Comm") is larger on PE-type B due to TLB misses (the DTU's TLB is not tagged and hence needs to be flushed on a context switch). The communication overhead on PE-type C is even larger due to the DTU's TLB misses, which are caused if, for example, the DTU needs to store a received message. The reason is that on PE-type C, in contrast to PE-type B, these TLB misses are not handled in hardware, but the DTU injects an interrupt and lets the virtual-memory assistant handle the TLB miss in software.

The core-local communication on M^3 ("M^3-C (local)") is even more expensive, because it requires two context switches: one context switch from the benchmark application that sends the request to the communication partner and another switch to send the reply back to the benchmark application. However, the overhead is less than twice as high, because the M^3 kernel knows that both communication partners share a PE. For that reason the M^3 kernel can directly switch from VPE_1 to VPE_2 without asking VPE_1 first whether it is currently idling, resulting in a shorter wakeup time ("Wake"). On the

[2] On NOVA, the core-local communication requires two context switches. The cross-core communication does not necessarily involve context switches, but has a high overhead due to the use of inter-processor interrupts and kernel entries and exits.

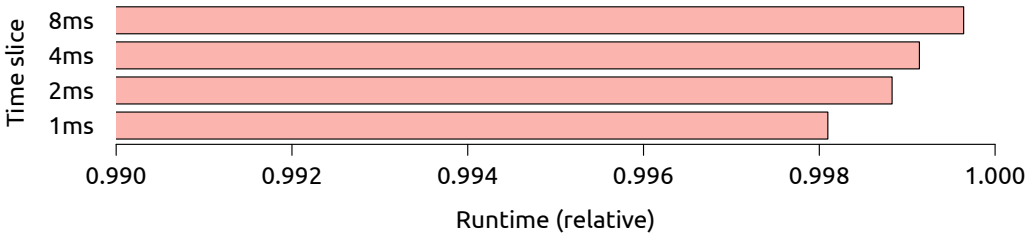


Figure 7.5: Total runtime of a simultaneous execution of two applications compared to a sequential execution, using different time slices

other hand, the communication overhead ("Comm") is larger, because both messages need to be forwarded, leading to more interrupts caused by TLB misses in the DTU.

In summary, the context-switching overhead is much higher on M^3 than on NOVA, because context switches are done remotely based on a communication protocol between the kernel and RCTMux. There is still room for optimizations by, for example, using a tagged TLB in the DTU, though. I would also like to highlight that core-local communication is fast on NOVA, whereas cross-core communication is fast on M^3, if the communication partner is running. This suggests that a combination of these two approaches could be used to get the advantages of both. I will discuss this combination in more detail in the conclusion in Chapter 9.

7.7.2 Non-communicating Applications

After the analysis of the context-switching overhead in the previous section, this section evaluates the impact on non-communicating applications. To this end, I use a compute-intensive application and run this application on a single PE in two ways: first, I run the same application two times in a row and second, I start the same application twice and run them simultaneously, leading context switches between the two applications. To evaluate the context switching overhead, I chose different time slices for the VPEs, resulting in a different number of context switches. In this case, only type-C PEs are used with the same configuration as in the previous section. Figure 7.5 shows the relative runtime of the simultaneous execution compared with the sequential execution, using three runs, preceded by one warm-up run. The standard deviation is less than 3 % and hence not shown in the plot. As can be seen, even relatively short time slices of 1 ms (Linux's completely fair scheduler uses time slices between 0.75 ms and 6 ms by default [12]) introduce almost no runtime overhead.

7.7.3 Communicating Applications

The more interesting workload to analyze the impact of the context switching overhead is a set of communicating applications, because they require more frequent context switches. To this end, I run different pairs of applications that exchange data via pipe. The used applications are:

1. `rand`: produces random numbers and writes them into a pipe,
2. `cat`: reads a file and writes it into a pipe,
3. `wc`: reads data from a pipe and counts the lines, words, and bytes, and
4. `sink`: reads data from a pipe and discards the data.

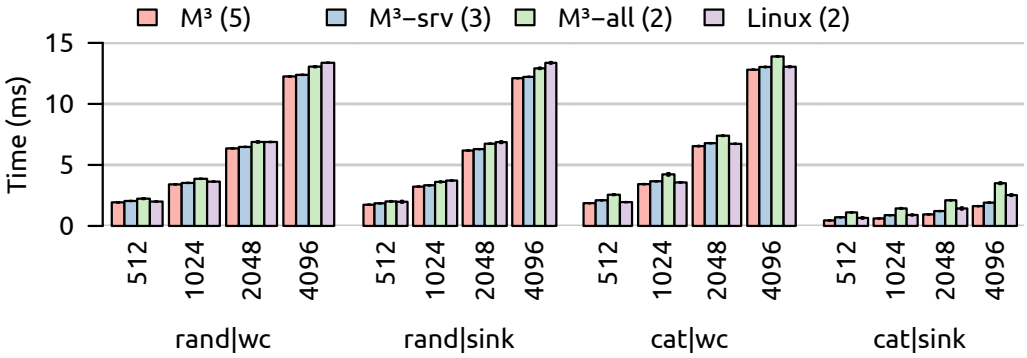


Figure 7.6: Runtime of different applications exchanging a varying amount of data via pipe, with two cores on Linux and different user PE counts on M³. The number of cores/PEs is shown in parentheses.

This set of applications has been chosen to have one compute-intensive application and one I/O-intensive application on each side of the pipe. Each of the four combinations (`rand|wc`, `rand|sink`, `cat|wc`, and `cat|sink`) are run with different amounts of data exchanged via pipe (512 KiB, 1024 KiB, 2048 KiB, and 4096 KiB). I compare the performance to Linux and analyze the behavior on M³ by varying the degree of PE sharing. On both M³ and Linux, the applications use an 8 KiB buffer to read from the pipe or write into the pipe, because 8 KiB achieves the best performance for these benchmarks on Linux. On M³, the pipe uses a 512 KiB shared memory area and the pipe server provides its clients access to at most 128 KiB at once. Figure 7.6 shows the runtime of these benchmarks over three runs, preceded by one warm-up run. The standard deviation is insignificant and hence shown in the plot. The x-axis of the plot shows the different combinations and different data sizes, whereas the y-axis shows the total runtime. For each variant and data size, there are four bars:

1. The first bar shows the runtime on M³ using five user PEs. One PE is used for M³FS, one PE for the pipe server, one PE for the pager, and one PE for each application.
2. The second bar shows the runtime on M³ using three user PEs, letting the servers share a PE. The applications still use one PE each.
3. The third bar shows the runtime on M³ with two user PEs by letting all five VPEs share these two PEs.
4. The fourth bar shows the runtime on Linux with two cores.

All PEs are type C PEs with the same configuration as in the previous sections. Linux runs on a dual-core system, with 32 KiB L1 instruction cache and 32 KiB L1 data cache each and a shared 512 KiB L2 cache. Linux is using 512 KiB instead of 256 KiB L2 cache to make it comparable to M³'s run with two PEs, having 256 KiB L2 each. However, the runtime differences between 256 KiB and 512 KiB L2 cache for Linux are negligible for these benchmarks. Note that M³ always uses an additional PE for the kernel due to M³'s OS design.

As can be seen in Figure 7.6, M³ outperforms Linux if it uses one PE per VPE. Based on the micro-benchmarks from earlier chapters (for example, see Section 4.8), this result is not surprising: M³FS and the pipe server provide their clients direct access to large amounts of data and the clients access this data via the DTU. If more VPEs share a PE as

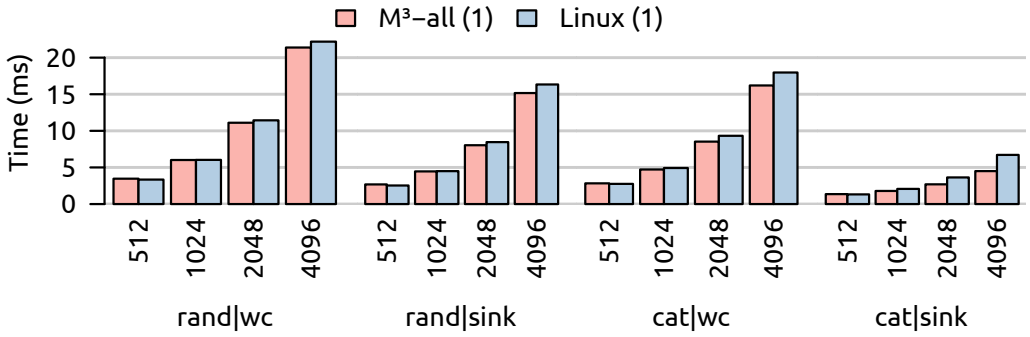


Figure 7.7: Runtime of different applications exchanging a varying amount of data via pipe, with one core on Linux and one user PE on M³

in "M³-srv (3)" and "M³-all (2)", M³'s performance degrades due to the context switching overhead. Interestingly, M³'s performance is still roughly on one level with Linux's performance. The results also show that `cat|sink` is significantly slower on M³ than on Linux. The reason is, that both `cat` and `sink` are I/O intensive, leading to frequent communication with servers and hence frequent context switches. However, I consider this as a pathological case, because doing nearly nothing on both ends of the pipe is rather uncommon. Another interesting result is that the colocation of all servers (M³FS, pipe server, and pager) on one PE as done in "M³-srv (3)" leads to almost no runtime increase.

The next step in this comparison is to run all processes/VPEs on a single core/PE. Since the colocation of reader and writer eliminates their parallelism, the total runtime increases and cannot directly be compared to the runtime with two cores/PEs. Hence, I show these results separately in Figure 7.7, again as the average of three runs, preceded by one warm-up run. The standard deviation is always below 2 % in this case and hence not shown in the plot. Surprisingly, M³ is again slightly faster than Linux if a single core/PE is used for the benchmark with an increasing difference between M³ and Linux if the data size increases. This has two main reasons. First, M³ does not migrate VPEs between PEs anymore, which required costly cache write-backs and invalidates due to the non-coherent caches. Second, when running reader and writer on a single core, Linux switches between reader and writer in the granularity of the application's buffer. In other words, after every 8 KiB of data, Linux performs a context switching (including an address space switch) between reader and writer. In contrast to that, M³'s pipe server provides each client access to 128 KiB at once, leading to fewer context switches.

The final question to answer is the amount of load that is put onto the kernel PE in these different settings. As can be seen in Figure 7.8, if all VPEs run on separate PEs, the kernel is barely used, ignoring the pathological case for now. The kernel load does also not increase much if all servers share a single PE. However, running all five VPEs on two PEs leads to an significant increase. If all five VPEs run on a single PE, the load decreases again, because the kernel has one PE less to perform context switches on. In the pathological case, the kernel load is higher, especially if a small amount of data is exchanged. Note however, that the kernel is loaded for a shorter amount of time, compared to the other benchmarks.

In summary, even if communicating applications share PEs, M³ performs well in most cases. However, the more context switches need to be performed, the more load is put on the kernel. Additionally, if two heavily-communicating VPEs *have to* be run on the same PE, the performance suffers and the kernel load increases. This suggests that

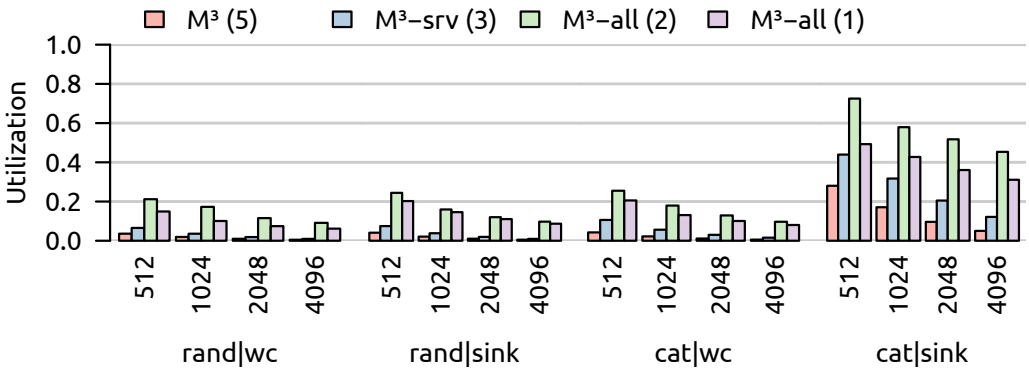


Figure 7.8: Utilization of the kernel PE with different kinds of PE sharing on M³. The number of user PEs is shown in parentheses.

the kernel should track communication channels and the context switching rate to place VPEs in a way that maximizes the performance and minimizes the kernel load. As has been shown in the evaluation, running (non-compute-intensive) servers on one PE has only a small impact on the performance and kernel load, suggesting that a colocation of servers is a good strategy.

7.8 Summary

In this chapter, I explained how multiple VPEs can share a single PE by performing context switches between the VPEs. In contrast to traditional operating systems, the M³ kernel does not perform the context switches on its own PE, but on the user PEs. To this end, M³ uses a component called remote-controlled time-multiplexer (RCTMux) on each user PE for the CU-specific actions (e.g., saving CPU registers). Context switches are always requested by the M³ kernel and RCTMux acts on its behalf. Interestingly, RCTMux is not a privileged component in the system, because the DTU in user PEs is still unprivileged, which prevents RCTMux from adding or modifying communication channels. Additionally, being able to reset a core into a defined state and using one RCTMux instance per VPE allows to run RCTMux in the same protection domain as the VPE. In other words, multiple mutually distrusting VPEs can be run on the same PE without needing to thrust the core to enforce the different CPU modes correctly.

