M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores

Nils Asmussen[†] Marcus Völp[†]* Benedikt Nöthen[‡] Hermann Härtig[†] Gerhard Fettweis[‡]

Operating-Systems Chair[†] and Vodafone Chair Mobile Communications Systems[‡]
Technische Universität Dresden
Nöthnitzer Straße 46, 01187 Dresden, Germany
{nils.asmussen,marcus.voelp,benedikt.noethen,hermann.haertig,gerhard.fettweis}@tu-dresden.de

Abstract

In the last decade, the number of available cores increased and heterogeneity grew. In this work, we ask the question whether the design of the current operating systems (OSes) is still appropriate if these trends continue and lead to abundantly available but heterogeneous cores, or whether it forces a fundamental rethinking of how systems are designed. We argue that:

1. hiding heterogeneity behind a common hardware interface unifies, to a large extent, the control and coordination of cores and accelerators in the OS,
2. isolating at the network-on-chip rather than with processor features (like privileged mode, memory management unit, ...), allows running untrusted code on arbitrary cores, and
3. providing OS services via protocols over the network-on-chip, instead of via system calls, makes them accessible to arbitrary types of cores as well.

In summary, this turns accelerators into first-class citizens and enables a single and convenient programming environment for all cores without the need to trust any application.

In this paper, we introduce network-on-chip-level isolation, present the design of our microkernel-based OS, M^3, and the common hardware interface, and evaluate the performance of our prototype in comparison to Linux. A bit surprising, without using accelerators, M^3 outperforms Linux in some application-level benchmarks by more than a factor of five.

* New affiliation: SnT-CritiX, University of Luxembourg

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA.

DOI: http://dx.doi.org/10.1145/2872362.2872371

Categories and Subject Descriptors C.1.3 [*Other Architecture Styles*]: Heterogeneous (hybrid) systems; D.4.7 [*Organization and Design*]: Distributed systems; D.4.6 [*Security and Protection*]: Access controls

Keywords Heterogeneous architectures; Accelerators; On-chip networks; Operating systems; Capabilities

1. Introduction

In recent years, computer architecture followed two major trends: increased parallelism and heterogeneity. We expect that future systems will continue to follow these trends and confront us with even more heterogeneous cores.

1.1 Heterogeneity

Today, workloads in many application areas run on computer systems, which combine general-purpose CPUs and special-purpose accelerators to achieve the required performance and energy efficiency. For example, in addition to CPU/GPU combinations, which have become mainstream in many daily life devices, we find accelerators to assist in cryptography [21], multimedia [47], object caching [28], signal processing [5, 37], the processing of massive multidimensional data-sets [46] and machine learning [30], to mention just a few.

In their respective domains, special-purpose solutions show large performance and energy advantages over general-purpose solutions. For example, Lim et al. have presented an FPGA-based accelerator for memcached [28] that shows a 16 times performance-per-watt improvement over an Atom CPU. Liu et al. demonstrated that an machine learning accelerator [30] can achieve 20% better performance than a GPU-based solution, while requiring 128 times less energy.

Increased leakage currents and limited heat dissipation capabilities already make it impossible to operate all parts of a chip at full frequency over extended periods of time. To mitigate these effects, which lead to dark silicon, computer architects already proposed mixtures of largely heterogeneous processing elements to accelerate frequently recurring patterns in general-purpose workloads [43], and also to

integrate special-purpose accelerators [18] for specific application domains.

In summary, we expect hardware platforms to continue to increase in heterogeneity. Unfortunately, current operating-systems (OSes) rely on specific hardware features such as a privileged mode, exceptions, and a memory management unit (MMU). Furthermore, the OS design requires a kernel, available on every core that should be a *first-class citizen*, i.e., which has access to OS services and can run untrusted code. Of course, special-purpose cores could be equipped with all the hardware features required by current OSes to make them first-class. However, this leads either to large porting efforts for the OS kernel because it needs to run on various different cores. Or it forces hardware vendors to produce cores that are equal at the architectural level, which limits the benefits of accelerators since their advantages stem also from their architectural differences to general-purpose cores.

1.2 Second-class vs. First-class Citizens

Accelerators are currently treated as devices, i.e., *second-class citizens*. Although this model is suitable for timers, network cards or storage devices, we consider it less appropriate for accelerators that execute software or for reconfigurable circuits like FPGAs, which are becoming increasingly common. At first, as more and more accelerators execute increasingly complex software, their need for OS services like filesystems, network stacks or inter-process communication mechanisms increases. Second, applications might want to offload parts of their computation onto accelerators and of course, do not want to switch to a completely different and limited programming environment. We therefore believe that it is crucial to remove the barrier between general-purpose cores and accelerators and treat both as first-class citizens.

1.3 Abundantly Available Cores

Although the increase of single-thread performance has mostly stopped due to the break-down of Dennard scaling [12], Moore's law still leads to about twice as many integrated transistors every 18 months, which are used to integrate more cores on a chip as a response to the stop of Dennard scaling. Today, there are already research projects with hundreds [3] to even one thousand [24] cores on a chip and we believe that chips with that many cores will become commercially available in the future. These large number of cores, while probably not all usable at all times due to power and heat constraints, offer new opportunities and trade offs. For example, having enough cores allows the OS to place each application on its own core. By cleverly balancing the load and powering off unused cores, the OS can make sure that it stays within the power budget. This approach is beneficial because it avoids both the direct and the indirect costs of context switches. That is, applications no longer share hardware resources like registers, caches, and translation lookaside buffers (TLBs).

1.4 Contributions

Our goals are therefore to, at first, integrate arbitrary cores as first-class citizens into the system, and second, leverage the fact that cores are abundantly available. To achieve that, we integrate cores and memories into a packet-switched network-on-chip (NoC) and equip each core with a *data transfer unit* (DTU) as the common hardware component. The DTU, offering message passing and memory access, is thereby the only means for the core to communicate with other cores or memories. Thus, controlling the DTU allows us to control the core and therefore also the software running on the core. OS services like filesystems and network stacks are provided based on a core-neutral communication protocol between DTUs. Our contributions are as follows:

1. We introduce data transfer units and the concept of NoC-level isolation, which enables uniform control and coordination of heterogeneous cores.
2. We evaluate new OS design opportunities and trade-offs that become possible through NoC-level isolation and abundantly available cores.
3. We describe the design of our OS prototype, M^3, where we combine the above findings into a microkernel-based system. As a proof of concept, we implemented a filesystem and pipes. M^3 is available as open source at `https://github.com/TUD-OS/M3`.
4. Finally, we evaluate the performance of our prototype to show the feasibility of the OS design. By trading system utilization for supporting heterogeneous cores and by accelerating data transfers via the DTU, M^3 outperforms Linux as a representative of a traditional OS by more than a factor of five in some application-level benchmarks.

The rest of the paper is organized as follows: After relating this paper to previous work in Section 2, we introduce our approach to tame heterogeneous manycores based on NoC-level isolation and DTUs in Section 3. In Section 4, we detail our hardware/OS co-design of the DTU and the OS prototype M^3. Finally, we evaluate our prototype in Section 5, conclude in Section 6, and sketch directions for future work in Section 7.

2. Related Work

As this paper is about a hardware/operating-system co-design, the related work can be split into similar hardware components and similar OS designs.

