M³X: Autonomous Accelerators via Context-Enabled Fast-Path Communication

Nils Asmussen[†‡] Michael Roitzsch[†‡] Hermann Härtig[†‡]
[†] *Technische Universität Dresden, Germany* [‡] *Barkhausen Institut, Dresden, Germany*

Abstract

Performance and efficiency requirements are driving a trend towards specialized accelerators in both datacenters and embedded devices. In order to cut down communication overheads, system components are pinned to cores and fast-path communication between them is established. These fast paths reduce latency by avoiding indirections through the operating system. However, we see three roadblocks that can impede further gains: First, accelerators today need to be assisted by a general-purpose core, because they cannot autonomously access operating system services like file systems or network stacks. Second, fast-path communication is at odds with preemptive context switching, which is still necessary today to improve efficiency when applications underutilize devices. Third, these concepts should be kept orthogonal, such that direct and unassisted communication is possible between any combination of accelerators and general-purpose cores. At the same time, all of them should support switching between multiple application contexts, which is most difficult with accelerators that lack the hardware features to run an operating system.

We present M³X, a system architecture that removes these roadblocks. M³X retains the low overhead of fast-path communication while enabling context switching for general-purpose cores and specialized accelerators. M³X runs accelerators autonomously and achieves a speedup of 4.7 for PCIe-attached image-processing accelerators compared to traditional assisted operation. At the same time, utilization of the host CPU is reduced by a factor of 30.

1 Introduction

The end of Dennard scaling [18] prevents further frequency gains and the prospect of dark silicon [21] hampers general-purpose parallelism. Hardware and system designers thus turn to new architectures to increase performance or reduce power consumption. These new ideas often revolve around specialization through custom accelerators [9, 26, 35, 37, 48, 67, 68] and streamlined communication that bypasses the operating system to avoid overheads [10, 39, 46].

TPUs [27] are a key example of the first approach. By creating a fixed-function accelerator for neural network training and inference, Google managed to increase performance per socket 30-fold and performance per watt 80-fold over a contemporary CPU. The second approach of preferring data fast paths to avoid indirections through the operating system can be observed today with technologies like single root I/O virtualization (SR-IOV) or Infiniband. System designs like M³ [10] and DLibOS [39] have shown that fast-path communication achieves latency reductions of 5× for a file system workload on M³, and 20× for memcached on DLibOS. Furthermore, our previous work M³ demonstrates that this idea can be generalized to provide fast-path communication between all compute units in the system.

Encouraged by these benefits, we expect ongoing development and increased deployment of these solutions. Use-case-driven accelerators will find their place in datacenters, also because of their deterministic execution model which helps to meet tail-latency requirements. We therefore assume that more applications will entail complex interactions between a mix of accelerators and general-purpose cores. Additionally, in multi-tenant cloud environments context switching is essential, because a single user will typically underutilize accelerators. We 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.

1.1 Problem Statement

We extract three fundamental architectural challenges from our assumptions: First, the system architecture should enable accelerators to run autonomously. Currently, accelerators are often treated as peripheral devices whose execution needs to be assisted by a general-purpose CPU [57]. The TPUs described in Google's paper burden their controlling CPU with 11 – 76% load just to operate the TPU [27]. Our comparison to the traditional usage of accelerators in § 7.6 confirms this experience by showing that even a 3 GHz out-of-order

x86-64 core is 86% loaded to assist three image-processing accelerators attached via PCIe. If accelerators had direct access to data sources and sinks, this overhead would not be necessary. However, such connectedness requires first-party interaction of accelerators with OS services like storage and network. Specialized solutions exist, like GPUfs [58], GPUnet [29], and PTask [52] for GPUs or BORPH [59] and FPGAFS [31] for FPGAs. But 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 a general-purpose core.

Second, fast-path communication without OS interaction is important for low-latency data and control transfer, but conflicts with context switching. This problem applies to communication channels involving general-purpose cores as well as accelerators. If communication partners are pinned and exclusively use dedicated resources, direct communication is easy. However, with context switching, communication needs to consider whether the intended recipient is currently running and how to deliver a message otherwise. A system design needs to answer, how the equivalent of a blocking system call should work on an accelerator that lacks the hardware features to run an operating system. Current solutions like M³ and DLibOS avoid this question and forgo context switching altogether.

Finally, all compute resources in the system — accelerators as well as general-purpose cores — should be accessible via the same communication primitives. The resulting system should enable developers to offload any job to the most suitable compute resource. Such unification was explored in previous work, but only for heterogeneous general-purpose cores [12, 19, 43].

1.2 Scope

We believe that accelerators should not be forced to adapt to operating system requirements, but focus on their main task: a fast and energy-efficient solution to a specific problem. In this paper, we take the extreme position and rethink the system architecture to enable a first-class integration of accelerators without imposing changes on them.

In contrast to conventional architectures, we do *not* build upon coherent shared memory for two reasons: First, providing global cache coherency is challenging for systems that consists of a wide variety of compute units such as general-purpose cores, DSPs, and fixed-function accelerators. Second, the costs of cache coherency in terms of chip area, power, complexity, and performance are expected to increase with an increasing number of compute units [28, 41]. For these reasons, it is still unclear whether and how future systems will support cache coherency. Therefore, we keep cache coherency optional.

Additionally, our long-term goal is to support arbitrary accelerators as first-class citizens. In this paper, we begin to address this challenge by demonstrating our approach for accelerators that are arguably the most difficult to support as first-class citizens: fixed-function accelerators [22, 26, 35, 37, 48, 65] that do not execute software and therefore provide none of the features that are required to run an operating-system kernel. We believe our efforts towards a unified interface for all compute units will generalize to more feature-rich accelerators in the future.

1.3 Contribution

We propose M³ˣ, a solution for the identified issues using a hardware-software co-design approach.

- We explore the design space for fast-path communication and context switching. We explain the fundamental problems, when combining both techniques (§ 2) and discuss solutions in terms of interaction modes (autonomous vs. assisted) and mechanisms (hardware vs. software).
- We converge on a design for M³ˣ that allows fast-path communication without involving the OS kernel and enables accelerators to access data sources and sinks without assistance by a general-purpose core. At the same time, M³ˣ supports context switching on both general-purpose cores and accelerators (§ 4).
- We implement these mechanisms within the M³ OS and hardware architecture (§ 5). M³ already supports fast-path communication within a tile-based architecture and unifies communication among heterogeneous instruction sets [10]. Thus, it constitutes a suitable starting point, which we extend with support for context switching and autonomous accelerators.
- In the evaluation (§ 7), we demonstrate how M³ˣ retains the low overhead of fast-path communication while enabling context switching. We show the performance and utilization benefits of autonomous accelerators using an accelerator benchmark suite and an application scenario that might occur in datacenters.

We rely on gem5 [16] as our simulation platform. Its high accuracy and modularity enable us to experiment with new hardware components. The implementation of M³ˣ[1] and our extensions to gem5[2] are available as open source.

2 Background and Motivation

Traditional communication via UNIX pipes, sockets, or microkernel IPC 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. Communication that bypasses the kernel offers significant gains in terms of latency and throughput, as has been shown by M³ [10] and DLibOS [39]. We call such communication *fast-path communication* in this paper. Using fast-path

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

communication with dedicated cores for the applications is easy, because none of the aforementioned actions are required, which is why M³ and DLibOS chose to omit context switching support altogether. We also believe the still increasing core counts and the dark silicon effect [21, 25] will reduce the context switching frequency and lead to dedicated cores for applications by default. However, in the foreseeable future, provisioning enough hardware resources to handle all load spikes is not feasible. These load spikes therefore require oversubscription of cores and accelerators. Thus, fast-path communication needs to be combined with context switching to use the system as efficiently as possible.