The evaluation has shown that M³'s cross-core context switching mechanism is significantly more expensive than the traditional context switching mechanism that operates on the local core. However, for non-communicating applications, this introduces almost no overhead. For communicating applications, I have shown that M³'s overall performance is still on the same level as Linux's performance, except for pathological cases. I have also shown that context-switching-heavy workloads put significant load onto the kernel PE, suggesting that the kernel should avoid them by assigning VPEs onto PEs according to their communication channels.

Chapter 8

Evaluation

The previous chapters presented the system architecture and evaluated its basic properties by using micro-benchmarks and by focusing on a single aspect of the system. This chapter evaluates the system architecture as a whole and in more realistic settings. In particular, this chapter addresses the following questions:

- What performance does the new system architecture achieve?
- How well does the M^3 kernel scale with the number of user PEs?
- How well does M^3 perform if PEs are shared?
- How does a real-world scenario with accelerators perform and scale?
- How efficient is the sharing of accelerators?
- How complex are the trusted software components of M^3?

8.1 Experimental Setup

Before I start with the evaluation, this section describes the evaluation platform and introduces the system-call tracing infrastructure that is used to replay traces of Linux applications on M^3.

8.1.1 Evaluation Platform

All benchmarks in the evaluation are performed with the cycle-accurate gem5-based prototype platform, using gem5's full-system mode and classical memory system. The different benchmarks use system setups with different mixtures of PEs. However, the configuration of the individual PEs is always the same. All type-C PEs (for general-purpose cores) contain a single out-of-order x86-64 core clocked at 3 GHz with 32 KiB L1 instruction cache, 32 KiB L1 data cache, 256 KiB L2 cache, 8 KiB DTUCache and 32 TLB entries in the DTU. The MMU of the x86-64 core has 128 TLB entries. Type-B PEs (for request-processing accelerators) are clocked at 1 GHz, use 32 KiB L1 cache, and 128 TLB entries in the DTU[1]. Type-A PEs (for stream-processing accelerators) are also clocked at 1 GHz and contain 2 KiB scratchpad memory. All DTUs have 16 endpoints. I use the `DDR3_1600_8x8` model of gem5 as the physical memory, clocked at 1 GHz.

[1]The reason for the different number of TLB entries is that the DTU in type-B PEs handles all memory accesses of the CU, leading to many address translations. In type-C PEs, the DTU's TLB is only used for DTU transfers.

As mentioned in Chapter 1, the prototype implementation supports both x86-64 and ARMv7. However, the implementation of the virtual-memory assistant is still missing for ARMv7 and gem5 does not support the simulation of systems that contain multiple instruction set architectures (ISAs). To prevent a distraction from the important points in this work by a comparison of ISAs, I decided to only evaluate M^3 on x86-64.

Due to the strong coupling of gem5's network-on-chip (NoC) model with the cache coherency protocol and to keep the simulation times manageable, I connect the PEs via a crossbar interconnect instead of a full NoC. A crossbar is feasible for small systems as used in this evaluation, but too expensive for large systems, because of the quadratic growth of wiring. Due to the still long simulation times I used representative, but short-running benchmarks.

Whenever feasible, I compare M^3's performance to Linux 4.10. I chose Linux, because Linux is a widespread and well-optimized operating system and therefore a good performance baseline. Linux is run on a gem5-based single-core or dual-core system, depending on the benchmark. To allow a fair comparison, the single core is configured in the same way as a type-C PE for M^3 and the dual core is also configured in the same way, except that both cores share 512 KiB L2 cache (which benefits Linux). Additionally, Linux uses the same DRAM model.

8.1.2 Systrace Infrastructure

Many benchmarks are using the system-call tracing infrastructure called *systrace*, introduced by Weinhold [148]. Systrace allows to run an application on Linux and trace its system calls to replay them later, for example on M^3. The application is first run with `strace` to determine the performed system calls including their arguments. Afterwards, the application is executed again without `strace` to obtain timestamps before and after each system call (Linux has been modified to support that). Note that these steps are separated due to the overhead of `strace`. In a post-processing step, the system call arguments and timestamps are merged into a data structure that allows to replay the trace on M^3. The *trace player* uses the corresponding API on M^3 for all supported system calls in the trace. The not yet supported system calls and the times between system calls are spent with spinning. In other words, I assume that the unsupported system calls and the application's computation require the same time on Linux and M^3. Note that the unsupported system calls are never important for the application's performance. Since I use the same core configuration for both Linux and M^3, the compute performance is the same on Linux and M^3.

8.2 Performance

This section starts by analyzing the performance of the described system architecture. To this end, I used systrace to trace different applications on Linux and replay them on M^3 to show M^3's performance in comparison to Linux. In a first step, I will show "standalone" applications, followed by pipelines of applications.

8.2.1 Standalone Applications

The first set of benchmarks uses the following applications:

1. *tar*: creates a tar archive with files between 128 KiB to 8192 KiB (16 MiB in total),
2. *untar*: unpacks the same archive,

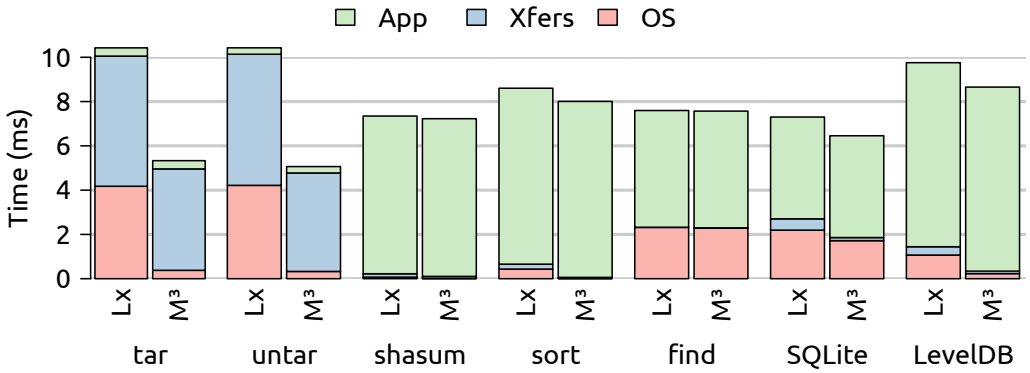


Figure 8.1: Performance comparison to Linux using standalone applications

3. *shasum*: reads a 512 KiB file and computes its SHA256 hash,
4. *sort*: sorts a 256 KiB file with 408 lines,
5. *find*: searches 24 directories with 40 files each,
6. *SQLite*: creates a table and inserts and selects 32 entries, and
7. *LevelDB*: creates a table and inserts and selects 512 entries.

The applications *tar*, *untar*, *shasum*, *sort*, and *find* have been taken from BusyBox 1.26.2 [4]. *SQLite* is an embedded and highly reliable database engine [18]. *LevelDB* is a light-weight and high-performance key-value store, created by Google [11]. The benchmarks for SQLite and LevelDB have been created by myself, using SQLite 3.18.0 and LevelDB 1.19, respectively. This mixture of applications has been chosen to stress the system in different ways: *tar* and *untar* are transfer intensive, *shasum* and *sort* are compute intensive, *find* performs many file-system requests, and *SQLite* and *LevelDB* show a mixture of these three patterns.

Due to the different OS designs, Linux and M³ use a different number of cores/PEs for the benchmarks. I will repeat the comparison with the same number of cores/PEs based on context switching on M³ in Section 8.4. Since all benchmarks are single threaded, Linux uses a single core, whereas M³ uses four PEs: one PE for the kernel, one for the pager, one for M³FS, and one for the application. However, M³ does not take advantage of multiple PEs, that is, at no point in time multiple PEs are doing useful work in parallel. The benchmark uses only type C PEs for M³ and a single core for Linux, all with the default configuration as outlined in Section 8.1. Since M³FS is an in-memory file system, I compared it to Linux's tmpfs. Both M³FS and tmpfs are using a 4 KiB block size. M³FS has been configured to use extents of at most 512 KiB for both existing and created files.

Figure 8.1 shows the average runtime over three runs, preceded by one warm-up run. The runtime does not include the time to start and shutdown the application, because this time is independent of the application's runtime, which is rather short in these benchmarks to keep the simulation times acceptable. The runtime is broken down into the application time ("App"), the time for data transfers ("Xfers"), and the OS overhead ("OS"). As mentioned in Section 8.1, when replaying the trace, M³ spins for the time that is spent in the application on Linux and for the time of system calls that are unsupported[2] on M³. Hence, for both M³ and Linux, the times for these system calls

[2]In these benchmarks, the system calls `access`, `brk`, `chdir`, `chmod`, `chown`, `dup2`, `fchown`, `fcntl`, `fdatasync`, `futex`, `geteuid`, `getpid`, `getrlimit`, `gettimeofday`, `getuid`, `ioctl`, and `utimes` were unsupported on M³. The sum of the times for the ignored system calls were at most 0.4 ms.

are accounted as application time in the plot. The standard deviation is below 3 % and therefore not shown in the plot.

As can be seen in Figure 8.1, Linux and M³ are mostly on the same level, but for *tar* and *untar* M³ outperforms Linux by roughly a factor of two. Having already discussed and analyzed M³'s advantage in data transfers in Chapter 4 on OS services, this result is not surprising: M³ shows both faster transfer times and less OS overhead for the transfers. The former stems from the fact that the DTU has a dedicated data transfer functionality, that (slightly) outperforms a memory-copy operation via caches. The latter is achieved, because M³FS grants its clients access to large contiguous file regions at once. After the access has been granted, the client can directly access the data via the DTU, which has almost no overhead. In contrast, Linux copies the data page by page in the page cache, which results in more OS overhead. Note that the OS overhead is higher on Linux despite the fact that *tar* and *untar* are using the `sendfile` system call to transfer an entire file with one system call.

The remaining benchmarks exhibit smaller performance differences between M³ and Linux. The performance of *shasum* and *sort* is dominated by the application time and hence the overall runtime is comparable on both systems. *find* performs primarily `stat` system calls, which behave similarly on both systems, except that the request to the file system is sent via message on M³ and performed via system call on Linux. *SQLite* and *LevelDB* are again slightly faster on M³ due to M³'s faster data accesses.

8.2.2 Pipelines of Applications

After the standalone applications, this section evaluates the performance of pipelines of applications in comparison to Linux. The following applications are combined:

1. *cat*: reads a 1 MiB file and writes into a pipe,
2. *grep*: reads a 1 MiB file and writes all lines that contain "ipsum" into a pipe,
3. *awk*: reads from a pipe and counts the lines that start with a "D", and
4. *wc*: reads from a pipe and counts the lines, words, and bytes.

The applications have been taken from BusyBox 1.26.2 [4]. Similarly to the standalone applications, Linux uses fewer cores/PEs than M³ due to the different OS designs. Linux is running on two cores to have separate cores for reader and writer, whereas M³ uses six PEs: one for the kernel, one for the pager, one for M³FS, one for the pipe server, and two for the applications. I will repeat these benchmarks with the same number of cores/PEs for Linux/M³ based on context switching on M³ in Section 8.4. As in the previous section, type C PEs are used for M³ with the default configuration. Linux is using the dual-core configuration. On both Linux and M³ the benchmark starts the writer and reader, connects them via a pipe, and pins them on dedicated cores/PEs. On M³, the pipe uses a 512 KiB shared memory area and the writer and reader get access to 128 KiB at a time.

To evaluate the performance, I ran all four combinations of the applications (*cat*|*awk*, *cat*|*wc*, *grep*|*awk*, and *grep*|*wc*). The first two bars for each of these four combinations in Figure 8.2 show the total runtime of the pipeline ("Total") on Linux and M³. The total runtime does again not include the time to start and shutdown the two applications. To analyze and explain the differences between Linux and M³, the next two bars for each of the four combinations show the time for the writer, whereas the final two bars show the time for the reader. The time for reader and writer denotes the time from their start until the `exit` system call. The times of the reader and writer are broken down

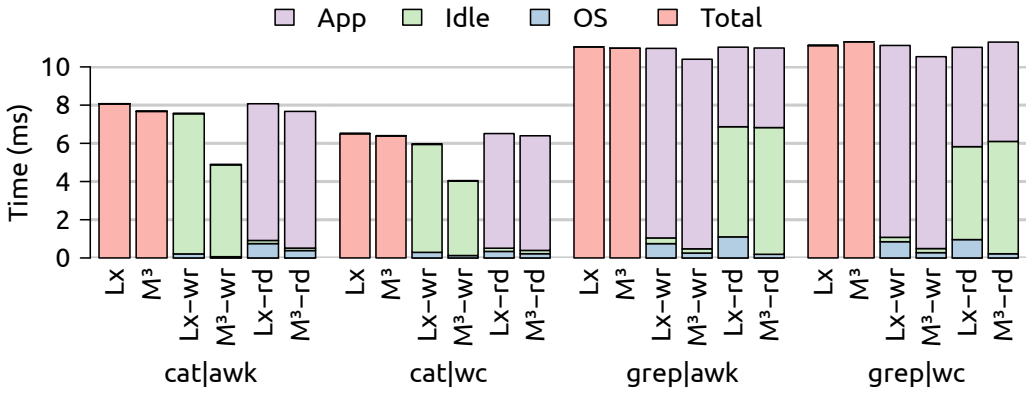


Figure 8.2: Performance comparison to Linux using a pipeline between two apps

into the time for the application ("App"), the idle time ("Idle"), and the OS overhead ("OS"). The application time[3] and the OS overhead have the same meaning as in the previous section. The idle time is the time the reader or writer waited for each other. The standard deviation is below 1 % and therefore not shown in the plot.