2.1 Hardware-Level Isolation and Message Passing

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

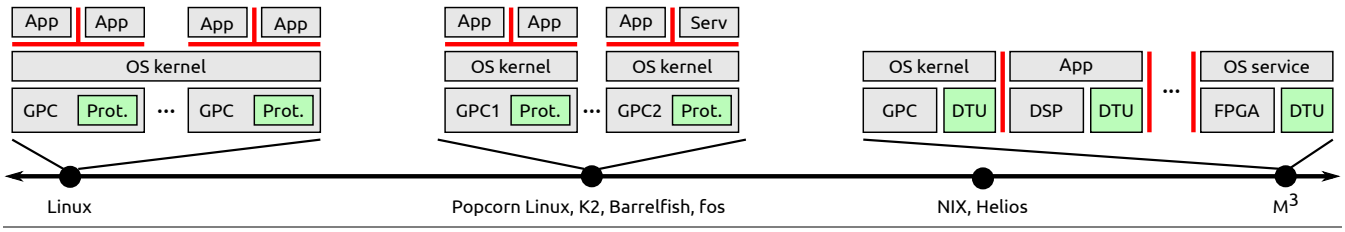

Figure 1. Design space for heterogeneity, enforcing isolation (red lines) and providing OS services.

devices. Furthermore, hardware components have been proposed to connect heterogeneous cores over a NoC in a secure way [16, 36]. However, all of them are dealing with memory accesses only. Instead, the purpose of the DTU is to abstract from the heterogeneity of the cores and provide the features that are required to make all cores first-class citizens. In particular, this includes the ability to perform secure and efficient message passing between cores.

Besides approaches for memory accesses, various work [4, 20, 25, 31, 34] has been done on (user-level) message passing architectures. For example, Alpert et al. describe protected message passing in userland on the Shrimp multicomputer [4]. Protection is thereby based on virtual memory support in the core. Similarly, the MAGIC component in the FLASH multiprocessor [20, 25], is designed as a core that executes software to allow the implementation of a variety of protocols like cache coherency. It does also require cooperation with an OS kernel. In summary, all approaches rely on core-specific features like virtual memory and/or an OS kernel on the sending/receiving core for special cases. Since our goal is to integrate arbitrary cores into the system and simple accelerators in particular, the DTU has to be core-agnostic and needs to handle all operations without involving an OS kernel on the same core.

2.2 Operating Systems

When exploring related OS work and comparing it with our design, we find different approaches to heterogeneity, isolation, and providing OS services. As depicted in Figure 1, traditional OSes like Linux are built to execute a shared kernel on homogeneous cores. Isolation between the OS kernel and applications and between applications is thereby implemented via processor features like a privileged mode, exceptions, and MMUs (denoted by *Prot.* for protection features in the figure) that we find in general-purpose cores (GPC). OS services are offered via system calls that perform a mode switch from user mode into privileged mode.

To let Linux support heterogeneous cores, approaches like Popcorn Linux [7] and K2 [29] have been suggested, which run multiple Linux instances on potentially heterogeneous cores. Since they are still based on Linux running on general-purpose cores, isolation and OS services are realized in the traditional way. Like with standard Linux, accelerators are treated as devices.

Moving further to the right, we find the multi-kernel design with Barrelfish [8] as one implementation. Barrelfish aims to support many heterogeneous cores as well and designs the OS as a distributed system to improve scalability. Similarly, fos [44] targets manycores and thus introduces the concept of *service fleets* that are spread among the whole chip. Both Barrelfish and fos use message passing to access OS services that may run on a different core. However, the design of Barrelfish and fos is based on the traditional way of isolation and executing a kernel on every core, required for message passing and resource access in case of fos and for memory isolation and capability management in Barrelfish.

NIX [6], based on Plan 9 [35], 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*, i.e., interference with the application caused by the OS through e.g., interrupt handling. Application cores can still access OS services by communicating via message passing with a time sharing core. Although NIX supports 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. In contrast to NIX, our approach does not rely on this.

Helios [33], a derivative from Singularity [14], reduces the requirements one step further by using software isolation instead of address space protection. Thereby, neither an MMU nor a privileged mode is required. However, Helios still executes a *satellite kernel* on every core and requires a timer device, an interrupt controller, exceptions, and a quite large amount of memory (at least 32 MiB).

In this paper, we evaluate the rightmost point in this spectrum (see Figure 1) by removing all requirements on processor features. We achieve that by abstracting the heterogeneity of the cores via a common hardware component per core, called DTU, which contains all the features that are required for an OS to treat the attached core as a first-class citizen. This allows us to hide any kind of processor, including very simple ones, behind a DTU. For example, a general-purpose core, a digital signal processor (DSP), an application-specific integrated circuit (ASIC), an FPGA, etc. Based on the DTU, we isolate at the NoC-level by remotely controlling the communication capabilities of DTUs and offer OS services via core-neutral communication protocols. In this paper, we use the term *processing element* (PE) to de-

note the combination of core, local memory (scratchpad or cache) and DTU.

Similar to M^3, GPUfs [40], GPUnet [22] and PTask [39] strive to make GPUs first-class citizens. However, they specifically target GPUs, while we aim to find a general solution to integrate arbitrary PEs as first-class citizens.

3. Taming Heterogeneous Manycores

To address the problem of heterogeneous PEs, we move the kernel to dedicated PEs and thereby let the applications run on bare-metal, as shown in Figure 1. That is, no applications are running on the *kernel PEs*, while no kernel is running on the *application PEs*. Similar to previous works [6, 10, 41], system calls are not handled on the same core by performing a mode switch, but by sending a message over the DTU to the corresponding kernel PE. However, the main motivation in the mentioned works is the improvement of performance, while ours is the support of heterogeneous cores.

Despite the differences between the kernel in M^3 and a traditional kernel, they share their main responsibility: making the final decision of whether an operation is allowed or not. Typically, a traditional kernel, in contrast to applications, is running in the privileged processor mode. With M^3, privilege is not defined by the processor mode, but by the DTU. Similarly to the processor mode, all DTUs are privileged at boot, i.e., any communication is allowed. During boot, the DTUs of the application PEs are downgraded by the kernel to become unprivileged. Due to the similarities to traditional kernels, we also use the term *kernel* for the entity that exercises the mentioned responsibility and the term *kernel PE* for naming the entire PE on which this entity runs.

3.1 Data Transfer Unit

In our approach, a DTU is a small hardware component that is present in each PE and thereby serves two purposes:

1. It is the only interface for the PE to PE-external resources like other PEs or memories.
2. It abstracts from the heterogeneity of the PEs to allow a uniform control of different PEs.

For our OS design, the DTU needs to support both message passing and remote memory access. Message passing is essential for the PE-neutral protocols used to provide filesystems, network stacks, access to devices and so on, while memory access is required to use PE-external memories.

The DTU should be attached to the core as a device with memory mapped registers. In this way, the DTU can be core-agnostic and existing instruction set architectures can be reused without any change.

3.2 NoC-level Isolation

Since we assume that all PEs are potentially different, the processor features used for isolation in classical OSes are different in every PE or are not even available. Therefore, we enforce isolation at the NoC-level instead of within one core. That is, applications run on bare-metal and can thus use their core in any way they like. Instead, to isolate applications, we control the communication capabilities of the PE, i.e., control the exchange of information with other PEs or memories. Since the DTU is the only interface to other PEs or memories, controlling the DTU suffices to control the application on that PE. This is done by a kernel, running on a different PE. In this way, a kernel isolates the applications at the NoC-level by remotely controlling their DTUs.

3.3 Abundantly Available Cores

Although we expect that future platforms will have many cores, we do not assume that, at any point in time, cores outnumber the threads of execution in the system. Instead, we plan to support the multiplexing of a core[1] among a group of threads that can run on that same core, but to not context-switch periodically, but only if required. With this, we optimize for the common case with abundantly available cores: two interacting applications are running on separate cores at the same time and can thus directly communicate with each other. That is, communication does not mean to transition the execution from the sender to the receiver, but both are running in parallel and are putting their cores in a low power state when waiting. For a communication that involves longer wait times, we plan to inform the kernel about a potentially reusable core, which can then perform a context switch to another thread of execution, if necessary. In this case, the kernel needs to switch back to the old thread before the interrupted communication can be completed. We leave the details of context switching (and migration, because it requires the same mechanism) as future work.

