Efficient and Scalable Core Multiplexing with M^{3}v

Nils Asmussen, Barkhausen Institut, Dresden, Germany
Sebastian Haas, Barkhausen Institut, Dresden, Germany
Carsten Weinhold, Barkhausen Institut, Dresden, Germany
Till Miemietz, Barkhausen Institut, Dresden, Germany
Michael Roitzsch, Barkhausen Institut, Dresden, Germany

ABSTRACT

The M^3 system (ASPLOS '16) proposed a hardware/software co-design that simplifies integration between general-purpose cores and special-purpose accelerators, allowing users to easily utilize them in a unified manner. M^3 is a tiled architecture, whose tiles (cores and accelerators) are partitioned between applications, such that each tile is dedicated to its own application.

The M^{3}x system (ATC '19) extended M^3 by trading off some isolation to enable coarse-grained multiplexing of tiles among multiple applications. With M^{3}x, if source tile t_1 runs code of application p and sends a message m to destination tile t_2 while t_2 is currently not associated with p, then m is forwarded to the right place through a "slow path", via some special OS tile.

In this paper, we present M^{3}v, which extends M^{3}x by further trading off some isolation between applications to support "fast path" communication that does not require the said OS tile's involvement. Thus, with M^{3}v, a tile can be efficiently multiplexed between applications provided it is a general-purpose core. M^{3}v achieves this goal by 1) adding a local multiplexer to each such core, and by 2) virtualizing the core's hardware component responsible for cross-tile communications. We prototype M^{3}v using RISC-V cores on an FPGA platform and show that it significantly outperforms M^{3}x and may achieve competitive performance to Linux.

CCS CONCEPTS

• **Computer systems organization** → *Architectures*; • **Security and privacy** → *Operating systems security*; • **Software and its engineering** → **Process management**; **Communications management**.

KEYWORDS

Context Switching, Hardware Virtualization

ACM Reference Format:
Nils Asmussen, Sebastian Haas, Carsten Weinhold, Till Miemietz, and Michael Roitzsch. 2022. Efficient and Scalable Core Multiplexing with M^{3}v. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3503222.3507741

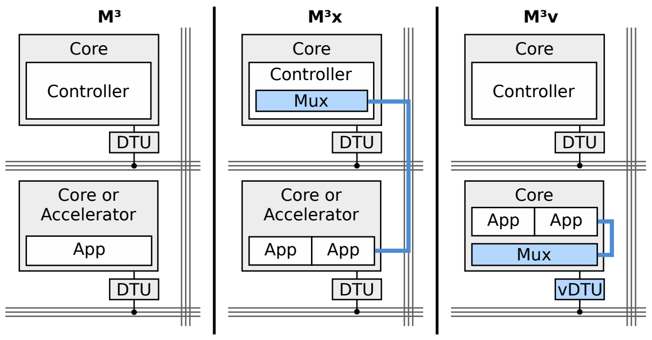


Figure 1: Tile multiplexing on M^3 (no multiplexing), M^{3}x (all tiles can be multiplexed using a single centralized OS tile), and M^{3}v (general-purpose tiles can multiplex themselves). Each of the lower tiles can exist multiple times in a real platform.

1 INTRODUCTION

M^3 [16] is a hardware/software co-design that addresses the trend towards increasingly heterogeneous systems [28, 29, 40, 41, 52]. It is based on a tiled hardware architecture [59] and allows users to easily utilize general-purpose cores and special-purpose accelerators in a unified manner. Communication between tiles is achieved with a custom per-tile hardware component called data transfer unit (DTU). The DTU provides a uniform interface to all tiles, which simplifies heterogeneous systems. Additionally, the DTU isolates tiles from each other, because cross-tile communication is denied by default. Communication channels between tiles are set up by the *communication controller*,[1] or *controller* for short, which runs on a dedicated OS tile. After the setup, applications can communicate directly via their DTU, bypassing the controller. Therefore, we call such communication *fast-path communication*.

As depicted in Figure 1 (left), M^3 does not support tile multiplexing and is therefore limited to one application per tile. The inability to multiplex tiles inhibits tile utilization when applications are occasionally idle. Multiplexing tiles among multiple applications therefore enables increased tile utilization. One reason that M^3 and similar systems like DLibOS [42] are limited to one application per tile is that tile multiplexing impedes fast-path communication: the OS is responsible for tile multiplexing, but the goal of fast-path communication is to bypass the OS. For example, if an application is waiting for an incoming message, the OS needs to suspend the application to allow forward progress for other applications on the same tile. Later, the OS needs to resume the suspended application upon message arrival.

M^{3}x [15] resolved this challenge in a manner that allows for multiplexing of both general-purpose cores and special-purpose accelerators. With M3x, the controller performs all context switches on

[1] We decided to use the name *communication controller* instead of "kernel" as in previous M^3 papers to prevent confusion with the traditional meaning of "kernel".

all tiles in the system remotely, as illustrated in Figure 1 (middle). Namely, the controller is responsible for scheduling decisions, asks other tiles to save or restore their state, and switches between contexts. M^{3}x retains fast-path communication if the recipient is running, but otherwise resorts to *slow-path communication*, which redirects the communication over the controller. When two applications share a tile and cause frequent slow-path communication, M^{3}x suffered from performance problems.

In this paper, we present M^{3}v, a new core-multiplexing approach for the M^3 system that replaces the general mechanism of M^{3}x by a specific one for general-purpose cores. We trade some of the isolation and generality of M^{3}x for improved efficiency. As sketched in Figure 1 (right), our design is based on 1) a core-local software multiplexer, which performs context switches on this core without involving the controller on the OS tile and on 2) hardware virtualization of the DTU (*vDTU*). In contrast to the previous M^3 prototypes that were simulated, we built a hardware FPGA-based implementation of M^{3}v including our core-multiplexing support. In the evaluation, we compare to M^{3}x in simulation using a context-switch heavy workload, showing a two-fold performance improvement and almost linear scalability up to 12 tiles for M^{3}v, whereas M^{3}x does not scale to two tiles. Additionally, we evaluate the performance of M^{3}v in comparison to Linux on an FPGA platform, showing that M^{3}v is competitive with single and multiple applications per core.

In terms of isolation, both M^{3}x and M^{3}v are less secure than M^3, because they weaken isolation, allowing multiple applications to share a physical core, whereas M^3 enforces physical isolation. The newly proposed M^{3}v further trades off some isolation relative to M3x, as multiplexing is supported in both M^{3}x and M^{3}v but M^{3}x implements it remotely in the controller, where it is isolated from untrusted, potentially malicious applications. In contrast, M^{3}v implements multiplexing in the application tiles, where no such physical isolation exists, which is most likely riskier, notably with respect to side-channel attacks.

Moreover, whereas M^{3}x multiplexing functionality is implemented in software and is operational only at the controller, M^{3}v multiplexing is implemented in hardware in each individual core. Thus, M^{3}v trades off *software* complexity (that affects only one core), with *hardware* complexity (affecting all cores). In all M^3 variants, taking over the controller implies taking over the entire machine.

Paper Roadmap. In section 2, we provide background on M^3 and M^{3}x and discuss how similar system properties can be achieved with traditional architectures. We then present the design and implementation of M^{3}v in Sections 3 and 4, respectively. Section 5 discusses the trade-off between isolation and efficiency. We evaluate M^{3}v in Section 6 and then discuss related work and future work in Sections 7 and 8, respectively.

2 BACKGROUND

This section introduces M^3 [16] and provides the required background on its extension M^{3}x [15], which implements core multiplexing. Furthermore, we compare the properties of the M^3 platform with traditional system architectures and discuss the differences.

2.1 The M^3 Hardware/Software Platform

M^3 [16] proposed a new system architecture based on a hardware/software co-design. On the hardware side, M^3 builds upon a tiled architecture [59], as shown in Figure 3. M^3 extends its tiles by adding a new hardware component called data transfer unit (DTU) to them. Each tile contains a DTU and either a core, an accelerator, or memory (e.g., a memory interface to off-chip DRAM). In contrast to conventional architectures, M^3 does not build upon coherent shared memory, but uses the DTU for cross-tile messaging and memory accesses. To perform message-passing or memory accesses, a corresponding *communication channel* needs to be established. Communication channels are represented as *endpoints* in the DTU. At runtime, each endpoint can be configured to different endpoint types: A *receive endpoint* allows to receive messages, a *send endpoint* allows to send messages to a specific receive endpoint, and a *memory endpoint* allows to issue DMA requests to tile-external memory. Message passing is performed between a pair of send and receive endpoint, whereas each memory endpoint refers to a region of memory without an endpoint on the memory side.

On the software side, M^3 runs a *communication controller*, or *controller* for short, on a dedicated *controller tile*, and applications and OS services on the remaining *user tiles*. Applications and OS services on user tiles are represented as *activities*, comparable to processes. An activity on a general-purpose tile executes code, whereas an activity on an accelerator tile uses the accelerator's logic. Activities can use existing communication channels, but only the controller is allowed to establish such channels. By default, no communication channels exist and thus tiles are isolated from each other. M^3 runs at most one activity per tile and cannot start a new activity on this tile until the current activity has terminated. In the expectation of abundantly available tiles, M^3 does not support tile multiplexing.

2.2 Tile Multiplexing and Autonomous Accelerators with M^{3}x

M^{3}x [15] introduced tile multiplexing and thereby trades some of the isolation of M^3 for the ability to use the available resources more efficiently. Since tiles do not share hardware resources such as caches, side-channel attacks based on these resources are not possible between activities on different tiles. Therefore, if activities share a tile, isolation between these activities is arguably weaker than the isolation between activities on different tiles.

M^{3}x proposed to multiplex cores and accelerators with the same mechanism: the controller performs all context switches on all user tiles in the system remotely from the controller tile. The controller is responsible for scheduling decisions, whereas user tiles are responsible for saving or restoring the activity's state upon request by the controller. Since all activities on a tile share the same DTU, the DTU endpoints need to be saved and restored as well. As the ability to restore endpoints also allows to create arbitrary communication channels, saving and restoring of endpoints is done by the controller in M^{3}x. Activities can use fast-path communication in case the recipient is currently running and fall back to slow-path communication otherwise. The slow path forwards the message to the recipient via the controller, which first schedules the recipient and delivers the message afterwards.