As can be seen in Figure 8.2, the performance on Linux and M³ is roughly on the same level for all four combinations. The writer exits earlier on M³ for *cat|awk* and *cat|wc*, because *cat* is not compute intensive and gets access to 128 KiB of the pipe's shared memory at a time. In contrast to that, Linux exchanges the data via the pipe in a granularity of less than 4 KiB (with these applications). Hence, after the writer is finished, the reader still has to process 128 KiB of data on M³ and less than 4 KiB on Linux. The granularity of the data exchange also leads to differences in *grep|awk* and *grep|wc*. In these cases, the reader spends more time idling on M³ than on Linux, which matches roughly the time *grep* requires to process 128 KiB of data. In contrast to that, Linux spends more time in the OS. Since reader and writer cannot work in parallel during the first and last step of the data exchange and the data exchange granularity on M³ is larger than on Linux, M³ cannot turn the lower OS overhead into an advantage for *grep|awk* and *grep|wc*, in contrast to *cat|awk* and *cat|wc* (to a smaller degree).

8.3 Scalability

This section evaluates the scalability of the single kernel PE with respect to the number of user PEs. As discussed in more detail in Chapter 9, my work spawned a new research project that investigates the scalability of capability systems based on M³. The key idea is to partition the user PEs into groups, using one kernel PE per group with local resources and capabilities. The kernel PEs communicate via message passing to enable cross-group interaction in a way transparent for applications. In this section, I evaluate how many user PEs a single kernel PE can manage without becoming the bottleneck.

Since the system's scalability depends on more components than the kernel and these components can be made scalable with means that are orthogonal to my system architecture, I use the following techniques to focus on the kernel's scalability:

1. **No memory accesses for data transfers:** Instead of accessing the files' data and the pipes' shared memory areas in RDMA fashion via the DTU, the benchmarks

[3] In these benchmarks, the system calls `brk`, `getuid`, and `ioctl` were unsupported on M³. The sum of the times for the ignored system calls were at most 0.012 ms.

spin for the corresponding time and therefore assume a perfectly scaling memory architecture. A scalable memory architecture can be designed in different ways, independent of my work. For example, instead of directly accessing a single DRAM from all PEs, a hierarchy of caches could be placed in front of one or multiple DRAMs.

2. **Multiple independent file-system servers:** Building a scalable file system is a challenge by itself and out of the scope of this work. For example, M^{3}FS can employ multiple worker VPEs and keep the global state consistent via messages. To evaluate whether the file system or the kernel is the bottleneck in the individual benchmarks, I start multiple independent file-system servers without synchronization. In other words, I assume that the synchronization requires no additional time.

3. **Crossbar instead of a NoC:** As mentioned in Section 8.1, the current prototype is using a crossbar to connect all PEs instead of a full network-on-chip. Since a crossbar uses a dedicated route between all pairs of PEs, communication causes no contention.

An important point is, that all three techniques put *more* load on the kernel rather than less. For example, a contended memory hierarchy would reduce the load on the kernel, because the time for data transfers increases, which in turn increases the intervals the kernel is contacted. In other words, the load on the kernel in these benchmarks is an upper bound.

To evaluate the scalability of the kernel, I performed multiple kinds of experiments that stress the kernel in different ways. First, I execute multiple instances of the benchmarks used in the previous section and determine the *parallel efficiency* for weak scaling. The parallel efficiency for n application instances in parallel is defined as $\frac{T_1}{T_n}$ with T_1 as the runtime of a single application instance running alone and T_n as the average runtime of the n application instances running in parallel. In other words, I analyze the slowdown of individual application instances due to resource contention. Ideally, the runtime per application instance should stay constant with an increasing number of instances, which corresponds to a parallel efficiency of 100 %. Second, I execute a web server with a varying number of workers and evaluate the total number of requests per second. Due to the long simulation times and the low variation on M^3 in the previous benchmarks, I use only a single run in the scalability evaluation.

8.3.1 Standalone Applications

I start with the standalone applications from Section 8.2. The results for the standalone applications are shown in Figure 8.3. Each plot shows the parallel efficiency depending on the number of file system servers (1, 2, 4, or 8) to determine whether the kernel or the file system is the bottleneck. The system for the benchmarks contains one PE for the kernel, one for the pager, n for M^{3}FS servers ("srv" in the figure), and m for applications (the x-axes in the figure). The PE configurations are the same as in the previous section. Note that *shasum* and *sort* are shown in one plot, because both scale almost perfectly (less than 1 % slowdown even with 32 applications).

As can be seen in Figure 8.3 and unsurprisingly after the analysis of these applications in the previous section, the behavior of the applications can be divided into groups. *tar* and *untar* are transfer intensive and therefore do neither cause much load in the kernel, nor in the file system servers. Hence, with 32 instances of *tar* or *untar* in parallel, the

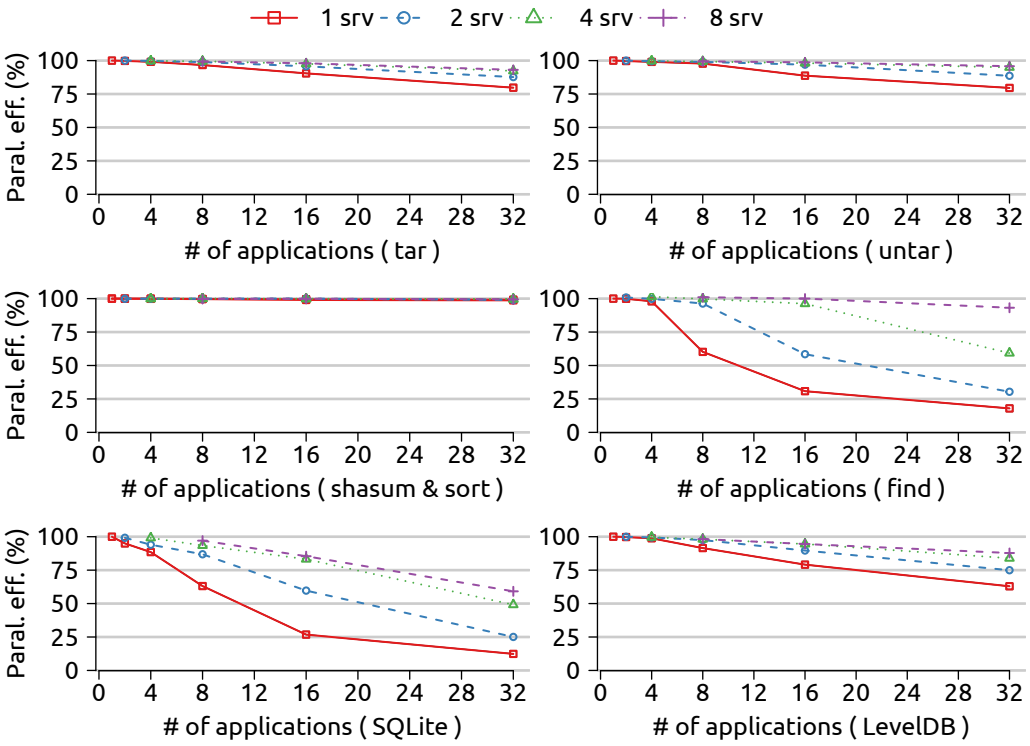


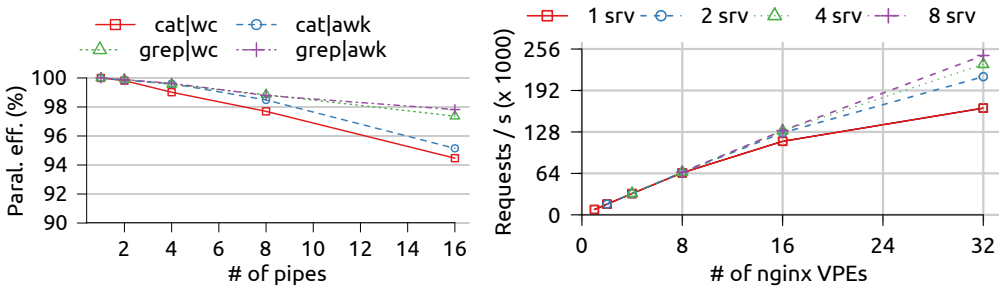
Figure 8.3: Parallel efficiency of the standalone applications, depending on the number of application instances.

parallel efficiency is always above 80 % and above 93 % with eight M^{3}FS servers. *shasum* and *sort* are compute intensive and therefore put even less load onto the kernel and servers. As mentioned, even with 32 applications, the parallel efficiency is always above 99 %. *find* and *SQLite* behave differently, though. The reason is that both send requests to the file system server with high frequency, leading to significant slowdowns with less than one server per four applications. Since *SQLite* reopens files and directories frequently, requiring capability exchanges and revokes, *SQLite* puts also more load onto the kernel. Therefore, even eight servers only lead to a parallel efficiency of 59 % for 32 *SQLite* instances. Finally, *LevelDB* shows a parallel efficiency of 87 % with 32 LevelDB applications and eight M^{3}FS servers, similar to *tar* and *untar* due to the long computation times and data transfers.

8.3.2 Pipelines of Applications

After the standalone benchmarks, this section evaluates the parallel efficiency of pipelines of applications, similar to Section 8.2. The shared memory area of the pipes is again 512 KiB and the writer and reader get access to 128 KiB at a time. The results are shown in Figure 8.4a. In this case, the system for the benchmarks contains one PE for the kernel, one for the pager, one for M^{3}FS, one for the pipe server, and two PEs for each pipe (the x-axis in the figure), that is, for each pair of applications. Since these applications cause little load in the file-system server and the pipe server, I only show the results with one instance of each.

As shown in Figure 8.4a, the parallel efficiency is above 94 % for all application combinations when running 16 pipes (32 applications) in parallel. The reason is that none of these applications are performing many requests to the file system server and



(a) Pipelines of applications

(b) The *nginx* web server

Figure 8.4: Parallel efficiency of application pipelines and scalability of *nginx*

none are frequently performing operations on capabilities. Most of the time, these applications compute or transfer data, which does not involve other components. The communication with the pipe server occurs infrequently due to the coarse grained data exchange (128 KiB) and the pipe server handles requests typically in less than 0.5 µs, leading to a good scalability of the pipe server.

8.3.3 Web Server

This section evaluates the scalability of the kernel using a web server as the workload. I chose *nginx*, because it is a fast, light-weight, and popular web server [120]. As for the other application-level benchmarks, I ran *nginx* on Linux using systrace and replayed it on M^3. In this case, I traced the system calls of nginx's worker process that handles the requests. The requests are sent by the Apache benchmark *ab* [1]. To test different file sizes, I sent HTTP requests for files between 1 KiB and 1 MiB in power-of-two steps. For each file, *ab* sent three requests to *nginx*.

Since M^3 does not have network support yet, I used a load generator that sends these requests to a replayer. The replayer replays the trace obtained from *nginx*'s worker process, reading the requests from the load generator and writing the responses back to the load generator. To communicate between the load generator and the replayer, one message-passing channel is used for signaling and shared memory is used to exchange the data (via the DTU).

To evaluate the scalability, I measured the total number of requests per second with a varying number of replayer instances (*nginx* instances) in parallel, as shown in Figure 8.4b. As with the standalone applications, I used a varying number of independent M^{3}FS instances. In these benchmarks, the system contained one PE for the kernel, one for the pager, *n* for M^{3}FS (1, 2, 4, and 8), *m* for the *nginx* instances (the x-axis in the plot), and sufficiently many PEs for the load generators to stress *m* *nginx* instances. All PEs are configured as before. The results in Figure 8.4b show that the requests per second increase almost linearly with sufficiently many file-system servers. In other words, the kernel is not significantly limiting the scalability of *nginx*.

8.4 Efficiency

In the previous sections, I used dedicated PEs for OS servers to evaluate the performance and scalability if sufficiently many PEs are available. In this section, I analyze the impact on the performance and scalability if PEs need to be shared using M^3's context switching mechanism. In other words, I evaluate how efficiently M^3 can utilize the available PEs.

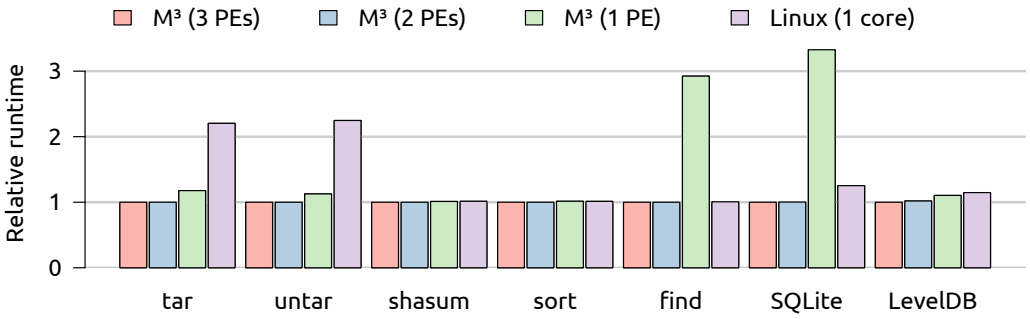


Figure 8.5: Performance of the standalone applications with a varying number of user PEs. The runtime is shown in relation to the runtime on M³ with three user PEs.