Combining fast-path communication with context switching is a hard problem, though. If the kernel is not involved in the communication, how can we determine whether the recipient is running and how can we deliver the message if it is not running? Even without relying on coherency (see § 1.2), we could buffer all messages in DRAM to cope with non-running recipients. However, this would effectively route all communication over DRAM, which increases latency and DRAM load and is therefore detrimental to the goals of fast-path communication. On the other hand, communicating directly between compute units, without involving the OS, has the consequence that a message cannot be delivered if the designated recipient is not running. The naive solution of waking up the recipient and retrying the fast-path 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.

Accelerators typically lack the architectural features to run an OS kernel locally. To avoid the indirection of all communication through a remote kernel, accelerators require fast-path communication to interact with other accelerators [57]. However, as described before, fast-path communication is possible only if the recipient is running. Thus, the combination of fast-path communication and context switching is necessary to run accelerators autonomously, without indirection through the kernel.

3 Related Work

There are industry solutions for accelerator integration such as the coherent accelerator processor interface (CAPI) [6, 62] and the heterogeneous system architecture (HSA) [4, 51]. Both allow the integration of accelerators into a cache coherent virtual memory system, but in contrast to M³x, these hardware solutions do not consider direct access to operating system services by accelerators. Such access is investigated by other works for specific OS services and specific accelerators like GPUs [15, 29, 34, 52, 58, 64, 69] or FPGAs [31, 47, 59]. In contrast, M³x does not target a specific kind of accelerator, but provides a general construction principle for fast-path communication of any accelerator with any OS service or application.

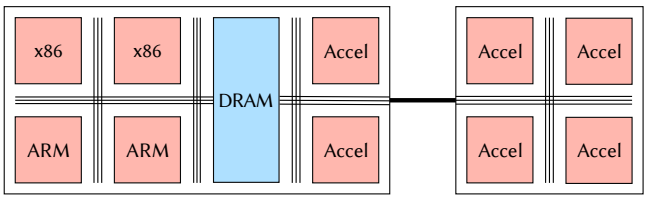

Figure 1: Overview of the system architecture.

K2 [36] and Popcorn Linux [12] demonstrate how the Linux kernel can be extended to support multiple coherence domains and potentially heterogeneous cores. However, heterogeneity in these cases is represented by general-purpose cores with different instruction sets and does not include fixed-function accelerators. Barrelfish [14] introduced the multikernel concept, where message passing is used to communicate between the operating system instances on each core. This concept is close to the design of M³x, but it assumes that all cores offer the architectural features to run an OS kernel. M³x sets out to remove this requirement to integrate fixed-function accelerators as well.

Arrakis [46] and OmniX [57] integrate peripherals and accelerators using SR-IOV. Instead of requiring the architectural features to run a kernel, these works assume the hardware to manage multiple contexts. M³x explores a more lightweight design, yet with enough hardware support to enable context switching and fast-path communication between components. Like NIX [11] or FlexSC [60], M³x adopts the idea of redirecting system calls to kernel cores to reduce the duties of non-kernel cores such as accelerators.

Horizontal system layouts [66] with different services on separate cores were explored in DLibOS [39] and M³ [10]. Both works have shown latency reductions due to the use of fast-path communication between cores, but they do not support context switching due to the problems explained in the previous section. Context switching of accelerators has been explored specifically for GPUs [13, 45], but without considering fast-path communication. M³x combines context-switching with the benefits of fast-path communication.

4 Design

We start this section with an introduction to the basic system architecture and discuss the design space for accelerator integration. Afterwards, we describe how fast-path communication is combined with context switching. Finally, we explain how this combination is used to run accelerators autonomously.

4.1 System Architecture

In this work, we assume a tiled system architecture as depicted in Fig. 1. The system uses an interconnect to communicate between tiles, similar to M³ [10] and DLibOS [39] and also similar to upcoming system architectures based on GenZ [3] or CCIX [2]. The tiles can contain heterogeneous *compute units* (CUs), ranging from general-purpose cores to DSPs

to fixed-function accelerators. These CUs can be part of the host system (left) and can be attached as an expansion card (right). The host system has a shared DRAM. We use the term *activity* to unify the active entities on these CUs. On general-purpose cores, an activity is typically a thread, whereas on accelerators it is the logic operating on a context.

4.2 Accelerator Integration

Adding accelerators into a system design poses the question of how to balance responsibilities between hardware and software. First, there are different ways to support arbitrary data sources and sinks for accelerators. Access to OS services like file systems or network stacks can be performed by software, which is the typical approach today. This approach requires a general-purpose core to assist the accelerator by continuously moving data back and forth. However, if the protocol to access OS services and data is sufficiently simple, it can be implemented in hardware. Such a hardware-friendly protocol allows accelerators to autonomously access arbitrary sources and sinks and removes software from the critical path. We present our protocol in § 4.8.

Second, 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, as described in § 4.9.

Finally, to improve the utilization of accelerators, support for multiple contexts is necessary. One solution is to require the accelerator to provide a sufficient number of contexts and multiplex the hardware accordingly (e.g., based on SR-IOV). Alternatively, a combination of hardware and software can be used, which requires only a single context in hardware. To keep the hardware simple, we chose the latter approach: We perform the potentially complex scheduling decisions in software and add a simple state machine to the hardware, which saves and restores contexts.

4.3 Activity-aware Communication

To increase the flexibility and applicability of our system design, we chose not to rely on shared memory. Hence, messages cannot be delivered if the receiving activity is suspended (e.g., by preemption). There are two basic solutions to this problem:

1. *Eagerly* invalidate all incoming communication channels to an activity before suspending it or
2. keep the communication channels alive, but *lazily* detect communication attempts with suspended activities.

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. We chose the lazy approach, because our system supports many incoming communication channels (at most $(n-1)*m$ per compute unit with n CUs and m communication endpoints per CU). Furthermore, many communication channels are typically not used while an activity is suspended. To this end, we inform the hardware of the running activity and of the intended recipient activity when communicating (see § 5.5 for details). The hardware compares both and reports an error upon communication attempts with suspended activities.

4.4 Message Forwarding

Independent of eager invalidation or lazy detection, the hardware reports an error to the sender if the intended recipient is not running. Unfortunately, the naive solution of scheduling the recipient and retrying the fast-path 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 suspends the recipient before the communication has been finished, the sender does not make progress. The problem is that context switching and communication are decoupled, because the kernel performs the context switching, but activities bypass the kernel when performing fast-path communication. For example, if multiple senders try to communicate with multiple recipients scheduled on the same CU, the kernel could decide to schedule the next recipient before the communication with the current recipient has been finished.

We resolve this race condition by falling back to the traditional kernel-based communication model, if a communication failed due to a suspended activity. The kernel performs both the context switching and the communication: If activity A receives an error after trying to send a message to activity B, it asks the kernel to *forward* this message to B. When receiving the forward request, the kernel will first schedule B and afterwards send the message to B. To guarantee progress, the kernel does not suspend B until the message has been successfully delivered.

4.5 Computing vs. Idling

Another consequence of fast-path communication is that the kernel does not know whether an activity is currently computing or idling, because it waits for a message. We solve this problem by sending an *idle notification* to the kernel, similarly to scheduler activations [8]. Alternatively, the kernel could poll all CUs periodically to check whether the current activity is performing useful work, but we opted against this solution in favor of a less loaded and more scalable kernel.