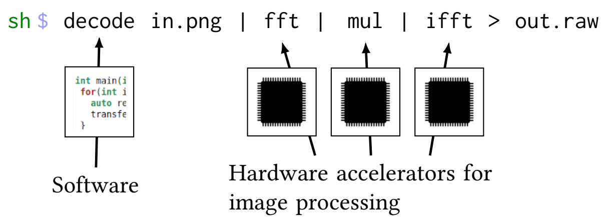

sh $ decode in.png | fft | mul | ifft > out.raw
Software
Hardware accelerators for image processing

Figure 2: Combination of software (`decode`) and hardware accelerators (`fft`, `mul`, and `ifft`) in M^{3}x's shell.

Besides tile multiplexing, M^{3}x also improved the user experience when utilizing accelerators. As an example, consider that a user wants to do edge detection on image files. The input image is stored in the file system and the output image should be stored as raw pixel data for later post processing. The actual image processing can be done faster and more energy-efficient on specific hardware accelerators (e.g., using FFT convolution [47]). However, accessing files from accelerators or pipelining accelerators with software is challenging [30, 48, 54, 55]. M^{3}x showed how accelerators can run "autonomously" by connecting them directly with OS services or activities on general-purpose cores. For that reason, the edge detection can be performed on M^{3}x's shell as illustrated in Figure 2. M^{3}x showed that the autonomous execution can lead to significant speedups and reduced CPU utilization.

Apart from these advantages, M^{3}x revealed performance problems for workloads on general-purpose cores that frequently require the slow-path for communication. For example, if two activities share a tile, communication between these activities is only possible through the slow path. The goal of this paper is to make multiplexing of general-purpose cores more efficient.

2.3 Comparison to Existing Architectures

To illustrate what we want to achieve in this paper (namely, the increment we establish over M^3/M^{3}x), let us describe how the desired additional functionality would have been deployed in a more conventional, "regular" system, by combining existing hardware and software components in a manner that achieves, more or less, similar isolation properties to that of M^{3}v. Consider a hypothetical multi-socket system in which each socket consists of a single CPU core. Such a system can run a multikernel operating system (e.g., Barrelfish [17]) which executes a separate kernel on each socket. Communication can then be performed via a network interface card (NIC) that is shared between all sockets and that provides hardware virtualization features such as SR-IOV. Similarly to M^3, one socket is used as the *controller socket*. The shared NIC is physically connected to the PCIe root complex of the controller socket. This allows the controller socket to bind itself to the physical function of the NIC and thus exclusively control its configuration. The controller socket can set up communication channels between two sockets by creating two virtual functions of the NIC that are tagged with a specific VLAN ID. Afterwards, the controller grants each socket that should participate in said communication channel access to one of the respective virtual functions by making the associated PCIe memory visible to the respective sockets. Since the user of a virtual function cannot see or modify the VLAN tag associated with it, communication channels are isolated from each other. However, in order to provide the same security properties as M^3, we need to additionally ensure memory isolation. When DRAM is shared among sockets, the controller socket must restrict memory accesses of all unprivileged sockets to individual ranges of the physical address space. If it was possible to arrange things such that 1) sockets are able to access the shared memory using only a per-CPU DMA engine that copies memory to memory (like Intel's I/O Acceleration Technology [11]), and 2) only the controller is able to program the IOMMU thereby creating restrictive address spaces for these per-CPU DMA engines, as required – then similar isolation to that of M^3 would have been achieved.

Such a setup would have had similar system properties as our proposed system M^{3}v. Namely, sockets would have been isolated from each other by default and all shared memory or communication channels would have been explicitly established by the controller socket. In this setup, each socket can be multiplexed among multiple applications by the socket-local kernel instance, similarly to our tile-local multiplexer described in further detail in section 3. Since the NIC supports SR-IOV, each application can have its own virtual NIC and applications can benefit from fast-path communication without involving the controller socket. Therefore, instead of virtualizing the DTU, it is imaginable to replace all DTUs with something that is conceptually similar to a single SR-IOV-enabled NIC.

Such a theoretical setup is functionally analogous to what we want to achieve with the M^{3}v design in terms of isolation and communication between compute cores. But using an SR-IOV-enabled NIC in this way is ill-suited for the tiled systems-on-a-chip (SoCs) that the M^3 architecture targets, for the following reasons. First, the NIC is typically off-chip, so that all on-chip traffic is routed over an off-chip NIC. Considering that the tile-to-tile latency within our on-chip network is dozens of nanoseconds and typical PCIe latencies are about 1µs [22, 31], we expect an increase of communication costs by multiple orders of magnitude. Second, independent of whether the NIC is on-chip or off-chip, it would be a central hub for all communication and therefore constitute an inherent bottleneck. Decentralizing the NIC and its SR-IOV enforcement is non-trivial, whereas our solution is distributed by design. And third, like the DTU, the NIC would become part of the trusted computing base, but it is arguably more complex. We show in this paper how the DTU as a simple on-chip communication device can be virtualized and thereby shared in an efficient and lightweight manner.

3 DESIGN

The overall goal of our work is to extend the M^3 system architecture by the ability to multiplex general-purpose cores efficiently among multiple applications. In more detail, our goals are:

Efficient and scalable multiplexing: The overhead of a context switch should be small, so tiles can be multiplexed efficiently. Furthermore, tile multiplexing should scale with the number of tiles.

Transparent multiplexing: Activities should be able to use the same communication mechanism, independent of whether the communication partner runs on the same tile or a different tile.

Strong isolation between tiles: Like in M^3/M^{3}x, tiles should not share resources to prevent that such resources can be used for side-channel attacks between activities on different tiles. Therefore, isolation between tiles is arguably stronger than isolation between activities on the same tile. Strong isolation between tiles requires that the tile-local multiplexer has no control over other tiles. In other words, access to tile-external resources can only be granted by the controller.

Weak isolation within tiles: Activities running on the same tile should be isolated from each other using the common mechanisms of address spaces and privilege levels, as in M^{3}x. Such isolation is arguably weaker than the isolation between tiles, because co-located activities share the tile's resources. For example, the shared resources open the possibility for side-channel attacks between activities on the same tile.

3.1 Overall Approach

Traditional OS kernels multiplex cores by saving and restoring its state (e.g., registers) with the very same core. Multiplexing a tile in our system architecture requires to additionally multiplex the DTU among the applications running on this tile. However, allowing a tile's core to multiplex its associated DTU by saving and restoring the DTU state and in particular the DTU endpoints, would also allow this core to create arbitrary communication channels. Thus, the core would have full control over the entire system and thereby break the isolation between tiles.

This challenge is resolved in M^{3}x by letting the controller save and restore the DTU state on each context switch. Since the controller runs on a single dedicated tile and performs all context switches remotely on all other tiles, M^{3}x shows performance and scalability problems (see subsection 6.4 for details). We therefore decided to not save/restore the DTU state at all, but enforce that each application can only access its own DTU state. To make that enforcement efficient we virtualized the DTU. This approach results in more efficient and scalable multiplexing of general-purpose cores, which we show in section 6, and retains strong isolation between tiles. However, our approach provides weaker isolation than M^{3}x, as discussed in more detail in section 5. The following provides an overview on the system architecture, introduces the per-tile multiplexer, and describes the individual extensions of the DTU to virtualize it including their interaction with TileMux.

3.2 System Architecture

Figure 3 depicts the system architecture of M^{3}v. The controller runs on a dedicated tile, whereas applications and OS services run on the remaining user tiles, represented as activities. On general-purpose cores, activities execute code, whereas accelerators work on a *context* associated with the current activity. User tiles with general-purpose cores contain the virtualized DTU (vDTU) and run the tile-local multiplexer called *TileMux*. The combination of TileMux and vDTU allows to run multiple activities on the same tile. In contrast, the controller tile does not need a vDTU as it only runs the controller. Similarly, accelerator tiles cannot be currently multiplexed by M^{3}v (see section 8) and thus also contain a non-virtualized DTU. As in M^3 and M^{3}x, memory tiles do not need to be multiplexed, but multiple tiles can get shared access to memory tiles via memory endpoints.

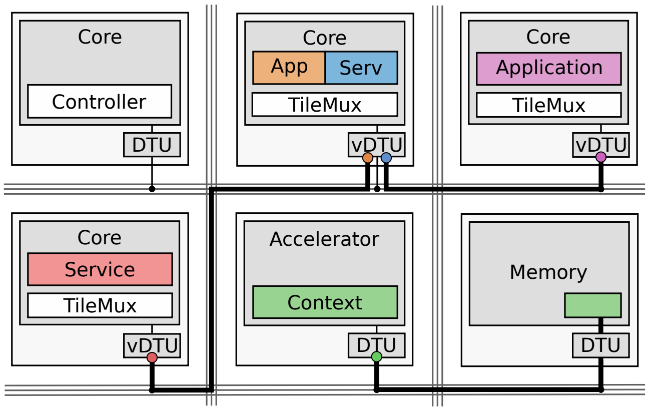


Figure 3: System architecture of M^{3}v. The DTU endpoints at the end of communication channels correspond to the activity that owns the endpoint.

Figure 3 also shows communication channels between tiles with their DTU endpoints on both ends. Communication channels between two DTUs represent message-passing channels, whereas channels between one DTU and memory allow to issue DMA requests to a region within that memory. The color of the endpoints corresponds to the activity that owns the endpoint.

3.3 Tile-Local Multiplexer

The tile-local multiplexer is called *TileMux* and is responsible for multiplexing the tile among all activities on this tile. TileMux only runs on general-purpose cores and leverages the different privilege modes and address spaces to isolate itself from tile-local activities and these activities from each other. TileMux is furthermore responsible for scheduling the tile-local activities and to perform low-level memory management (e.g., manipulation of page-table entries as explained in more detail in subsection 4.3) upon requests by the controller. Therefore, TileMux is comparable to traditional kernels and microkernels [32, 35, 57]. Like with a multikernel [17], we run one TileMux instance per tile. Additionally, TileMux has no control beyond its own tile, because TileMux cannot change DTU endpoints and we use memory endpoints to control memory accesses to shared tile-external memory (see subsection 4.3).