3.4 Discussion

This approach has several advantages:

1. Applications do not share resources like registers, caches or TLBs with kernels and, in the common case, not with other applications as well. Therefore, when performing system calls, registers do not need to be saved or restored and, when the call is finished, cache or TLB misses do not happen because no entries have been evicted.
2. As no kernel is running on application PEs, supporting arbitrary PEs becomes much easier. First, the kernel PEs could be homogeneous. Second, only the application needs to be compiled for the targeted PE(s) and, if desired, be prepared for hardware features it wants to make use of. Thus, new application PEs can be added without requiring any change to a kernel.
3. Furthermore, as no application is running on the kernel PEs, a kernel does not need to run in privileged mode, support context switching or use paging to isolate itself

[1] This will be restricted to the subset of the cores that support it, i.e., some accelerators might be excluded.

from the application. 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.

4. OS kernels and services do not necessarily benefit from the same hardware features (instruction extensions, number of functional units, memory architecture etc.) as the application. Giving them different PEs allows to accelerate both in the best possible way.

The disadvantage of this design is the decrease in system utilization, because a PE is idling (for a certain time) if the application on that PE is waiting for an incoming message or the completion of a memory transfer. However, it is expected that the power consumption and heat generation will be the limiting factors in the future. That is, even if an OS could fully utilize all PEs at all times, physical limits will prevent it from doing so. Furthermore, abundantly available cores will allow the OS in most cases to not reuse the core for a different application immediately, since enough free cores are available. For this reason we believe that this can be relaxed in favor of supporting heterogeneous cores (which are controlled remotely) and an increase in performance, because the state of the application (e.g., in the cache and TLB) is kept during most communications.

Note that, despite the separation in kernel PEs and application PEs, it does not mean that they require different cores. In fact, the kernel of our prototype implementation does not need any specific processor feature and can thus run on any core, where a DTU is attached and for which a C++ compiler is available. But of course our approach *allows* to specialize a kernel PE by e.g., adding specific instructions to the core to accelerate certain operations, if desired.

4. Design and Implementation

To evaluate the feasibility of the described approach, we built a prototype for both the DTU and the OS. The DTU was thereby co-designed with the OS to reflect the requirements of the OS in the feature-set of the DTU. In particular, our goal is to treat all PEs as first-class citizens whether or not they support OS kernels. For example, the core might have no privileged mode or memory protection to protect a kernel. Another reason is that the core can be an FPGA that should only host the circuit to accelerate a specific computation.

4.1 Prototype Platform

To investigate how an OS could look like if PEs do not support an OS kernel, we chose to use the Tomahawk platform [5], which is a multiprocessor system-on-a-chip (MPSoC) for mobile communication applications. Tomahawk consists of multiple PEs, connected over a network-on-chip and one DRAM module. The PEs are Xtensa RISC cores that do not have a privileged mode and also no MMU. Furthermore, they employ a scratchpad memory (SPM) instead of caches as the only directly addressable memory. SPM is often used for accelerators, because in contrast to caches, they can be tailored to the accelerators needs by adding multiple independent SRAM banks to perform multiple accesses in parallel [11].

M^3 supports two versions of Tomahawk: The first is a silicon chip, which employs 8 PEs, each having a SPM of 32 KiB for code and 32 KiB for data and a predecessor of the here described DTU, that does only support data exchange, but no message passing with the features described in Section 4.4. The second is cycle-accurate simulator of Tomahawk, based on the Cadence Xtensa SystemC framework [9], that we modified to include the DTU described in this paper. It supports an arbitrary number of PEs, each having a SPM of 64 KiB for code and 64 KiB for data. Although this version of Tomahawk is currently only available as a simulator, it has already been sent to the chip manufacturer and will thus also be available as a silicon chip next year. M^3 runs on both Tomahawk versions, but we describe and evaluate the simulator-based version because it supports more PEs and employs the described DTU (on the current hardware, some features need to be emulated in software).

4.2 Limitations

Our prototype platform has currently two major limitations:

1. Instead of a cache, a core employs only a small SPM, limiting the size of code and data and thus the complexity of applications that can run on it (without manually managing it like a cache).
2. Virtual memory is not supported, but a core accesses the SPM and external memory with physical addresses. The SPM can be accessed directly with load/store instructions, whereas the external DRAM (or other PE's SPMs) requires an explicit transfer into the SPM over the DTU first, leading to a distributed memory architecture.

However, to us, this was not a limitation, but a feature: simple accelerators, which we strive to support in particular, typically lack these features. Starting with a platform that consists of simple accelerator-like cores only, allowed us to explore how to design the system in a way that these cores are supported as first-class citizens. In future work, we will address these limitations to optionally support caches and virtual memory, as described in Section 7.

4.3 Overview and Terminology

Before diving into the details, this section briefly sketches the general concepts of isolation, communication, and remote memory access, and introduces the terminology used in this paper. At the hardware side, as shown in Figure 2, the DTU consists of a number of *endpoints* (EPs). Each endpoint can be configured to be a *send endpoint*, a *receive endpoint*, or a *memory endpoint*. The configuration registers (`buffer`, `target`, `credits`, and `label`) are only writable by kernel PEs, while the `data` register is writable by the application PE as well. Initially, all registers are writable at all PEs

and the kernels downgrade the permissions at the application PEs during boot[2]. To let applications communicate, a kernel establishes communication *channels* by remotely configuring endpoints. Afterwards, no kernel PE is involved, but the sender and receiver communicate directly (as indicated by the thick arrow in Figure 2).

At the software side, the kernels provide *Virtual Processing Elements* (VPEs) as an abstraction for PEs. Applications consist of at least one VPE, whereas each VPE is assigned to exactly one PE at any point in time. The kernels use *capabilities* [13] to manage the permissions of the VPEs by maintaining a capability table per VPE. In Figure 2, the receiver uses a *receive gate* object, which is associated with the *receive capability* and bound to a receive endpoint. Analogously, the sender uses a *send gate* associated with the *send capability* and bound to a send endpoint. For memory access, the concept is analogous, but the *target* register at the send endpoint specifies a region of memory instead of a receive endpoint.

this case. As the memory transfer occurs without involving an OS kernel and without involving software on the passive side (where the data is read from or written to), it is a form of RDMA.

An endpoint is therefore a hardware representation of a capability, similar to CHERI [45]. In contrast to CHERI, we did not extend the instruction set architecture, but the DTU acts as a device attached to the core.

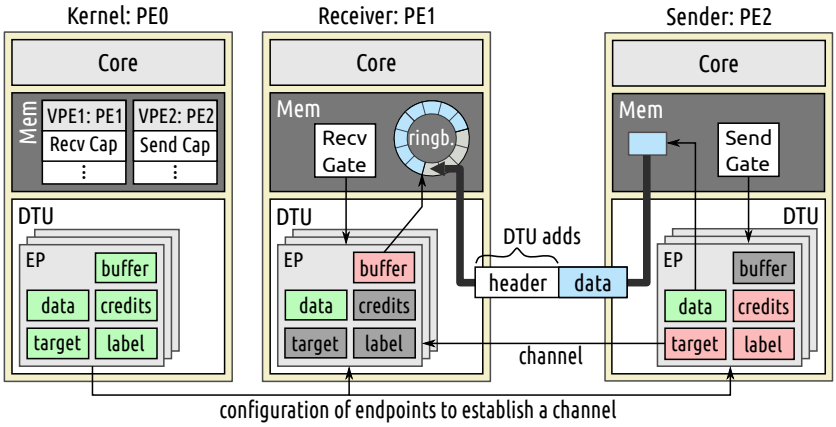

Figure 2. Communication between two PEs via the DTUs, set up by a kernel. The receiver has a receive endpoint configured for receiving messages and the sender has a send endpoint for sending messages to the receive endpoint.