We employ two optimizations. First, to prevent overeager context switches, we delay the sending of idle notifications

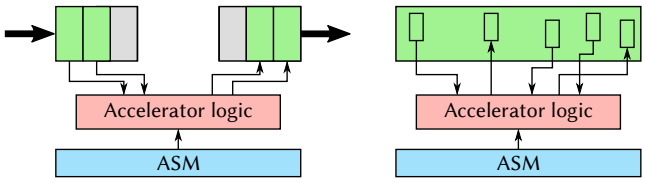

Figure 2: Stream-processing accelerators (left) and request-processing accelerators (right).

by a kernel-defined value called *idle delay*. The idle delay is stored in the address space of the current activity and updated by the kernel. Second, an activity does not need to send idle notifications at all if there is no ready activity that can run on its CU. In this case the idle delay is set to zero.

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

4.6 Gang Scheduling

The concepts described so far allow to suspend activities, resume activities based on communication attempts, and to use the system's resources efficiently by switching to a different activity in case the current activity idles. However, if a set of heavily communicating activities contend with other activities for the same CUs, a systematic scheduling approach is required to maintain good performance. For example, consider a chain of accelerator activities that perform stream processing and therefore exchange messages and data at a high rate. If multiple such chains are contending for the same accelerators, the kernel needs to context switch these activities. However, uncoordinated context switching among the activities of all chains leads to many failed communication attempts when activities of different chains run simultaneously.

We solve this problem by introducing a simple form of gang scheduling [44]. Applications define the gang of a new activity at its creation time and the kernel pins all activities in a gang on different CUs and schedules them at the same time. We use this to run all activities of a single chain simultaneously. As the evaluation shows, multiple sets of heavily-communicating activities can therefore efficiently share the same CUs.

4.7 Accelerator Types

In this work, we consider two types of accelerators, depicted in Fig. 2: *stream-processing accelerators* that process a stream of data in blocks and compute on each block exactly once (e.g., AES encryption) and *request-processing accelerators* that receive the entire data for the computation with a single request and can access all data during the entire computation (e.g., graph processing or garbage collection). The 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. The request-processing accelerators use cache-based memory access to the request data to support large requests while maintaining fine-grained data access. For both accelerators, we add an *accelerator support module* (ASM), implemented as a finite-state machine, to the accelerator logic. The accelerator logic performs the computation, whereas the ASM interacts with other CUs and invokes the accelerator logic. We implemented the ASM as a piece of hardware in the current gem5-based prototype to demonstrate its simplicity.

Running these two types of accelerators autonomously requires access to OS services such as file systems, network stacks, and pipes to load and store data. Furthermore, accelerators need to be interruptible without requiring assistance by a general-purpose core. We describe our solution for both problems in the following sections.

4.8 Access to OS Services

Enabling accelerators to access files or network sockets requires a simple and unified protocol to obtain access to these resources. To this end, we designed a simple protocol for all file-like objects, in the same spirit as UNIX's everything-is-a-file principle. In contrast to UNIX, we implement OS services as microkernel-style servers and support both applications and accelerators as clients.

The *file protocol* uses a fast-path messaging channel between client and server. The server is expected to make the data available in memory and to provide the client with access to the data via a fast-path data channel. This channel enables accelerators to access large amounts of data autonomously, preventing frequent client-server interactions.

The protocol comprises 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. For example, a file-system server will provide the client with access to a fragmented file piece by piece, as described in more detail in § 5.7. After providing the client access to the data, the server returns the offset and size of the piece. Upon receiving this reply, the client can access the data via the fast-path data channel without involving the server again. After finishing the current piece, the client issues another `next_in` or `next_out` request to the server. A piece of length zero from the server denotes end-of-file.

As the client accesses the data on its own, the server does not know how many bytes the client has actually read or written. Therefore, 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. 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.

Finally, some servers support the seek request to change the file position. As described in more detail in § 5.8, the file protocol is implemented within the ASM of the stream-processing accelerators to load input data from arbitrary file-like sources and store the result to arbitrary file-like sinks. Note that request-processing accelerators can access OS services via the file protocol as well, but this has not been implemented. To test the generality of the protocol, we added a POSIX-like API on top and implemented a file system server, pipe server, and virtual terminal.

4.9 Interruptible Accelerators

As discussed in § 4.2, 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. We achieve both by using the ASM as an indirection. Software performs the coarse-grained invocation of the hardware-implemented ASM, which in turn invokes the accelerator logic in a fine-grained fashion and is interruptible between these invocations. We implemented this scheme for request-processing accelerators, because the considered stream-processing accelerators already perform their computation block-wise with relatively small block sizes.

5 Implementation

Our prototype implementation is based on the hardware and software part of M³ [10]. The hardware platform of M³ exists by now as custom silicon in the Tomahawk 4 chip [24]. To extend the hardware part, we build on top of the already existing gem5 prototype. Both the gem5 prototype platform and the OS are open source and have been extended in this work to support context switching and autonomous accelerators.

5.1 Background on M³

The key idea of M³ is to introduce a new hardware component next to each CU, which serves as an abstraction for the heterogeneity of the CUs and supports fast-path communication between CUs. This hardware component is called data transfer unit (DTU) and is accessible over memory-mapped I/O (MMIO). Each CU is integrated with its DTU as a tile into the network-on-chip. The DTU provides a set of communication endpoints that can be configured as send, receive, or memory endpoints. Send and receive endpoints allow to establish a fast-path messaging channel, whereas memory endpoints are used for fast-path data channels. Data channels provide DMA-like access to a contiguous and byte-granular memory region.

The M³ kernel runs on a dedicated *kernel tile*, because not all tiles can be expected to run an OS kernel. The M³ kernel is different from traditional kernels, because it does not run user applications on the kernel tile. Instead, the kernel runs applications on other tiles, called *user tiles*, and waits for system calls in the form of messages, sent by applications via the DTU. Since only the kernel tile can configure DTU endpoints, applications are isolated from each other by default. The main responsibility of the kernel is to establish communication channels between applications by configuring DTU endpoints remotely. After a communication channel has been established, applications communicate directly with each other, bypassing the kernel.

On M³, the same activity abstraction[3] is used for all types of tiles, because the kernel is only concerned with their DTU state. The M³ kernel uses capabilities to manage the permissions in the system. Each activity has its own address space and capability space and system calls allow to exchange capabilities between activities. Since M³ does not support context switching, an activity is assigned to a free tile on creation and occupies this tile until its termination.

Outside of the kernel, M³ provides servers to host the actual functionality of the OS. M³ offers an in-memory filesystem, called *m3fs*, that organizes the data similarly to classical UNIX filesystems. The important difference is that m3fs grants applications direct access to file data via the DTU. Additionally, M³ offers a pipe server to connect activities via a unidirectional, first-in-first-out communication channel.

5.2 Virtual Memory Support

So far, M³ supports only simple general-purpose cores without virtual memory. Instead, cores have untranslated access to their dedicated scratchpad memory. To support more complex applications and be able to switch between them without first saving their entire memory state to DRAM, we added virtual memory support to M³. However, to prepare for future systems, M³X does not take advantage of cache coherency, but keeps it optional.

As virtual memory is also desirable for the cache-based memory access of request-processing accelerators, we added virtual memory support in two variants. For general-purpose cores, we use their memory management unit (MMU) and run a small helper on the core that receives page faults. For accelerators, we add an MMU to the DTU, consisting of a page table walker and translation lookaside buffer (TLB). In both cases, page faults are resolved by a *pager* in collaboration with the M³X kernel, which updates the page table entries, similar to other microkernel-based systems [33,61]. The pager is a server in M³X that supports copy-on-write and demand loading. On general-purpose cores, the helper sends a message to the pager to resolve page faults. On accelerators, the DTU sends the message to the pager, which is transparent to the accelerator.