Besides the strong isolation between tiles, we want to allow interactions between all activities on all tiles as permitted by the controller, like in M^3 and M^{3}x. Therefore like before, the controller knows all activities in the system and is responsible for establishing communication channels. The controller decides which channels are established via capability-based access control [46]. Activities send "system calls" in form of DTU messages to the controller in order to create, exchange, and revoke capabilities. M^{3}v extends the controller by communication channels to each TileMux instance. The controller has a send endpoint for requests to TileMux, which are used to, for example, create new activities or kill activities. TileMux has a send endpoint to notify the controller about activity terminations.

TileMux offers *TMCalls* via a trap (e.g., `ecall` on RISC-V) for the activities on its tile. TMCalls are used by activities to block for incoming messages or report a voluntary exit to TileMux.

3.4 Virtualizing the DTU

The DTU provides two interfaces: the *external interface* for the controller to change endpoints and thereby establish or tear down communication channels; and the *unprivileged* interface for activities that allows them to use existing channels. We added a third interface to virtualize the DTU, called *privileged interface.* The privileged interface can only be used by TileMux and enables TileMux to maintain the illusion for activities that each activity has its own vDTU. The following sections describe how the privileged interface is used to securely share tile-local resources (endpoints and memory) between tile-local activities and how communication with non-running activities can be supported. The similarities and differences between the virtualization of the DTU and SR-IOV are discussed in section 7.

3.5 Endpoint Protection

Sharing the vDTU with multiple activities raises the question of how to prevent that one activity can use endpoints of another activity. One approach is to multiplex the vDTU endpoints among all activities by letting TileMux mediate all vDTU accesses. However, we learned in a first design iteration that this is not sufficient, because it degraded the performance of all communication by an order of magnitude due to several involvements of TileMux. We concluded that activities need to be able to use the vDTU directly, without mediation by software.

The vDTU therefore tags all endpoints with the owning activity id and provides the register `CUR_ACT` containing the id of the current activity on the tile. The register is part of the privileged interface and can therefore only be accessed by TileMux. Attempts to use communication endpoints of another activity result in an "unknown endpoint" error to prevent that activities can gain information about other endpoints.

3.6 Tile-Local Memory Protection

Besides endpoint protection, we need to prevent activities from using the vDTU to access each others memory or the memory of TileMux. For example, when sending a message via vDTU, activities need to specify its address in memory. The vDTU needs a physical address to load the message from memory. However, allowing activities to specify the physical address would allow them to, for example, access memory from another activity on the same tile. For that reason, the vDTU accepts virtual addresses from activities and translates them to physical addresses. However, to keep the vDTU small and simple, some compromises must be made.

To translate virtual addresses to physical addresses, the vDTU contains a software-loaded translation lookaside buffer (TLB) for the recent translations. To further reduce the vDTU's complexity we restricted the memory range covered by a source or destination for reads, writes, and sends to a single page. Additionally, we decided against interrupt injections in case of a TLB miss. These two restrictions allow the vDTU to check the TLB once before every command execution instead of, for example, multiple times during large memory transfers. Thus, the vDTU lets the command fail in case of a TLB miss, in which case the activity uses a TMCall to pass the desired virtual address and access mode (read/write) to TileMux. TileMux translates the virtual address and inserts the resulting physical address into the TLB via the vDTU's privileged interface.

3.7 Waiting for Messages by Blocking Activities

If an activity wants to wait for incoming messages, a non-polling solution is preferable, because it gives other ready activities the chance to perform useful work. Therefore, TileMux tells the current activity via shared memory whether other activities are ready. If other activities are ready, the current activity uses a TMCall to be blocked by TileMux until a new message arrives. However, without further measures, TileMux cannot block activities without risking that message notifications are lost, similar to the lost wake-up problem [53]. Therefore, the solution must make sure that the check for the absence of new messages and the blocking operation are atomic. Note that our current implementation polls the vDTU for new messages if no other activities are ready. We leave a more energy-efficient solution for future work.

With many communication endpoints, atomically iterating over all endpoints to check for received messages is not desirable. Therefore, the `CUR_ACT` register contains not only the id of the current activity, but also the number of unread messages for this activity. The vDTU keeps track of messages by incrementing this counter whenever a message for the current activity arrives and decrements it on message consumption. For each non-running activity, TileMux maintains a counter in memory. Furthermore, the vDTU offers a command in its privileged interface that atomically switches to another activity and returns the contents of `CUR_ACT` for the old activity. This approach allows TileMux to check the message count of the old activity in order to decide whether it can be blocked until a new message arrives. The atomicity of the command guarantees that no other events within the vDTU can interfere with the activity switch.

3.8 Receiving Messages for Blocked Activities

If an activity that is currently blocked receives a message, TileMux needs to be notified to decide whether it should switch to the recipient. Like in M^3 and M^{3}x, messages are transferred between send endpoints and receive endpoints using credit-based flow control [44], which is maintained by the vDTUs. Each send endpoint is connected to exactly one receive endpoint, whereas receive endpoints can receive messages from multiple send endpoints. Received messages are stored in a per-endpoint *receive buffer* in memory. However, in contrast to M^{3}x, received messages can always be stored in the receive buffer of the targeted receive endpoint, independent of whether the recipient (the owner of the receive endpoint) is running. The reason is that the DTU in M^{3}x has only the endpoints of the currently running activity available. Therefore, the DTU does not know where to store a message for a non-running activity and M^{3}x needs to fall back to slow-path communication. The vDTU in M^{3}v knows all endpoints of all activities on the tile, independent of whether they are currently running or not, and can therefore always use fast-path communication.

If the recipient is not running, the vDTU additionally injects an interrupt into the core to notify TileMux. The mechanism is called *core request* and the core request tells TileMux which activity received a message. However, as multiple receive endpoints can receive messages simultaneously, the vDTU needs to maintain a small queue of core requests. The queue is not accessible for TileMux, but TileMux can handle the first core request through the vDTU's privileged interface. TileMux obtains information about the core request by reading a register in the privileged interface and has to acknowledge the core request by a write to the same register. Afterwards the vDTU might issue another core request via interrupt in case the queue is not empty. Queue overruns are handled via the packet-based flow control of the on-chip network, which operates independently of the higher-level credit-based flow control of the vDTU.

3.9 Transparent Multiplexing

Different placements of activities should not require different communication mechanisms. For that reason, we decided to use the vDTU for all communication, even if both communication partners run on the same tile. If the recipient is currently running on a different tile, it directly receives the message, otherwise TileMux on the receiving tile receives an interrupt and marks the recipient as ready. The sender does not need to distinguish between these two cases, which makes it transparent for the sender. In contrast, with M^{3}x messages cannot be delivered via vDTU if the recipient is not running. In this case, the activity needs to communicate with the controller, which first schedules the recipient and delivers the message afterwards.

4 IMPLEMENTATION

Our implementation is based on the openly available M^{3}x hardware/software platform [3]. We replaced M^{3}x's support for tile multiplexing with a more efficient multiplexing of general-purpose cores and virtualized its DTU model for gem5 [19]. Furthermore, we implemented the vDTU in hardware with all features to run complex workloads on our FPGA prototype. To ease the development, we used both gem5 and the FPGA prototype and made them binary compatible. M^{3}v, the vDTU model in gem5, and the vDTU hardware implementation are available as open source[2]. This section explains the most important aspects of our implementation.

4.1 Hardware Implementation

In contrast to our previous work on M^3, we built a hardware prototype of the M^{3}v system architecture. Our hardware platform depicted in Figure 4 is implemented on a Xilinx Virtex UltraScale+ FPGA (VCU118 board). The architecture allows to place hardware components such as cores, memories, and I/O devices on physically separated tiles. The current prototype contains eight processing tiles with a single RISC-V core each. One of these processing tiles has an on-chip NIC attached to its RISC-V core, which provides network access in our benchmarks. Additionally, our platform has two memory tiles with interfaces to external DDR4 DRAM. Finally, one tile runs a hardware UDP/IP stack, which is used exclusively for configuration and debugging purposes. This UDP/IP tile is only involved in benchmark setup and does not contribute to any measurements. Even though the architecture would allow to build larger designs, we are limited by the given resources of the FPGA.

All tiles are connected by a network-on-chip (NoC) using a 2x2 star-mesh topology. The NoC consists of routers, links between them, and links to the tiles. Furthermore, each processing tile integrates a vDTU to enable tile multiplexing, whereas other tiles integrate a DTU. For simplicity, the following speaks only of the vDTU, which additionally implements the privileged interface, in contrast to the DTU. The vDTU controls the interface between the core and the

[2]https://github.com/Barkhausen-Institut/M3

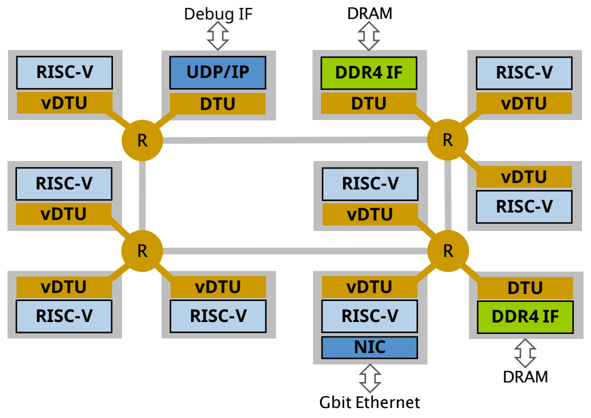


Figure 4: The M^{3}v hardware platform with eight RISC-V tiles, two memory tiles, and one debug tile. The tiles are connected via a NoC with four routers (R).

NoC. As described earlier, the vDTU enforces isolation policies on the hardware level as set up by the controller.

RISC-V Cores. Our hardware platform employs RISC-V cores in the processing tiles. We use Rocket cores [13] and BOOM cores [63], which are available as open source. Rocket is a 64-bit RISC-V in-order core with MMU and 16 kB L1 cache, each for instruction and data, as well as a shared 512 kB L2 cache. BOOM is the out-of-order variant of Rocket with the same cache configuration. We set the clock frequencies of the Rocket and BOOM cores to 100 MHz and 80 MHz, respectively, to fully meet timing requirements during FPGA synthesis and place-and-route. The main memory is located in the external DDR4 DRAM and can be accessed via the vDTU's PMP feature (see subsection 4.3).