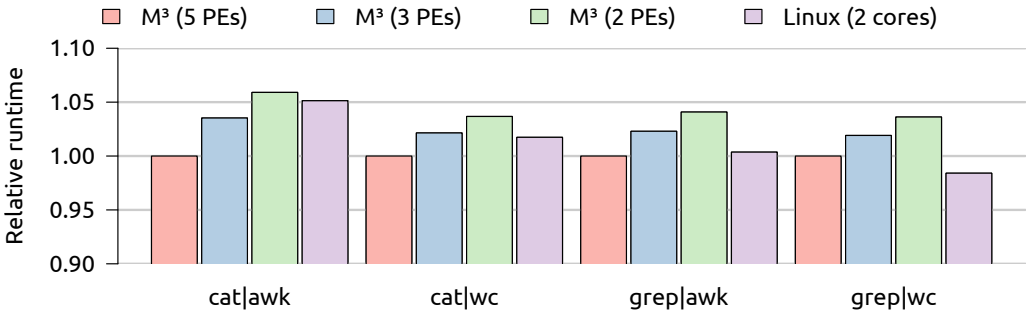


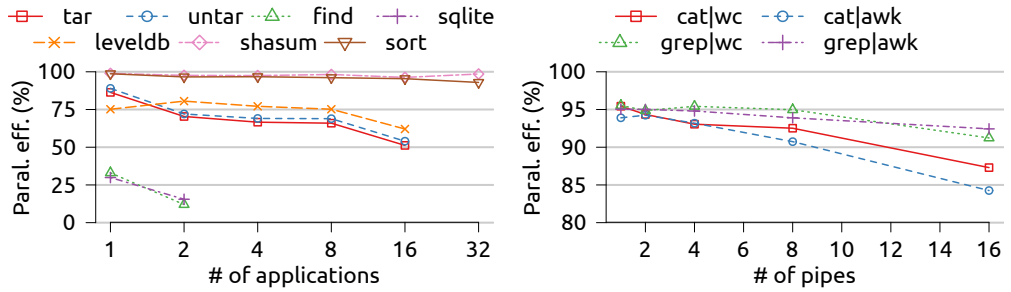
Figure 8.6: Performance of pipelines of applications with a varying number of user PEs. The runtime is shown in relation to the runtime on M³ with five user PEs.

8.4.1 Single Application Instances

The first question to answer is: how does the performance of the standalone applications and the pipelines of applications used in Section 8.2 change, if OS servers do not run on dedicated PEs, but share the PEs with the applications? To answer this question, I ran the benchmarks again and reduced the number of PEs in two steps. In the first step, I ran all OS servers on the same PE and in the second step, I ran the OS servers and the applications on a single PE.

The results for the standalone applications are shown in Figure 8.5, which shows the average runtime of three runs, preceded by one warm-up run, in relation to the average runtime on M³ with three user PEs (one PE for M³FS, one PE for the pager, and one PE for the application). The first step ("M³ (2 PEs)" in Figure 8.5) runs all OS servers on the first PE and the application second PE. The second step ("M³ (1 PE)") runs all OS servers and the application on a single PE. As a reference, I also show the performance on Linux with one core again. As the results show, using a single dedicated PE for both servers has almost no impact on the performance. Running both servers on the same PE as the application leads to a performance degradation in some cases. For *tar* and *untar*, the performance is reduced by 17 % and 12 %, respectively, but is still about twice as fast as on Linux. In other words, M³'s performance benefits for data-intensive applications that have been shown in Section 8.2 shrink slightly, because of M³'s slow remote context switches. *shasum* and *sort* show almost no performance degradation, whereas *find* and *SQLite* experience a significant slowdown. The reason is, that both *find* and *SQLite* communicate heavily with M³FS, leading to many context switches. The performance of *LevelDB* degrades slightly when using a single PE, but is still better than on Linux.

Figure 8.6 shows the results for the pipelines of applications, relative to the runtime



(a) Standalone applications

(b) Pipelines of applications

Figure 8.7: Parallel efficiency of the standalone applications and application pipelines without additional PEs for servers

on M^3 with five user PEs (one PE for M^{3}FS, one PE for the pipe server, one PE for the pager, and two PEs for the applications). Analogous to the standalone applications, the first step ("M^3 (3 PEs)" in Figure 8.6) runs all OS servers on the first PE and the two applications on the other two PEs. The second step ("M^3 (2 PEs)") runs all servers and the two applications on two PEs. In this case, all servers and applications can run on both PEs and the kernel performs context switches and migrations as necessary (see Section 7.4.3). As for the standalone applications, the runtimes are the averages from three runs, preceded by one warm-up run. In this case, the performance degradation is at most 6 % when using two user PEs instead of five. Therefore, the performance on M^3 with two user PEs is still on the same level as Linux with two cores.

8.4.2 Multiple Application Instances

I have shown in Section 8.3 that the M^3 kernel scales well in most cases to up to 32 parallel applications. However, I used dedicated PEs for the OS servers, preventing to use these PEs for applications. In this section, I evaluate the parallel efficiency that can be achieved if the application PEs are shared with the OS servers. To this end, I ran the standalone and pipeline benchmarks again without using dedicated PEs for the OS servers. In other words, the system consists of $n + 1$ PEs: one kernel PE and n user PEs for application instances. Each application instance is pinned on its PE, whereas the OS servers can freely migrate between the n user PEs.

The results are depicted in Figure 8.7, which shows the parallel efficiency using the runtime of a single application instance with dedicated PEs for OS servers as the baseline. In other words, a parallel efficiency of 100 % means that the average runtime of all application instances *without* dedicated PEs for OS servers is the same as the runtime of a single application instance *with* dedicated PEs for OS servers. Note that Figure 8.7a does not show the results for higher application counts, if the results were already unacceptable for lower application counts. Additionally, remember that Section 8.3 showed the scalability for the standalone benchmarks with a varying number of M^{3}FS instances. To keep the plot simple, Figure 8.7a only shows the highest parallel efficiency that was achieved with any number of M^{3}FS instances.

As depicted in Figure 8.7a, *find* and *SQLite* show disappointing results. The parallel efficiency starts at 33 % and 29 %, respectively, and further degrades with more application instances. As already outlined in the previous section, letting a single instance of these applications share a PE with OS servers results in bad performance, because of the many calls to the OS servers, requiring many context switches. Since these context

switches already cause a significant load in the kernel, running multiple applications results in even worse performance. The results for *tar*, *untar*, and *LevelDB* are better due to less frequent calls to OS servers, but also lead to unacceptable results with more than eight application instances. In contrast to that, *shasum* and *sort* scale well even when sharing the PE with OS servers. As shown in Figure 8.7b, letting pipelines of applications share their PEs with OS servers also leads to good results.

A counter-intuitive effect in these benchmarks is that an increasing number of application or pipe instances sometimes leads to better performance. For example, *LevelDB*'s parallel efficiency increases from a single application instance to two instances, as shown in Figure 8.7a. The reason is that the performance depends on whether a call to an OS server leads to a context switch on the caller's PE (PE-local call), to a context switch on another PE (remote call), or no context switch at all (remote fast call) because the server is already running on a different PE. The PE-local call is the most expensive type, because it requires two context switches (from the application to the server and back). The remote call is cheaper, because it only requires a single context switch, whereas the remote fast call is the cheapest type of call. Which of these call types is used depends on the exact timing of the call and the calls of other applications. However, the M^3 kernel still lacks good heuristics to decide which of these call types should be used. At the moment, the kernel uses the PE-local call whenever a remote call is not immediately possible, because no other PE is idling. As the other PEs in these benchmarks are typically busy with running other applications, the PE-local call is used in most cases. Hence, there is still room for optimization.

8.4.3 System Efficiency

As shown in the previous section, some workloads yield a good parallel efficiency if PEs are shared among applications and OS servers. In other cases, PE sharing leads to a degradation of the parallel efficiency. Furthermore, Section 8.3 showed that the parallel efficiency depends on the number of M^{3}FS instances if OS servers run on dedicated PEs. This raises the question how efficiently M^3 can run a given set of applications by choosing the best number of OS servers and placing the OS servers in the best possible way. I evaluate that by a measure I call *system efficiency*. The system efficiency is defined as $\frac{n}{m} \cdot e$, whereas n denotes the number of application instances, m denotes the total number of used PEs, and e denotes the achieved parallel efficiency[4]. In other words, the system efficiency determines how efficient the system runs a specific workload with a specific number of PEs. In this work, I only vary the number and placement of OS servers and always use dedicated PEs for all application instances. For example, running eight application instances on eight PEs with a parallel efficiency of 80 % results in a system efficiency of $\frac{8}{8} \cdot 0.8 = 0.8$. If these eight instances achieve a parallel efficiency of 85 % with two additional PEs for OS servers, the system efficiency is only $\frac{8}{10} \cdot 0.85 = 0.68$. Hence, this case prefers to share the application PEs with OS servers.

To determine the optimal number and placement of OS servers, I calculate the system efficiency for the different number of OS servers as done in Section 8.3 and calculate the system efficiency when sharing application PEs with OS servers as done in Section 8.4.2. The system efficiency with dedicated PEs for OS servers is calculated as $\frac{n}{n+1+s} \cdot e$, because it requires n PEs for applications, one kernel PE and s server PEs. For shared PEs, the system efficiency is calculated as $\frac{n}{n+1} \cdot e$, because it only requires n

[4]Remember that parallel efficiency is defined as $e = \frac{T_1}{T_n}$ with T_1 as the runtime of a single application instance running alone and T_n as the average runtime of n application instances running in parallel.

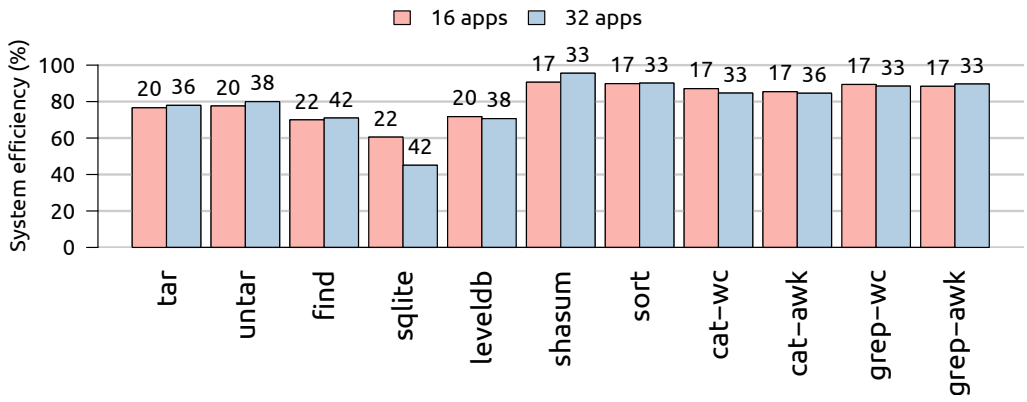


Figure 8.8: System efficiency for 16 and 32 application instances, using the best possible number and placement of OS servers. The number on top of each bar shows the total number of used PEs.

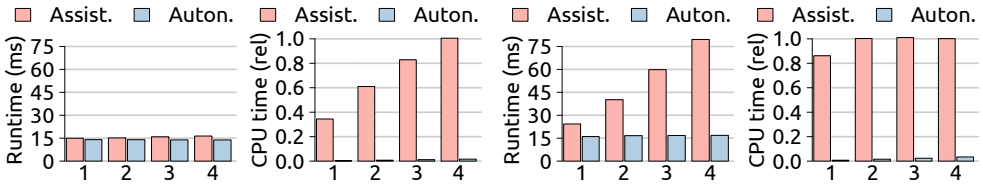
PEs for applications/servers and one kernel PE. Figure 8.8 shows the highest system efficiency for each benchmark, using the best result of the just described calculations for 16 and 32 application instances. The number on top of each bar in the figure denotes the number of PEs used to achieve this result. Note that for the pipe benchmarks, 16 application instances denotes 8 pipes and 32 application instances denotes 16 pipes.

As can be seen in Figure 8.8, for most applications, the system efficiency is above 75 %, for the compute-intensive applications even above 90 %. Additionally, the results show that the pipe benchmarks mostly prefer to share the application PEs with OS servers, whereas the standalone benchmarks mostly prefer dedicated PEs for OS servers. The system efficiency for *find*, *SQLite*, and *LevelDB* is rather low due to their frequent calls of OS servers. Note however, that system efficiency is defined rather strict in this evaluation, because the time spent in OS servers is classified as “useless”, which is not generally true. Additionally, I assume that all PEs can be fully utilized at the same time and over longer periods of time. Intel processors starting with the Nehalem microarchitecture already use Intel’s Turbo Boost technology to increase the clock frequency to the highest level only if permitted by the thermal and power constraints. Current predictions of the dark silicon effect [59, 142] suggest that this trend continues and leaves significant portions of the chip area “dark” for most of the time. In this case, dedicating a few PEs to OS servers that are typically underutilized or are clocked at a lower frequency, might be a good strategy to clock the application PEs with a high frequency over longer periods of time.

8.5 Autonomous Image Processing

This section shows the benefits of autonomous accelerators based on a real-world scenario. I am using an image-processing scenario as it is imaginable in data centers, similar to Google’s TPU [69] workloads. The cloud provider offers a set of image-processing accelerators as a service and allows customers to perform large-scale image processing on these accelerators. An efficient method for large images is FFT convolution [103], which performs a 2D fast Fourier transformation (FFT) on the image, multiplies the result pointwise with an image filter, and performs the inverse FFT. Depending on the filter, FFT convolution can be used for example for edge detection or low-pass filtering.