5.3 Context Switching Overview

Context switches are performed remotely on the user tiles, initiated by the M³X kernel. This approach is required for accelerators that do not have the architectural features to run an OS kernel, but is optional for general-purpose cores with these features. In other words, the implementation could be extended to perform context switches on general-purpose cores locally.

[3] M³ calls tiles *processing elements* (PEs) and activities *virtual PEs* (VPEs).

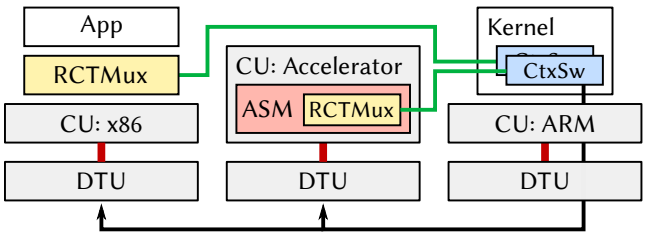

Figure 3: The involved components for context switches and their interfaces, shown on an exemplary assembly of CUs.

A context switch involves four components, depicted in Fig. 3: the CU, the DTU, the *context switcher* (CtxSw) in the M³x kernel, and a small component on each user tile, called *remotely controlled time multiplexer* (RCTMux). RCTMux saves and restores the CU-state (e.g., CPU registers or the accelerator's local memory) during a context switch. The security-critical DTU-state (e.g., communication endpoints) is saved and restored by the kernel. RCTMux is CU-specific and either a piece of software on programmable CUs or a piece of hardware as part of the ASM for accelerators. The M³x kernel maintains one context switcher for each user tile and performs scheduling and placement decisions.

These four components have two important interfaces. The first interface between the context switcher and RCTMux (green in Fig. 3) is used by the kernel to request saves and restores from RCTMux and by RCTMux to acknowledge their completion. Second, the DTU-CU interface (red) is used by the kernel to signal the CU about an imminent context switch. Depending on the type of CU, the signal injects an interrupt request into a core or notifies the ASM of an accelerator.

5.4 Kernel Extensions

We incorporated the context switcher module into the M³ kernel to perform context switches on user tiles. First, the context switcher asks RCTMux to save the CU state. The context switcher then saves the DTU state of the current activity, restores the DTU state of the new activity, and asks RCTMux to restore said activity's CU state. Each of these steps is executed individually to be able to handle other requests (e.g., system calls) in the meantime.

Furthermore, we introduced a system call to forward messages upon communication attempts with suspended activities. The kernel buffers the message to forward, schedules the recipient, and delivers the message to the recipient as soon as it is running. Finally, we added a system call for idle notifications, upon which the kernel switches to the next ready activity or work-steals an activity from another tile in case no activity was ready. For application activities, the kernel sets the idle delay to 20,000 cycles[4]. For server activities, the kernel uses an idle delay of one cycle, because servers are typically only activated on demand. Hence, switching to an application is more beneficial for the system's performance.

To facilitate fast-path communication, the kernel migrates activities in two situations. First, if two activities are scheduled on the same CU and attempt to communicate, the kernel tries to migrate the currently suspended activity to another CU. If migration is not possible (e.g., no other compatible CU is available), the kernel instead performs a context switch from the activity that attempted the communication to the suspended activity. Second, if an activity is idling (see § 4.5), the kernel tries to work-steal a ready activity from a compatible CU.

5.5 DTU Extensions

To detect communication attempts with suspended activities, we equipped the DTU with the ID of the current activity and added the destination activity ID to the message header. If the destination activity ID at the recipient's DTU does not match the current activity ID, the DTU reports an error to the sender. In this case, the sender asks the kernel to forward the message to the recipient via the forward system call.

If the kernel decides to perform a context switch on a user tile, the DTU might currently be busy with a communication. As explained in § 5.1, the DTU supports messaging channels and data channels. Messages need to be delivered exactly once, whereas data accesses can be repeated. To keep the DTU simple, we decided against a complicated protocol to abort potentially ongoing communication. Instead, the DTU has an abort command, which consists of two parts. First, further communication attempts are rejected with an error until re-enabled by the kernel. Second, the DTU waits until all message-based communication is completed, whereas data accesses are aborted with an error and need to be repeated later.

5.6 RCTMux

We implemented RCTMux both for accelerators and for general-purpose cores. As mentioned before, on accelerators, RCTMux receives a signal from the kernel if a context switch is desired. The ASM checks for the signal only at convenient points in time, because the accelerator logic is not assumed to be interruptible. Upon the signal, it saves the ASM's state and the accelerator's local memory via DTU to a previously allocated space in DRAM, uses the DTU's abort command, and notifies the kernel that the state has been saved. Analogously, the state is restored upon a restore request from the kernel.

We also implemented RCTMux for x86-64 as a small piece of software running in ring 0. In this case, RCTMux is activated by an interrupt injection, saves the CPU registers, uses the DTU's abort command, and notifies the kernel. Upon a restore request from the kernel, it restores the CPU registers and resumes a previously aborted data access, if necessary.

5.7 File Protocol Servers

M³ already features an in-memory file system, called m3fs, and a pipe server. However, since M³ was only evaluated

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

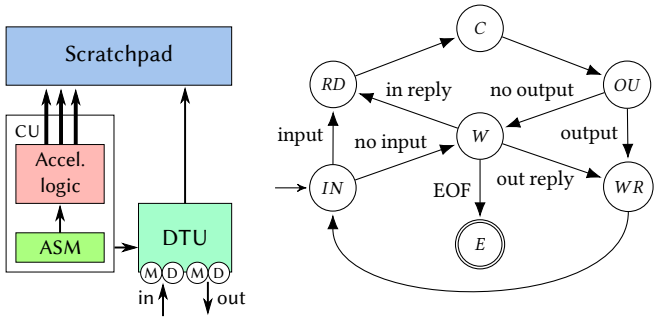

Figure 4: Stream-processing accelerator.

on general-purpose cores (in some cases with instruction extensions), the protocols to access these OS services are not suited for accelerators. First, M³ uses a different protocol for m3fs than for the pipe server, requiring accelerators to implement multiple protocols. Second, m3fs's protocol is based on the exchange of capabilities to obtain access to the data and requires clients to manage the file position. In summary, the existing protocols are too complex to be implemented in hardware. For that reason, we replaced them with the file protocol, as introduced in § 4.8.

On M³ˣ, the file protocol is based on a messaging channel between client and server and a data channel to access the file data. Since m3fs manages the file data in extents, similar to other modern file systems [40, 50], the `next_in` request provides the client with access to the next extent of the file by asking the kernel to establish a corresponding data channel. The file position is managed and advanced by m3fs and can also be changed by the `seek` request. For appends, the `next_out` request allocates new space and provides the client with access to this space. Upon commit, m3fs truncates this space, if necessary, and makes it visible to other clients.

The pipe server uses a single and contiguous shared memory area in DRAM per pipe to exchange data between clients. For that reason, the pipe server asks the kernel to configure the client's data channel only once for the complete area and tells the clients where to read or write next. If no data can be read or written, the pipe server delays its response to the `next_in` or `next_out` request correspondingly.