Additionally, we integrated a NIC into a selected processing tile, which provides the necessary hardware basis for our network stack running on the RISC-V core (see subsection 4.4). The on-chip NIC is built out of Xilinx Ethernet IP blocks including an AXI-based Ethernet subsystem with the Ethernet MAC, which is connected to the external Ethernet PHY. In addition, an AXI DMA block is also connected to the core's cache-coherent system bus to access data of sent and received Ethernet packets. The RISC-V core has interrupt-driven access to the AXI DMA.

Core-vDTU Interface. The core can use the unprivileged and privileged interface of the vDTU via memory-mapped I/O (MMIO). Both interfaces offer *command registers* to operate the vDTU. For example, the unprivileged interface allows to send a message via the SEND command, which expects the id of the send endpoint, the message address in memory, and the message size as arguments in the command registers. The vDTU first verifies whether the command can be executed based on the endpoint with given id (e.g., the message is not larger than the maximum size set in the send endpoint). To execute the command, the vDTU has access to the core's cache-coherent system bus via DMA. For example, the vDTU loads a message from memory via the cache-coherent system bus before sending the message to the receiver over the NoC.

Virtualized Data Transfer Unit. From a hardware perspective, the feature set of the vDTU is provided by three main functionalities:

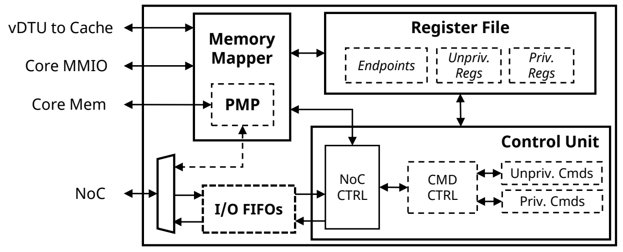


Figure 5: Hardware implementation of the vDTU. Dashed blocks and arrows indicate optional components.

1) command execution to enable data transfers, 2) interfacing the NoC and tile-internal components (e.g., the core), and 3) providing access to the vDTU's register file.

First, the vDTU executes commands to support message passing and memory transfers (*unprivileged commands*) as well as commands to handle virtual memory and switch between activites (*privileged commands*). Commands are implemented as finite state machines. They are processed and executed in the vDTU control unit as shown in Figure 5. Furthermore, the vDTU includes logic to access the NoC, which allows the vDTU to send and receive packets via the NoC.

Second, a memory mapper multiplexes the access of the core to the vDTU via MMIO and the vDTU's access to the core's cache. Furthermore, to control accesses to physical memory (e.g., shared DRAM), the memory mapper routes all last-level cache misses of the core through *physical-memory protection* (PMP) within the vDTU. The vDTU decides whether the access is allowed based on memory endpoints. The current implementation uses the first four endpoints as memory endpoints for PMP and selects the PMP endpoint with the upper two bits of the physical address.

Third, a register file holds the necessary hardware registers which are used by the core within the tile to communicate with the vDTU. In our current hardware implemention, the register file contains 2 external registers, 4 unprivileged registers, 4 privileged registers, and 128 endpoints. The number of endpoints is a trade-off between usability and chip area consumption. Significantly increasing the number of endpoints would require to outsource endpoints to external memory. We leave this open for future work. As shown in Figure 5, parts of the vDTU drawn with dashed lines can be omitted for specific tiles. The non-virtualized DTU in the controller tile and accelerator tiles can omit the privileged registers and privileged commands. DTUs in memory tiles omit all components with dashed lines.

4.2 TileMux

TileMux is a small software component running on a single tile and is implemented in Rust. We chose Rust due to the memory-safety guarantees the language provides and also rewrote the controller and all system services of M^3 in Rust. TileMux supports the x86-64, ARMv7, and RISC-V architectures, but we focus on RISC-V in this work, because the FPGA prototype employs RISC-V cores.

Similar to an OS kernel, TileMux is running in the core's privileged mode (e.g., supervisor mode on RISC-V) and is entered via TMCalls by activities (e.g., using the `ecall` instruction on RISC-V) and upon interrupts or exceptions. TileMux uses address spaces as provided by the core's MMU to isolate activities from each other. Additionally, TileMux maps itself into every address space using pages that cannot be accessed by activities. If the running activity causes a page fault, the core raises a page fault exception, which is handled by TileMux in collaboration with a pager, as explained in more detail in subsection 4.3.

TileMux is responsible for scheduling the activities on its tiles. TileMux implements a preemptive round-robin scheduler with time slices and uses timer interrupts to preempt activities as soon as their time slice is depleted. However, to keep TileMux simple, interrupts are disabled while TileMux is running. Activities can use TileMux to wait for events such as received messages and hardware interrupts of tile-local devices (used by the network driver as described in subsection 4.4). If no activities are ready to run, TileMux runs an *idle activity*. As soon as a non-running activity received a message and has time left to execute, TileMux switches to that activity. Note that our current implementation pins activities to tiles, but could be extended to support activity migration between tiles.

As described in subsection 3.4, the vDTU offers an external, privileged, and unprivileged interface. The external interface can only be used via "external" requests from the controller. The privileged and unprivileged interface are accessible through MMIO from the core that is attached to the vDTU. To ensure that activities can only use the unprivileged interface, TileMux maps the privileged interface only for itself. Hence, only TileMux can switch between different activities, maintain the vDTU's software-loaded TLB, and handle interrupts upon receiving messages for non-running activities. Since TileMux requires endpoints to communicate with the controller, it has a special activity id and these endpoints are tagged with this activity id. However, the vDTU does not allow to use endpoints of non-running activities. Therefore, TileMux needs to switch to its own activity id before being able to use its own endpoints. In general when switching between activities, TileMux ensures that no message notifications are lost by checking the old value of the `CUR_ACT` register after the atomic switch provided by the vDTU.

4.3 Memory Management

The memory management in M^{3}v is split into physical-memory management and virtual-memory management. Physical-memory access of each tile needs to be granted by the controller and is enforced by the tile's vDTU. If a tile's core has a memory-management unit (MMU), activites on this core can use virtual memory.

As described in subsection 4.1, access to physical memory is controlled via the vDTU's physical-memory protection (PMP) based on dedicated memory endpoints. As with all endpoints, only the controller can configure them. The first endpoint is predefined by the controller to a per-tile region in DRAM for TileMux, whereas the other endpoints are usable by activities.

The virtual-memory management is split between the controller, TileMux, and a *pager*. Like in M^{3}x, the pager is an OS service represented as an activity. The pager is responsible for address space layouts of other activities and policies such as demand loading and copy-on-write, similar as in L4 [32, 35, 57]. If the pager wants to map memory for an activity under its control, the pager sends a memory-mapping request to the controller. The pager needs to provide capabilities to the controller to proof the legitimacy of the mapping request. In contrast to M^{3}x, page-table entries are not updated by the controller, but the controller forwards the mapping request to the TileMux instance that is responsible for the activity. For that reason, the controller does not need to know the page-table format on other tiles. TileMux trusts the controller that the mapping is valid and manipulates the page-table entries accordingly. In any case, neither activities nor TileMux can access any memory outside the defined PMP regions and the M^{3}v controller makes sure that two tiles do not have access to the same memory region unless explicitly allowed.

4.4 Network Stack

To enable networking on M^{3}v, we added an OS service called *net* based on smoltcp [10]. Smoltcp is a standalone Rust-based TCP/IP stack for bare-metal systems. Net provides POSIX-like sockets for clients and uses a per-socket communication channel to exchange data and events with clients. For example, the client sends network packets as vDTU messages through this communication channel to net, which enqueues the packet for transmission.

Towards the network side, we adapted the AXI Ethernet standalone driver from Xilinx [5] and integrated it with smoltcp (resulting in a single software component). In our current hardware platform, the corresponding AXI Ethernet NIC is attached to a dedicated core. Therefore, net is always placed on this core.

5 ISOLATION VS. EFFICIENCY TRADE-OFF

The M^3 architecture provides better security—in terms of per-core isolation—when compared to prevalent off-the-shelf CPUs like the x86 family. Isolation is improved because M^3 cores are physically isolated from each other and share no resources, whereas x86 cores share resources like caches, opening the possibility of side-channel exploits [33, 39].

Both M^{3}x and M^{3}v trade some isolation of M^3 for the ability to multiplex tiles. Namely, if activities share a tile, these activities share the tile's resources, opening the possibility of side-channel attacks based on these shared resources. M^{3}x implements tile multiplexing in the controller, whereas M^{3}v uses a tile-local multiplexer and a virtualized DTU. Due to these differences, M^{3}v further trades some isolation of M^{3}x for more efficiency. Compared to M^3, M^{3}v increases the complexity of the DTU, in hardware, in all tiles, whereas M^{3}x increases the complexity of the controller, in the software that runs on a single core. Assuming that more complexity increases the probability for exploitable bugs, both M^{3}x and M^{3}v are less secure than M^3. By compromising either the controller or any DTU, an attacker can take over the system in both M^{3}x and M^{3}v.

M^{3}v additionally introduces TileMux as a tile-local multiplexer. Like the multiplexer in M^{3}x, TileMux needs to maintain meta data about its activities (e.g., their timeslices), which can possibly be exploited by attackers. In contrast to M^{3}x, TileMux keeps the meta data on the same tile as the potentially malicious applications, whereas M^{3}x stores them in the controller and thereby physically isolated from applications. We therefore conclude that M^{3}v further trades off some isolation from M^{3}x to improve efficiency.

6 EVALUATION

In our evaluation, we first discuss the hardware/software complexity of the vDTU, the controller, and TileMux. Afterwards, we evaluate the efficiency of tile multiplexing with M^{3}v. We start with microbenchmarks to provide a basic understanding for the system's behavior, followed by a comparison to M^{3}x in terms of performance

Table 1: FPGA area consumption: Logic and LUT-RAM (LUTs), registers (Flip-flops, FFs), block RAM (BRAM) with 36 kbit per block.