To evaluate this scenario, I use three types of stream-processing accelerators called



(a) Runtime (NoC) (b) CPU time (NoC) (c) Runtime (PCIe) (d) CPU time (PCIe)

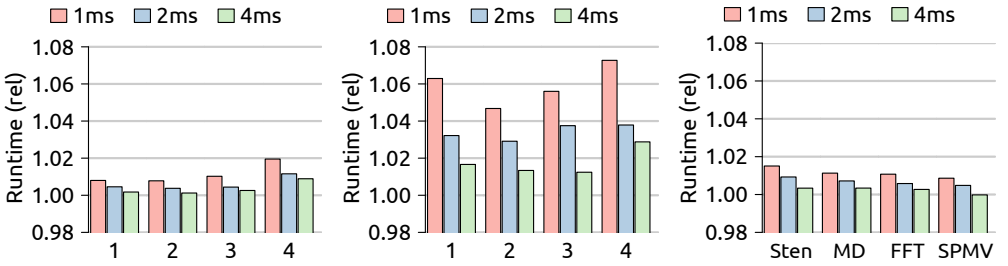
Figure 8.9: Total runtime (a and c) and the required CPU time (b and d) for different numbers of accelerator chains when integrating the accelerators into the NoC and attaching them via PCIe

FFT, MUL, and IFFT. As described in Chapter 6, the accelerators use a local scratchpad memory (SPM) of 2 KiB in this case (the block size for the 32×32 point FFT) and use the file protocol to stream the data block-wise from the input stream via the SPM to the output stream. I used Aladdin [129] to determine the computation times for the three accelerators offline. As described in Section 6.7.1, Aladdin uses a configuration file to analyze trade-offs between power, chip area, and performance by, for example, specifying loop unrolling factors. To get reasonable results, I generated all sensible configurations and picked the sweet spot between performance and the product of chip area and power consumption. I obtained 5856 cycles for FFT and IFFT and 1189 cycles for MUL.

These three types of accelerators run VPEs that form an FFT-MUL-IFFT chain to process a 4 MiB large image file and store the resulting image as a file as well. In this experiment, I run 1 to 4 such chains simultaneously without context switching, thus using 1 to 4 instances of each accelerator type. To show the benefits of autonomous accelerators, I compare M^3's autonomous approach with the assisted approach. The assisted approach drives the accelerators from software using a single general-purpose core. Hence, software is responsible for loading the input data into the SPM, starting the accelerator via a message, asking the accelerator's DTU to move the result to the next SPM, and to write the final result to the output file. The autonomous approach connects the DTU endpoints of the accelerators as follows. The input of the first and the output of the last accelerator is connected to a file. The middle accelerator is connected directly to its neighbors, as explained in Section 6.6.2.

I simulate two ways to attach accelerators to the system: network-on-chip (NoC) and PCI express (PCIe). Integrating accelerators with the CPU into a NoC leads to superior performance due to the lower latency. Connecting accelerators as an add-on card via PCIe to the host system provides more flexibility, because accelerators can be designed independently of the CPU. I compare the assisted and autonomous approach in terms of performance and CPU time for both the NoC version and the PCIe version. Figure 8.9 shows the overall runtime (a and c for NoC and PCIe, respectively) and the CPU time spent to drive the accelerators (b and d), depending on the number of accelerator chains. I show the averages of three runs, preceded by one warm-up run. The standard deviation is below 3 %. I simulate the PCIe-attached add-on card by connecting the accelerators via a bridge with 500 ns delay to the host system, which is the typical one-way latency for PCIe gen 3 [50, 54, 76, 125]. The one-way latency within the NoC, as simulated by the crossbar, is about 10 ns. For both the NoC and the PCIe version, the DRAM is part of the host system and stores the in-memory file system.

As depicted in Figure 8.9a, using the assisted approach leads to a slightly worse



(a) Stream processing (NoC) (b) Stream processing (PCIe) (c) Request processing (NoC)

Figure 8.10: Context switching overhead for stream processing, depending on the number of accelerator chains, and request processing, depending on the workload. Each plot shows the overhead for different time slice lengths (1 ms to 4 ms).

overall runtime with an increasing number of accelerator chains when integrating the accelerators into the NoC. When using PCIe, the overall runtime increases significantly with the number of accelerator chains, leading to a slowdown of factor 4.7 with four chains. In contrast, the autonomous approach always achieves the same runtime, independent of the number of chains. Even more importantly, the assisted approach keeps the CPU busy most of the time. Within the NoC, the CPU is utilized 100 % of the time starting at four accelerator chains, whereas with PCIe, the CPU is already utilized 100 % of the time starting with two chains. The autonomous approach does not cause significant CPU load in either case. Additionally, Figure 8.9c shows that the autonomous approach outperforms the assisted approach for PCIe-based accelerators even if the assisted approach does not fully utilize the CPU. The reason is the 500 ns delay when communicating with the accelerators, which prevents the assisted approach from fully utilizing the accelerators.

Note that the performance can still be improved for both the assisted and the autonomous approach. For the assisted approach, batching could be used to reduce the interaction frequency with the accelerators. However, batching is only possible by increasing the SPM sizes of the accelerators, which is expensive in terms of area and energy and increases the time the accelerator is not interruptible. Additionally, the assisted approach can trade more CPU time for more accelerator performance by using multiple cores to drive the accelerators, until the PCIe bus becomes the bottleneck. The autonomous approach does not suffer from the trade-off between SPM size and CPU utilization and can further improve the performance by overlapping data transfers to the DRAM instead of issuing one transfer at a time.

8.6 Accelerator Sharing

After showing the benefits of autonomous image processing, this section evaluates the context-switching overhead if accelerators need to be shared. To this end, I compare the runtime of two sequential accelerator usages with two interleaved usages. I start with the image-processing scenario from the previous section. In this case, I only use the autonomous approach and put each chain of VPEs into the same gang to benefit from gang scheduling (see Section 7.4.5). Figure 8.10a and Figure 8.10b show the average context-switching overhead using three runs, preceded by one warm-up run. The standard deviation is less than 2 %. I vary the time slice length for context switching between 1 ms and 4 ms. As the results show, using a still rather short time slice of 4 ms

Component	SLOC
Kernel	5398
├ PEs & VPEs	1839
├ Syscalls	970
├ Capabilities	676
├ Remote control	658
└ Memory	551
Pager	852
M^{3}FS	1794
Pipe server	619

(a) Complexity of kernel and servers

Component	SLOC
CU-specific helper	1049
├ x86-64-specific code	362
├ ARMv7-specific code	149
├ RCTMux	176
└ VMA (x86-64)	301
Base libraries	5204
├ x86-64-specific code	209
└ ARMv7-specific code	178
libm3	5843

(b) Complexity of support components

Table 8.1: Complexity of software components

leads to less than 0.9 % overhead when integrating the accelerators into the NoC and less than 2.9 % overhead when attaching them via PCIe.

For the request-processing accelerators, I use the same workloads as in Chapter 6 and the accelerator support module performs all invocations of the accelerator logic in a single batch from software. Like for the image-processing scenario, the input and output data is stored in files. Figure 8.10c shows the average context-switching overhead using three runs, preceded by one warm-up run. The standard deviation is less than 1 %. Similarly to the stream-processing accelerators integrated into the NoC, the overhead is less than 0.4 % with a time slice of 4 ms. Note that I only evaluate this case by integrating these accelerators into the NoC, because these accelerators perform fine-grained memory accesses to the files that are stored in the host system. Storing the data into a memory on the PCIe-attached addon card is supported as well, but out of scope in this work. As in other system architectures, it requires to copy the data to the addon card in advance and to copy the result back to the host system afterwards.

8.7 Software Complexity

Finally, I evaluate the complexity of the software components that are or can be part of the trusted computing base (TCB). A simple and widely used metric to measure the complexity of software is the number of code lines. I use the tool cloc [2], which counts the physical lines of source code (SLOC), excluding blank lines and comments. Note that the results depend on the coding style and programming language and therefore only allow rough comparisons of different software components.

Table 8.1 shows the SLOC numbers for the different components of M^3. For some of the components I show the sizes of interesting subcomponents. The kernel is part of the TCB of all applications, because the kernel has the full control over the system. If an application uses a server such as the pager or M^{3}FS, this server is also part of the application's TCB, because the application depends on the server's correctness. However, in contrast to the kernel, a server has typically less power over an application. For example, a malicious pipe server can prevent that an application can successfully use pipes, but cannot access the application's private data or influence the application. The CU-specific helper on a specific PE is in the TCB of all applications running on this PE. For example, the CU-specific helper for ARMv7 is not in the TCB of applications running on x86 PEs. Finally, M^3 provides multiple libraries for the other software components.

The base libraries are used by the kernel, the CU-specific helper, servers, and applications. These libraries provide routines to access the DTU, generic data structures, and other generic utilities that can be used in any context. In contrast, libm3 is only used by servers and applications and uses system calls to implement its functionality. For that reason, the kernel cannot use libm3.

As shown in Table 8.1, the M^3 kernel has almost no ISA-specific code, except for a few lines in the base libraries that are used by the kernel. ISA-specific code is not required, because the M^3 kernel is a simple program that runs alone on a dedicated PE and interacts with the user PEs via DTU-based message passing and memory accesses. The small amount of ISA-specific code allows to port the kernel to any general-purpose ISA with little effort. Another interesting property of M^3 is that the kernel with about 10 000 lines in total (the sum of the lines for the kernel and the base libraries) is similarly small as NOVA [137], one of the smallest L4 kernels, with about 9000 lines. At the same time, M^3 supports more complex platforms with a large variety of CUs, ranging from complex general-purpose cores to fixed-function accelerators. In contrast, NOVA only supports homogeneous x86-based platforms.

Chapter 9

Conclusion and Future Work

This chapter starts by summarizing the most important points and discusses the advantages and disadvantages that the described system architecture provides. Afterwards, I elaborate on extensions and future work.

9.1 Conclusion

In this work, I described and evaluated a new system architecture that integrates accelerators as first-class citizens. The system architecture uses three key principles:

1. adding new hardware component, called data transfer unit (DTU), next to each compute unit (CU), leading to a system that consists of multiple and heterogeneous processing elements (PEs) with uniform interfaces,

2. running the kernel on a dedicated kernel PE that controls applications and accelerators on the user PEs remotely by configuring their DTUs, and

3. enabling DTU-based communication channels between user PEs, established by the kernel and bypassing the kernel during the actual communication.

The uniform interface that the DTU provides allows my system architecture to integrate very heterogeneous CUs, ranging from general-purpose cores to fixed-function accelerators, as first-class citizens. The investigation of this system architecture lead to various insights, which I discuss in the following.

Access to OS services and direct communication The two main features of the DTU are message passing between CUs and RDMA-like memory access. Based on these features, I showed how a generic and accelerator-friendly *file protocol* (see Section 4.3) allows all types of CUs to access arbitrary file-like objects such as files, pipes, or sockets. The direct access to OS services is one of the key principles to run accelerators autonomously, that is, without continuous assistance by the CPU. Furthermore, the uniform communication interface allows easy interaction between *all* CUs. I have also shown in Section 6.6.2 that the file protocol can be extended for direct accelerator-to-accelerator communication, which leads to performance improvements and reduces the system load over a pipe-based communication, if multiple accelerators operate in a chain (see Section 6.7.3).

Context switching on all CUs Although the kernel-bypassing communication via the DTU has proven beneficial for performance and the autonomous operation of accelerators (see Section 8.5), the combination with context switching is challenging. I showed in Chapter 7 that VPE-aware communication, message forwarding by the kernel, and idle notifications are sufficient to successfully combine kernel-bypassing communication with context switching. Furthermore, I showed that the context-switching concept supports context switching on all types of CUs by performing the potentially complex decisions in software by the M^3 kernel and only simple, but CU-dependent save and restore actions in hardware. The evaluation in Section 8.6 demonstrated that accelerator sharing leads to an overhead of less than 1 % with time slices of at least 4 ms and when integrating the accelerators into the NoC. Attaching accelerators via a PCIe-like interconnect leads to an overhead of less than 2.9 %.

Autonomous accelerators I have shown in Chapter 6, on the example of request-processing accelerators, how fine-grained interruptibility can be combined with autonomous operation. This combination achieves the best performance and allows at the same time to interrupt the accelerator with low latency in case a more important job for the accelerator arrives. Additionally, I have demonstrated that stream-processing accelerators can run autonomously based on the direct access to OS services and unassisted accelerator-to-accelerator communication. The evaluation in Section 8.5 showed that the autonomous operation improves the performance by a factor of 4.7 and reduces the CPU load by a factor of 30 when attaching image-processing accelerators via PCIe. My approach also offers to trade larger file extents or larger shared-memory areas when using pipes for better performance and improved autonomy: increasing the size of the extent or the shared-memory area reduces the number and frequency of interactions with the server.

Easy integration of accelerators Tiled architectures [143, 151] are already used today, because they enable a modular system design and an easy integration of different CUs into one system. My approach takes tiled architectures one step further by additionally providing accelerator designers with the necessary interfaces to integrate an accelerator as a first-class citizen. In other words, accelerator designers only need to use these interfaces to get access to OS services, to enable context switching support, or to communicate with arbitrary other CUs.

Support of non-coherent and mixed-ISA systems The prototype implementation does not take advantage of cache coherency, but handles the underlying platform as a distributed system. Additionally, the M^3 kernel is almost ISA-independent and the CU-specific helper has only a few hundred lines of ISA-specific code. This design allows M^3 to support non-coherent systems, mixed-ISA systems, systems with multiple coherence domains, and distributed-memory systems such as Tomahawk comparatively easily. Furthermore, system designers are not forced to provide global cache coherency, which can be challenging for very heterogeneous platforms.

Performance with sufficiently many PEs Evaluating the performance of my system architecture in Section 8.2 revealed that most workloads achieve a comparable performance as Linux on traditional architectures, if sufficiently many PEs are available to use a dedicated PE for each application and server. For data-intensive file system

workloads, M³ outperforms Linux by a factor of two by taking advantage of the DTU and providing applications direct access to large amounts of file data at once.