5.8 File Protocol Clients

On the client side, we implemented the file protocol in software (for general-purpose cores) and in hardware (for accelerators). The software version is part of M³ˣ's standard library and allows applications to use a POSIX-like file API. The library maps this API to the corresponding `next_in`, `next_out`, `commit`, and `seek` requests.

To enable access to file-like resources for stream-processing accelerators, we implemented the file protocol as part of the accelerator support module (ASM). The stream-processing accelerator has an input stream and an output stream, each using one messaging channel (M) to the server and one data channel (D) to access to data, as shown in Fig. 4. Like many other accelerators [17, 38, 55, 56, 63], the computation is performed on scratchpad memory (SPM), because it allows many parallel memory accesses (indicated by the thick arrows) and has predictable access latency.

The ASM loads data via the DTU from the input stream into the accelerator's SPM, activates the accelerator logic, and writes the result to the output stream. The ASM starts in state *IN*, which checks whether the input data channel has data left to read. If so, it directly transitions to state *RD* to read the next block of data into the SPM. Otherwise, it 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 message arrives and transitions to *RD* as soon as the reply to the input request has been received. After the next data block has been read into the SPM, the accelerator logic is activated and the ASM transitions to state *C* for the computation. As soon as the computation has been completed, the ASM transitions to state *OU*. Analogously to the input phase, *OU* first checks whether the output data channel has space left for the result of the computation. If so, it directly transitions to *WR* and writes the data. Otherwise it first requests new space from the output server (`next_out`). Afterwards, the ASM transitions back to state *IN*, which repeats the procedure until the reply to an input request indicates end-of-file. In this case, the ASM commits the written data by sending `commit` to the output server, if required, and transitions to state *E*.

6 Discussion

We believe that our architecture provides a good foundation for very heterogeneous systems, but we are aware that CUs will be diverse and have different requirements. This section discusses a few examples of how our current prototype can be extended to support other use cases.

Our context switching mechanism handles the simple save and restore actions on user tiles and the decision making in the M³ˣ kernel. While we show in the evaluation that context switches on fixed-function accelerators have acceptable overhead, the mechanism is probably not a good fit for accelerators that have a large state such as GPUs. General-purpose cores provide native mechanisms to save/restore their state, which are used by the software version of RCTMux. Therefore we believe that large-state accelerators need tailored context-switching mechanisms as already partially supported by modern GPUs.

As described in § 5.4, the M³ˣ kernel currently migrates an activity to a different tile if two activities on the same tile try to communicate. This policy assumes that the communication attempt starts a series of interactions between these activities, which mostly holds true for our current workloads. Clearly this is not the best solution in all cases. For example, if the activities communicate just once, a local communication channel with context switching can be preferable, if supported by the compute unit.

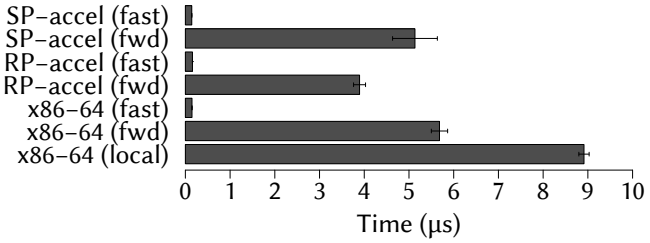

Figure 5: Fast-path vs. forwarded communication.

And finally, our current prototype does not queue messages at the recipient's compute unit if the recipient is suspended, but forwards the message to the recipient via the M³X kernel. We chose this solution to keep the hardware extensions small and, most importantly, minimize the burden on accelerators. If accelerators support message queues or such queues are added externally, the number of kernel involvements can be reduced. Thus, arguably our solution uses a queue size of zero, which can be extended to queue a few messages locally and only resort to the M³X kernel if the queue is full.

7 Evaluation

Our evaluation answers the following questions:

- How does fast-path and forwarded communication perform?
- Do the changes to the file protocol reduce its performance?
- What is the performance impact if activities share tiles?
- What are the benefits of autonomous accelerators?

7.1 Evaluation Platform

We used the gem5-based prototype platform for our evaluation. General-purpose tiles contain a single out-of-order x86-64 core with 32 KiB L1 instruction cache, 32 KiB L1 data cache, and 256 KiB L2 cache. The request-processing accelerators use 32 KiB L1 cache, whereas the stream-processing accelerators use 2 KiB scratchpad memory. General-purpose cores are simulated with a 3 GHz clock frequency, whereas accelerators are clocked with 1 GHz. All DTUs are configured to have 16 endpoints available. We use the `DDR3_1600_8x8` model of gem5 as the physical memory, clocked at 1 GHz. To keep the simulation times manageable, we connect the tiles via a crossbar instead of a full network-on-chip, which was sufficient, because our evaluation does not require large numbers of tiles. Due to the still long simulation times we used representative, but short-running benchmarks.

7.2 Fast-Path vs. Forwarding

M³X combines fast-path communication with context switching. In a first step, we use micro-benchmarks to determine the costs of forwarded communication, requiring a context switch, compared to fast-path communication. We measure the round-trip-time between activities on different CUs. Fig. 5 shows the average time over 16 runs with warm caches. The uppermost two rows show the time for stream-processing accelerators (SP), first if the recipient is running, resulting in fast-path communication, and second if the recipient is suspended, resulting in forwarded communication. The next two rows show the results for request-processing accelerators (RP), followed by two rows for an x86-64 core. The final row shows the time for a core-local round trip, using forwarded communication for both the request and the response.

As the results in Fig. 5 show, fast-path communication is more than one order of magnitude faster than forwarded communication on our system. All forwarded communication requires a forward system call, upon which the kernel performs a context switch to the receiving activity and forwards the message to the recipient. Most of the time is spent with the actual context switch, because it requires multiple steps and is carried out partially by RCTMux and partially by the kernel. Since stream-processing accelerators use a local scratchpad memory, the content needs to be saved and restored, leading to additional overhead. On x86-64, the overhead is larger, because the RCTMux is implemented in software. Finally, the core-local round trip requires two context switches. However, it is not twice as expensive as a single context switch, because the kernel optimizes this case by omitting the idle notification.

7.3 Application-level Benchmarks

As described in § 5.7, we simplified and unified the file protocol to be hardware-friendly. To evaluate whether these changes impact performance, we used the system call tracing infrastructure from M³. It allows to run an application on Linux, trace the system calls including timing information and replay the trace on M³. We used the following applications:

1. *tar*: creates a tar archive from files with sizes between 128 and 8192 KiB and 16 MiB in total,
2. *untar*: unpacks the same archive,
3. *shasum*: computes the SHA256 hash of a 512 KiB file,
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/selects 32 entries, and
7. *LevelDB*: creates a table and inserts/selects 512 entries.

The applications *tar*, *untar*, *shasum*, *sort*, and *find* have been taken from BusyBox 1.26.2 [1]. *SQLite* is an embedded and highly reliable database engine [7]. *LevelDB* is a light-weight and high-performance key-value store, created by Google [5]. We chose these applications to stress the system in different ways: *tar* and *untar* are data intensive, *shasum* and *sort* are compute intensive, *find* performs many small file-system requests, and *SQLite* and *LevelDB* are mixtures of these.

We used these applications to compare the performance between Linux 4.10, M³ with the original file protocol, and M³X using the unified and hardware-friendly file protocol. In this section, M³ and M³X use three dedicated x86-64 tiles (without accelerator tiles) for the application, m3fs, and the pager, whereas Linux uses a single x86-64 core. However, M³ and M³X

Figure 6: Performance comparison between Linux (Lx), M³, and M³X.