	LUTs [k]	FFs [k]	BRAMs
BOOM	143.8	71.8	159
Rocket	46.6	22.0	152
NoC router	3.4	2.2	0
vDTU	15.2	5.8	0.5
Control Unit	10.3	3.3	0.5
NoC CTRL	3.2	1.5	0
CMD CTRL	7.1	2.8	0.5
Unpriv. IF	6.2	2.5	0.5
Priv. IF	0.9	0.3	0
Register file	2.0	1.0	0
Memory mapper + PMP	0.6	0.2	0
I/O FIFOs	2.3	0.3	0

and scalability with the number of tiles. Finally, we use application-level benchmarks to compare M^{3}v's performance to Linux 5.11. To run POSIX applications on M^{3}v, we use a port of the musl [9] C library, which translates a subset of the Linux system calls to the corresponding API calls of M^{3}v. If not explicitly stated otherwise, all measurements run on the FPGA prototype, described in subsection 4.1. To enable performance comparisons to Linux, we run Linux bare-metal on a single tile of our FPGA prototype using the bootloader from RISC-V-PK [12]. Note that Linux runs only on a single tile, because tiles are not cache coherent, as required by Linux.

6.1 Hardware and Software Complexity

In contrast to previous M^3 works, we built a hardware prototype. Therefore, we evaluate the hardware complexity of the DTU and the costs for virtualizing it. Furthermore, we show the complexity of the software components that enable efficient tile multiplexing. For both hardware and software we use metrics as established proxies for complexity (FPGA gates, source lines of code).

Table 1 shows the consumed FPGA resources of the major components in our hardware platform. The presented vDTU configuration includes all extensions and corresponds to the implementation in the processing tiles including the RISC-V core. In comparison to the BOOM and Rocket core, the vDTU only requires 10.6% and 32.6% of the FPGA LUTs, respectively. Since the vDTU contains no memory or caches, the number of required BRAMs is negligible compared to the cores. This is an advantage especially when considering a real chip implementation where memory consumes a substantial part of the area. The DTU is virtualized by adding the privileged interface and privileged registers. This increases the size of the DTU's logic by 6% and adds four additional registers.

On the software side, the M^{3}v controller consists of 11.5k SLOC (Rust; 900 unsafe according to cargo-count [1]), to which TileMux adds 1.7k SLOC (Rust; 50 unsafe). In comparison, the NOVA microkernel [57], which is similar to the controller, consists of about 9k SLOC (C++).

6.2 Microbenchmarks

To get a better understanding of the system's behavior, we start the performance evaluation with microbenchmarks on the basic primitives in M^{3}v. As a reference, we show the performance of similar functionality in Linux. Applications on M^{3}v use remote procedure

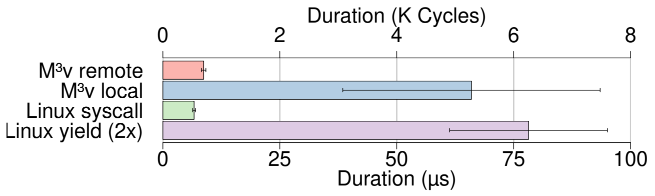


Figure 6: Local/remote communication on M³v and similar primitives on Linux as a reference.

calls (RPCs), consisting of a request and a response, to access system services or the M³v controller. Therefore, we measured the performance of no-op RPCs between tiles on M³v and show the performance of no-op system calls on Linux as a reference. Additionally, we measured the performance of tile-local no-op RPCs on M³v, which requires two context switches and is therefore similar to two switches between processes on Linux via the `yield` system call. Both M³v and Linux run on our FPGA platform. M³v runs the communication partners on one or two BOOM cores, and Linux uses a single BOOM core. We performed 1000 runs with a warm system.

The results in Figure 6 show that cross-tile communication ("M³v remote") is roughly as fast as a system call on Linux. On M³v, the RPC requires multiple interactions with the vDTU via MMIO to send the message, fetch the message on the receiver side, reply on the received message, fetch the reply on the sender side and mark the message as read. These vDTU interactions are done without involving TileMux or the M³v controller. Tile-local communication ("M³v local") is significantly more expensive, because it involves two interrupts as messages are received for non-running activities. Thus, TileMux is involved twice and needs to schedule the corresponding activity. However, as the results show, the costs are on a similar level as two yields (two context switches) on Linux, even though M³v requires several vDTU interactions for the two message transfers.

For reference, we note that M³x consumes 9 µs for tile-local RPC on gem5's 3 GHz out-of-order x86-64 core, translating into about 27k cycles. M³v requires about 5k cycles for the same operation, indicating a performance advantage over M³x. However, we acknowledge that these results cannot be directly compared, because M³x obtained these results on gem5's out-of-order CPU model, whereas M³v used an out-of-order BOOM core on our FPGA platform. Note also that M³v could employ a different mechanism for tile-local RPC, similar to microkernel-based systems [32, 35], to further increase its performance. However, we opted against this optimization to keep communication agnostic regarding the placement of activities.

6.3 OS Services

Following up on the previous microbenchmarks, we now compare the performance of basic OS services such as file systems and network stacks. The file system benchmarks compare the performance when reading and writing files using the POSIX read/write API and access the in-memory file system of M³v and Linux' tmpfs. For the extent-based file system of M³v, we limited the size of extents to 64 blocks. On M³v, we show the read and write performance with ("shared") and without tile sharing ("isolated"). All involved components (pager, file system, and the benchmark) share the same BOOM core in the former case, and run on separate BOOM cores in the latter

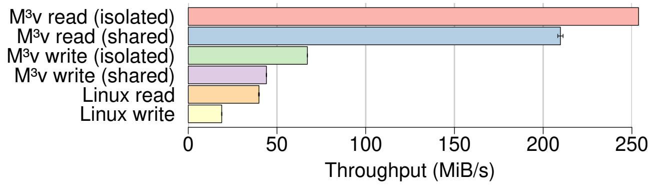


Figure 7: File read/write throughput comparison. The results without tile sharing ("isolated") cannot be compared to Linux, because in contrast to Linux, M³v uses multiple tiles.

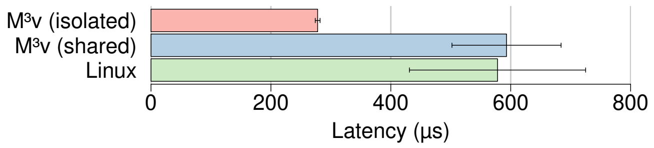


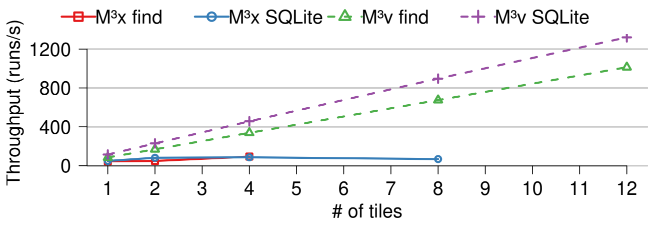
Figure 8: UDP latency comparison. The results without tile sharing ("isolated") cannot be compared to Linux, because in contrast to Linux, M³v uses multiple tiles.

case. On both systems we used 2 MiB files, a 4 KiB buffer for read and write operations, and performed 10 runs after 4 warmup runs.

As shown in Figure 7, the performance of read and write operations differs significantly between M³v and Linux. The reason for the performance differences is that a single read or write request to M³v's file system grants the application direct access to an entire extent. Afterwards, the application can perform the actual reads and writes via vDTU without involving the file system again until access to another extent is required. In contrast, every read or write operation on Linux is a system call involving the kernel. On both M³v and Linux, writes are much slower than reads, because blocks need to be allocated, cleared, and appended to a file. Note that these performance differences were also shown by earlier M³ prototypes in simulation [15, 16] and are now confirmed by M³v on an FPGA platform.

M³v shows better performance than Linux both with and without tile sharing. However, the results without tile sharing on M³v ("isolated") cannot be directly compared to Linux, because Linux cannot leverage multiple non-coherent tiles. In contrast, the results with tile sharing ("shared") are comparable to Linux, because both run the involved components on a single tile. Note that the M³v controller is rarely used during the benchmark, but is always called synchronously, so that M³v does not take advantage of the additional tile. As shown in Figure 7, tile sharing has an impact on M³v's read and write performance. The reason is that requests for new extents require a call to the file system via a tile-local RPC, involving two context switches. However, independent of tile sharing, all data accesses are performed directly via vDTU. Note that the throughput numbers are smaller than the throughput of today's systems, because the BOOM cores on our FPGA prototype are clocked with only 80 MHz.

To compare basic network performance, we measured the UDP latency between a sender running on a BOOM core of our FPGA platform and a receiver on an AMD Ryzen machine that is directly connected via Ethernet. We used 50 repetitions of sending and receiving 1 byte packets after 5 warmup runs. The results are shown in Figure 8. Like in the file-system benchmarks, the numbers without



M³x find throughput (runs/s): 45 (1 tile), 49 (2 tiles), 94 (4 tiles)

M³x SQLite throughput (runs/s): 49 (1 tile), 82 (2 tiles), 86 (4 tiles), 68 (8 tiles)

Figure 9: Scalability of context-switch-heavy applications on M³x and M³v with tile multiplexing.

tile sharing on M³v ("isolated") cannot be compared to Linux. With tile sharing ("shared"), M³v shows competitive performance to Linux.

6.4 Comparison with M³x

After the microbenchmarks, we now compare the efficiency of tile multiplexing on M³v with M³x. Since M³x does not run on the FPGA platform, we use gem5 and replicate benchmarks previously performed with M³x [14] on M³v with the same settings on gem5. In contrast to RISC-V as in the other benchmarks, we use a 3 GHz out-of-order x86-64 core in each tile. To stress both systems, the benchmark uses communication-heavy applications based on system call traces. We use traces from "find" and "SQLite", both accessing an in-memory filesystem. The find benchmark searches through 24 directories with 40 files each, whereas the SQLite benchmark performs 32 database inserts and selects. These traces were recorded on Linux and are replayed on both systems using a *traceplayer*. We execute one traceplayer on each tile and connect it to a filesystem instance on the same tile. Therefore, all calls to the file system require a context switch from the traceplayer to the filesystem and back.