4.4 Data Transfer Unit

For our prototype, we built a DTU and integrated one instance in each PE of the Tomahawk platform.

4.4.1 Endpoints

To support both memory access and message-based communication we provide endpoints to establish communication channels. Each DTU contains a set of endpoints, which can be configured for different operations.

With a send endpoint, a message can be sent by specifying the address and size of the data to send via the `data` register. This requires that the `target` register has been configured to point to the receive endpoint and that the registers `credits` and `label` have been set accordingly (the following sections provide more detail on them). The `buffer` register is not used for send endpoints.

For receive endpoints, the `buffer` register specifies the location and size of the ringbuffer, which can only be done by a kernel PE (see Section 4.4.4). Received messages are placed into the ringbuffer by the DTU without involving any software. If permitted by the sender, a reply can be send via the `data` register. The other registers are unused.

For memory endpoints, the `target` register specifies the remote memory region and the permissions (read or write). When reading, the `data` register denotes the location the read data should be transferred to, while when writing, it denotes the data to write. `Label` and `credits` are not used in

[2] If desired, a kernel can also upgrade the permissions of an application PE again to turn it into a kernel PE, but this is not done in our prototype.

In the current implementation, the software polls a DTU register to wait for received messages or completed memory transfers. In future work, we will put the core into a low-power state and the DTU will wake it up for these events.

4.4.2 Messages

Messages consist of a header and a payload. The header is automatically prepended to the payload by the DTU and contains a label, the length of the message, and information for a potential reply (see Section 4.4.4). A label, originally introduced by KeyKOS [19] as *numeric tag*, is a value that is chosen by the receiver when the channel is created and unforgeable by the sender to securely identify the sender. Typically, the receiver 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.

We believe that device interrupts should be sent as messages as well to integrate them with the existing concepts. This would allow to wait for them as for any other message, interpose them, sent them to any PE, independent of the core, etc. However, we have not yet implemented this idea, because of the lack of devices in the prototype platform.

4.4.3 Ringbuffer

Ringbuffers at the receive endpoints allow receivers to simultaneously accept messages from multiple senders. Ringbuffers are allocated in the local memory of the PE and organized in fixed-size slots. The size of the slots, which cor-

responds to the maximum message size, is configurable per endpoint. Upon the reception of a message, the DTU writes the received message at the current write position in the ringbuffer and moves the write position forward. The software in turn advances the buffer's current read position to indicate that a message has been processed. To manage the ringbuffer space, we use a credit system, similar to the one in Intel QuickPath [2]. That is, the receiver hands out credits to its senders and thereby limits the number of messages per sender. If a sender has no credits left, message sending is denied by the DTU until the credits have been refilled by either the receiver (typically when replying) or an OS kernel. For most scenarios, the receiver should not hand out more credits than buffer space is available, because messages are dropped if no space is left.

4.4.4 Replies

A channel is unidirectional in the sense that only the sender can start a communication. But since a reply is an important and also performance-critical operation, the DTU supports direct replies on received messages without requiring a dedicated channel back to the sender.

To enable replies, the sender specifies any of its endpoints as receive endpoint for the reply and transmit this information to the receiver. We decided to store the reply information in the message header, i.e., in the receiver's ringbuffer. To reply, the receiver selects the message to reply to and the DTU accesses the message header to extract the destination. Naturally, storing this security-critical information at a software accessible location requires additional protection. Therefore, if replies should be enabled for a ringbuffer, an OS kernel ensures these ringbuffers are placed in read-only memory and do not overlap before configuring a receive endpoint.

4.5 Operating System

In parallel to the just described DTU, we have built an operating-system prototype called M^3, which is short for **m**icrokernel-based syste**m** for heterogeneous **m**anycores (or L4 [27] $\pm$ 1), and follows the design introduced in Section 3. M^3 consists of a kernel, running on a dedicated PE[3], OS services and the library *libm3*.

4.5.1 Microkernel Approach

We chose to build the OS as a microkernel-based system. That is, the kernel provides only the necessary mechanisms to let applications implement the actual functionality of the OS. In particular, filesystems, network stacks and drivers are therefore implemented as applications; hence we use the term *application* to refer to both user applications and OS services in this paper.

Besides the well known security and reliability advantages of microkernels, the DTU and our approach of having dedicated application and kernel PEs eliminates the most important criticism of microkernels: Since message passing is performed directly between applications via DTUs, without kernel involvement or context or address-space switches, it is very fast. As reported in Section 5, it is even faster than a Linux system call on Xtensa and ARM.

[3] In the future, multiple instances of the kernel will be supported.

4.5.2 Library

The library libm3 provides abstractions for communicating with the kernel or OS services, accessing files, using the DTU etc. and contains only little architecture-specific code. Currently, only the PE initialization is architecture-specific, which makes libm3 easily portable to architectures other than Xtensa. Due to the small SPMs in our prototype, which we believe is a common limitation of accelerators and other PEs without kernel support, libm3 provides lightweight abstractions rather than a POSIX-compliant environment. This choice also results in performance improvements, as demonstrated in the evaluation.

4.5.3 Capabilities

Inspired by L4 microkernels [23, 26, 42], M^3 uses capabilities to manage the permissions of applications. A capability is thereby a pair consisting of a kernel object and permissions for this object. The kernel maintains a table of capabilities per VPE, similar to the file descriptor table in UNIX systems.

M^3 provides two operations to *exchange* capabilities between VPEs (*delegate* and *obtain*) and one operation to undo the exchange (*revoke*):

1. *Delegate*: Grant another VPE access to a capability.
2. *Obtain*: Request access to a capability from another VPE.
3. *Revoke*: Undo all grants of a capability recursively.

Naturally, VPEs can only delegate and revoke capabilities they possess themselves. To revoke a capability recursively, i.e., including all grants, the kernel maintains a tree that records all delegation/obtain operations, similar to the *mapping database* found in some L4 microkernels [23, 26, 42].

In L4 microkernels, a capability exchange is performed via inter-process communication (IPC). In particular, a single message allows to exchange both information and capabilities. In contrast, in M^3, IPC does not involve the kernel and is instead performed directly between applications. However, to exchange capabilities, the kernel needs to be involved as it manages all capabilities. For that reason, M^3 offers two options to exchange capabilities, which are both realized as system calls to the kernel: first, one can exchange capabilities with another VPE, provided that the requesting VPE has a capability for the other. Second, applications can exchange capabilities with services. The second additionally involves communication with the service to negotiate details and allow the service to deny the capability exchange. The required communication channel between kernel and service is created at service registration. In the current prototype,

send and receive capabilities are virtualizable, i.e., they can be interposed by a proxy to e.g., monitor the communication, but memory capabilities are not. This is due to the lack of virtual memory support in our prototype platform.

4.5.4 Gates

A *gate* is the software abstraction used for communication and memory access over the DTU. M^3 provides three different kinds of gates:

- *receive gates* to receive messages,
- *send gates* to send messages to receive gates and
- *memory gates* to access remote memory.

As receive gates can receive messages at any point in time, they can only be moved to different endpoints or PEs after invalidating all connected send gates and ensuring that no transfer is in progress. In contrast, send gates and memory gates are easily movable. Hence, the kernel only allows to delegate/obtain send and memory capabilities, but not receive capabilities.

The kernel is responsible for managing the memories in the system. That is, it decides which application can use which parts of which memories. Our current prototype contains only one DRAM module and all PEs have SPMs instead of caches. Therefore, code, static data, heap, and stack are placed in the SPM, owned by the application currently running on that PE. Applications can however request a region of the DRAM via a system call to obtain a memory gate, which provides access to the region via explicit data transfers over the DTU.