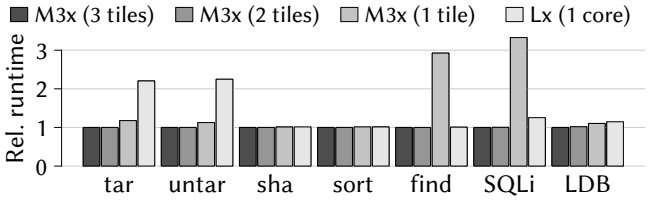

Figure 7: Application performance with a varying number of tiles, relative to the runtime on M³X with 3 tiles.

do not take advantage of multiple tiles, because all cross-tile interactions are synchronous and therefore, at no point in time is useful work done in parallel. On M³ and M³X, we use extents of at most 512 KiB, requiring multiple requests to m3fs to read and write files. On Linux, we use `tmpfs` as the in-memory file system. All file systems use a block size of 4 KiB. Fig. 6 shows the average runtime of 7 runs after one warmup run, broken down into the application time, time for data transfers, and OS overhead. We account Linux's time for the system calls, which are unsupported[5], as application time as well. Since the standard deviation is below 1%, we omit error bars.

In our previous work, we have already shown that data-intensive workloads like tar and untar have significantly less OS overhead on M³ than on Linux when running on simple Xtensa cores. As Fig. 6 shows, these improvements can be seen on x86-64 cores as well. Note however, that the difference is about a factor of two on x86-64 instead of five as on Xtensa, because the Xtensa cores did not have a cacheline prefetcher, resulting in poor performance on Linux [10]. On both architectures, the DTU's data channel can be configured in constant time for any byte-granular and contiguous region of memory, independent of its size. After the channel has been established, applications access the data via DMA with almost no overhead. Therefore, M³ and M³X outperform Linux significantly.

For the remaining applications, computation dominates the runtime, leading to smaller overall performance improvements. Note that SQLite is slightly faster on M³, because the new file protocol in M³X currently does not provide clients with read-write access to data and SQLite often switches between reading and writing of the same file. These switches require a `commit` request and a new `next_in` or `next_out` request, causing additional overhead.

7.4 Tile Sharing

After the performance comparison using three tiles, we show the performance impact when activities share tiles by means of context switching. In the first step, we ran both OS servers (m3fs and pager) on the same tile and in the second step, we ran the OS servers and the application on a single tile.

Fig. 7 shows the average runtime of three runs, preceded by one warmup run, normalized to the average runtime on M³X with three tiles. The standard deviation is less than 2%. As the results show, using the same tile for both servers and a dedicated tile for the application (two tiles in total) has almost no performance impact. Running servers and application on a single tile leads to a performance degradation in some cases. For *tar* and *untar*, the runtime is increased by 17% and 12%, respectively, but M³X is still about twice as fast as Linux. *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, but is still better than on Linux. We conclude that some workloads require faster core-local context switches. We could improve M³X by running a kernel with context-switching support directly on the core, in case the necessary hardware features are available.

7.5 Autonomous Request Processing

After the evaluation on general-purpose cores, we want to demonstrate the benefits of autonomous accelerators. In this section, we start with request-processing accelerators. As described in § 4.9, software invokes the ASM, which in turn invokes the accelerator logic. In this section, we evaluate the impact of the invocation granularity on performance and CPU wake-up frequency.

We simulate the accelerators using gem5-Aladdin [56], which is a power-performance accelerator modeling framework that can be used to explore the design space for fixed-function accelerators. gem5-Aladdin simulates the accelerator logic and uses the memory subsystem of gem5 to perform memory accesses. gem5-Aladdin achieves an error of less than 6% for the accelerator's performance compared to real hardware. We adapted gem5-Aladdin to be invoked by the ASM and to notify the ASM of completions.

To use a request-processing accelerator, an application creates an activity for the desired accelerator, creates the memory mappings for the input and output data in the activity's virtual address space, and establishes the communication channel to invoke the ASM. 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 activity.

We use different accelerator workloads from MachSuite [49]. MachSuite has been analyzed by gem5-Aladdin

[5] 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/M³X. The sum of the times for the ignored system calls was at most 0.4 ms.

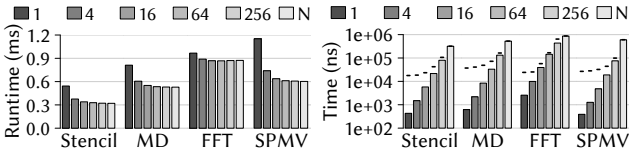

Figure 8: Total runtime (left) and the average (bars on the right) and maximum execution times (lines on the right) for different batch sizes.

with the result that some accelerators benefit from DMA-based memory access and others benefit from cache-based memory access. For this evaluation, we 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.

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

Fig. 8 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 (bars) and maximum (lines) accelerator execution times for each ASM invocation. For example, using the MD workload and a batch size of 16 shows acceptable performance, but leads to a context switching latency of 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 [32, 54]). Hence, deeper sleep states are only beneficial during longer idle periods.

M³X performs all accelerator invocations in a single batch and uses an ASM that is interruptible between invocations to get the best of both worlds: On the one hand, a single batch leads to the best performance. On the other hand, the fine-grained interruptibility allows to context-switch to a more important activity with a low latency. Note that an immediate interruption can be achieved by resetting the accelerator logic, but requires to repeat the last step of the computation. If all invocations are done by the ASM in hardware (*autonomous*), the accelerator needs to repeat only a single indivisible step. If all invocations are done in software (*assisted*), the accelerator needs to repeat as many steps as the performance and energy constraints allow.

7.6 Autonomous Stream Processing

Finally, we want to show the benefits of autonomous stream-processing accelerators. Stream processing is used in various domains such as mobile communication, image processing, and audio processing. In this work, we consider an image processing scenario as is imaginable in data centers, similar to Google's TPU [27] 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 [42], which first performs a 2D fast Fourier transform (FFT) on the input image, then multiplies the result pointwise with an image filter, and finally 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, we use three types of accelerators called FFT, MUL, and IFFT. Each accelerator has 2 KiB (the block size for the 32×32 point FFT) of local scratchpad memory (SPM) and uses the file protocol (see § 5.8) to stream the data block-wise from the input stream via the SPM to the output stream. Due to the deterministic execution model of these accelerators, we did not use gem5-Aladdin to simulate the accelerator logic, but used Aladdin [55] to determine the computation times offline. To get reasonable results, we generated all sensible configurations and picked the sweet spot between performance and the product of chip area and power consumption. We obtained 5,856 cycles for FFT and IFFT and 1,189 cycles for MUL. To put these numbers into perspective, the FFT/IFFT accelerator is about three times as fast as a simple software implementation and Aladdin reports a three orders of magnitude lower power consumption than a typical modern x86 core.

These three types of accelerators run activities that form an FFT-MUL-IFFT chain to process a 4 MiB image file and store the resulting image as a file. In our first experiment, we 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, we compare M³X's autonomous approach with the assisted approach. The assisted approach drives the accelerators from software using a single general-purpose core. Hence, software loads the input data into the SPM, starts the accelerator via a message, and asks the accelerator's DTU to move the result from the SPM to the next accelerator or to the output file. The autonomous variant connects the DTU endpoints of the accelerators as follows: The input of the first and the output of the last accelerator are connected to a file. The output of the first and second accelerator are directly pushed to the successor.