Figure 9 shows the results when executing these benchmarks with one up to 12 tiles. The y-axis shows the number of application runs per second after one warmup run. As can be seen, the throughput of M³x improves only slightly with an increasing number of tiles. The reason for this behavior is the single-threaded controller performing all context switches on all tiles. Note that the benchmark on M³x does not run reliably with higher tile counts (8 and 12 for find, 12 for SQLite) and therefore these numbers are missing in Figure 9. In contrast, M³v scales almost linearly up to 12 tiles. Since context switches are performed tile-local, scalability is only limited by other shared resources in the system such as the controller. Finally, with a single tile, M³x executes 45 and 49 applications per second with "find" and "SQLite", respectively, whereas M³v executes 84 and 111, constituting a performance improvement of about 2x.

6.5 Macrobenchmarks

We want to frame our application-level performance evaluation within a larger scenario to show the applicability of our results. We did not implement this scenario end to end, but we zoom into specific aspects, which we quantify with individual experiments. We envision our platform within an example Internet-of-Things (IoT) device and within a cloud server. Our IoT example is a simplified voice assistant, which continuously listens to room audio and scans for a trigger word. Once the trigger is detected, audio data is compressed and sent to the cloud for further processing. As one part of a larger voice recognition pipeline, our cloud service stores extracted observations in a key-value store for retrieval by other services.

In a world where we consider software and hardware vulnerable, we must think of the trust domains within our applications and how to map them to cores. On the IoT voice assistant, it is crucial that audio data remains on the device until the trigger word is detected. Access to the audio stream should be protected even when the attacker compromises the network stack and exploits a vulnerability in the underlying core. A similar argument can be made for the cloud use case, where multi-tenancy is an important requirement. The key-value store may hold sensitive data, which needs to be protected from breaches of inter-tenant isolation.

6.5.1 Voice Assistant. For the first part of the application-level benchmarks, we zoom into the introduced voice assistant as it could run on an IoT device. The voice assistant consists of four components: 1) the *scanner* that looks for trigger words, 2) a *compressor* that receives selected audio samples from the scanner and sends them to the cloud, 3) the network stack for these transmissions, and 4) the pager that manages the address spaces of the former two. The scanner does not use a pager, but gets all pages mapped right away to minimize its trusted computing base. To receive audio samples, the scanner delegates a memory capability to the data in memory to the compressor. The compressor uses libFLAC [2] as a lossless compression and sends the result to another machine.

To show the overhead of tile multiplexing, we map this scenario to our platform by either placing all components but the scanner on the same tile to save resources or by placing each component on a dedicated tile. The scanner runs on a separate Rocket core to strongly isolate it from the other components, which run on a complex out-of-order BOOM core. In reality, the compressor would send the audio data via TCP to the cloud service. Unfortunately, even with a direct Ethernet connection between the FPGA and another machine, we observed several packet drops, which made it impossible to get reproduceable results on M³v[3]. We therefore decided to send the data out via UDP and ignore lost packets.

With 16 repetitions after warmup we obtained 384 ms without sharing and 398 ms with sharing, constituting an overall sharing overhead of 3.6%. However, note that this overhead does not only stem from the actual context switches, but also from the fact that the compressor, the network stack, and the pager compete for the same core.

6.5.2 Cloud Service. Finally, we evaluate the cloud side of our voice-activation system. The cloud service hosts leveldb [8] as a key-value store to aggregate data from IoT devices and perform further analysis on the data. In particular, it offers the ability to answer requests on the stored data. Therefore, the scenario comprises four components: 1) the *database* using leveldb and handling requests, 2) the file system as a backend for leveldb, 3) the network stack to receive and answer requests, and 4) the pager to manage their address spaces. The question we strive to answer is whether M³v delivers competitive performance for such a complex and communication-heavy application scenario with and without tile sharing.

To support networking on Linux, we used the available Xilinx Linux driver for AXI Ethernet NICs with DMA support [4]. Like for

[3]We suspect problems in smoltcp in combination with the vast performance difference between the 80 MHz BOOM core on the FPGA and an AMD Ryzen 7 2700X on the other side, which prevents sender and receiver to get in sync.

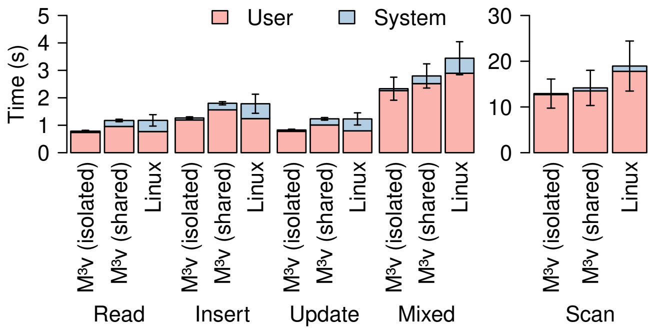


Figure 10: Cloud service in comparison to Linux. The results without tile sharing ("isolated") cannot be compared to Linux, because in contrast to Linux, M3v uses multiple tiles.

the voice assistant, TCP was not usable due to huge variations on M³v. Since UDP does not allow to receive data reliably and, most importantly, reproducably between runs and fair between M³v and Linux (random packet drops make the results incomparable), we decided to not just send the results to another machine via UDP, but also the requests. For that reason, the database reads the requests ahead of time from a file, executes them afterwards and sends the requests and results via UDP to another machine.

To let the database handle a realistic workload, we use the Yahoo! Cloud Serving Benchmark (YCSB) [6], which supports insert, update, read, and scan operations. Scan operations are the most expensive, because they need to walk through large parts of the data to find a range of values. For that reason and also to understand the behavior of different operations, we execute read-heavy, insert-heavy, update-heavy, and scan-heavy workloads. The first three omit the scan operation and use a proportion of 80-10-10 (e.g., 80% reads, 10% inserts, 10% updates). The scan-heavy workload omits updates and has a 80-10-10 proportion for the other three. Finally, we use a mixed workload using a 50-10-30-10 proportion for reads, inserts, updates, and scans, respectively. All workloads are generated with the Zipfian distribution and are based on 200 records that are created first, followed by the execution of 200 operations on these records according to the stated proportions.

Figure 10 shows the result of 8 runs after 2 warmup runs as a comparison between M³v with isolated tiles ("isolated"), with one shared tile for all four components ("shared"), and Linux. The plot shows the total runtime in seconds split into user and system time. On Linux, user and system time was obtained by `getrusage`. On M³v, time spent by the file system and network stack is considered system time, whereas the remainder is accounted as user time. For implementation-specific reasons, time spent by TileMux and the pager is also accounted as user time on M³v, resulting in more user time than on Linux. The results depend on the workload, because inserts and updates cause file system writes, whereas scans are memory intensive. All workloads cause UDP network traffic.

In general, M³v benefits from direct data transfers with the vDTU as indicated by the microbenchmarks. As before, the results of M³v with isolated tiles ("isolated") cannot be compared to Linux, because Linux runs on a single tile. We therefore show the results with isolated tiles only for completeness. Placing all components on a single tile ("shared") leads to a slowdown as more components compete for the same core, but M³v shows still competitive performance for reads, inserts, and updates. However, Linux performs worse than M³v ("shared") for scans. We suspect, that the small L1 instruction cache (16 kB) and Linux' code size cause the application to lose most of its state on every system call, which happen frequently during this benchmark. M³v's smaller components lead to less cache footprint and due to the M³v architecture, many file system calls can be handled via the vDTU without context switches. However, we believe that Linux could benefit from extensions like FlexSC [56] to reduce its cache footprint. Note that the M³v controller runs on a simpler and more trustworthy Rocket core, but is rarely involved during the benchmark. If it is involved, it is called synchronously, so that M³v does not take advantage of the additional tile. Finally, we want to highlight that even with tile sharing, M³v isolates all participating components from each other like in microkernel-based systems, whereas Linux executes the file system, network stack, and pager in privileged mode without any isolation.

7 RELATED WORK

M³v implements core multiplexing by virtualizing the DTU in the context of a tiled architecture. We therefore discuss related work for I/O device virtualization, architectures distributed across physically separate compute resources, and inter-process communication.

I/O Device Virtualization. Bypassing the kernel has been identified as a key design principle [50] for fully utilizing the performance offered by modern I/O devices [44]. Kernel-bypass also enables application-specific shortcuts for interactions across multiple devices like streaming data from storage to the network [43]. Contemporary userspace driver libraries like DPDK [7] and SPDK [60] have evolved into complete system architectures like Demikernel [62]. Naturally, the performance benefits of kernel bypass should be retained when virtualizing devices to share them among multiple clients. For instance with storage devices, software-based approaches like SPDK-vhost [61] or MDev-NVMe [49] implement device virtualization by means of dedicated virtualization servers that use shared memory for communicating with their clients, thus avoiding the tax of entering and returning from the kernel. However, such approaches consume a significant amount of CPU resources, as they have to poll on said memory regions. ELI [24] has demonstrated how I/O device virtualization can use blocking communication while still achieving near-native performance by handling interrupts directly within virtual machines, but supports only direct device assignment to a single client. Solutions like LeapIO [37] extend the I/O device with additional hardware to allow secure sharing among multiple clients. I/O devices later added standard functionality similar to ELI (called IOMMU posted interrupts) that also allows multiple clients.

To compare the vDTU against SR-IOV, let us establish a M³v-like system based on SR-IOV: Consider a set of virtual machines (VMs) as applications. Assume that 1) all these application VMs executes on a single physical server, 2) each VM is allocated its own SR-IOV NIC instance, 3) the memory accesses of each NIC instance are constrained by the IOMMU to only access the private memory address space of the corresponding application, and 4) the system is configured such that the only means of communication between applications is via their SR-IOV NIC instances. In such a system, these SR-IOV NIC instances share several commonalities with vDTU communication channels, notably, that they are implemented as replicated state in hardware, and they provide direct communication between applications, shielding them from each other in the presence of application multiplexing. The differences are: 1) the SR-IOV channels operate across PCI, whereas vDTU channels operate across the much faster network-on-chip, 2) the SR-IOV hardware state is implemented in a single component (the NIC), whereas vDTU state is distributed (each vDTU stores state of the applications that execute on the associated core), and 3) the IOMMU provides full address translation services, including page table walks and, potentially [36], a page fault protocol, whereas the vDTU only supports a software-loaded (IO)TLB.