To actually send messages or access memory, receiving a send or memory gate is not sufficient. A kernel needs to configure an endpoint at the application PE first, which is requested via a system call, because only a kernel has the permissions to configure endpoints. This indirection allows a kernel to defer the reply to the system call until the receiver is ready to receive messages. Furthermore, since the DTU provides only a limited number of endpoints (8 in our prototype platform) and applications might need more send gates or memory gates[4] than endpoints are available, multiplexing is used to share the endpoints among these gates. This is done by libm3, which checks before the usage of a gate whether the endpoint is appropriately configured. If not, the corresponding system call is performed.

4.5.5 Virtual PEs

Applications consist of multiple Virtual PEs, whereas each VPE is bound to a specific PE at any point in time. Although our current prototype does not yet support it, we plan to allow the migration of VPEs and also time-sharing of PEs among multiple VPEs. As far as the kernel is concerned, each VPE represents a single activity, i.e., does not use parallelism. Instead, new VPEs can be created to make use of more than one PE. However, an application is of course free to implement user-level thread-switching on a single PE or, if a PE supports it, use a timer interrupt and a small interrupt routine to switch between threads preemptively.

VPEs are created via a system call to the kernel, which instructs the kernel to select a suitable and unused PE. Thereby, the application can request a specific type of PE – for example a specific accelerator. If found, the kernel creates a VPE kernel object and a VPE capability for the VPE that requested it. Furthermore, the requesting VPE receives a memory gate for the memory that the VPE can access. In the current prototype, the memory gate refers to the local memory of the PE. If caches are available, it will be some PE-external memory and the kernel will configure the cache/DTU to access it.

The memory gate is used by libm3 to perform application loading as it provides complete control of the PE. libm3 supports two operations: First, applications can clone themselves onto this PE, similar to a `fork` in UNIX. Second, they can load an executable from the filesystem (see Section 4.5.8) onto the PE, similar to `exec`. The former is intended only for homogeneous PEs, the latter for both homogeneous and heterogeneous PEs.

When using the clone operation, libm3 transfers the code, static data, the used portion of the heap and the stack to the corresponding locations of the memory denoted by the memory gate. Since each PE has its own local memory, this does neither require virtual memory nor position independent code: the regions are simply copied to the same addresses in the other PE.

In the current C++ implementation of libm3, the clone operation can be used to execute C++ lambdas on other PEs, as the following example shows:

```
int a = 4, b = 5;
VPE vpe("test");
vpe.run([a,&b]() {
    auto &s = Serial::get();
    s << "Sum: " << (a + b) << "\n";
    return 0;
});
int result = vpe.wait();
```

At first, the VPE is created via a system call, selecting a PE of the same type as the requesting PE in this case. Second, all state is copied to the other PE and the given function is called. Arguments can be passed to the lambda by capturing them – either by value or by reference. However, since our prototype platform does not support to access PE-external memories with load/store instructions, values of the caller cannot be directly manipulated. Instead, this has to be done explicitly by using message passing or memory gates. This can be achieved by exchanging capabilities with the VPE before and/or after running the lambda. Note that

[4] Multiplexing is currently unsupported for receive gates, because they are more difficult to move, as described before.

the execution of the lambda happens asynchronously. The method `wait` can be called to wait until the lambda has finished execution and to receive an exit code. In case the lambda does not return after some time, the owner of the VPE capability could revoke it to let the kernel reset the associated PE, thereby making it available again for others.

4.5.6 Message Passing

We chose to make message passing asynchronous on the lowest level. That is, after a message is sent, which is done by programming the memory-mapped registers of the DTU accordingly, the application is free to perform other work until the reply has been received (if any is expected). This is because the kernel is executing on a different PE and not involved if two applications communicate. Since an asynchronous model is often more difficult, most abstractions of libm3 combine the send operation with waiting for the reply, making it synchronous again.

To simplify the message-based communication, libm3 provides easy-to-use C++ abstractions for marshalling and unmarshalling. Inspired by previous L4 marshalling frameworks [15], it overloads the C++ shift operators to marshal an object into the message or unmarshal it again.

4.5.7 Pipes

Based on the so far described basic primitives, libm3 offers a pipe concept, similar to a pipe in UNIX. On M^3, a pipe is a unidirectional data channel between exactly one writer and exactly one reader. The data is thereby transferred over a software-managed ringbuffer in the DRAM, to which both reader and writer have access. Although the SPM and the ringbuffer provided by the DTU could be used, we decided in favor for the DRAM, because the SPM might be too small, if an application needs several pipes to other applications. By using the DRAM, large ringbuffers can be used to maximize the parallelism of readers and writers.

To manage access to the ringbuffer, messages are used to synchronize reader and writer. That is, after writing new data to the ringbuffer, the writer notifies the reader with a message, which in turn will read the data from the ringbuffer, after it received the message. In this way, the state of the ringbuffer is exchanged with messages, while the ringbuffer in DRAM contains only the data that is transmitted through the pipe. It is important to note that, after setting up the pipe, the kernel is not involved in the communication. That is, the actual pipe-based communication happens directly between the PE of the reader and PE of the writer.

4.5.8 Filesystem

As a filesystem is an essential and performance-critical component of an OS, we have designed and implemented a filesystem, called *m3fs*, as a proof of concept for a service provided by an application in M^3. m3fs is currently an in-memory filesystem, because our prototype platform lacks persistent storage. However, the organization of the data has been chosen to be suitable for persistent storage as well, so that we can support it later.

For opening files, closing files, meta-data operations like `mkdir`, `link` etc., the service is contacted to perform these operations. The actual data transfers are done without involving m3fs, because the applications directly read or write to the memory, where the file is stored[5]. The application needs to ask m3fs for the locations of the file fragments that it wants to access first. m3fs will then delegate memory capabilities for the requested fragments in memory to the application. The concept is therefore somewhat similar to GoogleFS [17], where meta-data is separated from data storage as well.

The filesystem is organized like classical UNIX filesystems, consisting of a superblock, an inode and block bitmap, an inode table and directories with pointers to the inodes. The data of an inode is stored in a tree of tables containing extents. As in other modern filesystems [32, 38], an extent is a pair of a starting block number and a number of blocks. The reason for this organization is that the applications get access to the data in form of memory capabilities, representing contiguous pieces of memory. To maximize the performance and scalability, these pieces of memory should be as large as possible to reduce the number of times m3fs has to be contacted. Storing the contiguous pieces in the form of extents avoids the need to first scan through a list of block numbers to determine the contiguous pieces.

To relieve the application programmers from obtaining memory capabilities from m3fs, determining at which offset of that memory the application needs to access etc., libm3 offers POSIX-like abstractions (`open`, `read`, `write`, `seek`, `close`, ...) to the application. That is, the application uses a local buffer for reading and writing, and libm3 will translate that into memory reads or writes at the appropriate location and will, if necessary, request further memory capabilities.

In this way, we optimize the data transfers for performance and scalability, because we believe that they are more performance-critical than meta-operations. For example, because a file consists of a list of extents, each potentially having a different length, seeking gets more expensive. However, most seek operations can be done in libm3 by seeking within the already obtained memory capabilities (extents).

A further consequence of this design is that, even if m3fs is storing the data in random access memory, like in our prototype, the fragmentation of files (of how many extents they consist) is important. The reason is, that the more extents a file has, the more often the application needs to communicate with m3fs to request further memory capabilities. Additionally, the seeking performance suffers with an increasing number of extents. For that reason, write operations extend files by a large number of blocks at once to minimize the

[5] In the current prototype, files are already in DRAM. If persistent storage was involved, m3fs would first load the file into DRAM, i.e., into the buffer cache.

fragmentation and the close operation truncates it to the actually used space. As the evaluation in Section 5.7 shows, as long as the extents are sufficiently large, the performance degradation with multiple extents is minimal.