We simulate two ways to attach accelerators to the system: network-on-chip and PCI Express (PCIe). The former leads to superior performance due to lower latency, whereas the latter

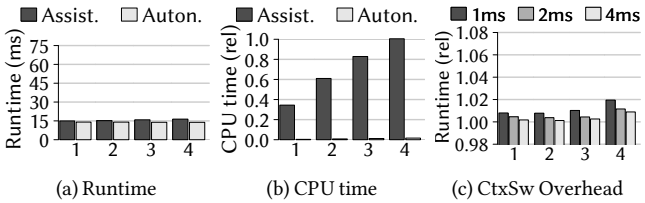

Figure 9: Total runtime, CPU time, and context switching overhead for different numbers of accelerator chains when integrating the accelerators into the NoC.

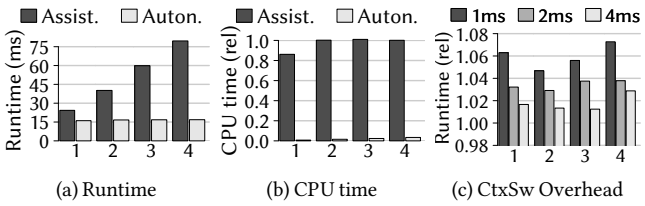

Figure 10: Like Fig. 9, but with PCIe-attached accelerators.

allows a flexible combination of independently developed components. The result for the NoC version is depicted in Fig. 9, whereas the PCIe version is depicted in Fig. 10. We show the overall runtime (a) and the CPU time spent to drive the accelerators (b), depending on the number of accelerator chains using three runs, preceded by one warmup run. The standard deviation is below 3%. We simulate the PCIe-attached add-on card by connecting the accelerators via a bridge with a delay of 500 ns to the host system, which is the typical one-way latency for PCIe gen 3 [20, 23, 30, 53]. In other words, we do not simulate a complete PCIe interconnect, but only the latency PCIe introduces. The one-way latency within the NoC is about 10 ns. In both cases, the DRAM is part of the host system and stores the in-memory file system.

Using the assisted approach leads to slightly worse overall runtime with an increasing number of accelerator chains when integrating the accelerators into the NoC. With PCIe, the overall runtime increases significantly, leading to a slowdown of factor 4.7. In contrast, the autonomous approach always achieves the same runtime, independent of the number of chains. 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 fully utilized starting with two chains. The autonomous approach does not cause significant CPU load in either case. Additionally, Fig. 10 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.

Finally, we evaluate the context switching overhead when two chains of activities compete for the same accelerators. In this case, we use only the autonomous approach and put each chain of activities into the same gang to benefit from gang scheduling. Plot (c) in Fig. 9 and Fig. 10 shows the context switching overhead by comparing the runtime of two activity chains running consecutively with the runtime of two chains running interleaved. We 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 leads to less than 0.9% overhead with accelerators integrated into the NoC and less than 2.9% overhead when attaching them via PCIe.

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 performance by overlapping data transfers to the DRAM instead of issuing one transfer at a time.

8 Conclusion

In this work, we presented M³x, which combines fast-path communication, bypassing the kernel, with context switching and thereby enables autonomous accelerators. To this end, we re-evaluated the boundary between hardware and software. We found that (1) introducing a hardware-friendly file protocol enables accelerators to autonomously access file systems or network stacks, (2) performing potentially complex scheduling decisions in software and simple save and restore actions in hardware allows to context switch accelerators, and (3) attaching a simple hardware component to the accelerator logic allows to combine autonomous operation and interruptibility.

We demonstrated in our evaluation that M³x retains the performance advantages of M³'s fast-path communication, while using the system's resources more efficiently by performing context switches, if required. Additionally, we have shown that running PCIe-attached image processing accelerators autonomously achieves a speedup of 4.7 and reduces the CPU utilization by a factor of 30.

In future work, we plan to study other types of accelerators such as FPGAs and GPUs and apply our insights from simulation to real hardware.

9 Acknowledgments

We would like to thank our shepherd, Christopher Rossbach, and the anonymous reviewers for their helpful suggestions. This work was funded by the German Research Council DFG through the Cluster of Excellence Center for Advancing Electronics Dresden (cfaed) and by public funding of the state of Saxony/Germany.

References

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

[2] Cache Coherent Interconnect for Accelerators (CCIX). http://www.ccixconsortium.com. Accessed: 08/03/2018.

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

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

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

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

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

[8] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 53–79.

[9] Arnold, O., Matus, E., Noethen, B., Winter, M., Limberg, T., and Fettweis, G. Tomahawk: Parallelism and heterogeneity in communications signal processing MPSoCs. *ACM Transactions on Embedded Computing Systems (TECS) 13*, 3s (Mar 2014), 107:1–107:24.

[10] Asmussen, N., Völp, M., Nöthen, B., Härtig, H., and Fettweis, G. 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* (2016), ASPLOS'16, ACM, pp. 189–203.

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

[12] Barbalace, A., Sadini, M., Ansary, S., Jelesnianski, C., Ravichandran, A., Kendir, C., Murray, A., and Ravindran, B. Popcorn: Bridging the programmability gap in heterogeneous-ISA platforms. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys'15, ACM, pp. 29:1–29:16.

[13] Basaran, C., and Kang, K.-D. Supporting preemptive task executions and memory copies in GPGPUs. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems* (Washington, DC, USA, 2012), ECRTS'12, IEEE Computer Society, pp. 287–296.

[14] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP'09, ACM, pp. 29–44.

[15] Bergman, S., Brokhman, T., Cohen, T., and Silberstein, M. SPIN: seamless operating system integration of peer-to-peer DMA between SSDs and GPUs. In *Proceedings of the Seventeenth USENIX Annual Technical Conference* (2017), vol. 17 of *USENIX ATC'17*.

[16] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. The gem5 simulator. *SIGARCH Computer Architecture News 39*, 2 (u 2011), 1–7.

[17] Cota, E. G., Mantovani, P., Di Guglielmo, G., and Carloni, L. P. An analysis of accelerator coupling in heterogeneous architectures. In *Proceedings of the 52nd Annual Design Automation Conference* (New York, NY, USA, 2015), DAC'15, ACM, pp. 202:1–202:6.

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

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

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

[21] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2011), ISCA'11, ACM, pp. 365–376.

[22] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Neural acceleration for general-purpose approximate programs. *IEEE Micro 33*, 3 (May 2013), 16–27.

[23] Flajslik, M., and Rosenblum, M. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX Annual Technical Conference* (San Jose, CA, 2013), USENIX ATC'13, USENIX, pp. 333–346.

[24] Haas, S., Seifert, T., Nöthen, B., Scholze, S., Höppner, S., Dixius, A., Adeva, E. P., Augustin, T., Pauls, F., Moriam, S., Hasler, M., Fischer, E., Chen, Y., Matúš, E., Ellguth, G., Hartmann, S., Schiefer, S., Cederström, L., Walter, D., Henker, S., Hänzsche, S., Uhlig, J., Eisenreich, H., Weithoffer, S., Wehn, N., Schüffny, R., Mayr, C., and Fettweis, G. A heterogeneous SDR MPSoC in 28 nm CMOS for low-latency wireless applications. In *Proceedings of the 54th Annual Design Automation Conference 2017* (New York, NY, USA, 2017), DAC'17, ACM, pp. 47:1–47:6.

[25] Henkel, J., Khdr, H., Pagani, S., and Shafique, M. New trends in dark silicon. In *Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference* (2015), DAC'15, IEEE, pp. 1–6.

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

[27] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2017), ISCA'17, ACM, pp. 1–12.