Physically Separate Compute Resources. Barrelfish [17] introduced the multikernel concept, which runs an independent kernel instance on each core and uses message passing to communicate between kernel instances. The idea of remote system calls by placing kernel code on dedicated cores was explored by FlexSC [56]. Within large-scale distributed systems, CapNet [20] has promoted the idea of using capabilities to manage communication permissions. Similarly, Caladan [58] proposed a distributed capability-based OS for data centers. Caladan runs a distributed kernel on Smart NICs and uses RDMA for cross-node communication. M^3 and its extension M^{3}v run a capability-based OS on a tiled architecture, communicate between tiles via vDTU, and run a controller on a dedicated tile to setup communication channels.

Inter-Process Communication. Inter-process communication (IPC) has a long history [18, 21, 25, 32, 34, 38] and exists in different flavors such as shared memory, pipes, or messages. We focus on message-based IPC and discuss works related to the vDTU-based message passing between activities on M^{3}v. Message-based IPC is used in many microkernel-based systems [16, 26, 32, 35, 57] to exchange data between otherwise isolated applications and OS services. Early L4 prototypes [38] transferred messages in CPU registers between applications, whereas modern L4 systems [32, 35, 57] transfer messages based on pinned pages that are shared between application and kernel. DLibOS [42] is based on the Tilera architecture [59] and leverages its intercore messaging facility for IPC between cores without involving the kernel. Similarly, SkyBridge [45] proposed the usage of the `VMFUNC` instruction for kernel-bypassing communication. All these systems implement IPC using standard CPU features or extensions of the CPU instruction set. In contrast, IPC in M^{3}v is based on a dedicated and per-core hardware component. Similarly, the MAGIC component of the FLASH multiprocessor [34] is a dedicated hardware component that allows message passing between cores. However, in contrast to the fixed-function hardware implementation of the vDTU, MAGIC is implemented in software on a general-purpose processor. In addition to message passing, M^{3}v also uses the vDTU to control cross-tile memory accesses, which is similar to components like NoC-MPU [51] and DPU [23].

8 DISCUSSION AND FUTURE WORK

Legacy Support. Our implementation already employs a port of the musl [9] C library, which translates Linux system calls to the corresponding API calls of M^{3}v. This approach allowed us to easily run existing software such as leveldb or libFLAC on our platform. Further legacy support is imaginable by running a complete Linux on a tile with restricted access to tile-external resources through the vDTU.

Cache Coherency. Our current prototype does not maintain cross-tile cache coherency, which we deem sufficient for typical IoT devices. However, more complex use cases could benefit from this feature. We believe that M^{3}v could support cross-tile cache coherency, while still providing strong isolation between tiles. Building upon our physical memory protection, we could allow cross-tile cache-coherency traffic only if it conforms to the PMP restrictions of the participating tiles. In this way, each tile could have private memory areas, but also share selected memory areas with other tiles. Implementing and evaluating this idea is left for future work.

Accelerator Support. M^{3}v shares the simplified integration between general-purpose cores and special-purpose accelerators with M^3 and M^{3}x, allowing users to easily utilize them in a unified manner. For example, accelerators can run “autonomously” and access OS services like in M^{3}x. However, M^{3}v focuses on efficient multiplexing of general-purpose cores and does currently not support multiplexing of accelerators. We believe that accelerators can be multiplexed remotely as in M^{3}x, but the multiplexing should be implemented by an OS service on a user tile rather than by the controller. However, we leave accelerator multiplexing for future work.

Scalability. Our current prototype uses a single instance of the M^{3}v controller. Although the controller is rarely involved during runtime and primarily when new activities are set up, it will become a bottleneck with a large number of user tiles. However, as already shown by SemperOS [27], M^3 can scale to hundreds of user tiles when employing multiple controller instances. We believe that our current implementation can be extended similarly.

9 SUMMARY

M^{3}v builds upon M^{3}x but trades some of the isolation and generality of M^{3}x for more efficient core sharing. M^{3}x multiplexes cores and accelerators with the same mechanism and remotely from a dedicated core, whereas our work introduces a new approach to multiplex cores efficiently. M^{3}v improves the multiplexing efficiency with a core-local multiplexer and by virtualizing the on-chip hardware component for cross-core communication. Our results show that the additional hardware costs are modest. In comparison to M^{3}x, M^{3}v achieves a two-fold performance improvement when running workloads with frequent context switches on a single core and scales almost linearly with the number of cores. However, M^{3}v loses some isolation in comparison to M^{3}x, because the multiplexer keeps its state on the same core as the potentially malicious applications. Finally, we show that M^{3}v exhibits competitive performance in comparison to Linux, even for workloads with frequent context switches.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Dan Tsafrir, for his feedback and guidance in the revision process, as well as Boris Pismenny for his helpful suggestions. We further thank the anonymous reviewers for their comments and insights. This research was co-financed by public funding of the state of Saxony/Germany. It has also received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957216.

A ARTIFACT APPENDIX

A.1 Abstract

Our work studies a hardware/operating-system co-design and therefore requires custom hardware. The artifact contains the hardware platform in form of bitfiles for the Xilinx VCU118 FPGA. For the software side, it includes the source code of all M³v components, the modified Linux kernel we compared M³v against, and all scripts to run the benchmarks. Executing the scripts will build all required parts, run the experiments on the FPGA, and produce the plots. For comparison, the artifact also contains the raw results used for the plots in this paper. The artifact covers the results of subsection 6.2, subsection 6.3, and subsection 6.5.

A.2 Artifact Check-List (Meta-Information)

- **Program:** M³v operating system, Linux, RISC-V PK, buildroot, LevelDB, and Yahoo! Cloud Serving Benchmark (YCSB). All are included in the provided source code.
- **Compilation:** GCC cross compiler, Rust nightly-2021-04-19, and Vivado Lab 2019.1. The cross compiler is included and built automatically. The Rust compiler is downloaded and built automatically.
- **Hardware:** Xilinx VCU118 Evaluation Board Rev 2.0, a Quad Gigabit Ethernet FMC Card OP031-1V8 for an additional Ethernet port, and another machine that is connected to this Ethernet port.
- **Execution:** The experiments will run for about 1 hour.
- **Metrics:** We use execution time, latency, and throughput as metrics.
- **Output:** The scripts produce files with raw numbers and plots. The expected results are included.
- **How much time is needed to prepare workflow?:** 1 hour
- **How much time is needed to complete experiments?:** 1 hour
- **Publicly available?:** The source of the complete software part and the vDTU and NoC of the hardware part are publicly available. The FPGA bitfiles are also publicly available.
- **Code licenses:** M³v is available under GPLv2. Please refer to the LICENSE file in root directory for the licenses of the other contained components.
- **Archived:** `10.5281/zenodo.5863686`.

A.3 Description

A.3.1 How to Access. Both the hardware and software platform are available on Zenodo (10.5281/zenodo.5863686) and on Gitlab (https://gitlab.com/Nils-TUD/m3bench).

A.3.2 Hardware Dependencies. M³v requires a custom hardware platform, which we provide in form of bitfiles for the Xilinx VCU118 FPGA. Additionally, the FPGA board needs an Ethernet FMC card (Quad Gigabit Ethernet FMC Card OP031-1V8) for an additional Ethernet port besides the port to load programs onto the FPGA. This additional port should be connected to another machine to host the communication partners for our benchmarks. In our benchmarks, we connected the FPGA to an AMD Ryzen 7 2700X with a Realtek RTL8111/8168/8411 1Gb/s NIC and Linux 5.4.0.

A.3.3 Software Dependencies. Both M³v and Linux require specific C/C++ and Rust compilers for RISC-V. These will be downloaded and built automatically. Vivado Lab is used to program the FPGA with the provided bitfiles.

A.4 Installation

The repository needs to be cloned as follows:

```
git clone https://gitlab.com/Nils-TUD/m3bench.git --recursive
```

The repository contains a `README.md` with all instructions to build everything and run the experiments.

A.5 Evaluation and Expected Results

The expected raw results are contained in the `expected-results` directory and can be compared with the raw results produced by the evaluation in the `results` directory. For example, the following commands can be used to compare the results:

```
tail results/*.dat > res.txt
tail expected-results/*.dat > expected.txt
vimdiff res.txt expected.txt
```

A.6 Notes

Since the measurements are done on an FPGA platform and involve networking, small differences between the results reported in this paper and new runs are expected. For that reason, the plots in this paper show the standard deviation of the measurements.

Note also that benchmarks will be automatically repeated on failure, because some occasional failures are unavoidable (e.g., sometimes loading programs onto the FPGA fails due to UDP packet drops). Additionally, there are still some hardware/software bugs due to system's complexity that we have not found yet.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] 2017. kbknapp/cargo-count: a cargo subcommand for counting lines of code in Rust projects. https://github.com/kbknapp/cargo-count. (Accessed on August 11, 2021).

[2] 2019. FLAC - Free Lossless Audio Codec. https://xiph.org/flac/. (Accessed on August 11, 2021).

[3] 2020. M³: microkernel-based system for heterogeneous manycores. https://github.com/TUD-OS/M3. (Accessed on August 11, 2021).

[4] 2021. AXI Ethernet Linux Driver. https://github.com/Xilinx/linux-xlnx/tree/master/drivers/net/ethernet/xilinx. (Accessed on January 25, 2022).

[5] 2021. AXI Ethernet Standalone Driver. https://github.com/Xilinx/embeddedsw/tree/master/XilinxProcessorIPLib/drivers/axiethernet. (Accessed on January 25, 2022).

[6] 2021. brianfrankcooper/YCSB: Yahoo! Cloud Serving Benchmark. https://ycsb.site/. (Accessed on August 11, 2021).