Scalability with sufficiently many PEs Similarly to the delivered performance, Section 8.3 showed that the M³ kernel scales well for most workloads, achieving a parallel efficiency of above 90 % with applications on 32 user PEs, as long as sufficiently many PEs are available to run OS services on dedicated PEs. Some workloads such as *SQLite* that perform many capability operations, achieve a parallel efficiency of 83 % with 16 application instances, but only 59 % with 32 instances. In other words, the single kernel PE becomes a bottleneck in this case with more than 16 application instances.

Performance and scalability with PE sharing The evaluation also revealed a shortcoming of the current OS design: if applications need to share PEs with servers, the performance degrades significantly for OS-intensive workloads as shown in Section 8.4.1. For example, the runtime of *find* and *SQLite* increases by about a factor of three if these applications share a PE with the pager and file system server. This slowdown is caused by the context switches between applications and servers that are performed remotely by the M³ kernel and therefore significantly slower than traditional core-local context switches. Performing context switches remotely and centralized by the M³ kernel leads to scalability problems as well. As has been evaluated in Section 8.4.2, the current OS design cannot run multiple OS-intensive applications efficiently if their PEs need to be shared with servers. Despite these performance and scalability problems for some workloads, remote context switches are required to support context switches on accelerators. However, these problems indicate that a special case is required to run OS-intensive general-purpose workloads efficiently even if PEs need to be shared. I discuss a solution to this problem in more detail in the next section.

Dedicated kernel PE Running the kernel on a dedicated PE is one of the key principles to support arbitrary user PEs and has numerous benefits. For example, the kernel does not share hardware resources such as registers, caches, and TLBs with applications, which improves performance and prevents side-channel attacks on the kernel based on these resources, such as Spectre [78] and Meltdown [89], by design. Furthermore, kernel PEs and user PEs can be specialized independently. However, a dedicated kernel PE can reduce the system's scalability even though the kernel is primarily responsible to setup communication channels. Additionally, a dedicated kernel PE can lead to less system utilization, because applications cannot run on the kernel PE. However, the dark silicon effect [59, 142] might already prevent OSes from fully utilizing all PEs on future platforms.

Dedicated user PEs vs. user PE sharing As has been shown in the evaluation, M³ achieves the best performance if all applications and servers run on dedicated user PEs and performance typically degrades if PEs need to be shared. For these reasons, M³ prefers to pin applications and servers onto dedicated PEs and distribute them horizontally in the system instead of co-locating them on a single PE. The horizontal distribution is used by other research projects as well [35, 96, 150], but whether this approach is preferable on future platforms is an open question. On the one hand, increasing core counts, the dark silicon effect, and recent security vulnerabilities based on speculative execution and resource sharing [78, 89, 149] suggest that horizontal

distribution of applications will be the default in the future. On the other hand, co-location and resource sharing is still important today to use systems efficiently.

Reduced OS complexity I have shown that the uniform interface provided by the DTU simplifies the management of heterogeneous systems. This makes the M^3 kernel independent of the user PEs it manages and results in a similar kernel size as NOVA [137], one of the smallest L4 kernels (see Section 8.7). Despite the comparable size, M^3 supports more complex platforms with a large variety of CUs, ranging from complex general-purpose cores to fixed-function accelerators, whereas NOVA only supports homogeneous x86-based platforms.

Reduced trusted computing base My system architecture isolates PEs at the network-on-chip (NoC) by controlling the PEs' access to PE-external resources through the DTU. In contrast to traditional architectures, this allows to remove the CUs in user PEs from the trusted computing base (TCB), because the means of these CUs (e.g., different CPU modes) are not required for isolation. However, I described in Section 5.11 that additional means are required for PE-type C, which reuses the memory management unit (MMU) of general-purpose cores for virtual-memory support. Since the MMU of the supposedly untrusted general-purpose core determines the physical addresses on memory accesses, the DTU needs to restrict these addresses to prevent that the untrusted core can access all physical memory in the system.

Reuse of existing hardware components My system architecture reuses existing hardware components such as general-purpose cores, caches, memories, network-on-chips, etc. without requiring modifications. Additionally, an existing accelerator logic can be reused without changes as explained in Section 6.4. However, I also presented possible improvements that may require changes. For example, reducing the context-switching latency of an accelerator can require changes to the accelerator logic, independent of my system architecture, as described in Section 6.5.2. Similarly, I explained in Section 7.6 that the ability to reset a general-purpose core allows to share the core among multiple applications without being forced to trust the core to properly enforcement its different CPU modes. Whether the reset requires changes to the core is an open question.

Architecture is based on custom hardware Finally, the proposed system architecture is based on the addition of a DTU next to each CU, which prevents M^3 to run on commercial-off-the-shelf (COTS) hardware. However, upcoming interconnects such as GenZ [6] already provide the DTU's most important features: message passing and RDMA-like memory access and the ability to control these features to build secure systems. In other words, it is imaginable that M^3 can be adapted for these interconnects to run on future COTS hardware without requiring a DTU.

9.2 Extensions and Future Work

After the discussion of the system architecture with its advantages and disadvantages, this section elaborates on extensions that either already exist, but were not ready in time for this disseration, or are imaginable for future work.

Support for peripheral devices Based on the work of Lukas Landgraf and Georg Kotheimer, I was able to take the idea of the system architecture further by integrating peripheral devices as PEs as well. Lukas focused on the integration of an IDE controller to support persistent storage, whereas Georg added support for network interface cards [79]. Both works are based on a hardware proxy between the peripheral device and the DTU that translates port-mapped I/O (PMIO), memory mapped I/O (MMIO), direct memory access (DMA), and interrupts to mechanisms of the DTU. The generality of the proxy enables the integration of arbitrary PCI devices into the system architecture without requiring modifications to the PCI device.

Network stack In Georg's undergraduate thesis [79], he did not stop at the hardware support for networking, but also ported lwIP [49], a lightweight TCP/IP stack, to M^3. Initial measurements of latency and throughput in comparison to Linux showed similar or superior results.

File system for persistent storage Based on the support for IDE controllers by Lukas Landgraf, Sebastian Reimers designed and implemented extensions to the in-memory file system M^{3}FS for persistent storage in his bachelor thesis [121]. The focus of his work was the addition of a page cache to support persistent storage efficiently, while at the same time maintaining M^{3}FS' advantages in terms of performance and autonomous accelerators.

Core-local context switching As discussed in the previous section, M^3's performance and scalability degrades for some workloads if PEs are shared. This problem stems from the fact that M^3 performs context switches on the user PEs remotely from the kernel PE. These context switches are significantly more expensive than core-local context switches and the single kernel PE quickly becomes the bottleneck with an increasing number of user PEs that demand context switches. Although the remote context switches are required for accelerators, general-purpose cores typically have all necessary architectural features to support core-local context switches. For these reasons, a follow-up project could introduce a special case for general-purpose cores by extending the CU-specific helper to a *second-level kernel*. Each second-level kernel could get a set of VPEs from the M^3 kernel and switch between these VPEs locally without involving the M^3 kernel. Additionally, the second-level kernel could provide an L4-like IPC mechanism between these VPEs. In other words, this extension can be seen as a combination of M^3 and traditional OSes such as L4 [77, 83, 137] or Barrelfish [35]. However, further research is necessary to determine how independent of the M^3 kernel the second-level kernel can operate and whether a privileged DTU is required.

Scaling to larger systems This work uses a single kernel PE, which inherently limits the system's scalability with the number of user PEs. As evaluated in Section 8.3, the M^3 kernel scales well to 32 user PEs for most workloads, because the kernel is primarily used to establish communication channels and not involved in the actual communication. However, scaling to more user PEs requires multiple kernel PEs and their synchronization. Matthias Hille is addressing this challenge in his dissertation by partitioning the user PEs into groups and by using one kernel PE per group with local resources and capabilities. The kernel PEs communicate via message passing to enable cross-group interaction in a way transparent for applications. We have shown that this approach achieves a parallel efficiency of 70 % to 78 % when examining a system with

576 PEs executing 512 application instances, while using 11 % of the system's PEs for the OS [66].

Support for GPUs and FPGAs The goal of this work is to integrate all types of CUs as first-class citizens into the system, which enables access to OS services, direct communication, and context switching for all CUs. As a proof of concept, I demonstrated these benefits for the two extreme points in the design space of CUs in this work. On the one end of the spectrum, complex general-purpose cores provide all architectural features that are required for an OS kernel. Fixed-function accelerators on the other end of the spectrum have none of these features and do not even execute software. For these reasons, I believe that the concepts and mechanisms developed in this thesis can be applied to other types of CUs as well. For example, FPGAs are similar to fixed-function accelerators with the difference that the logic on an FPGA can be reconfigured at runtime. I believe that GPUs can be integrated by adding one DTU for each of the groups of cores that execute the same instruction stream (*Warp* in Nvidia's terminology). However, how exactly FPGAs and GPUs can be integrated in the best possible way is an open question and demands further research.

Support of PEs with multiple cores or hyper-threads In this work, each general-purpose PE uses a single core without hardware multithreading. As mentioned in Section 3.4.8, the DTU can be shared among multiple hardware threads by, for example, employing per-thread DTU registers, but sharing buffers and logic. Note that this approach is transparent to software. Another option that is not transparent to software is to use the DTU as is for multiple cores or hardware threads and multiplex the DTU in software. For example, a full OS can be run on such a PE, as discussed in the following, that performs all DTU accesses in the kernel and protects these accesses accordingly with synchronization primitives.

Running a full OS on a user PE The current prototype runs only a single application in each VPE. As described in this work, M^3 does not require an OS kernel on the user PEs, but only a few hundred lines of support code (the CU-specific helper) on each PE. For that reason, it is imaginable to run a full OS on a user PE, represented as a single VPE by M^3. For example, running Linux on a user PE by integrating the support code into the Linux kernel would allow to run legacy applications or to reuse a device driver from Linux. Adding a DTU driver to the Linux kernel would enable the communication between Linux and M^3 applications or accelerators. In such a scenario the DTU acts as an additional layer of protection on top of the protection features of the CU. This additional layer allows to isolate Linux from the rest of the system with the means of the DTU without being required to trust the CU. In other words, this approach goes one step further than virtualization, because virtualization is inherently based on the CU's protection features. However, how peripheral devices can be shared securely among multiple parties with this approach is an open research question.

Support for legacy applications Besides running a full OS on a user PE, legacy applications can be supported by providing a POSIX-emulation layer on top of libm3. Such an emulation layer already exists for the file-system API in form of the systrace infrastructure, which allows to trace Linux applications and replay their system calls on M^3. Furthermore, initial support for the C standard library *musl* has been provided by Sherif Abdalazim [23].

Acknowledgements

First, I would like to thank my advisor Professor Hermann Härtig for giving me the opportunity to work at the operating systems chair and for supporting me during my research. In particular, I am grateful for the enthusiastic and inspiring athmosphere he created and for the freedom that he granted us all. I would also like to thank all members of the operating systems chair for the various interesting and fruitful discussions and for all the fun we had during the last years. Additionally, I thank Marcus Völp for his support and guidance during the early phase of my work.

Furthermore, I would like to thank the Vodafone Chair Mobile Communications Systems at TU Dresden and in particular Benedikt Nöthen for the interesting collaboration on the Tomahawk platform and for giving me the opportunity to put my ideas into real hardware.

I am also grateful that I had the opportunity to work with many talented and motivated students. Christian Menard implemented the first prototype of the DTU model for gem5. René Küttner added initial support for context switching. Lukas Landgraf added support for storage devices, whereas Georg Kotheimer added network support. Sherif Abdalazim ported the musl C library to M^3 and finally, Sebastian Reimers extended the existing in-memory file system, M^3FS, for storage devices. Thank you all!

I would also like to thank Jan Bierbaum, Hermann Härtig, Matthias Hille, Maksym Planeta, Michael Roitzsch, Timothy Roscoe, Till Smejkal, Carsten Weinhold, and Hannes Weisbach for proof-reading parts of my dissertation.

Finally and most importantly, I want to thank my parents and my wife Daniela for always believing in me and supporting me. Additionally, I thank my children Nele and Lara for distracting me from work whenever necessary. Without you, this dissertation would not have been possible!

Glossary

M³FS is short for M³'s file system and the name for the in-memory file system on M³.

M³ is short for **m**icrokernel-based syste**m** for heterogeneous **m**anycores and the name for the operating system prototype in my work.

ASM is short for accelerator support module and is a piece of hardware that is used in accelerator PEs to invoke the accelerator logic and interact with the DTU. The ASM comes in two flavors for the two considered accelerator types: request-processing accelerator support module (RASM) and stream-processing accelerator support module (SASM).

Capabilities are used by M³ to manage the application's access to resources. A capability is a pair, consisting of a pointer to the resource and permissions for this resource. M³ offers operations to exchange capabilities between applications and also to undo these exchanges.

CU is short for compute unit and is used in this work to subsume all hardware components that perform computations. Examples are general-purpose cores, digital signal processors, graphics processing units, field-programmable gate arrays, and fixed-function accelerators.

DTU is short for data transfer unit, which is used as uniform hardware interface for all compute units and allows the communication via messages and bulk data transfers between the compute units.

EP is short for endpoint and is used to establish communication channels between processing and memory elements. Each data transfer unit has a set of endpoints represented as hardware registers to store the required information for the communication such as the destination element.

FPGA is short for field-programmable gate array, which is an integrated circuit designed to be configured (many times) after manufacturing [5].