To support multiple filesystems, libm3 offers a virtual filesystem (VFS) that allows to mount filesystems at specific paths. Besides m3fs, it provides a pipe filesystem to integrate pipes into the VFS, making it transparent for applications whether they access a pipe or a file in m3fs.

5. Evaluation

Our evaluation strives to answer the following questions:

- How fast are system calls via DTU?
- How fast are file operations with m3fs?
- What is the performance impact of file fragmentation?
- How fast are OS-intensive applications?
- How far does M^3 scale with a single service/kernel?
- What performance can be achieved by accelerators?

5.1 Methodology

To answer these questions, we compare our prototype to Linux in as many benchmarks as possible. We used two general approaches: first, we conducted micro-benchmarks to compare the performance of system calls and filesystem operations. Second, to prevent large porting efforts, we obtained traces from applications on Linux to compare both OSes in more realistic settings. Due to prohibitively long simulation times, we could not use standard benchmarks, but were required to use short running applications.

For both M^3 and Linux we used cycle-accurate simulators, whose computation performances are equivalent. M^3 was running on the Tomahawk platform, which can be started with an arbitrary number of PEs, each containing 64 KiB of SPM for instructions and 64 KiB for data with no caches and no MMU. Linux 3.18 was running on the simulator provided by Cadence with an Xtensa core, which contains an instruction and data cache, each with a capacity of 64 KiB, and an MMU. Furthermore, we configured the cost for a cache miss on Linux to be equivalent to the transfer time for loading a cache line (32 Bytes) via the DTU. That is, loading data from DRAM takes the same time in both setups. Due to the larger variation of the results on Linux, we discarded the results of the first two runs of all benchmarks to ensure that the caches are warm. For results still subject to a standard deviation of more than 1%, we show error bars in the figures.

Unfortunately, Linux does not provide support for multiple PEs in the simulator. To achieve a fair comparison, we made sure that, if comparing to Linux, M^3 did not take advantage of multiple PEs, i.e., at no point in time multiple PEs were doing useful work in parallel. This is because a second PE was only involved if the first PE sent a message to the second, in which case the first one waited until the second one replied, so that the execution merely transitioned from PE to PE.

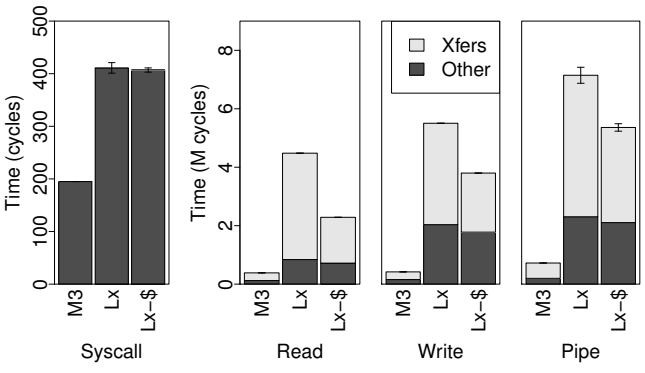

Figure 3. Comparison of system calls and file operations. Lx-$ shows the time on Linux without cache misses.

5.2 Linux on Xtensa vs. Linux on ARM

To ensure that our results are not specific to Xtensa, because e.g., Linux could be less optimized for Xtensa than for other architectures, we did also run some of the benchmarks on an ARM Cortex-A15. Apart from data transfers being slower on Xtensa, because the core does not have a cache line prefetcher (see Section 5.4 for more details), which prevents Xtensa to saturate the memory bandwidth, we saw comparable results. For example, a Linux system call requires 320 cycles on ARM and 410 cycles on Xtensa, creating a 2 MiB large file has 2.4 million cycles overhead on ARM and 2.2 million cycles on Xtensa, and copying a 2 MiB file has 3.2 million cycles overhead on both architectures.

5.3 System Calls

System calls on M^3 are done by sending a message via the DTU to the kernel PE and waiting for the reply of the kernel, while classical system calls switch from user mode into privileged mode on the same PE. To compare their performance, we performed a *null system call* on both M^3 and Linux, i.e., the system call function has an empty body. As can be seen in the left part of Figure 3, a system call on M^3 via DTU takes about 200 cycles, while it takes about 410 cycles on Linux. On M^3, the actual message transfers take about 30 cycles and the other 170 cycles are required for marshalling the messages, programming the DTU registers, unmarshalling the messages and figuring out the system call function to call. On Linux, most of the time is spent with saving and restoring the machine state, because the user application and the kernel share these hardware resources. Note that on Linux, we have removed outliers caused by interrupts from the results since they would have had large effects on the standard deviation in this case due to the short time for a single system call.

5.4 Filesystem and Pipe

Next, we conducted micro-benchmarks to evaluate m3fs and pipes. Since m3fs is an in-memory filesystem, we compared it to Linux's tmpfs. For all our benchmarks, we transferred 2 MiB of data, using a buffer size of 4 KiB, because 4 KiB is the sweet spot on Linux (M^3 benefits from larger buffer sizes until all available space in the SPM is used for the buffer). On M^3, the file was not fragmented (see Section 5.5 for the influence of fragmentation) and m3fs used a block size of 1 KiB. On Linux, tmpfs used a block size 4 KiB. We performed the following benchmarks:

1. *Read*: read a file, discarding the data,
2. *Write*: write precomputed data into a new file, and
3. *Pipe*: transfer data between two processes/VPEs.

On Linux, all three benchmarks used `read`/`write` system calls to perform the transfer. We also compared copying a file using `mmap` on Linux, but do not show it here, because of Linux's bad performance due to cache thrashing between the page fault handling of the kernel and the `memcpy` of the application. On M^3, the time for copying is the sum of the times for reading and writing the file.

To understand the differences between M^3 and Linux, we have broken down the results into the time for the data transfer ("Xfers") using the DTU or `memcpy` in the kernel, and the remaining time ("Other"). As Figure 3 shows, a large portion of the difference is made up by data transfers. This is because the data transfer on M^3 is performed by the DTU, which transfers 8 Byte per cycle without involving the core (similar to DMA). In contrast to that, Linux is copying the data via `memcpy`. Unfortunately, Xtensa does not have a cache line prefetcher, which could detect the data access pattern of `memcpy` and thus prefetch the next cache lines in the background. Therefore, `memcpy` cannot saturate the memory bandwidth on Xtensa. We believe that, with a cache line prefetcher and load/store instructions that operate on more bytes at once, the transfer time could reach the theoretical limit of 8 Byte per cycle, like with the DTU.

Apart from the differences for the data transfers, Figure 3 shows that M^3 incurs much less OS overhead. This is because on Linux, a system call is done for every 4 KiB block, doing in part the same work over and over again. On M^3, after the locations of the file fragments in the DRAM have been obtained, only libm3 is involved, which only needs to determine where to read or write. For example, `read` on Linux requires ∼380 cycles for entering/leaving the kernel, ∼400 cycles for retrieving the file pointer, doing security checks and executing function prologs/epilogs and ∼550 cycles for page cache related operations (`get`, `put`, etc.). M^3 on the other hand needs ∼70 cycles to get to the read function and ∼90 cycles to determine the location for reading.

Another difference is that Linux is overwriting each block with zeros before handing it out to a writing application, while m3fs can instruct the DTU to zero blocks in the background, i.e., in parallel to handling requests. In the benchmarks, we assume that enough zero blocks are available, which will be the case in realistic scenarios since there will always be time where m3fs is idling.