[7] 2021. Data Plane Development Kit. https://www.dpdk.org. (Accessed on August 11, 2021).

[8] 2021. google/leveldb. https://github.com/google/leveldb. (Accessed on August 11, 2021).

[9] 2021. musl libc. https://musl.libc.org/. (Accessed on August 11, 2021).

[10] 2021. smoltcp | M-Labs. https://m-labs.hk/software/smoltcp/. (Accessed on August 11, 2021).

[11] 2022. Intel® I/O Acceleration Technology. https://www.intel.com/content/www/us/en/wireless-network/accel-technology.html. (Accessed on January 30, 2022).

[12] 2022. RISC-V Proxy Kernel and Boot Loader. https://github.com/riscv-software-src/riscv-pk. (Accessed on January 25, 2022).

[13] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, and John Koenig. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).

[14] Nils Asmussen. 2019. *A New System Architecture for Heterogeneous Compute Units.* Ph. D. Dissertation. Dresden University of Technology. https://os.inf.tu-dresden.de/papers_ps/asmussen-phd.pdf

[15] Nils Asmussen, Michael Roitzsch, and Hermann Härtig. 2019. M³x: Autonomous Accelerators via Context-Enabled Fast-Path Communication. In *2019 USENIX*

Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 617–632. https://www.usenix.org/conference/atc19/presentation/asmussen

[16] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. ACM, 189–203. https://doi.org/10.1145/2872362.2872371

[17] Andrew Baumann, Paul Barham, Pierre-Évariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 29–44. https://doi.org/10.1145/1629575.1629579

[18] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. 1989. Lightweight Remote Procedure Call. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles, SOSP 1989, The Wigwam, Litchfield Park, Arizona, USA, December 3-6, 1989*, Gregory R. Andrews (Ed.). ACM, 102–113. https://doi.org/10.1145/74850.74861

[19] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Computer Architecture News* 39, 2 (u 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[20] Anton Burtsev, David Johnson, Josh Kunz, Eric Eide, and Jacobus E. van der Merwe. 2017. CapNet: security and least authority in a capability-enabled cloud. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 128–141. https://doi.org/10.1145/3127479.3131209

[21] Kevin Elphinstone and Gernot Heiser. 2013. From L3 to seL4 what have we learnt in 20 years of L4 microkernels?. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 133–150. https://doi.org/10.1145/2517349.2522720

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

[23] L. Fiorin, G. Palermo, S. Lukovic, V. Catalano, and C. Silvano. 2008. Secure Memory Accesses on Networks-on-Chip. *IEEE Trans. Comput.* 57, 9 (Sept 2008), 1216–1229. https://doi.org/10.1109/TC.2008.69

[24] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. 2012. ELI: Bare-Metal Performance for I/O Virtualization. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 411–422. https://doi.org/10.1145/2150976.2151020

[25] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. 1997. The Performance of μKernel-Based Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, Michel Banâtre, Henry M. Levy, and William M. Waite (Eds.). ACM, 66–77. https://doi.org/10.1145/268998.266660

[26] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. 2006. MINIX 3: a highly reliable, self-repairing operating system. *ACM SIGOPS Oper. Syst. Rev.* 40, 3 (2006), 80–89. https://doi.org/10.1145/1151374.1151391

[27] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. 2019. SemperOS: A Distributed Capability System. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 709–722. https://www.usenix.org/conference/atc19/presentation/hille

[28] Tung Thanh Hoang, Amirali Shambayati, Calvin Deutschbein, Henry Hoffmann, and Andrew A. Chien. 2014. Performance and energy limits of a processor-integrated FFT accelerator. In *IEEE High Performance Extreme Computing Conference, HPEC 2014, Waltham, MA, USA, September 9-11, 2014*. IEEE, 1–6. https://doi.org/10.1109/HPEC.2014.7040951

[29] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA'17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[30] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 201–216. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kim

[31] Seonbong Kim and Joon-Sung Yang. 2018. Optimized I/O determinism for emerging NVM-based NVMe SSD in an enterprise system. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*. ACM, 56:1–56:6. https://doi.org/10.1145/3195970.3196085

[32] 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. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP'09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[33] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *meltdownattack.com* (2018). https://spectreattack.com/spectre.pdf

[34] Jeffrey Kuskin, David Ofelt, Mark A. Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John L. Hennessy. 1994. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture. Chicago, IL, USA, April 1994*, David A. Patterson (Ed.). IEEE Computer Society, 302–313. https://doi.org/10.1109/ISCA.1994.288140

[35] Adam Lackorzynski and Alexander Warg. 2009. Taming Subsystems: Capabilities As Universal Resource Access Control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems* (Nuremburg, Germany) *(IIES'09)*. ACM, New York, NY, USA, 25–30. https://doi.org/10.1145/1519130.1519135

[36] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. 2017. Page Fault Support for Network Controllers. *SIGARCH Comput. Archit. News* 45, 1 (apr 2017), 449–466. https://doi.org/10.1145/3093337.3037710

[37] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. 2020. *LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs*. Association for Computing Machinery, New York, NY, USA, 591–605. https://doi.org/10.1145/3373376.3378531

[38] Jochen Liedtke. 1995. On micro-Kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, Michael B. Jones (Ed.). ACM, 237–250. https://doi.org/10.1145/224056.224075

[39] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *meltdownattack.com* (2018). https://meltdownattack.com/meltdown.pdf

[40] Dao-Fu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Temam, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. PuDianNao: A Polyvalent Machine Learning Accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 369–381. https://doi.org/10.1145/2694344.2694358

[41] Weichen Liu, Wenyang Liu, Yichen Ye, Qian Lou, Yiyuan Xie, and Lei Jiang. 2019. HolyLight: A Nanophotonic Accelerator for Deep Learning in Data Centers. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, Jürgen Teich and Franco Fummi (Eds.). IEEE, 1483–1488. https://doi.org/10.23919/DATE.2019.8715195

[42] Stephen Mallon, Vincent Gramoli, and Guillaume Jourjon. 2018. 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* (Williamsburg, VA, USA) *(ASPLOS'18)*. ACM, New York, NY, USA, 737–750. https://doi.org/10.1145/3173162.3173209

[43] Ilias Marinos, Robert N. M. Watson, Mark Handley, and Randall R. Stewart. 2017. Disk|Crypt|Net: rethinking the stack for high-performance video streaming. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM*. ACM, 211–224. https://doi.org/10.1145/3098822.3098844

[44] Mellanox Technologies. [n. d.]. RDMA Aware Networks Programming User Manual.

[45] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 9:1–9:15. https://doi.org/10.1145/3302424.3303946

[46] Mark Samuel Miller. 2006. Robust composition: towards a unified approach to access control and concurrency control (Ph. D. thesis). *Johns Hopkins Univ ersity, Baltimore, Maryland, USA* (2006).

[47] Kenneth Moreland and Edward Angel. 2003. The FFT on a GPU. In *Proceedings of the 2003 ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware, San Diego, California, USA, July 26-27, 2003*, Bill Mark and Andreas Schilling (Eds.). Eurographics Association, 112–119. http://diglib.eg.org/handle/10.2312/EGGH.EGGH03.112-119

[48] Vincent Nollet, Paul Coene, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. 2003. Designing an Operating System for a Heterogeneous Reconfigurable So. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*. IEEE Computer Society, 174. https://doi.org/10.1109/IPDPS.2003.1213320

[49] Bo Peng, Jianguo Yao, Yaozu Dong, and Haibing Guan. 2022. MDev-NVMe: Mediated Pass-Through NVMe Virtualization Solution With Adaptive Polling. *IEEE Trans. Computers* 71, 2 (2022), 251–265. https://doi.org/10.1109/TC.2020.3045785

[50] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 1–16. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[51] J. Porquet, A. Greiner, and C. Schwarz. 2011. NoC-MPU: A secure architecture for flexible co-hosting on shared memory MPSoCs. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'11)*. 1–4. https://doi.org/10.1109/DATE.2011.5763291

[52] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark Horowitz. 2015. Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing. *Commun. ACM* 58, 4 (Mar 2015), 85–93. https://doi.org/10.1145/2735841

[53] Jerome Howard Saltzer. 1966. *Traffic control in a multiplexed computer system.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[54] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: integrating a file system with GPUs. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, Vivek Sarkar and Rastislav Bodík (Eds.). ACM, 485–498. https://doi.org/10.1145/2451116.2451169

[55] Hayden Kwok-Hay So and Robert W. Brodersen. 2008. File system access from reconfigurable FPGA hardware processes in BORPH. In *FPL 2008, International Conference on Field Programmable Logic and Applications, Heidelberg, Germany, 8-10 September 2008*. IEEE, 567–570. https://doi.org/10.1109/FPL.2008.4630010

[56] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, Remzi H. Arpaci-Dusseau and Brad Chen (Eds.). USENIX Association, 33–46. http://www.usenix.org/events/osdi10/tech/full_papers/Soares.pdf

[57] Udo Steinberg and Bernhard Kauer. 2010. NOVA: a microhypervisor-based secure virtualization architecture. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, Christine Morin and Gilles Muller (Eds.). ACM, 209–222. https://doi.org/10.1145/1755913.1755935

[58] Lluis Vilanova, Lina Maudlej, Matthias Hille, Nils Asmussen, Michael Roitzsch, and Mark Silberstein. 2020. Caladan: A Distributed Meta-OS for Data Center Disaggregation. *10th Workshop on Systems for Post-Moore Architectures, SPMA* (2020).

[59] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. 2007. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro* 27 (10 2007), 15–31. https://doi.org/10.1109/MM.2007.89

[60] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2017, Hong Kong, December 11-14, 2017*. IEEE Computer Society, 154–161. https://doi.org/10.1109/CloudCom.2017.14

[61] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang Cao. 2018. SPDK Vhost-NVMe: Accelerating I/Os in Virtual Machines on NVMe SSDs via User Space Vhost Target. In *8th IEEE International Symposium on Cloud and Service Computing, SC2 2018, Paris, France, November 18-21, 2018*. IEEE, 67–76. https://doi.org/10.1109/SC2.2018.00016

[62] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 195–211. https://doi.org/10.1145/3477132.3483569

[63] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. (May 2020).