[28] Kelm, J. H., Johnson, D. R., Tuohy, W., Lumetta, S. S., and Patel, S. J. Cohesion: An adaptive hybrid memory model for accelerators. *IEEE micro 31*, 1 (2011), 42–55.

[29] Kim, S., Huh, S., Hu, Y., Zhang, X., Witchel, E., Wated, A., and Silberstein, M. GPUnet: Networking abstractions for GPU programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 201–216.

[30] Kim, S., and Yang, J.-S. Optimized I/O determinism for emerging NVM-based NVMe SSD in an enterprise system. In *Proceedings of the 55th Annual Design Automation Conference* (2018), DAC'18, ACM, p. 56.

[31] Krill, B., Amira, A., and Rabah, H. Generic virtual filesystems for reconfigurable devices. In *2012 IEEE International Symposium on Circuits and Systems* (2012), IEEE, pp. 1815–1818.

[32] Kurd, N., Chowdhury, M., Burton, E., Thomas, T. P., Mozak, C., Boswell, B., Mosalikanti, P., Neidengard, M., Deval, A., Khanna, A., et al. Haswell: A family of IA 22 nm processors. *IEEE Journal of Solid-State Circuits 50*, 1 (2015), 49–58.

[33] Lackorzynski, A., and Warg, A. Taming subsystems: Capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems* (New York, NY, USA, 2009), IIES'09, ACM, pp. 25–30.

[34] LeBeane, M., Potter, B., Pan, A., Dutu, A., Agarwala, V., Lee, W., Majeti, D., Ghimire, B., Van Tassell, E., Wasmundt, S., Benton, B., Breternitz, M., Chu, M. L., Thottethodi, M., John, L. K., and Reinhardt, S. K. Extended task queuing: Active messages for heterogeneous systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Piscataway, NJ, USA, 2016), SC '16, IEEE Press, pp. 80:1–80:12.

[35] Lim, K., Meisner, D., Saidi, A. G., Ranganathan, P., and Wenisch, T. F. Thin servers with smart pipes: Designing SoC accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA'13, ACM, pp. 36–47.

[36] Lin, F. X., Wang, Z., and Zhong, L. K2: A mobile operating system for heterogeneous coherence domains. *ACM Transactions on Computer Systems 33*, 2 (u 2015), 4:1–4:27.

[37] Liu, D., Chen, T., Liu, S., Zhou, J., Zhou, S., Teman, O., Feng, X., Zhou, X., and Chen, Y. PuDianNao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ASPLOS'15, ACM, pp. 369–381.

[38] Liu, Z., Severance, A., Singh, S., and Lemieux, G. G. Accelerator compiler for the venice vector processor. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays* (2012), FPGA'12, ACM, pp. 229–232.

[39] Mallon, S., Gramoli, V., and Jourjon, G. 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* (New York, NY, USA, 2018), ASPLOS'18, ACM, pp. 737–750.

[40] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium* (2007), vol. 2, pp. 21–33.

[41] Mattson, T. G., Van der Wijngaart, R., and Frumkin, M. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC'08, IEEE Press, pp. 38:1–38:11.

[42] Moreland, K., and Angel, E. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2003), Eurographics Association, pp. 112–119.

[43] Nightingale, E. B., Hodson, O., McIlroy, R., Hawblitzel, C., and Hunt, G. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP'09, ACM, pp. 221–234.

[44] Ousterhout, J. K., et al. Scheduling techniques for concurrent systems. In *ICDCS* (1982), vol. 82, pp. 22–30.

[45] Park, J. J. K., Park, Y., and Mahlke, S. 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* (New York, NY, USA, 2015), ASPLOS'15, ACM, pp. 593–606.

[46] Peter, S., Li, J., Zhang, I., Ports, D. R., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems 33*, 4 (2016), 11.

[47] Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P. Y., and Burger, D. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 13–24.

[48] Qadeer, W., Hameed, R., Shacham, O., Venkatesan, P., Kozyrakis, C., and Horowitz, M. A. Convolution engine: Balancing efficiency & flexibility in specialized computing. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA '13, ACM, pp. 24–35.

[49] Reagen, B., Adolf, R., Shao, Y. S., Wei, G.-Y., and Brooks, D. Machsuite: Benchmarks for accelerator design and customized architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization* (2014), IISWC'14, IEEE, pp. 110–119.

[50] Rodeh, O., Bacik, J., and Mason, C. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (Aug 2013), 9:1–9:32.

[51] Rogers, P., and Fellow, A. Heterogeneous system architecture overview. In *Hot Chips* (2013), vol. 25.

[52] Rossbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP'11, ACM, pp. 233–248.

[53] Rota, L., Vogelgesang, M., Perez, L. A., Caselle, M., Chilingaryan, S., Dritschler, T., Zilio, N., Kopmann, A., Balzer, M., and Weber, M. A high-throughput readout architecture based on PCI-Express Gen3 and DirectGMA technology. *Journal of Instrumentation 11*, 02 (2016), P02007.

[54] Schöne, R., Molka, D., and Werner, M. Wake-up latencies for processor idle states on current x86 processors. *Computer Science-Research and Development 30*, 2 (2015), 219–227.

[55] Shao, Y. S., Reagen, B., Wei, G.-Y., and Brooks, D. 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* (2014), ISCA'14, IEEE, pp. 97–108.

[56] Shao, Y. S., Xi, S. L., Srinivasan, V., Wei, G.-Y., and Brooks, D. Co-designing accelerators and SoC interfaces using gem5-aladdin. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture* (2016), MICRO'16, IEEE, pp. 1–12.

[57] Silberstein, M. OmniX: An accelerator-centric OS for omni-programmable systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2017), HotOS'17, ACM, pp. 69–75.

[58] Silberstein, M., Ford, B., Keidar, I., and Witchel, E. GPUfs: Integrating a file system with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS'13, ACM, pp. 485–498.

[59] So, H. K.-H., and Brodersen, R. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Transactions on Embedded Computing Systems 7*, 2 (Jan 2008), 14:1–14:28.

[60] Soares, L., and Stumm, M. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–8.

[61] Steinberg, U., and Kauer, B. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys'10, ACM, pp. 209–222.

[62] Stuecheli, J., Blaner, B., Johns, C., and Siegel, M. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development 59*, 1 (2015), 7–1.

[63] Thanh-Hoang, T., Shambayati, A., Deutschbein, C., Hoffmann, H., and Chien, A. A. Performance and energy limits of a processor-integrated FFT accelerator. In *Proceedings of the 2014 IEEE High Performance Extreme Computing Conference* (2014), HPEC'14, IEEE, pp. 1–6.

[64] Tseng, H.-W., Zhao, Q., Zhou, Y., Gahagan, M., and Swanson, S. Morpheus: creating application objects efficiently for heterogeneous computing. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture* (2016), ISCA'16, IEEE, pp. 53–65.

[65] Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., Swanson, S., and Taylor, M. B. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2010), ASPLOS XV, ACM, pp. 205–218.

[66] Wentzlaff, D., and Agarwal, A. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review 43*, 2 (Apr 2009), 76–85.

[67] Wu, L., Barker, R. J., Kim, M. A., and Ross, K. A. Navigating big data with high-throughput, energy-efficient data partitioning. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA'13, ACM, pp. 249–260.

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

[69] Zhang, J., Donofrio, D., Shalf, J., Kandemir, M. T., and Jung, M. NVMMU: A non-volatile memory management unit for heterogeneous GPU-SSD architectures. In *Proceedings of the International Conference on Parallel Architecture and Compilation* (2015), PACT'15, IEEE, pp. 13–24.