Gate is a kernel object in M³ to represent DTU-based communication channels. A *receive gate* allows to receive messages, a *send gate* allows to send messages to a specific receive gate, and a *memory gate* allows to access PE-external memory. Before a gate can be used, a DTU endpoint needs to be configured for the gate.

IPC is short for inter-process communication and denotes in my work the message-based communication between two virtual processing elements via the DTU, analogous to IPC on L4 [85] via the kernel.

Kernel PE is short for kernel processing element and denotes the PE that runs the M^3 kernel, in contrast to user PEs that run applications and servers.

ME is short for memory element and represents a physical memory (e.g., DRAM or NVRAM) that is reachable via interconnect, analogously to a processing element.

MMIO is short for memory-mapped input/output, which is a way for the CPU to interact with peripheral devices. With MMIO, the registers of the device are mapped into the physical address space and the CPU interacts with the device via ordinary memory loads and stores. In the current implementation, MMIO is used to interact with the DTU.

MMU is short for memory management unit, which is a hardware component that provides memory protection and translates virtual addresses to physical addresses. MMUs are typically tightly integrated with a general-purpose core.

NoC is short for network-on-chip and allows a network-based communication between the processing and memory elements on a chip.

OS is short for operating system.

PE is short for processing element and represents the combination of a compute unit, local memory (scratchpad memory or caches), and a data transfer unit. Therefore, a PE is comparable to a tile in tiled architectures.

PE-type A denotes a PE that is primarily intended for **A**ccelerators. The compute unit in PE-type A uses an untranslated access to local scratchpad memory, which is exclusively used by this PE.

PE-type B denotes a PE that is intended for compute units, for which virtual memory is desired, but unsupported by the compute unit itself. Type B PEs integrate the data transfer unit between the CU and caches and the DTU performs an address translation on the way.

PE-type C denotes a PE that is intended for **C**omplex general-purpose cores that have a memory management unit and tightly integrated caches and should be reused without any modification. In this case, the data transfer unit relies on the virtual memory assistant that runs on the CU to translate virtual addresses.

RDMA is short for remote direct memory access and is supported in this work by the read and write commands of the data transfer unit. These commands allow to access data in other processing and memory elements without involving any compute unit.

Server is in this work used to denote an application that provides a service. For example, the server M^{3}FS provides a file system service to applications. Since the only difference between applications and servers is that a server has registered a service at some point, I often subsume both under the term “application”.

Service is in this work used to denote the function provided by a server.

SPM is short for scratchpad memory and is often used in accelerators as a temporary memory for the data of the computation, because SPM can be easily tailored to the accelerator's requirements. In this work and in the Tomahawk platform, SPM can also be used by general-purpose cores as dedicated physical memory for all code and data of the application running on the core.

TCB is short for trusted computing base, which is a common concept in security research and subsumes all components (hardware, firmware, or software) the security of a system depends on.

Tomahawk is a heterogeneous multiprocessor system-on-chip (MPSoC) designed at TU Dresden and is primarily intended for mobile communication applications. Tomahawk consists of multiple processing elements integrated into a network-on-chip and a memory controller that provides access to an external DRAM.

User PE is short for user processing element and denotes a PE that run an application or server, in contrast to the kernel PE that runs the M^3 kernel.

VMA is short for virtual memory assistant and is a small piece of code running on type C PEs to assist the data transfer unit in the virtual address translation and page fault handling.

VPE is short for virtual processing element and is an abstraction used by the M^3 kernel to manage the access to user processing elements. A VPE is a resource container and an execution context and hence a combination of a process and a thread in traditional operating systems. VPEs are used for both general-purpose and accelerator PEs and provide the illusion that an entire PE, consisting of the compute unit, memory, and data transfer unit, belongs to the owner of the VPE.

Bibliography

[1] ab - Apache HTTP server benchmarking tool.
https://httpd.apache.org/docs/2.4/programs/ab.html.
Accessed: 06/22/2018.

[2] AlDanial/cloc: cloc counts blank lines, comment lines, and physical lines of source code in many programming languages.
https://github.com/AlDanial/cloc.
Accessed: 08/24/2018.

[3] Benchmarking GPUDirect RDMA on Modern Server Platforms.
https://devblogs.nvidia.com/parallelforall/benchmarking-gpudirect-rdma-on-modern-server-platforms/.
Accessed: 21/09/2018.

[4] BusyBox.
https://www.busybox.net.
Accessed: 10/27/2018.

[5] Field-programmable gate array.
https://en.wikipedia.org/w/index.php?title=Field-programmable_gate_array&oldid=863571676.
Accessed: 10/12/2018.

[6] Gen-Z Consortium: Computer Industry Alliance Revolutionizing Data Access.
https://genzconsortium.org/.
Accessed: 08/03/2018.

[7] HSA foundation ARM, AMD, Imagination, MediaTek, Qualcomm, Samsung, TI.
http://www.hsafoundation.com.
Accessed: 12/15/2017.

[8] InfiniBand Architecture Specification.
https://www.infinibandta.org/ibta-specification.
Accessed: 21/09/2018.

[9] Intel® 64 and IA-32 Architectures Software Developer's Manual.
https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.
Accessed: 01/10/2018.

[10] An introduction to the Intel® QuickPath interconnect.
https://www.intel.de/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf.
Accessed: 01/19/2015.

[11] LevelDB.
https://leveldb.org.
Accessed: 06/15/2018.

[12] Linux source code: kernel/sched/fair.c (v4.16.13) - Bootlin.
https://elixir.bootlin.com/linux/v4.16.13/source/kernel/sched/fair.c.
Accessed: 05/30/2018.

[13] Mill computing, inc.
https://millcomputing.com.
Accessed: 08/30/2018.

[14] Nvidia Pascal microarchitecture.
https://en.wikipedia.org/w/index.php?title=Pascal_(microarchitecture)&oldid=859816379.
Accessed: 10/10/2018.

[15] OpenCAPI consortium.
https://opencapi.org/.
Accessed: 12/15/2017.

[16] RDMA over Converged Ethernet (RoCE).
https://cw.infinibandta.org/document/dl/7148.
Accessed: 21/09/2018.

[17] RISC-V Foundation | Instruction Set Architecture (ISA).
https://riscv.org/.
Accessed: 08/09/2018.

[18] SQLite.
https://www.sqlite.org.
Accessed: 07/12/2017.

[19] Tech Brief: AMD FireProTM SDI – Link and AMD DirectGMA Technology.
https://www.amd.com/Documents/SDI-tech-brief.pdf.
Accessed: 21/09/2018.

[20] Verilog.
https://en.wikipedia.org/w/index.php?title=Verilog&oldid=861112673.
Accessed: 10/10/2018.

[21] VHDL.
https://en.wikipedia.org/w/index.php?title=VHDL&oldid=860613986.
Accessed: 10/10/2018.

[22] Xtensa customizable processor.
https://ip.cadence.com.
Accessed: 01/19/2015.

[23] Sherif Abdalazim. Porting Musl to the M3 microkernel.
https://os.inf.tu-dresden.de/papers_ps/sherif_muslm3.pdf, 2018.

[24] Reto Achermann. Message passing and bulk transport on heterogeneous multiprocessors. `http://www.barrelfish.org/publications/ma-acreto-transport-heterogeneous-mp.pdf`, 2014.

[25] Jacob T. Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. The case for GPGPU spatial multitasking. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, HPCA'12, pages 1–12. IEEE, 2012.

[26] Andreas Agne, Markus Happe, Ariane Keller, Enno Lubbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. ReconOS: An operating system approach for reconfigurable computing. *IEEE Micro*, 34(1):60–71, 2014.

[27] R. Alpert, C. Dubnicki, E.W. Felten, and K. Li. Design and implementation of NX message passing using Shrimp virtual memory mapped communication. In *Proceedings of the 1996 International Conference on Parallel Processing*, volume 1 of *ICPP'96*, pages 111–119, Aug 1996.

[28] Oliver Arnold, Emil Matus, Benedikt Noethen, Markus Winter, Torsten Limberg, and Gerhard Fettweis. Tomahawk: Parallelism and heterogeneity in communications signal processing MPSoCs. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(3s):107:1–107:24, Mar 2014.

[29] Nils Asmussen, Hermann Härtig, and Marcus Völp. Turning x86 into a hardware simulator for future manycores. In *Proceedings of the 3rd Workshop on Systems for Future Multicore Architectures*, SFMA'13, 2013.

[30] Nils Asmussen, Michael Roitzsch, and Hermann Härtig. M³x: Autonomous accelerators via context-enabled fast-path communication. In *2019 USENIX Annual Technical Conference*, USENIX ATC'19, pages 617–632, Renton, WA, 2019. USENIX Association.

[31] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 189–203. ACM, 2016.

[32] Francisco J. Ballesteros, Noah Evans, Charles Forsyth, Gorka Guardiola, Jim McKie, Ron Minnich, and Enrique Soriano-Salvador. Nix: A case for a manycore system for cloud computing. *Bell Labs Technical Journal*, 17(2):41–54, 2012.

[33] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-ISA platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys'15, pages 29:1–29:16, New York, NY, USA, 2015. ACM.

[34] Can Basaran and Kyoung-Don Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS'12, pages 287–296, Washington, DC, USA, 2012. IEEE Computer Society.

[35] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh 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, pages 29–44, New York, NY, USA, 2009. ACM.

[36] Andrew Baumann, Chris Hawblitzel, Kornilios Kourtis, Tim Harris, and Timothy Roscoe. Cosh: Clear OS data sharing in an incoherent world. In *Proceedings of 2014 Conference on Timely Results in Operating Systems*, TRIOS'14, Broomfield, CO, 2014. USENIX Association.

[37] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: seamless operating system integration of peer-to-peer DMA between SSDs and GPUs. In *Proceedings of the Seventeenth USENIX Annual Technical Conference*, volume 17 of *USENIX ATC'17*, 2017.

[38] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, u 2011.

[39] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, volume 215 of *FAST'03*, 2003.

[40] Jeronimo Castrillon, Matthias Lieber, Sascha Klüppelholz, Marcus Völp, Nils Asmussen, Uwe Assmann, Franz Baader, Christel Baier, Gerhard Fettweis, Jochen Fröhlich, Andrés Goens, Sebastian Haas, Dirk Habich, Hermann Härtig, Mattis Hasler, Immo Huismann, Tomas Karnagel, Sven Karol, Akash Kumar, Wolfgang Lehner, Linda Leuschner, Siqi Ling, Steffen Märcker, Christian Menard, Johannes Mey, Wolfgang Nagel, Benedikt Nöthen, Rafael Peñaloza, Michael Raitza, Jörg Stiller, Annett Ungethüm, Axel Voigt, and Sascha Wunderlich. A hardware/software stack for heterogeneous systems. *IEEE Transactions on Multi-Scale Computing Systems*, 4(3):243–259, Jul 2018.

[41] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'10, pages 225–236. IEEE, 2010.

[42] Joel Coburn, Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. Seca: security-enhanced communication architecture. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES'05, pages 78–89. ACM, 2005.

[43] Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. An analysis of accelerator coupling in heterogeneous architectures. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC'15, pages 202:1–202:6, New York, NY, USA, 2015. ACM.

[44] Yi Cui, Zhaohui Zhong, Deli Wang, Wayne U. Wang, and Charles M. Lieber. High performance silicon nanowire field effect transistors. *Nano letters*, 3(2):149–152, 2003.

[45] Bruno da Silva, An Braeken, Erik H. D'Hollander, Abdellah Touhafi, Jan G. Cornelis, and Jan Lemeire. Comparing and combining GPU and FPGA accelerators in an image processing context. In *Proceedings of the 23rd International Conference on Field Programmable Logic and Applications*, FPL'13, pages 1–4. IEEE, 2013.

[46] R.H. Dennard, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974.

[47] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. Execution migration in a heterogeneous-ISA chip multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'12, pages 261–272, New York, NY, USA, 2012. ACM.

[48] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart SSDs: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1221–1230, New York, NY, USA, 2013. ACM.

[49] Adam Dunkels. Design and implementation of the lwIP TCP/IP stack. *Swedish Institute of Computer Science*, 2:77, 2001.

[50] Keith G. Erickson, M. Dan Boyer, and D. Higgins. NSTX-U advances in real-time deterministic PCIe-based internode communication. *Fusion Engineering and Design*, 133:104–109, 2018.

[51] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA'11, pages 365–376, New York, NY, USA, 2011. ACM.

[52] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys'06, pages 177–190, New York, NY, USA, 2006. ACM.

[53] L. Fiorin, G. Palermo, S. Lukovic, V. Catalano, and C. Silvano. Secure memory accesses on networks-on-chip. *IEEE Transactions on Computers*, 57(9):1216–1229, Sept 2008.

[54] Mario Flajslik and Mendel Rosenblum. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX Annual Technical Conference*, USENIX ATC'13, pages 333–346, San Jose, CA, 2013. USENIX.

[55] Claudio Föllmi. Applying the multikernel approach to a heterogeneous OMPA4460 SoC. `http://www.barrelfish.org/publications/ba-foellmic-hetero-panda.pdf`, 2013.

[56] Isaac Gelado, Enric Morancho, and Nacho Navarro. Experimental support for reconfigurable application-specific accelerators. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjuction with the International Symposium on Computer Architecture*, WIOSCA'06, pages 50–57, 2006.

[57] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA'16, pages 153–165, Piscataway, NJ, USA, 2016. IEEE Press.