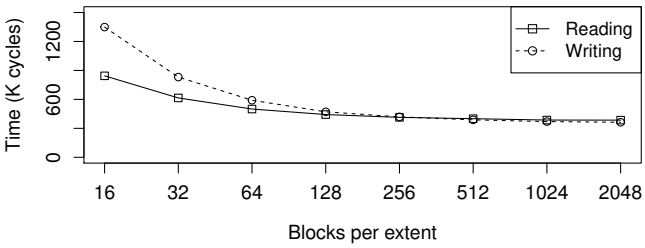

Figure 4. Read/write time, depending on file fragmentation

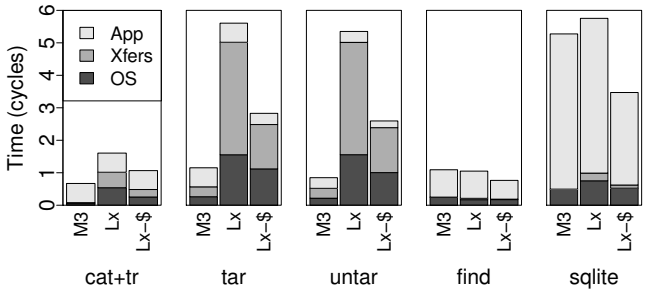

Figure 5. Comparison with application-level benchmarks. Lx-$ shows the time on Linux without cache misses.

5.5 Impact of File Fragmentation

As mentioned, the performance of reading and writing files with m3fs depends on file fragmentation, i.e., the number of extents the file consists of. To quantify this, we ran the read-/write benchmarks with varying numbers of extents per file, as shown in Figure 4. That is, for reading, the 2 MiB large file was prepared to have 16 to 2048 blocks per extent. And for writing we let the application allocate the corresponding number of blocks at once. As the results show, the sweet spot is 256 blocks, so that we chose to allocate that number of blocks at once when appending to a file. This leads to a good write performance and will probably also limit the file fragmentation, so that the read performance is good as well.

5.6 Application-level Benchmarks

To compare M^3 to Linux in more realistic settings, we performed application-level benchmarks. Since the computation performance of the cores is identical for both M^3 and Linux, we chose mostly applications that make extensive use of the OS. For a more compute-heavy application, we used sqlite. We ran the following five benchmarks:

1. *cat+tr*: creates a child process/VPE and lets it write a 64 KiB large file into a pipe, while the parent reads from

that pipe, replaces all occurrences of "a" with "b" and writes the result into a new file

2. *tar*: creates a tar archive with files between 60 and 500 KiB and 1.2 MiB in total
3. *untar*: unpacks the same tar archive
4. *find*: searches for files within a directory tree of 40 items
5. *sqlite*: creates a table, inserts 8 entries and selects them

As the first benchmark is simple to implement, we did that ourselves, using the same code for M^3 and Linux, except for programming against libm3 in case of the former. The reasons for including it in the benchmarks are that it uses all presented concepts of M^3 (application loading, pipes, and the filesystem) and that all of them have their equivalent on Linux. The other four benchmarks were first run on Linux with BusyBox [1], once running it with `strace` and again to record the execution times of the performed syscalls (to exclude the overhead of `strace`). The results were combined into a data structure that specifies which syscall to execute including its arguments. For all unsupported syscalls[6] and for the computation time, wait commands were inserted into the data structure. On M^3, we ran a program that replays the syscalls from the data structure using the corresponding API on M^3 or waits as long as specified. That is, we assume that computation and the unsupported syscalls require the same time on both systems.

The results are shown in Figure 5, which we have broken down into the time for the application (computation and on M^3 unsupported system calls), data transfers and the OS overhead to explain the differences between M^3 and Linux. Note however, that the results are in favor of Linux because on Linux, only the system call itself (e.g., `open`) is added to the OS overhead, while the time spent in a library call (e.g., `fopen`) is added to the application time. On M^3, the entire library call is added to OS overhead.

In case of *cat+tr*, M^3 is about twice as fast, which stems from `VPE::run` being faster than `fork`, less overhead for accessing files/pipes and the avoidance of context switches. Note that, as mentioned, although both the reader and writer are running time-multiplexed on one core on Linux, M^3 does not take advantage of the two cores involved. That is, like Linux with multiple cores, M^3 could achieve better performance by letting reader and writer work in parallel.

For *tar* and *untar*, M^3 requires only 20% and 16%, respectively, of the time Linux takes. This is caused on the one hand by faster data transfers and on the other hand by less OS overhead for `read` and `write`, as already described in Section 5.4. However, Linux does not suffer from many system calls in this case, because both benchmarks use `sendfile` to transfer the data.

[6] In these benchmarks, we ignored `access`, `ioctl`, `getuid`, `geteuid`, `getpid`, `getrlimit`, `setrlimit`, `rt_sigaction`, `rt_sigprocmask`, `brk`, `fcntl`, `fchown`, `chown`, `chmod` and `chdir`.

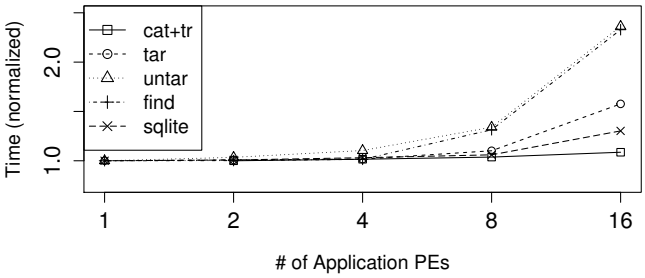

Figure 6. Scalability of the OS design, showing the average time per benchmark instance, normalized to the time with only one benchmark instance (flatter is better).

Find shows a different picture as Linux is slightly faster than M^3. This is because *find* consists mostly of `stat` calls, which work similarly on both systems. On Linux, a traditional syscall is done, which searches for the inode denoted by the given path, whereas M^3 does the same by calling m3fs, using message passing. Furthermore, `stat` is well optimized on Linux, so that M^3 is actually a bit slower.

Finally, *sqlite* is only slightly faster on M^3, because computation makes up the majority of the execution time.

5.7 Scalability

Another interesting point of this OS design is of course the scalability. Before using multiple instances of services or the kernel, we wanted to evaluate how far it scales using a single instance. That is, how many PEs a single instance can handle without significantly decreasing the performance. To evaluate that, we ran the application-level benchmarks again, with varying number of benchmark instances[7] in parallel. That is, we run one instance of the same problem on each core, so that equal time per instance means perfect scalability. Thereby, we replaced the reading/writing from/to the DRAM with a spinning loop of the same time to evaluate only the scalability of the software, i.e., we assume that the NoC (in terms of memory transfers; messages are still sent) and the DRAM scale perfectly.

As the results in Figure 6 indicate, all benchmarks scale very well with up to 4 instances, while degrading a bit with 8 instances. Using 16 instances, the performance of *find* and *untar* decreases significantly, while *tar* and *sqlite* are still acceptable and *cat+tr* show nearly no degradation. Thus, we believe that a single service/kernel instance can handle at least 8 PEs. Of course, *cat+tr* scales almost perfectly because after the setup phase, only the reader and writer communicate with each other. The *sqlite* benchmark spends the majority of the time with computation, so that it scales very well. For *tar* and *untar*, most of the time, files are read and written, which scales almost perfectly, too. Occasionally in *tar* and *untar* and quite often in *find*, m3fs is called, which queue up with an increasing number of benchmark instances.

[7] Since *cat+tr* requires two PEs per benchmark instance, there are no results for 1 Application PE.

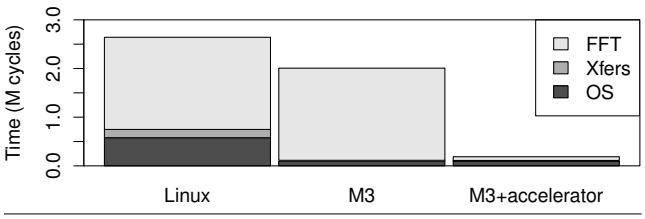

Figure 7. Performance benefits of an FFT accelerator core.