[58] Sebastian Haas, Tobias Seifert, Benedikt Nöthen, Stefan Scholze, Sebastian Höppner, Andreas Dixius, Esther Pérez Adeva, Thomas Augustin, Friedrich Pauls, Sadia Moriam, Mattis Hasler, Erik Fischer, Yong Chen, Emil Matúš, Georg Ellguth, Stephan Hartmann, Stefan Schiefer, Love Cederström, Dennis Walter, Stephan Henker, Stefan Hänzsche, Johannes Uhlig, Holger Eisenreich, Stefan Weithoffer, Norbert Wehn, René Schüffny, Christian Mayr, and Gerhard Fettweis. A heterogeneous SDR MPSoC in 28 nm CMOS for low-latency wireless applications. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC'17, pages 47:1–47:6, New York, NY, USA, 2017. ACM.

[59] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July 2011.

[60] Norman Hardy. KeyKOS architecture. *ACM SIGOPS Operating Systems Review*, 19(4):8–25, Oct 1985.

[61] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The nizza secure-system architecture. In *Proceedings of the 1st International Conference on Collaborative Computing: Networking, Applications and Worksharing*, CollaborateCom'05, pages 10–pp. IEEE, 2005.

[62] Hermann Härtig, Michael Roitzsch, Carsten Weinhold, and Adam Lackorzynski. Lateral thinking for trustworthy apps. In *Proceedings of the IEEE 37th International Conference on Distributed Computing Systems*, ICDCS'17, pages 1890–1899. IEEE, 2017.

[63] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integration of message passing and shared memory in the Stanford FLASH multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'94, pages 38–50, New York, NY, USA, 1994. ACM.

[64] Jörg Henkel, Heba Khdr, Santiago Pagani, and Muhammad Shafique. New trends in dark silicon. In *Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference*, DAC'15, pages 1–6. IEEE, 2015.

[65] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *SIGOPS Operating Systems Review*, 40(3):80–89, u 2006.

[66] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. SemperOS: A distributed capability system. In *2019 USENIX Annual Technical Conference*, USENIX ATC'19, pages 709–722, Renton, WA, 2019. USENIX Association.

[67] Galen Hunt, George Letey, and Ed Nightingale. The seven properties of highly secure devices. *Technical report MSR-TR-2017-16*, 2017.

[68] K.U. Jarvinen and J.O. Skytta. High-speed elliptic curve cryptography accelerator for koblitz curves. In *Proceedings of the 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 109–118, April 2008.

[69] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA'17, pages 1–12, New York, NY, USA, 2017. ACM.

[70] Heiko Kalte and Mario Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, FPL'05, pages 223–228. IEEE, 2005.

[71] Tomas Karnagel, Rene Mueller, and Guy M. Lohman. Optimizing GPU-accelerated group-by and aggregation. *ADMS@ VLDB*, 8:20, 2015.

[72] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott A Brandt. Gdev: First-class GPU resource management in the operating system. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, pages 401–412. Boston, MA;, 2012.

[73] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 67–81, New York, NY, USA, 2016. ACM.

[74] John H. Kelm, Daniel R. Johnson, William Tuohy, Steven S. Lumetta, and Sanjay J. Patel. Cohesion: An adaptive hybrid memory model for accelerators. *IEEE micro*, 31(1):42–55, 2011.

[75] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. GPUnet: Networking abstractions for GPU programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 201–216, Berkeley, CA, USA, 2014. USENIX Association.

[76] Seonbong Kim and Joon-Sung Yang. Optimized I/O determinism for emerging NVM-based NVMe SSD in an enterprise system. In *Proceedings of the 55th Annual Design Automation Conference*, DAC'18, page 56. ACM, 2018.

[77] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 207–220, New York, NY, USA, 2009. ACM.

[78] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.

[79] Georg Kotheimer. Network support on M3. `https://os.inf.tu-dresden.de/papers_ps/kotheimer_beleg.pdf`, 2018.

[80] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. Hardware acceleration in the IBM PowerEN processor: Architecture and performance. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT'12, pages 389–400. ACM, 2012.

[81] Nasser Kurd, Muntaquim Chowdhury, Edward Burton, Thomas P Thomas, Christopher Mozak, Brent Boswell, Praveen Mosalikanti, Mark Neidengard, Anant Deval, Ashish Khanna, et al. Haswell: A family of IA 22 nm processors. *IEEE Journal of Solid-State Circuits*, 50(1):49–58, 2015.

[82] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA'94, pages 302–313, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[83] Adam Lackorzynski and Alexander Warg. Taming subsystems: Capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES'09, pages 25–30, New York, NY, USA, 2009. ACM.

[84] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP'91, pages 165–182, New York, NY, USA, 1991. ACM.

[85] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP'95, pages 237–250, New York, NY, USA, 1995. ACM.

[86] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with smart pipes: Designing SoC accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA'13, pages 36–47, New York, NY, USA, 2013. ACM.

[87] Felix Xiaozhu Lin and Xu Liu. Memif: Towards programming heterogeneous memory asynchronously. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 369–383, New York, NY, USA, 2016. ACM.

[88] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A mobile operating system for heterogeneous coherence domains. *ACM Transactions on Computer Systems*, 33(2):4:1–4:27, u 2015.

[89] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, abs/1801.01207, 2018.

[90] Richard J. Lipton and Lawrence Snyder. A linear time algorithm for deciding subject security. *Journal of the ACM (JACM)*, 24(3):455–464, 1977.

[91] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. PuDianNao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 369–381. ACM, 2015.

[92] Xiaocheng Liu, Ziming Zhong, and Kai Xu. A hybrid solution method for CFD applications on GPU-accelerated hybrid HPC platforms. *Future Generation Computer Systems*, 56:759–765, 2016.

[93] Zhiduo Liu, Aaron Severance, Satnam Singh, and Guy GF Lemieux. Accelerator compiler for the venice vector processor. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA'12, pages 229–232. ACM, 2012.

[94] Björn Lüssem, Max L. Tietze, Hans Kleemann, Christoph Hoßbach, Johann W. Bartha, Alexander Zakhidov, and Karl Leo. Doped organic transistors operating in the inversion and depletion regime. *Nature communications*, 4:2775, 2013.

[95] K. Mackenzie, J. Kubiatowicz, M. Frank, W. Lee, W. Lee, A. Agarwal, and M.F. Kaashoek. Exploiting two-case delivery for fast protected messaging. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, HPCA'98, pages 231–242, Feb 1998.

[96] Stephen Mallon, Vincent Gramoli, and Guillaume Jourjon. DLibOS: Performance and protection with a network-on-chip. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'18, pages 737–750, New York, NY, USA, 2018. ACM.

[97] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, volume 2, pages 21–33, 2007.

[98] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC'08, pages 38:1–38:11, Piscataway, NJ, USA, 2008. IEEE Press.

[99] Dominik Menzi. Support for heterogeneous cores for Barrelfish. `http://www.barrelfish.org/publications/menzi-master-heterogeneouscores.pdf`, 2011.

[100] Inanc Meric, Natalia Baklitskaya, Philip Kim, and Kenneth L. Shepard. RF performance of top-gated, zero-bandgap graphene field-effect transistors. In *Proceedings of the 2008 IEEE International Electron Devices Meeting*, IEDM'08, pages 1–4. IEEE, Dec 2008.

[101] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L. Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K. Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D. Flickner, William P. Risk, Rajit Manohar, and Dharmendra S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[102] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, HPCA'15, pages 126–136. IEEE, 2015.

[103] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119. Eurographics Association, 2003.

[104] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'07, pages 89–100, New York, NY, USA, 2007. ACM.

[105] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 221–234, New York, NY, USA, 2009. ACM.

[106] Benedikt Noethen, Oliver Arnold, Esther Perez Adeva, Tobias Seifert, Erik Fischer, Steffen Kunze, Emil Matus, Gerhard Fettweis, Holger Eisenreich, Georg Ellguth, et al. 10.7 A 105GOPS 36mm 2 heterogeneous SDR MPSoC with energy-aware dynamic scheduling and iterative detection-decoding for 4G in 65nm CMOS. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, ISSCC'14, pages 188–189. IEEE, 2014.

[107] Vincent Nollet, Paul Coene, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Designing an operating system for a heterogeneous reconfigurable SoC. In *Proceedings of the International Parallel and Distributed Processing Symposium*, IPDPS'03, pages 7–pp. IEEE, 2003.

[108] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. Accelerating binarized neural networks: comparison of FPGA, CPU, GPU, and ASIC. In *Proceedings of the 26th International Conference on Field-Programmable Logic and Applications*, FPT'16, pages 77–84. IEEE, 2016.

[109] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. Border control: Sandboxing accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO'15, pages 470–481, New York, NY, USA, 2015. ACM.

[110] John K. Ousterhout et al. Scheduling techniques for concurrent systems. In *ICDCS*, volume 82, pages 22–30, 1982.

[111] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, pages 407–418, New York, NY, USA, 2013. ACM.

[112] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 593–606, New York, NY, USA, 2015. ACM.

[113] Mike Parker, Al Davis, and Wilson Hsieh. Message-passing for the 21st century: Integrating user-level networks with SMT. In *Proceedings of the 5th Workshop on Multithreaded Execution, Architecture and Compilation*, 2001.

[114] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems*, 33(4):11, 2016.

[115] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing systems*, 8(2):221–254, 1995.

[116] J. Porquet, A. Greiner, and C. Schwarz. NoC-MPU: A secure architecture for flexible co-hosting on shared memory MPSoCs. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, DATE'11, pages 1–4, March 2011.

[117] Andrew Putnam, Aaron Smith, and Doug Burger. Dynamic vectorization in the E2 dynamic multicore architecture. *SIGARCH Computer Architecture News*, 38(4):27–32, a 2011.

[118] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark Horowitz. Convolution engine: Balancing efficiency and flexibility in specialized computing. *Communications of the ACM*, 58(4):85–93, Mar 2015.

[119] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC'14, pages 110–119. IEEE, 2014.

[120] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[121] Sebastian Reimers. Extension of an accelerator-friendly in-memory file system for persistent storage.
`https://os.inf.tu-dresden.de/papers_ps/reimers_m3fs.pdf`, 2018.

[122] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9:1–9:32, Aug 2013.

[123] Phil Rogers and AC Fellow. Heterogeneous system architecture overview. In *Hot Chips*, volume 25, 2013.

[124] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP'11, pages 233–248, New York, NY, USA, 2011. ACM.

[125] L. Rota, M. Vogelgesang, LE. Ardila Perez, M. Caselle, S. Chilingaryan, T. Dritschler, N. Zilio, A. Kopmann, M. Balzer, and M. Weber. A high-throughput readout architecture based on PCI-Express Gen3 and DirectGMA technology. *Journal of Instrumentation*, 11(02):P02007, 2016.

[126] Robert Schöne, Daniel Molka, and Michael Werner. Wake-up latencies for processor idle states on current x86 processors. *Computer Science-Research and Development*, 30(2):219–227, 2015.

[127] Sudharsan Seshadri, Mark Gahagan, Meenakshi Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 67–80, 2014.

[128] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'18. USENIX Association, 2018.

[129] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*, ISCA'14, pages 97–108. IEEE, 2014.

[130] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. Co-designing accelerators and SoC interfaces using gem5-aladdin. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'16, pages 1–12. IEEE, 2016.

[131] Min Si and Yutaka Ishikawa. Design of direct communication facility for many-core based accelerators. In *Processing of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW'12, pages 924–929. IEEE, 2012.

[132] Mark Silberstein. OmniX: An accelerator-centric OS for omni-programmable systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS'17, pages 69–75, New York, NY, USA, 2017. ACM.

[133] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, pages 485–498, New York, NY, USA, 2013. ACM.

[134] Harald Simmler, Lorne Levinson, and Reinhard Männer. Multitasking on FPGA coprocessors. In *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, pages 121–130, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[135] Hayden Kwok-Hay So and Robert Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Transactions on Embedded Computing Systems*, 7(2):14:1–14:28, Jan 2008.

[136] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[137] Udo Steinberg and Bernhard Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys'10, pages 209–222, New York, NY, USA, 2010. ACM.

[138] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7–1, 2015.

[139] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[140] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on GPUs. In *Proceedings of the 41st International Symposium on Computer Architecture*, ISCA'14, pages 193–204. IEEE, 2014.

[141] Michael B. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference*, DAC'12, pages 1131–1136. IEEE, 2012.

[142] Michael B. Taylor. A landscape of the new dark silicon design regime. *IEEE Micro*, 33(5):8–19, Sept 2013.

[143] Michael B. Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA'04, Washington, DC, USA, 2004. IEEE Computer Society.

[144] Tung Thanh-Hoang, Amirali Shambayati, Calvin Deutschbein, Henry Hoffmann, and Andrew A. Chien. Performance and energy limits of a processor-integrated FFT accelerator. In *Proceedings of the 2014 IEEE High Performance Extreme Computing Conference*, HPEC'14, pages 1–6. IEEE, 2014.

[145] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. Fast convolutional nets with fbfft: A GPU performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014.

[146] Ashish Venkat and Dean M. Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA'14, pages 121–132, Piscataway, NJ, USA, 2014. IEEE Press.

[147] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'11, pages 163–174. IEEE, 2011.

[148] Carsten Weinhold. Reducing size and complexity of the security-critical code base of file systems. `https://os.inf.tu-dresden.de/papers_ps/weinhold-phd.pdf`, 2013.

[149] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. Technical report, 2018.

[150] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, Apr 2009.

[151] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27:15–31, 10 2007.

[152] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA'14, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.

[153] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA'13, pages 249–260, New York, NY, USA, 2013. ACM.

[154] Wei Yu and Yun He. A high performance CABAC decoding architecture. *IEEE Transactions on Consumer Electronics*, 51(4):1352–1359, Nov 2005.

[155] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling cores, kernels, and operating systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 17–31, Berkeley, CA, USA, 2014. USENIX Association.