5.8 Accelerator

So far, the benchmarks used standard Xtensa cores. To show the feasibility of integrating an accelerator and to demonstrate the performance advantages, we added a core with instruction extensions for a fast fourier transformation (FFT) to our prototype platform. In mobile communication, FFTs are typically used within a filter chain, where e.g., data is received over the mobile network, transformed using the FFT, and passed forward to the next element in the chain. Thus, we built a similar scenario that lets an application (the *parent*) create a VPE, assigned to the FFT core. Afterwards, it executes the FFT application (the *child*) on that VPE and creates a pipe between itself and the child. The parent is generating random numbers, 32 KiB of data in total, and writes them into the pipe, while the child reads from that pipe, performs the FFT and writes the result into a file (it could be another pipe as well).

To show the performance benefits, we ran this benchmark first on Linux using a software FFT, second, on M^3 using only standard Xtensa cores with the same software FFT, and third, on M^3 using the FFT accelerator. It is worth noting that the code for the parent is identical for the software version and the accelerator version. It merely receives a different path to the executable to run on the VPE. The results in Figure 7 show at first, that the accelerator has a huge performance benefit over the software version (about a factor of 30). Second, `exec`, the pipe communication and the write to the file on Linux has much more overhead than their equivalent on M^3, especially compared to the FFT time when using the accelerator. This shows that the fast abstractions of M^3 lower the bar for using accelerators.

6. Conclusion

In this paper, we presented a new point in the design space for operating systems (OSes) by co-designing the OS with the hardware to support arbitrary cores as first-class citizens. That is, we support isolation on all cores to run untrusted code and provide access to OS services on all cores. This is achieved by abstracting from the heterogeneity of the cores by introducing a common hardware component, called data transfer unit (DTU), next to each core that supports remote memory access and message passing. By leveraging the fact that it is expected that future platforms will contain a large number of cores, we introduce kernel cores, where only the kernel is running and application cores, which are owned by one application at a given time. Cores and thus applications are thereby isolated at the network-on-chip-level via remotely controlling their DTUs, because a DTU is the only component that has access to core-external resources.

We demonstrated the feasibility of this design by implementing the DTU and the OS for a prototype platform, whose cores do not have support for an OS kernel and where we could integrate accelerators. We showed how basic OS abstractions like application loading, pipes, and filesystems can be built and that they work across heterogeneous cores. In our evaluation, we illustrated how this design can outperform Linux in some application-level benchmarks by more than a factor of five without exploiting accelerators.

7. Future Work

As we have laid the foundation for supporting arbitrary PEs as first-class citizens by starting with the challenging part of building an OS for PEs without kernel support in this work, we will extend this to more feature-rich PEs in future work. In particular:

- We plan to add caches to the PEs or replace the SPM with caches. The cache will use the DTU to load/store cache lines from/into DRAM. In this way, the DTU remains the only component with access to PE-external resources and it thus suffices to control the DTU.
- Furthermore, we want to support virtual memory to enable copy-on-write, demand paging, etc. This can be done by by managing the page tables remotely, similarly to managing the DTU endpoints remotely.
- Finally, we plan to support POSIX-compliant applications. So far, our prototype platform prevented us from directly running existing and complex applications due to the small SPM and the distributed memory architecture. As soon as caches and virtual memory are supported, we will add a POSIX emulation layer, similar to the already existing emulation layer for the filesystem API, that was used to replay system call traces.

As already mentioned in the paper, we currently do not support multiple instances of services or the kernel, because we wanted to explore first how far a single instance can scale. In the future, we will design synchronization protocols to allow multiple instances.

8. Acknowledgments

We would like to thank our shepherd, Gernot Heiser, the anonymous reviewers, Björn Brandenburg, Jeronimo Castrillon, Pramod Bhatotia, and Michael Roitzsch for their helpful suggestions. This work is in part funded through the German Research Council DFG through the Cluster of Excellence Center for Advancing Electronics Dresden (cfaed).

References

[1] BusyBox. `http://www.busybox.net/`. last checked: 01/19/2015.

[2] An introduction to the Intel® QuickPath interconnect. `http://www.intel.de/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf`. last checked: 01/19/2015.

[3] J. Ahn, M. Fiorentino, R. G. Beausoleil, N. Binkert, A. Davis, D. Fattal, N. P. Jouppi, M. McLaren, C. M. Santori, R. S. Schreiber, S. M. Spillane, D. Vantrease, and Q. Xu. Devices and architectures for photonic chip-scale integration. *Applied Physics A*, 95(4):989–997, 2009.

[4] 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, pages 111–119, Aug 1996.

[5] 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*, 13(3s):107:1–107:24, March 2014.

[6] F.J. Ballesteros, N. Evans, C. Forsyth, G. Guardiola, J. McKie, R. Minnich, and E. Soriano-Salvador. NIX: A case for a manycore system for cloud computing. *Bell Labs Technical Journal*, 17(2):41–54, 2012.

[7] 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.

[8] 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.

[9] Cadence. Xtensa customizable processor. `http://ip.cadence.com`. last checked: 01/19/2015.

[10] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, pages 283–292, New York, NY, USA, 2006. ACM.

[11] 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.

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

[13] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[14] 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*, pages 177–190, New York, NY, USA, 2006. ACM.

[15] Norman Feske. A case study on the cost and benefit of dynamic RPC marshalling for low-level system components. *ACM SIGOPS Operating Systems Review*, 41(4):40–48, July 2007.

[16] 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.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, New York, NY, USA, 2003. ACM.

[18] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July 2011.

[19] Norman Hardy. KeyKOS architecture. *ACM SIGOPS Operating Systems Review*, 19(4):8–25, October 1985.

[20] 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*, pages 38–50, New York, NY, USA, 1994. ACM.

[21] K.U. Jarvinen and J.O. Skytta. High-speed elliptic curve cryptography accelerator for koblitz curves. In *16th International Symposium on Field-Programmable Custom Computing Machines (FCCM '08)*, pages 109–118, April 2008.

[22] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Amir Wated, Emmett Witchel, and Mark Silberstein. GPUnet: Networking abstractions for GPU programs. In *Proceedings of the International Conference on Operating Systems Design and Implementation*, pages 6–8, 2014.

[23] 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*, pages 207–220, New York, NY, USA, 2009. ACM.

[24] George Kurian, Jason E. Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel C. Kimerling, and Anant Agarwal. ATAC: A 1000-core cache-coherent processor with on-chip optical network. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*, pages 477–488, New York, NY, USA, 2010. ACM.

[25] 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*, pages 302–313, Apr 1994.

[26] 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.

[27] J. 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.

[28] 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.

[29] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A mobile operating system for heterogeneous coherence domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, pages 285–300, New York, NY, USA, 2014. ACM.

[30] 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*, pages 369–381. ACM, 2015.

[31] 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 *Fourth International Symposium on High-Performance Computer Architecture*, pages 231–242, Feb 1998.

[32] 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.

[33] 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.

[34] 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.

[35] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, et al. Plan 9 from Bell Labs. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9. London, UK, 1990.

[36] J. Porquet, A. Greiner, and C. Schwarz. NoC-MPU: A secure architecture for flexible co-hosting on shared memory MPSoCs. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–4, March 2011.

[37] 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, March 2015.

[38] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9:1–9:32, August 2013.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] M.B. Taylor. A landscape of the new dark silicon design regime. *IEEE Micro*, 33(5):8–19, Sept 2013.

[44] 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, April 2009.

[45] 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.

[46] 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.

[47] Wei Yu and Yun He. A high performance CABAC decoding architecture. *IEEE Transactions on Consumer Electronics*, 51(4):1352–1359, Nov 2005.